

激光 SLAM 第 6 次作业

1. 补充代码,实现高斯牛顿方法对 Pose-Graph 进行优化

首先从 main.cpp 中的 main 函数开始, 程序从 VertexPath 和 EdgePath 文件中分别读取顶点和边的信息, 并将其存储在两个容器中:

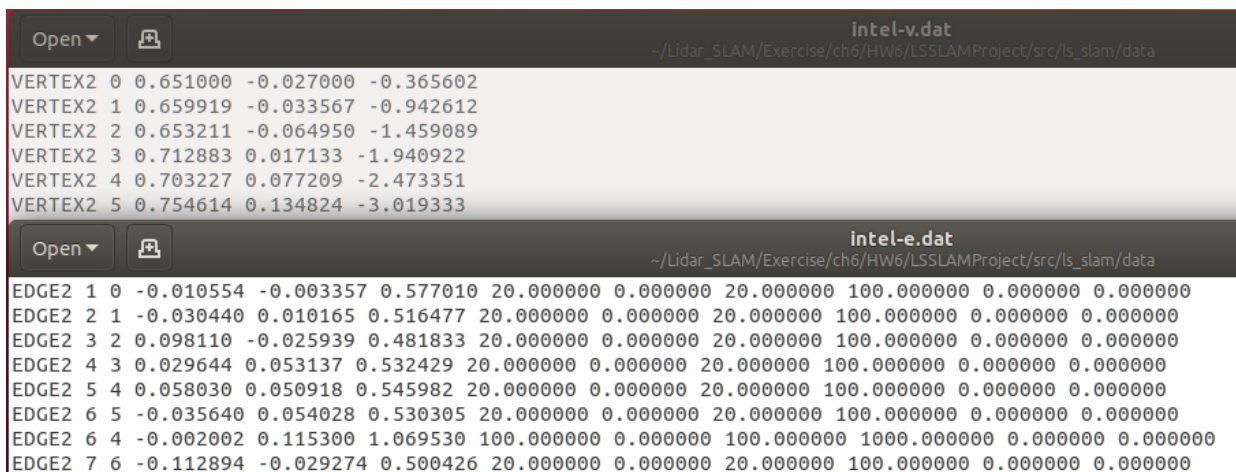
```
std::vector<Eigen::Vector3d> Vertices;
std::vector<Edge> Edges;

ReadVertexInformation(VertexPath, Vertices);
ReadEdgesInformation(EdgePath, Edges);
```

这里的边使用了自定义的数据结构 edge, 可存储该条边连接的顶点的 id, 该条边的测量值(即顶点 x_j 相对于顶点 x_i 的位姿关系), 以及测量值的信息矩阵(可看作是测量协方差矩阵的倒数):

```
7 // 自定义边的数据结构
8 typedef struct edge
9 {
10     int xi, xj;           // 边连接的两个顶点的序号
11     Eigen::Vector3d measurement;
12     Eigen::Matrix3d infoMatrix;
13 }Edge;
```

dat 文件的数据存储形式如下, 比如存储顶点的文件 inter-v.dat 内每一行存储一个顶点的 id, x, y, theta 存储边的文件 inter-e.dat 内每一行存储该条边连接的两个顶点的 id, dx, dy, dtheta, 信息矩阵的六个值:



intel-v.dat
~/Lidar_SLAM/Exercise/ch6/HW6/LSSLAMProject/src/ls_slam/data

```
VERTEX2 0 0.651000 -0.027000 -0.365602
VERTEX2 1 0.659919 -0.033567 -0.942612
VERTEX2 2 0.653211 -0.064950 -1.459089
VERTEX2 3 0.712883 0.017133 -1.940922
VERTEX2 4 0.703227 0.077209 -2.473351
VERTEX2 5 0.754614 0.134824 -3.019333
```

intel-e.dat
~/Lidar_SLAM/Exercise/ch6/HW6/LSSLAMProject/src/ls_slam/data

```
EDGE2 1 0 -0.010554 -0.003357 0.577010 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 2 1 -0.030440 0.010165 0.516477 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 3 2 0.098110 -0.025939 0.481833 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 4 3 0.029644 0.053137 0.532429 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 5 4 0.058030 0.050918 0.545982 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 6 5 -0.035640 0.054028 0.530305 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
EDGE2 6 4 -0.002002 0.115300 1.069530 100.000000 0.000000 100.000000 1000.000000 0.000000 0.000000
EDGE2 7 6 -0.112894 -0.029274 0.500426 20.000000 0.000000 20.000000 100.000000 0.000000 0.000000
```

