HPA整体流程及gtsam简介

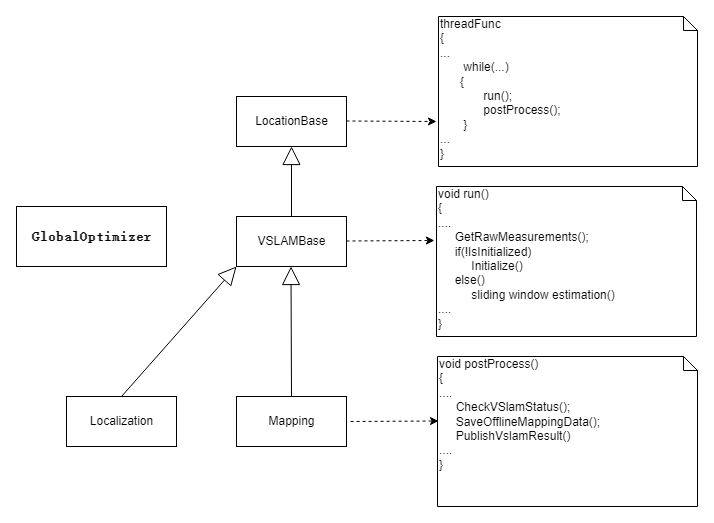
**一、HPA整体介绍**

**1.1 三个主要过程**

1、**在线建图** Online Mapping

2、**离线建图** Offline Global Optimizer

3、**定位跟踪** Localization



LocationBase 位于holo\_localization\_base 仓库中，其余都在 holo\_localization\_parking 仓库中。

**非线性优化三要素**

* 误差项
* 优化变量
* 协方差

**1.2 主要传感器输入**

**1、PointFeature 特征点信息(感知fd节点)**

**2、ChassisState 底盘状态信息(轮脉冲和车身速度信息)**

**3、imu消息 (imu预积分)**

**1.3 在线建图**

主要流程包括 传感器数据接收、初始化、滑窗优化、后处理部分等。

**1.3.1 静止初始化**

// comput initial rotation and imu bias

Vector3 sum\_angular\_vel(0.0, 0.0, 0.0);

Vector3 sum\_linear\_acc(0.0, 0.0, 0.0);

for (const auto& imu : imu\_msg\_buffer)

{

sum\_angular\_vel += imu.GetAngularVelocity();

sum\_linear\_acc += imu.GetLinearAcceleration();

}

// initialize the gyro bias

ImuBias bias;

bias.SetAngularVelocityBias(sum\_angular\_vel / imu\_msg\_buffer.size());

// initialize the initial orientation

const Vector3 mean\_linear\_acc = sum\_linear\_acc / imu\_msg\_buffer.size();

const Vector3 earth\_gravity\_direction(0.0, 0.0, 1.0);

const auto **q\_bw** = Eigen::Quaterniond::**FromTwoVectors**(

HoloGtsam::Convert(mean\_linear\_acc),

HoloGtsam::Convert(earth\_gravity\_direction)).coeffs();

Rot3 r\_wb(**Quaternion**(q\_bw[3], q\_bw[0], q\_bw[1], q\_bw[2]).ToRotationMatrix().

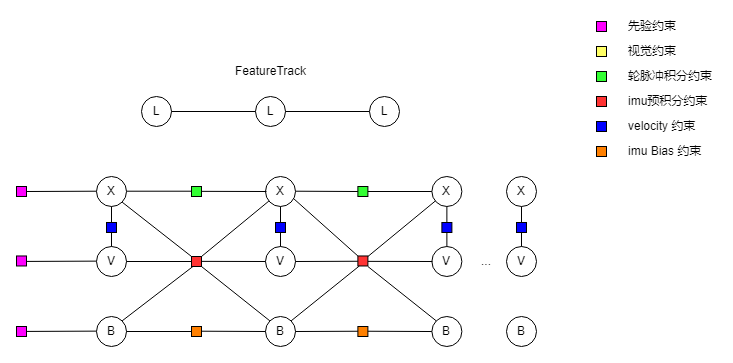
Transpose());

通过静止状态，计算imu bias, imu至少要求200帧 ，如果100Hz的话，静止时间2s以上。

初始化过程计算了初始位姿 (主要是计算rotation ，translation 默认为 (0，0，0)),根据重力方向和静止时间内的加速度计mean 值进行旋转姿态计算（个人感觉应该是考虑到车辆处于坡道之中时的状态 ）。

初始化过程中只计算了陀螺仪的bias 偏置 bg，忽略了加速度的bias偏置ba，因为加速度计与重力耦合，并且重力向量很大，初始化过程动态过程很短，幅度又不大，加速度偏置很难观测到。（这一段可以参考VINS的初始化过程）

**1.3.2 滑窗优化**



视觉部分不参与计算，只计算 FeatureTrack ,视觉 MapPoint的计算在离线阶段进行 。

**1.3.3 后处理部分**

在线建图的中间数据采用本地文件保存的方式进行。

**1.4 离线建图**

主要步骤梳理如下：

1、三角化所有candidate tracks , 创建 FramesDatabase ；

2、局部地图关联（ local map association）；

3、回环检测和几何一致性校验；

4、根据回环检测结果做 feature track Merging ;

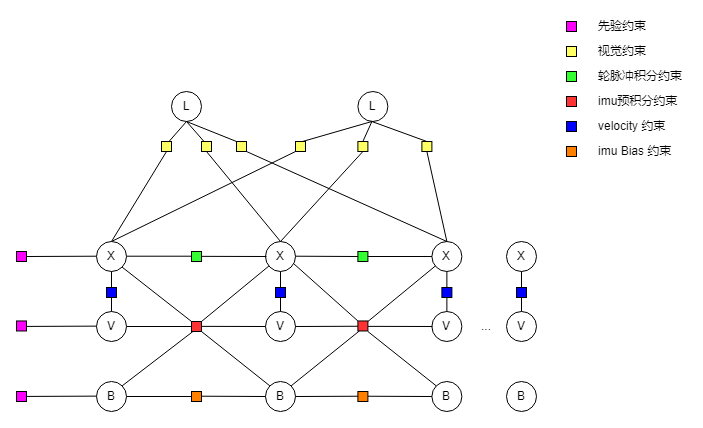
5、因子图构建；

6、Global Bundle Adjustment 后去除外点；

7、根据全局优化之后的结果重新优化输出结果。

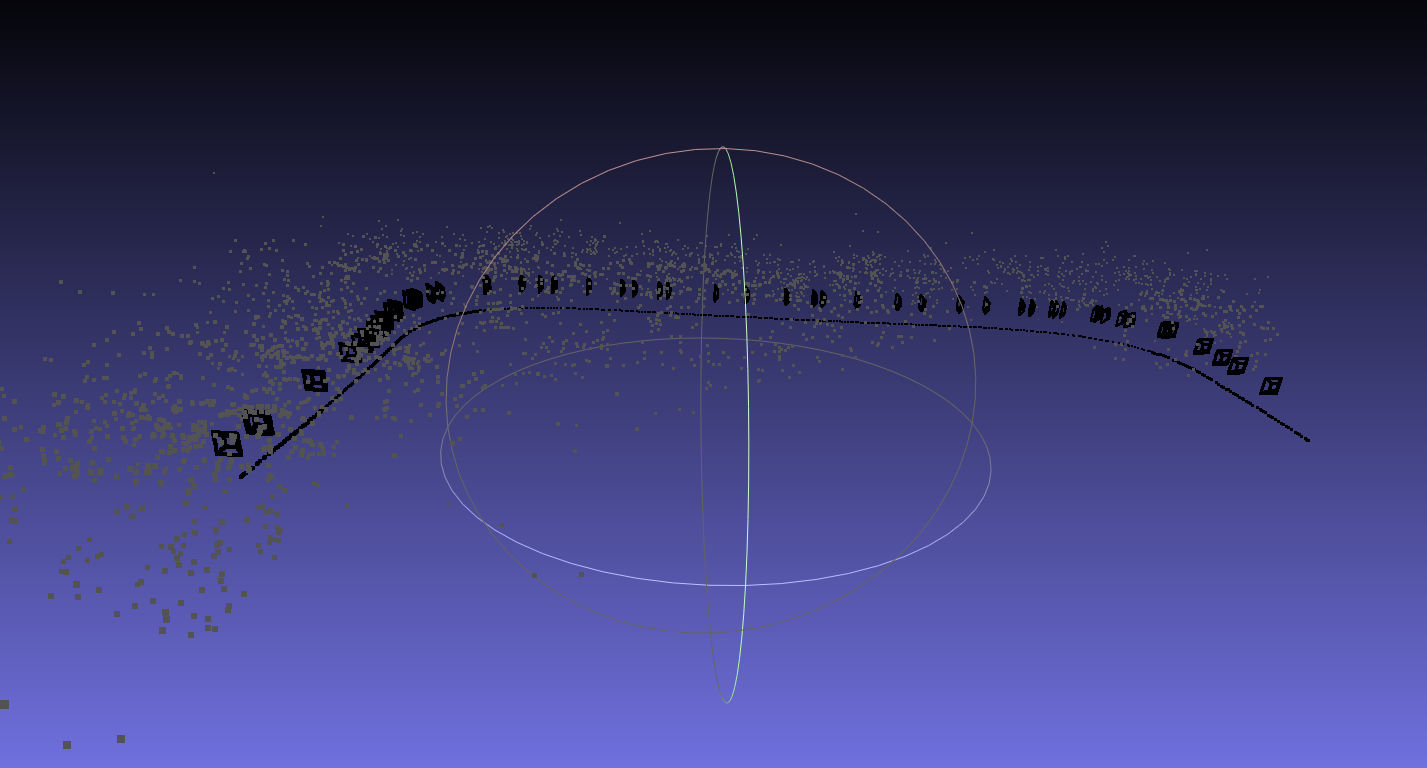
三角化部分是调用 gtsam 现成的函数 gtsam::triangulatePoint3() ，会判定三角化出来的点是否在相机前方，否则会throw bad 。

对在线建图得到的OfflineMap数据进行处理，进行全局因子图优化。初始值为，在线建图时生成的**xvb\_values** ，因子图构建还是采用 XVB模型，新增了视觉约束因子，新增了视觉变量L ，值为三角化生成的点。整体因子图如下所示，构建约束方程，进行整体优化。



构建完成后，通过 GlobalBundleAdjustment() 进行整体优化求解。

离线建图结果：



离线建图数据包括：Frame, KeyFrame, 点云数据.