读取完数据后，使用高斯牛顿法对节点的位姿进行优化。即设置一个 for 循环，通过调用函数 LinearizeAndSolve(Vertexss, Edges)来计算每一步的节点位姿更新量 dx ，将 dx 累加到原来的节点位姿上。一旦满足收敛条件，就结束循环，优化结束。

```

142 double initError = ComputeError(Vertexss,Edges);
143 std::cout <<"initError:"<<initError<<std::endl;
144
145 int maxIteration = 100;
146 double epsilon = 1e-4;
147
148 for(int i = 0; i < maxIteration;i++)
149 {
150     std::cout <<"Iterations:"<<i<<std::endl;
151     Eigen::VectorXd dx = LinearizeAndSolve(Vertexss,Edges);
152
153     //进行更新
154     //TODO--Start
155     for(int j = 0; j != Vertexss.size(); ++j)
156         Vertexss[j] += dx.segment(j*3, 3);
157     //TODO--End
158
159     double maxError = -1;
160     for(int k = 0; k < 3 * Vertexss.size();k++)
161     {
162         if(maxError < std::fabs(dx(k)))
163         {
164             maxError = std::fabs(dx(k));
165         }
166     }
167
168     if(maxError < epsilon)
169         break;
170 }
171
172
173 double finalError = ComputeError(Vertexss,Edges);

```

然后，进入到源文件 gaussian_newton.cpp 中的函数 LinearizeAndSolve(Vertexss, Edges)内。首先我们需要根据位姿节点数量来初始化矩阵 H 和向量 b 。然后为了使得每次优化的位姿更新量都相对于同一个坐标系而言，我们需要加入一个约束让某一个位姿固定。这里我们给 Hessian 矩阵的前三维加上一个单位阵，这样第一个位姿节点的更新量就始终为 0，使得第一个位姿节点得以固定：

```

139 Eigen::VectorXd LinearizeAndSolve(std::vector<Eigen::Vector3d>& Vertexss,
140                                   std::vector<Edge>& Edges)
141 {
142     //申请内存
143     Eigen::MatrixXd H(Vertexss.size() * 3,Vertexss.size() * 3);
144     Eigen::VectorXd b(Vertexss.size() * 3);
145
146     H.setZero();
147     b.setZero();
148
149     //固定第一帧
150     Eigen::Matrix3d I;
151     I.setIdentity();
152     H.block(0,0,3,3) += I;

```

然后我们使用容器 Edges 中的回环边信息和 Vertexs 中计算出的帧间边信息来构建矩阵 H 和向量 b。然后通过解方程 $H dx = -b$ 来计算获得节点位姿更新量 dx。

```

154 //构造H矩阵 & b向量
155 for(int i = 0; i < Edges.size();i++)
156 {
157     //提取信息
158     Edge tmpEdge = Edges[i];
159     Eigen::Vector3d xi = Vertexs[tmpEdge.xi];
160     Eigen::Vector3d xj = Vertexs[tmpEdge.xj];
161     Eigen::Vector3d z = tmpEdge.measurement;
162     Eigen::Matrix3d infoMatrix = tmpEdge.infoMatrix;
163
164     //计算误差和对应的Jacobian
165     Eigen::Vector3d ei;
166     Eigen::Matrix3d Ai;
167     Eigen::Matrix3d Bi;
168     CalcJacobianAndError(xi,xj,z,ei,Ai,Bi);
169
170     //TODO--Start
171     int index_i, index_j;
172     index_i = tmpEdge.xi;
173     index_j = tmpEdge.xj;
174
175     // 这里节点和边是从0开始计数的
176     H.block(index_i*3, index_i*3, 3, 3) += Ai.transpose() * infoMatrix * Ai;
177     H.block(index_i*3, index_j*3, 3, 3) += Ai.transpose() * infoMatrix * Bi;
178     H.block(index_j*3, index_i*3, 3, 3) += Bi.transpose() * infoMatrix * Ai;
179     H.block(index_j*3, index_j*3, 3, 3) += Bi.transpose() * infoMatrix * Bi;
180
181     b.segment(index_i*3, 3) += Ai.transpose() * infoMatrix * ei;
182     b.segment(index_j*3, 3) += Bi.transpose() * infoMatrix * ei;
183
184     //TODO--End
185 }
186
187 //求解
188 Eigen::VectorXd dx;
189
190 //TODO--Start
191 dx = H.ldlt().solve(-b); // 由于对称矩阵H可以保证为半正定矩阵，所以可以用ldlt求解
192 //TODO--End