备注：

这一部分耗时比较长（4～5s），因为HPA实际业务场景，可以把全局优化部分从在线建图阶段拎出来。

**1.5 定位**

**1.5.1 初始化**

通过 **Relocalization** 对当前位置进行初始化。

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

boost::optional<Pose3> Relocalization::**RunSynchronously**(

const std::vector<Features>& multi\_camera\_features,

const boost::optional<Pose3>& **prior\_vehicle\_pose**) const

{

// trying to relocaization by each camera data

for (uint16\_t camera\_index = 0; camera\_index < multi\_camera\_features.size(); camera\_index++)

{

const auto& **query\_features** = multi\_camera\_features.at(camera\_index);

const auto& intrinsic = camera\_parameters\_->intrinsics.at(camera\_index);

const auto& T\_bc = camera\_parameters\_->extrinsics.at(camera\_index);

const auto **associated\_results** =

**SolvePoseByFrameToFrameMatching**(

**key\_frames\_database\_**,

**query\_features**,

**prior\_vehicle\_pose**);

// check if solved pose successfully

if (associated\_results.empty())

{

continue;

}

// retrieve best camera pose

auto **best\_Twc** = **associated\_results**.front().T\_wc;

// create new pose only projection factor measurements

auto inliers\_mpms =

**CreatePoseOnlyProjectionFactorMeasurements**(

key\_frames\_database\_,

**query\_features**,

**associated\_results**);

// optimize camera pose

if (parameters\_.enable\_pose\_only\_ba)

{

const auto ba\_result = **PoseOnlyBundleAdjustment**(

inliers\_mpms,

associated\_results.front().T\_wc,

intrinsic);

best\_Twc = ba\_result.first;

inliers\_mpms = ba\_result.second;

}

// check if optimization failed

if (inliers\_mpms.empty())

{

continue;

}

return best\_Twc \* T\_bc.Inverse();

}

return boost::none;

}

总结：

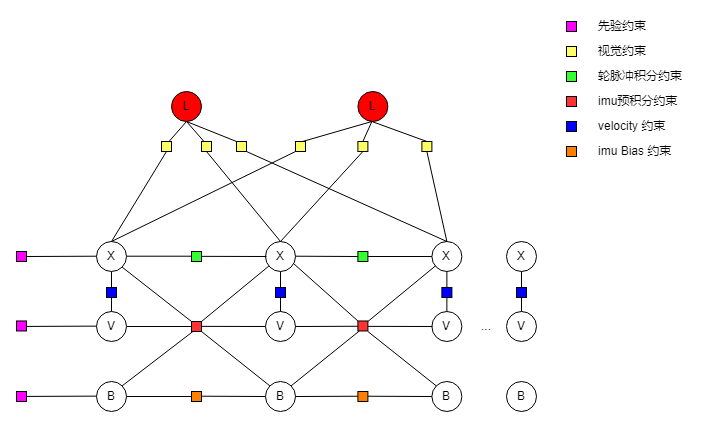
1、根据 关键帧找到关联的features；

2、根据 BruteForceMatching ,进行 features assocation, 找到mappoints;

3、根据 ransac PnP solver 进行位姿解算（3D - 2D）；

4、PoseOnlyBundleAdjustment 进行优化。

**1.5.2 滑窗优化**



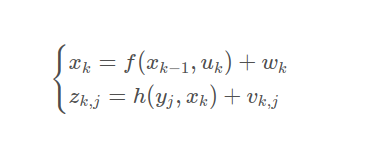
Landmark 在 Localization 阶段并不参与因子图的优化，仅优化位姿。**（PoseOnlyUnit3Factor）**

**二、非线性优化**

https://blog.csdn.net/weixin\_44413191/article/details/107623351

**2.1 状态估计问题**

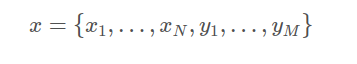
**2.1.1 SLAM问题数学模型**



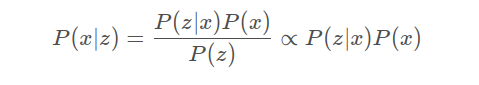
假设噪声服从高斯分布



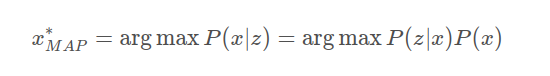
状态变量：所有待求解的量，即所有时刻的位姿和路标。



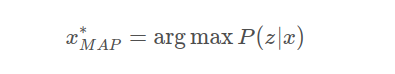
根据贝叶斯法则：



最大后验估计（MAP）(Maximize a Posterior)



最大似然估计（MLE）(Maximize Likelihood Estimation)



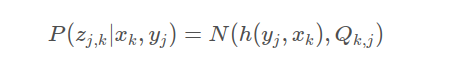
物理意义:

在哪种状态下，最容易产生当前的观测值。

**2.1.2 最小二乘问题**

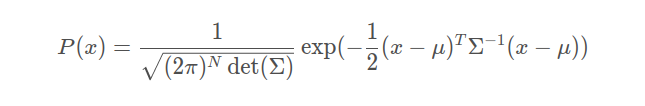
对于某次观测，噪声满足高斯分布 $$v\_{k,j} \backsim (0, Q\_{k,j} )$$



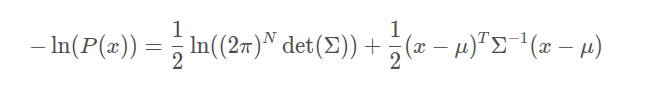


现在要使得该高斯分布的值最大，即求解这个高斯分布的最大似然。

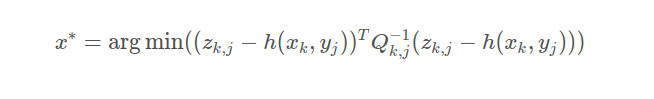
对于一般的高斯分布：



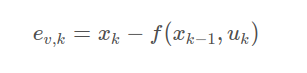
其负对数形式如下：

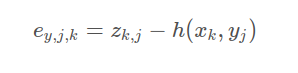


将二次型项带入 SLAM 观测模型中有：

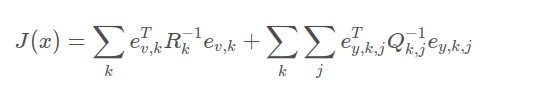


定义误差项





误差的平方和：



直观的解释是，由于噪声的存在，我们估计的优化变量带入 SLAM 的方程中不能很好地成立，我们需要调整状态的估计，使得误差最小化。

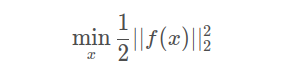
该问题的结构：

1. 由多个误差的平方和组成；
2. 虽然整体维度高，但每个误差项简单，仅与一两个状态变量有关；
3. 如果用李代数表达位姿，是无约束优化问题。

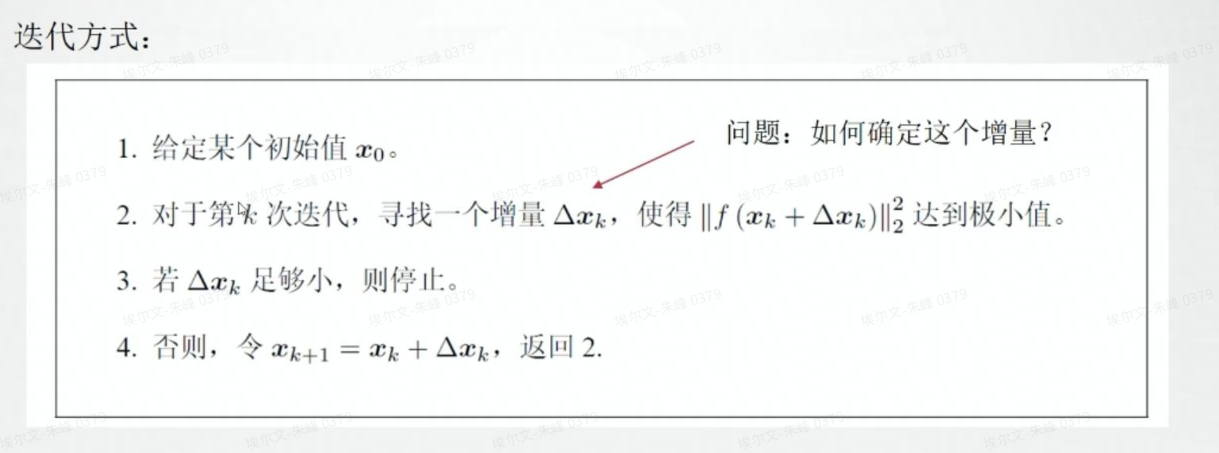
至此，我们将一个最大似然问题转换成了非线性最小二乘问题。

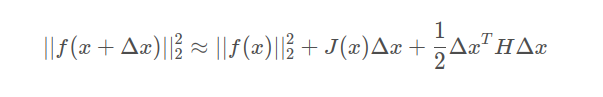
**2.2 高斯牛顿法**

**2.2.1 非线性最小二乘**



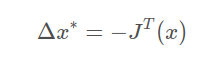
**通过梯度下降法迭代求解（负梯度方向）。**





**a、最速下降法**

如果保留一阶项，令增量方程导数为 0，增量的解为：



**b、牛顿法**

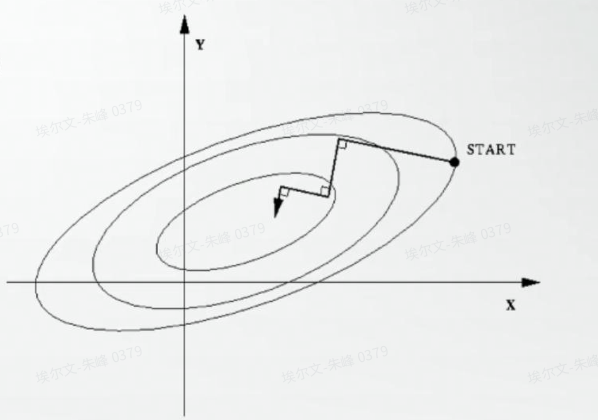
如果保留二阶项，令增量方程导数为 0，增量的解为：



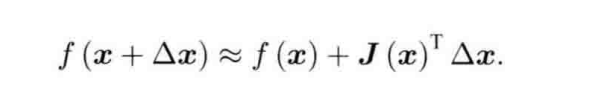
注：

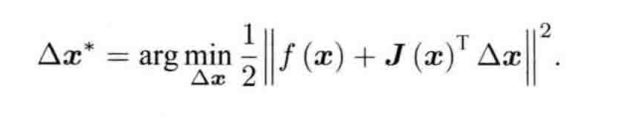
一阶和二阶梯度下降法十分直观，但各自都有缺点：

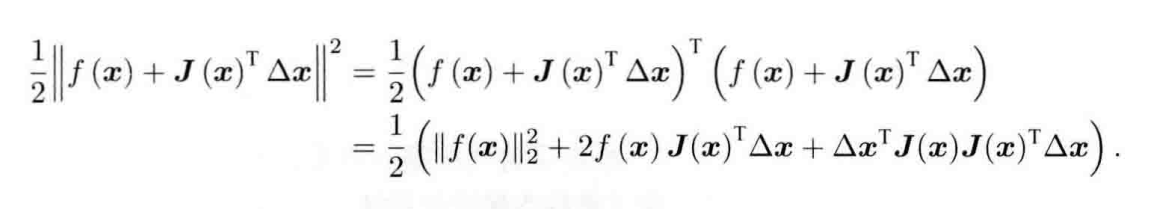
1. 最速下降法过于贪心，走出锯齿线路，反而增加了迭代次数；
2. 牛顿法要计算 Hessian 矩阵，在问题规模较大时难以求解。