```

这里为了计算雅可比矩阵 A_i 和 B_i ，以及预测和观测的误差项 e_i ，我们调用函数 `CalcJacobianAndError(xi,xj,z,ei,Ai,Bi)`。

```

85 void CalcJacobianAndError(Eigen::Vector3d xi,Eigen::Vector3d xj,Eigen::Vector3d z,
86                           Eigen::Vector3d& ei,Eigen::Matrix3d& Ai,Eigen::Matrix3d& Bi)
87 {
88     //TODO--Start
89
90     // Estimated Pose
91     Eigen::Matrix2d Ri;
92     Ri << cos(xi(2)), -sin(xi(2)),
93          sin(xi(2)),  cos(xi(2));
94     Eigen::Vector2d tj = Eigen::Vector2d(xj(0), xj(1));
95     Eigen::Vector2d ti = Eigen::Vector2d(xi(0), xi(1));
96     double theta_j = xj(2);
97     double theta_i = xi(2);
98
99     // Measured Pose
100    Eigen::Matrix2d Rij;
101    Rij << cos(z(2)), -sin(z(2)),
102          sin(z(2)),  cos(z(2));
103    Eigen::Vector2d tij = Eigen::Vector2d(z(0), z(1));
104    double theta_ij = z(2);
105
106    // 计算error
107    ei << Rij.transpose() * (Ri.transpose()*(tj - ti) - tij),
108          theta_j - theta_i - theta_ij;
109    // 注意：必须进行角度正则化，保证角度在-pi~pi之间，否则优化容易出问题!!!
110    // 因为有时候当我们朝着梯度下降的方向去优化角度时，由于没有正则化，角度反而会增大，导致优化产生错乱
111    ei(2) = GN_NormalizationAngle(ei(2));
112
113    // 当然也可以使用变换矩阵求解，由于TransToPose函数中使用了atan2,所以角度也会被自动限定在-pi~pi之间，而无需进行角度正则化
114    // Eigen::Matrix3d Ti = PoseToTrans(xi), Tj = PoseToTrans(xj);
115    // Eigen::Matrix3d Tij = PoseToTrans(z);
116    // ei = TransToPose(Tij.inverse() * Ti.inverse() * Tj);
117
118    // Jacobian dei / dxi
119    Eigen::Matrix2d dRi_dtheta;
120    dRi_dtheta << -sin(xi(2)), -cos(xi(2)),
121                 cos(xi(2)), -sin(xi(2));
122    Ai << -Rij.transpose() * Ri.transpose(), Rij.transpose() * dRi_dtheta.transpose() * (tj - ti),
123          Eigen::Vector2d(0,0).transpose(), -1;
124
125    // Jacobian dei / dxj
126    Bi << Rij.transpose() * Ri.transpose(), Eigen::Vector2d(0,0),
127          Eigen::Vector2d(0,0).transpose(), 1;
128
129    //TODO--end
130 }

```

这里使用了课上给出的公式来计算 e_i , A_i , B_i :

- 误差函数的矩阵形式：

$$e_i \longrightarrow e_{ij}(x) = \begin{Bmatrix} R_{ij}^T(R_i^T(t_j - t_i) - t_{ij}) \\ \theta_j - \theta_i - \theta_{ij} \end{Bmatrix}$$

- 对应的Jacobian矩阵：

$$A_i \longrightarrow \frac{\partial e_{ij}(x)}{\partial x_i} = \begin{bmatrix} -R_{ij}^T R_i^T & R_{ij}^T \frac{\partial R_i^T}{\partial \theta} (t_j - t_i) \\ 0 & -1 \end{bmatrix}$$

$$B_i \longrightarrow \frac{\partial e_{ij}(x)}{\partial x_j} = \begin{bmatrix} R_{ij}^T R_i^T & 0 \\ 0 & 1 \end{bmatrix}$$

其中值得注意的有两点:

1. 这里需要计算 $\frac{\partial R_l^T}{\partial \theta}$

其中矩阵对标量的导数定义为[1]:

矩阵 \mathbf{Y} 对标量 x 的导数定义如下:

$$\frac{\partial \mathbf{Y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{12}}{\partial x} & \dots & \frac{\partial y_{1n}}{\partial x} \\ \frac{\partial y_{21}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \dots & \frac{\partial y_{2n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \frac{\partial y_{m2}}{\partial x} & \dots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

即:

$$\left(\frac{\partial \mathbf{Y}}{\partial x}\right)_{ij} = \frac{\partial y_{ij}}{\partial x}$$

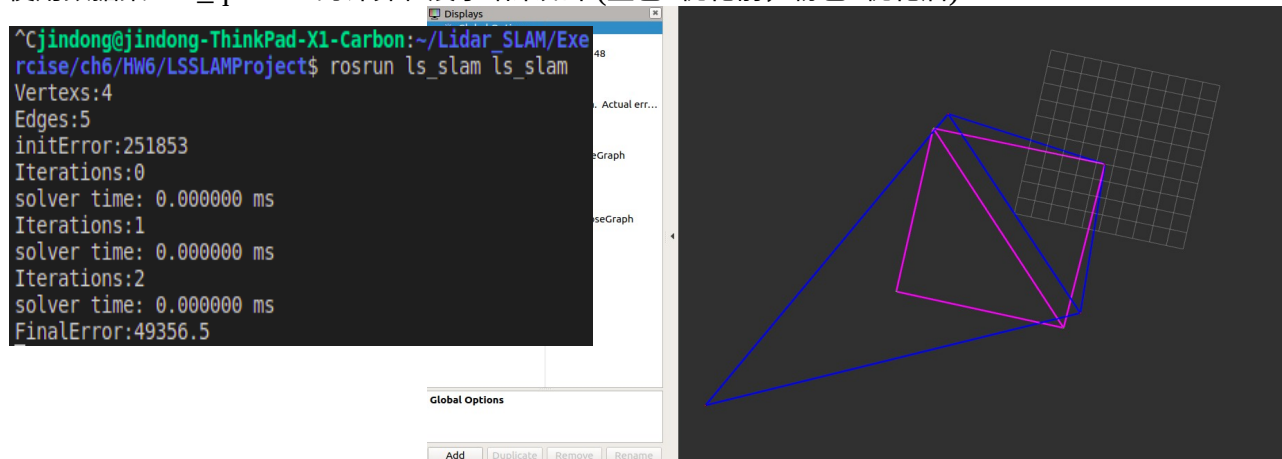
2. 计算预测和观测的误差项 e_i 时, 我们一定要将其中的**角度误差项进行正则化**, 即保证其取值范围为 $-\pi \sim \pi$ 之间, 否则会使优化结果产生错乱, 无法快速收敛。因为有时候由于角度没有正则化, 当我们朝着梯度下降的方向去优化角度时, 角度反而会增大。比如角度误差为-182度时, 当它向梯度下降的方向, 即-180度进行优化时, 此时-180度反而离0度更远了, 优化产生了错乱。角度误差正则化如下:

```
10 const double GN_PI = 3.1415926;
11
12 //进行角度正则化.
13 double GN_NormalizationAngle(double angle)
14 {
15     // 使角度居于-pi~pi之间
16     if(angle > GN_PI)
17         angle -= 2*GN_PI;
18     else if(angle < -GN_PI)
19         angle += 2*GN_PI;
20
21     return angle;
22 }
```