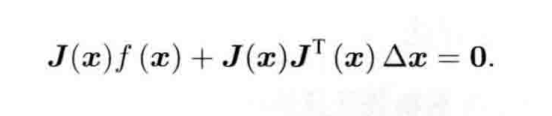


**2.2.2 高斯牛顿法**

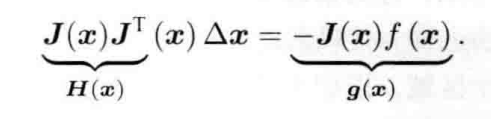




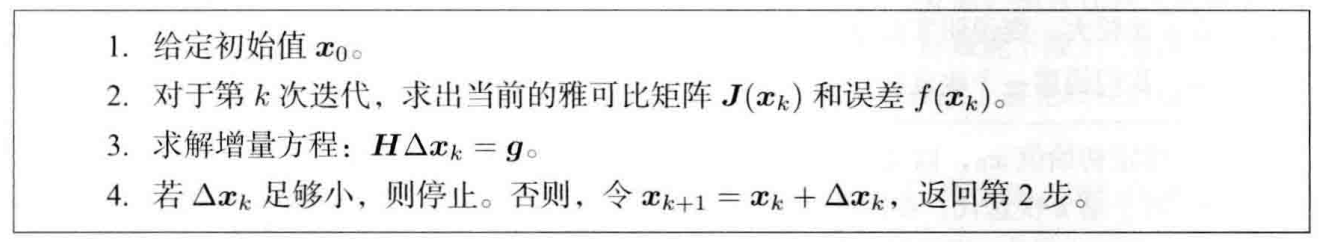




增量方程



高斯牛顿法迭代步骤



1、高斯牛顿法的不足：

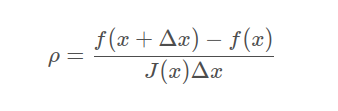
$$JJ^T$$是半正定阵，有可能出现病态矩阵的情况

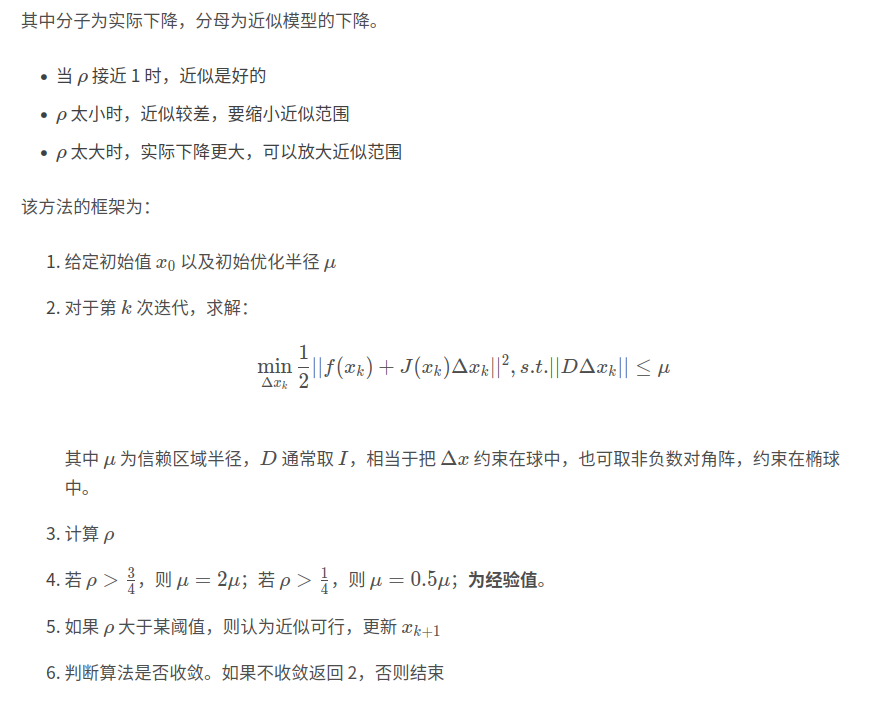
2、非线性优化**三要素**

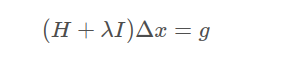
* 误差项
* 优化变量
* 协方差

**2.2.3 Levenberg-Marquardt (LM方法)**

使用近似模型和实际函数的差异来进行近似程度的描述，其中分子为实际下降，分母为近似模型的下降。







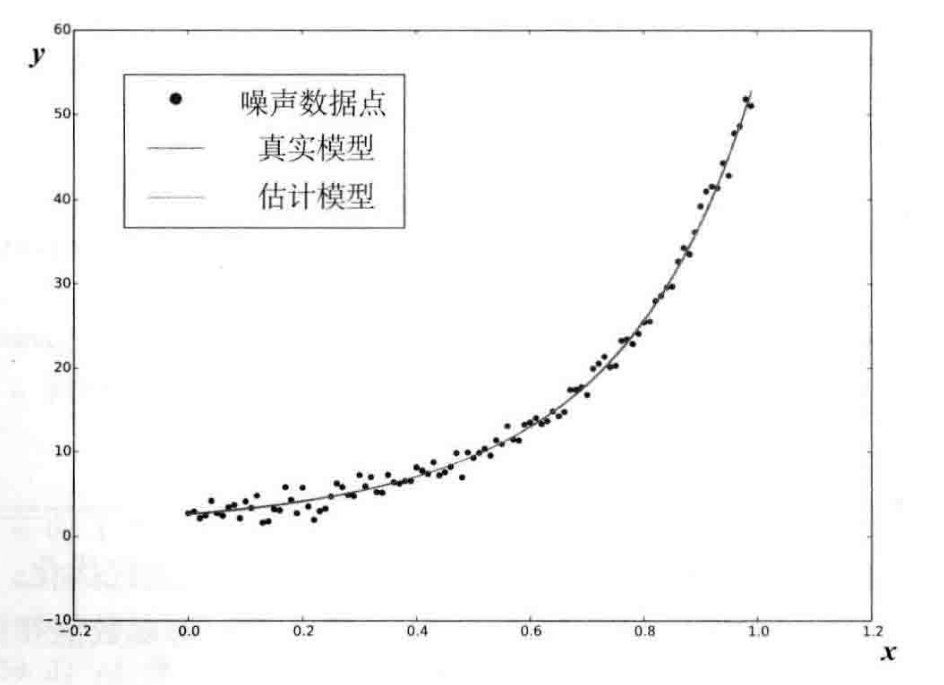
**2.3 案例**

问题描述

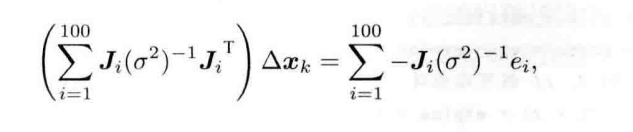
考虑一条满足以下条件的曲线.

$$y = exp (ax^2+ bx + c) + w$$

其中 a,b,c为曲线参数，w为高斯噪声，满足 $$w \backsim (0, \sigma^2 )$$



高斯牛顿增量方程：



代码实现

#include <iostream>

#include <chrono>

#include <opencv2/opencv.hpp>

#include <Eigen/Core>

#include <Eigen/Dense>

using namespace std;

using namespace Eigen;

int main(int argc, char \*\*argv) {

double ar = 1.0, br = 2.0, cr = 1.0; // 真实参数值

double ae = 2.0, be = -1.0, ce = 5.0; // 估计参数值

int N = 100; // 数据点

double w\_sigma = 1.0; // 噪声Sigma值

double inv\_sigma = 1.0 / w\_sigma;

cv::RNG rng; // OpenCV随机数产生器

vector<double> x\_data, y\_data; // 数据