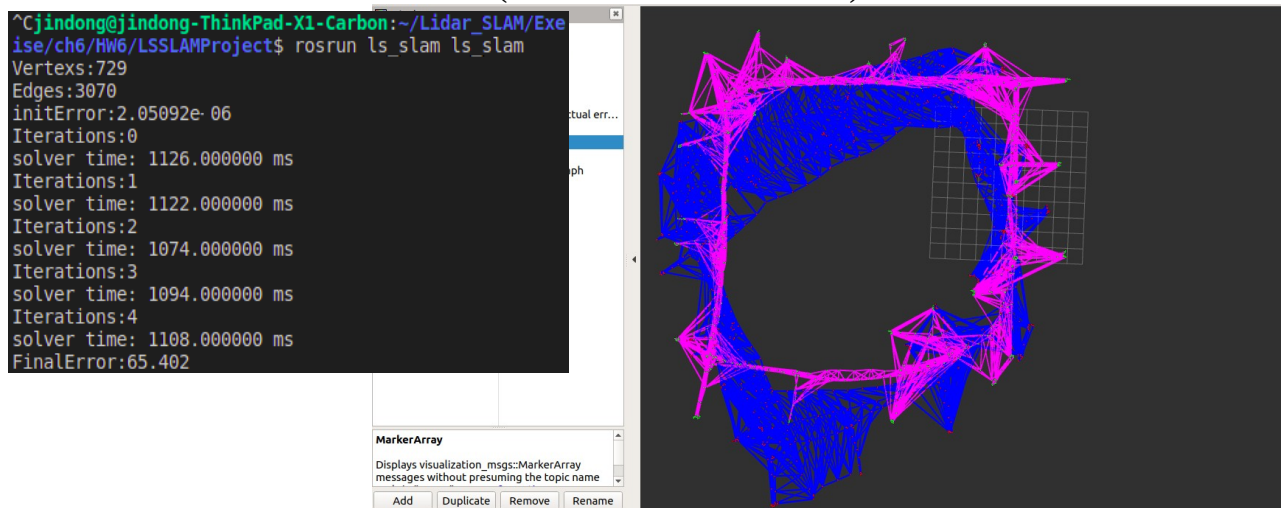
为了展示节点优化结果, 我们使用了消息类型 `visualization_msgs::MarkerArray`, 将图优化前后节点的位姿进行发布:

```
117 // beforeGraph
118 ros::Publisher beforeGraphPub, afterGraphPub;
119 beforeGraphPub = nodeHandle.advertise<visualization_msgs::MarkerArray>("beforePoseGraph", 1, true);
120 afterGraphPub = nodeHandle.advertise<visualization_msgs::MarkerArray>("afterPoseGraph", 1, true);
```

使用数据集 `test_quadrat` 的计算和展示结果如下(蓝色: 优化前, 粉色: 优化后):



使用数据集 intel 的计算和展示结果如下(蓝色: 优化前, 粉色: 优化后):



2. 简答题,开放性答案:你认为第一题的优化过程中哪个环节耗时最多?是否有什么改进的方法可以进行加速?

应该是解方程 $H dx = -b$ 来计算获得节点位姿更新量 dx 这个地方耗时最多。这里使用了 eigen 的 `ldlt` 方法来进行稠密矩阵求解。由于 H 矩阵的稀疏性质,我们可以使用 Eigen 提供的稀疏矩阵求解器[2][3]来进行加速(比如 `ldlt` 对应的稀疏求解法 `SimplicialLDLT`)。同时头文件需要包含 Cholesky 稀疏矩阵求解库 `#include<Eigen/SparseCholesky>` 或者整个稀疏矩阵求解库 `#include<Eigen/Sparse>`。代码如下:

```

189 // 使用稀疏矩阵求解
190 // 构建稀疏矩阵
191 Eigen::SparseMatrix<double> S = H.sparseView();
192 // 使用稀疏矩阵求解器
193 Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> solver;
194 solver.compute(S);
195 if(solver.info()!=Eigen::Success) {
196     std::cerr << "Decomposition H failed";
197 }
198 dx = solver.solve(-b);
199 if(solver.info()!=Eigen::Success) {
200     std::cerr << "Solving failed";
201 }
202
203 return dx;
204 }
  
```

我们使用数据集 intel, 计算用时如下:

```

^Cjindong@jindong-ThinkPad-X1-Carbon:~/Lidar_SLAM/Exe
ise/ch6/HW6/LSSLAMProject$ roslaunch ls_slam ls_slam
Vertices:729
Edges:3070
initError:2.05092e-06
Iterations:0
solver time: 32.000000 ms
Iterations:1
solver time: 34.000000 ms
Iterations:2
solver time: 36.000000 ms
Iterations:3
solver time: 31.000000 ms
Iterations:4
solver time: 31.000000 ms
FinalError:65.402
  
```

我们发现,相比于稠密矩阵 `ldlt` 求解每次迭代约 1000ms, 稀疏矩阵求解 `SimplicialLDLT` 每次迭代只需要大概 30ms, 快了约 30 倍。

3. 学习相关材料。除了高斯牛顿方法,你还知道哪些非线性优化方法?请简述它们的原理

除了高斯牛顿法以外, SLAM 中用的比较多的还有 Levenberg Margquart (LM)法, Dogleg 法等。

a) LM 法。VIO 课程中对 LM 法进行了介绍:

The Levenberg-Marquardt Method

Levenberg (1944) 和 Marquardt (1963) 先后对高斯牛顿法进行了改进, 求解过程中引入了阻尼因子:

$$(\mathbf{J}^\top \mathbf{J} + \mu \mathbf{I}) \Delta \mathbf{x}_{lm} = -\mathbf{J}^\top \mathbf{f} \quad \text{with } \mu \geq 0$$

阻尼因子的作用

- $\mu > 0$ 保证 $(\mathbf{J}^\top \mathbf{J} + \mu \mathbf{I})$ 正定, 迭代朝着下降方向进行。
- μ 非常大, 则 $\Delta \mathbf{x}_{lm} = -\frac{1}{\mu} \mathbf{J}^\top \mathbf{f} = -\frac{1}{\mu} F'(\mathbf{x})^\top$, 接近最速下降法。
- μ 比较小, 则 $\Delta \mathbf{x}_{lm} \approx \Delta \mathbf{x}_{gn}$, 接近高斯牛顿法。

阻尼因子 μ 的更新策略

定性分析, 直观感受阻尼因子的更新:

- ① 如果 $\Delta \mathbf{x} \rightarrow F(\mathbf{x}) \uparrow$, 则 $\mu \uparrow \rightarrow \Delta \mathbf{x} \downarrow$, 增大阻尼减小步长, 拒绝本次迭代。
- ② 如果 $\Delta \mathbf{x} \rightarrow F(\mathbf{x}) \downarrow$, 则 $\mu \downarrow \rightarrow \Delta \mathbf{x} \uparrow$, 减小阻尼增大步长。加快收敛, 减少迭代次数。

定量分析, 阻尼因子更新策略通过比例因子来确定的:

$$\rho = \frac{F(\mathbf{x}) - F(\mathbf{x} + \Delta \mathbf{x}_{lm})}{L(\mathbf{0}) - L(\Delta \mathbf{x}_{lm})} \quad (10)$$

其中:

$$\begin{aligned} L(\mathbf{0}) - L(\Delta \mathbf{x}_{lm}) &= -\Delta \mathbf{x}_{lm}^\top \mathbf{J}^\top \mathbf{f} - \frac{1}{2} \Delta \mathbf{x}_{lm}^\top \mathbf{J}^\top \mathbf{J} \Delta \mathbf{x}_{lm} \\ &\stackrel{\mathbf{b} = -\mathbf{J}^\top \mathbf{f}}{=} -\frac{1}{2} \Delta \mathbf{x}_{lm}^\top (-2\mathbf{b} + (\mathbf{J}^\top \mathbf{J} + \mu \mathbf{I} - \mu \mathbf{I}) \Delta \mathbf{x}_{lm}) \\ &= \frac{1}{2} \Delta \mathbf{x}_{lm}^\top (\mu \Delta \mathbf{x}_{lm} + \mathbf{b}) \end{aligned} \quad (11)$$

Marquardt 策略

首先比例因子分母始终大于 0, 如果:

- $\rho < 0$, 则 $F(\mathbf{x}) \uparrow$, 应该 $\mu \uparrow \rightarrow \Delta \mathbf{x} \downarrow$, 增大阻尼减小步长。
- 如果 $\rho > 0$ 且比较大, 减小 μ , 让 LM 接近 Gauss-Newton 使得系统更快收敛。
- 反之, 如果是比较小的正数, 则增大阻尼 μ , 缩小迭代步长。

1963 年 Marquardt 提出了一个如下的阻尼策略:

$$\begin{aligned} &\text{if } \rho < 0.25 \\ &\quad \mu := \mu * 2 \\ &\text{elseif } \rho > 0.75 \\ &\quad \mu := \mu / 3 \end{aligned} \quad (12)$$

b) 信赖域(Dogleg)法

该方法使用二次型模型来近似任意非线性函数:

Typically a quadratic model function is chosen

$$J(\underline{\theta}^{(k)} + \underline{p}) = J(\underline{\theta}^{(k)}) + \underbrace{\left[\frac{dJ(\underline{\theta})}{d\underline{\theta}} \right]_{\underline{\theta}^{(k)}}^T \underline{p} + \frac{1}{2} \underline{p}^T \underline{B}^{(k)} \underline{p}}_{m^{(k)}(\underline{p})}$$