for (int i = 0; i < N; i++) {

double x = i / 100.0;

x\_data.push\_back(x);

y\_data.push\_back(exp(ar \* x \* x + br \* x + cr) + rng.gaussian(w\_sigma \* w\_sigma));

}

// 开始Gauss-Newton迭代

int iterations = 100; // 迭代次数

double cost = 0, lastCost = 0; // 本次迭代的cost和上一次迭代的cost

chrono::steady\_clock::time\_point t1 = chrono::steady\_clock::now();

for (int iter = 0; iter < iterations; iter++) {

Matrix3d H = Matrix3d::Zero(); // Hessian = J^T W^{-1} J in Gauss-Newton

Vector3d b = Vector3d::Zero(); // bias

cost = 0;

for (int i = 0; i < N; i++) {

double xi = x\_data[i], yi = y\_data[i]; // 第i个数据点

double error = yi - exp(ae \* xi \* xi + be \* xi + ce);

Vector3d J; // 雅可比矩阵

J[0] = -xi \* xi \* exp(ae \* xi \* xi + be \* xi + ce); // de/da

J[1] = -xi \* exp(ae \* xi \* xi + be \* xi + ce); // de/db

J[2] = -exp(ae \* xi \* xi + be \* xi + ce); // de/dc

H += inv\_sigma \* inv\_sigma \* J \* J.transpose();

b += -inv\_sigma \* inv\_sigma \* error \* J;

cost += error \* error;

}

// 求解线性方程 Hx=b

Vector3d **dx** = **H.ldlt().solve(b)**;

if (isnan(dx[0])) {

cout << "result is nan!" << endl;

break;