当最优解位于信赖域内时, 二次型模型近似该函数较为准确。根据最优解位于信赖域的位置, 可以确定不同的梯度下降方向, 来进行迭代求解。

2.4 Trust Region Methods

2.4.1 Main Idea

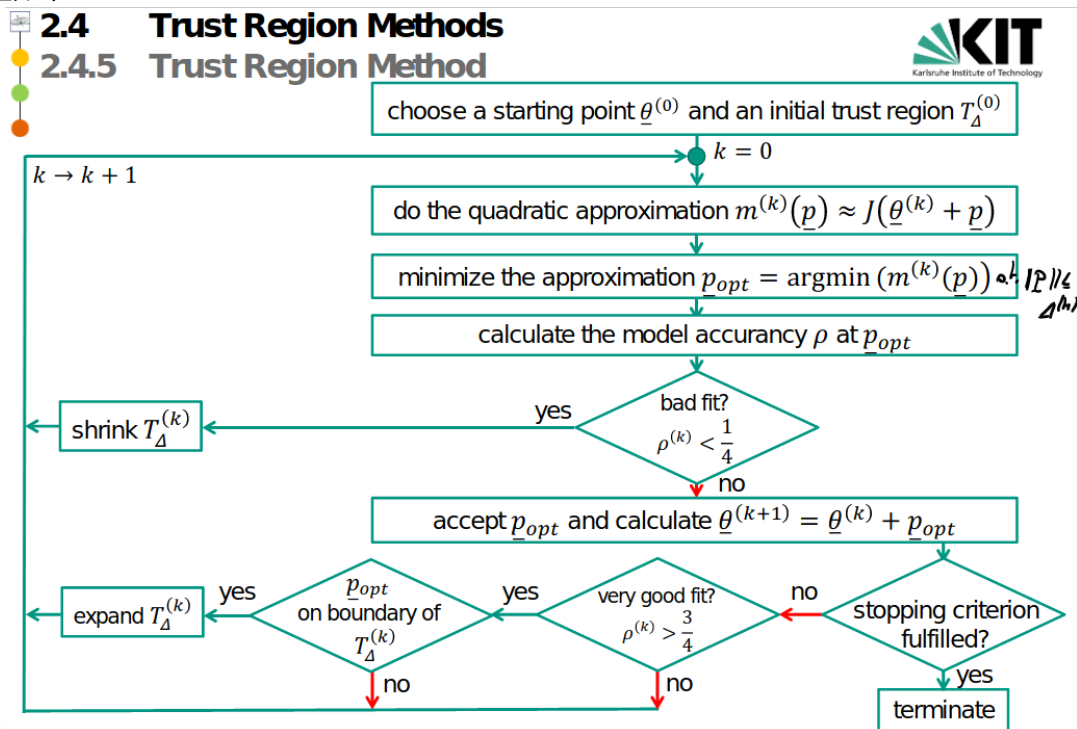


- Trust region (TR) methods calculate furthermore a **region of trust** T_Δ around the current iterate. If the next iterate $\underline{\theta}^{(k+1)} = \underline{\theta}^{(k)} + \underline{p}$ stays inside this trust region T_Δ the model is a good approximation of the original objective function.

J. Nocedal: Numerical Optimization, 2. Edition, p. 68

- Probably, the approximation does not hold outside the trust region
- The quality of the approximated model is calculated using $\rho^{(k)} = \frac{J(\underline{\theta}^{(k)}) - J(\underline{\theta}^{(k)} + \underline{p})}{m^{(k)}(\underline{0}) - m^{(k)}(\underline{p})}$
 - $\rho^{(k)} < 1$ means that the model function falls faster than the original function
 - $\rho^{(k)} = 1$ denotes a perfect fit
 - $\rho^{(k)} > 1$ means that the original function falls faster than the model function

算法流程如下:



Dogleg 法更复杂, 但基本思想与 LM 法类似, 都是考虑了模型近似程度(比例因子 ρ)对收敛效果的影响。

4. 将第一题改为使用任意一种非线性优化库进行优化(比如 Ceres, Gtsam 或 g2o 等)

可以输入 1, 2 或 3, 来使用 Eigen 稀疏求解器, Ceres 求解, g2o 求解

```
167 int solver_select = 0;
168 std::cout << "Which solver would you like to use? enter [1: Eigen, 2: Ceres, 3: g2o]" << std::endl;
169 std::cin >> solver_select;
```