}

if (iter > 0 && **cost** >= lastCost) {

cout << "cost: " << cost << ">= last cost: " << lastCost << ", break." << endl;

break;

}

ae += dx[0];

be += dx[1];

ce += dx[2];

lastCost = cost;

cout << "total cost: " << cost << ", \t\tupdate: " << dx.transpose() <<

"\t\testimated params: " << ae << "," << be << "," << ce << endl;

}

chrono::steady\_clock::time\_point t2 = chrono::steady\_clock::now();

chrono::duration<double> time\_used = chrono::duration\_cast<chrono::duration<double>>(t2 - t1);

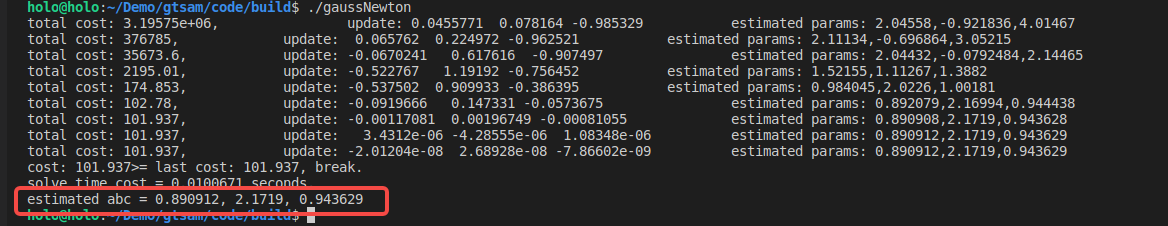
cout << "solve time cost = " << time\_used.count() << " seconds. " << endl;

cout << "estimated abc = " << ae << ", " << be << ", " << ce << endl;

return 0;

}

运行结果



**三、GTSAM简介**

**3.1 简介**

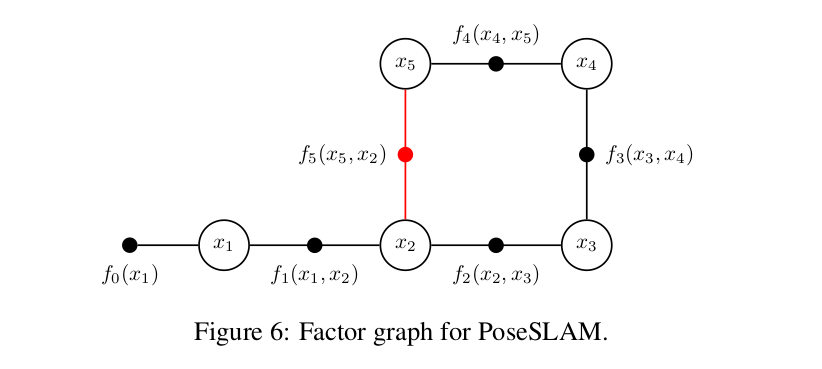
非线性优化库

GTSAM（Georgia Tech Smoothing and Mapping）是基于因子图的C++库，它由佐治亚理工学院的教授和学生们创造，可以解决复杂的非线性状态估计问题。因子图是一种图模型，非常适合于复杂的估计问题的建模，比如SLAM或者是SFM ( Structure from Motion)。

原文链接：https://blog.csdn.net/QLeelq/article/details/111368277

[GTSAM 库学习](https://holomatic.feishu.cn/docs/doccnbt5J7lJUOXnONbR6mbS9YR)

**3.2 经典案例**



**3.2 曲线拟合案例(实践环节)**

自定义因子图：

using namespace gtsam;

class MyFittingFactor: public gtsam::**NoiseModelFactor1**<gtsam::Vector3> {

private:

// measurement information

double mx\_, my\_;

public:

**MyFittingFactor**(gtsam::Key key, const gtsam::Vector2 m,

gtsam::SharedNoiseModel model) :gtsam::NoiseModelFactor1<

gtsam::Vector3>(model, key), mx\_(m.x()), my\_(m.y()) {}

// error function

// @param p the value Vector3

// @param H the optional Jacobian matrix, which use boost optional and has default null pointer

gtsam::Vector **evaluateError**(const gtsam::Vector3& p,

boost::optional<gtsam::Matrix&> H = boost::none) const

{

double expA = exp(p[0]\*mx\_\*mx\_ + p[1]\*mx\_ + p[2]);

if (H)

{

\*H = (gtsam::Matrix13()<< - expA \* mx\_ \* mx\_, - expA \* mx\_, -expA

).finished();

}

// return error vector

return (gtsam::Vector1() << (my\_ - expA)).finished();

}

};

gtsam部分代码：

int main(int argc, char\*\* argv)

{

double a=1.0, b=2.0, c=1.0; // 真实参数值

int N=100; // 数据点

double w\_sigma=0.1; // 噪声Sigma值

cv::RNG rng; // OpenCV随机数产生器

double abc[3] = {2.0 , -1.0 , 5.0}; // abc参数的估计值

vector<double> x\_data, y\_data; // 数据

std::cout<<"generating data: "<<std::endl;

for ( int i=0; i<N; i++ ){

double x =(1.0\* i) / N;

x\_data.push\_back ( x );

y\_data.push\_back (

exp(a\*x\*x + b\*x + c ) + rng.gaussian (w\_sigma\*w\_sigma )// 表达式+噪声

);

std::cout<<x\_data[i]<<" "<<y\_data[i]<<std::endl;

}

// Create a factor graph container

**NonlinearFactorGraph** **graph**;

using gtsam::symbol\_shorthand::X;

auto sigmalModel = gtsam::noiseModel::Isotropic::Sigma(1, 0.1);

for(int i=0; i< x\_data.size();i++)

{

**graph**.**add**(**MyFittingFactor**(X(0), Vector2(x\_data[i],y\_data[i]),

sigmalModel));

}

// print factor graph

graph.print("\nFactor Graph:\n");

**Values** **initials**;

**initials**.insert(X(0), Vector3(abc[0],abc[1], abc[2]));

**initials**.print("\nInitial Values:\n");

// Use Gauss-Newton method optimizes the initial values

**GaussNewtonOptimizer** **optimizer**(**graph**, **initials**);

Values results = optimizer.optimize();

// print final values

results.print("Final Result:\n");

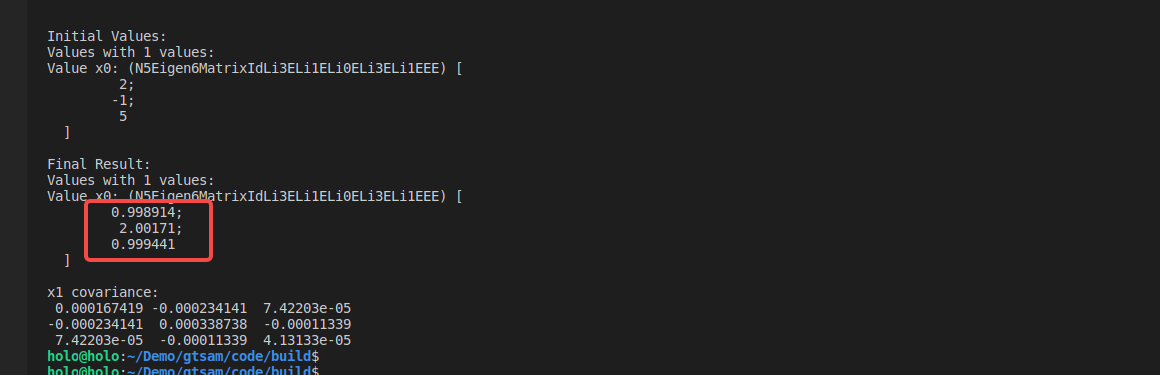
// Calculate marginal covariances for all poses

Marginals marginals(graph, results);

cout << "x1 covariance:\n" << marginals.marginalCovariance(X(0)) << endl;

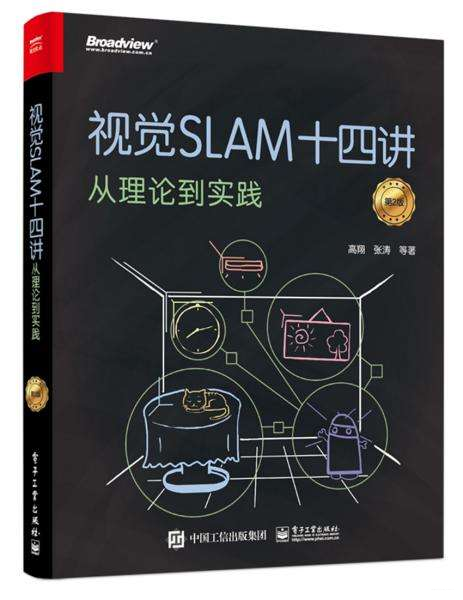
}

运行结果如下：

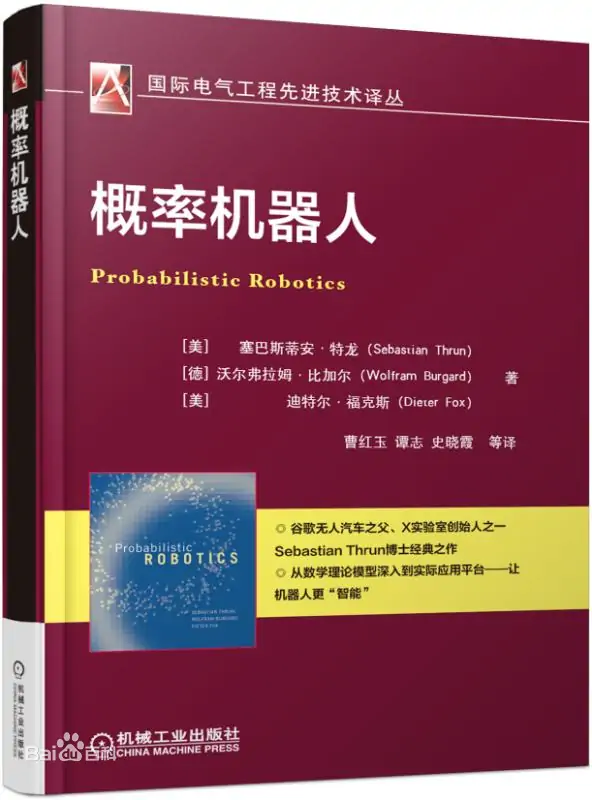


**四、入门书籍介绍**

**《视觉SLAM十四讲》**



**《概率机器人》**



**《机器人学中的状态估计》**



**《ROS机器人开发实践》**