可以比较三种方法的用时情况, 这里统一使用 intel 数据集:

使用 Eigen 的稀疏求解器(使用 SimplicialLDLT 方法):

```
^Cjindong@jindong-ThinkPad-X1-Carbon:~/Lidar_SLAM/Exercise/ch6/HW6/LSSLAMProject
$ rosrn ls_slam ls_slam
Vertices:729
Edges:3070
initError:2.05092e+06
Which solver would you like to use? enter [1: Eigen, 2: Ceres, 3: g2o]
1
Iterations:0
solver time for one step: 46.000000 ms
Iterations:1
solver time for one step: 51.000000 ms
Iterations:2
solver time for one step: 77.000000 ms
Iterations:3
solver time for one step: 60.000000 ms
Iterations:4
solver time for one step: 56.000000 ms
solver used all time: 292 ms
FinalError:65.402
```

使用 Ceres (使用稀疏求解方法 Sparse_Normal_Cholesky 以及 Dogleg 方法):

```
^Cjindong@jindong-ThinkPad-X1-Carbon:~/Lidar_SLAM/Exercise/ch6/HW6/LSSLAMProject
$ rosrn ls_slam ls_slam
Vertices:729
Edges:3070
initError:2.05092e+06
Which solver would you like to use? enter [1: Eigen, 2: Ceres, 3: g2o]
2

Solver Summary (v 2.0.0-eigen-(3.3.4)-lapack-suitesparse-(5.1.2)-cxspase-(3.1.9)
)-eigensparse-no_openmp)

               Original              Reduced
Parameter blocks      2187             2184
Parameters            2187             2184
Residual blocks       3070             3070
Residuals             9210             9210

Minimizer              TRUST_REGION

Sparse linear algebra library  SUITE_SPARSE
Trust region strategy         DOGLEG (TRADITIONAL)

               Given              Used
Linear solver    SPARSE_NORMAL_CHOLESKY  SPARSE_NORMAL_CHOLESKY
Threads          1                   1
Linear solver ordering    AUTOMATIC             2184

Cost:
Initial          1.025461e+06
Final            3.270102e+01
Change           1.025429e+06

Minimizer iterations          5
Successful steps              5
Unsuccessful steps            0

Time (in seconds):
Preprocessor                  0.009076

  Residual only evaluation    0.003638 (4)
  Jacobian & residual evaluation 0.014557 (5)
  Linear solver               0.026165 (4)
Minimizer                    0.052745

Postprocessor                  0.000205
Total                         0.062026

Termination:                  CONVERGENCE (Gradient tolerance reached. Gradient max norm: 2.573441e-04 <= 1.000000e-03)

solver used all time: 75 ms
FinalError:65.402
```

使用 g2o (使用稀疏求解方法 LinearSolverCSparse 以及 GaussNewton):

```
^Cjindong@jindong-ThinkPad-X1-Carbon:~/Lidar_SLAM/Exercise/ch6/HW6/LSSLAMProject
rosrun ls_slam ls_slam
Vertexs:729
Edges:3070
initError:2.05092e+06
Which solver would you like to use? enter [1: Eigen, 2: Ceres, 3: g2o]
3
iteration= 0      chi2= 365235.618940      time= 0.00998346      cumTime= 0.0099
8346      edges= 3070      schur= 0
iteration= 1      chi2= 134.648697      time= 0.00715837      cumTime= 0.0171
418      edges= 3070      schur= 0
iteration= 2      chi2= 65.402453      time= 0.00845398      cumTime= 0.0255
958      edges= 3070      schur= 0
iteration= 3      chi2= 65.402048      time= 0.00709108      cumTime= 0.0326
869      edges= 3070      schur= 0
solver used all time: 76 ms
FinalError:65.402
```

三种求解器看来, Ceres 和 g2o 较快, 两者用时差不多, 可能选择不同的求解方法, 比如 Dogleg, LM, GaussNewton 等方法两者会表现出细微的差别。

参考文献:

[1] 矩阵求导

<https://fei-wang.github.io/matrix.html>

[2] Eigen Tutorial- Solving Sparse Linear Systems

http://eigen.tuxfamily.org/dox/group_TopicSparseSystems.html

[3] Eigen 学习与使用笔记 3 - 稀疏矩阵求解

<http://zhaoxuhui.top/blog/2019/08/28/eigen-note-3.html>