从零开始手写VIO 第三课作业

边城量子 2019.7.06

1. 样例代码给出了使用LM算法来估计曲线 $y = exp(ax^2 + bx + c)$ 参数 a, b, c 的完整过程

- ① 请绘制样例代码中LM阻尼因子μ随着迭代变化的曲线图
- ② 将曲线函数改成 $y = ax^2 + bx + c$, 请修改样例代码中残差计算,雅可比计算等函数, 完成曲线参数估计.
- ✎ 如果有实现其他阻尼因子更新策略可加分(选做).

回答:

- **目标1: 绘制μ的随迭代变化的曲线图**
    - 1. 修改`problem.cc`代码, 对`currentLambda_`的值进行打点并追加输出到`points.txt`文件, 修改后的`bool Problem::Solve()`函数如下:

```
bool Problem::Solve(int iterations) {
    if (edges_.size() == 0 || verticies_.size() == 0) {
        std::cerr << "\nCannot solve problem without edges or
verticies" << std::endl;
        return false;
    }

    TicToc t_solve;
    // 统计优化变量的维数，为构建 H 矩阵做准备
    SetOrdering();
    // 遍历edge, 构建 H = J^T * J 矩阵
    MakeHessian();
    // LM 初始化
    ComputeLambdaInitLM();
    // LM 算法迭代求解
    bool stop = false;
    int iter = 0;

    // -----  新增代码 Start -------
    // 删除旧的数据文件
    std::string fpath = "points.txt";
    remove( fpath.c_str() );
    // -----  新增代码 End -------

    while (!stop && (iter < iterations)) {
        std::cout << "iter: " << iter << " , chi= " << currentChi_ << "
, Lambda= " << currentLambda_
                  << std::endl;
        bool oneStepSuccess = false;
        int false_cnt = 0;
        while (!oneStepSuccess)  // 不断尝试 Lambda, 直到成功迭代一步
        {
            // setLambda
```

```
            AddLambdatoHessianLM();

            // -----  新增代码 Start -------
            // 对最新的lambda进行记录, 记录到points.txt文件中
            ofstream fin( fpath , ios::app);
            fin << currentLambda_ << endl;
            cout << "saved currentLambda_: " << currentLambda_ << endl;
            // -----  新增代码 End -------

            // 第四步, 解线性方程 H X = B
            SolveLinearSystem();
            //
            RemoveLambdaHessianLM();

            // 优化退出条件1:  delta_x_ 很小则退出
            if (delta_x_.squaredNorm() <= 1e-6 || false_cnt > 10) {
                stop = true;
                break;
            }

            // 更新状态量 X = X+ delta_x
            UpdateStates();
            // 判断当前步是否可行以及 LM 的 lambda 怎么更新
            oneStepSuccess = IsGoodStepInLM();
            // 后续处理,
            if (oneStepSuccess) {
                // 在新线性化点 构建 hessian
                MakeHessian();
                false_cnt = 0;
            } else {
                false_cnt++;
                RollbackStates();   // 误差没下降, 回滚
            }
        }
        iter++;

        // 优化退出条件3:  currentChi_ 跟第一次的chi2相比, 下降了 1e6 倍则退出
        if (sqrt(currentChi_) <= stopThresholdLM_)
            stop = true;
    }
    std::cout << "problem solve cost: " << t_solve.toc() << " ms" <<
std::endl;
    std::cout << "   makeHessian cost: " << t_hessian_cost_ << " ms" <<
std::endl;
    return true;
}
```

- 2. 编译运行CurveFitting工程
  - 2.1 建立`build`目录, 运行可执行程序, 生成`points.txt`

```
mkdir  build
cd  build
cmake ..
make
cd  app
./testCurveFitting
```

- 3. 输出如下, 可以看到优化后的参数和真实值差别很小, 在1e-2级别:

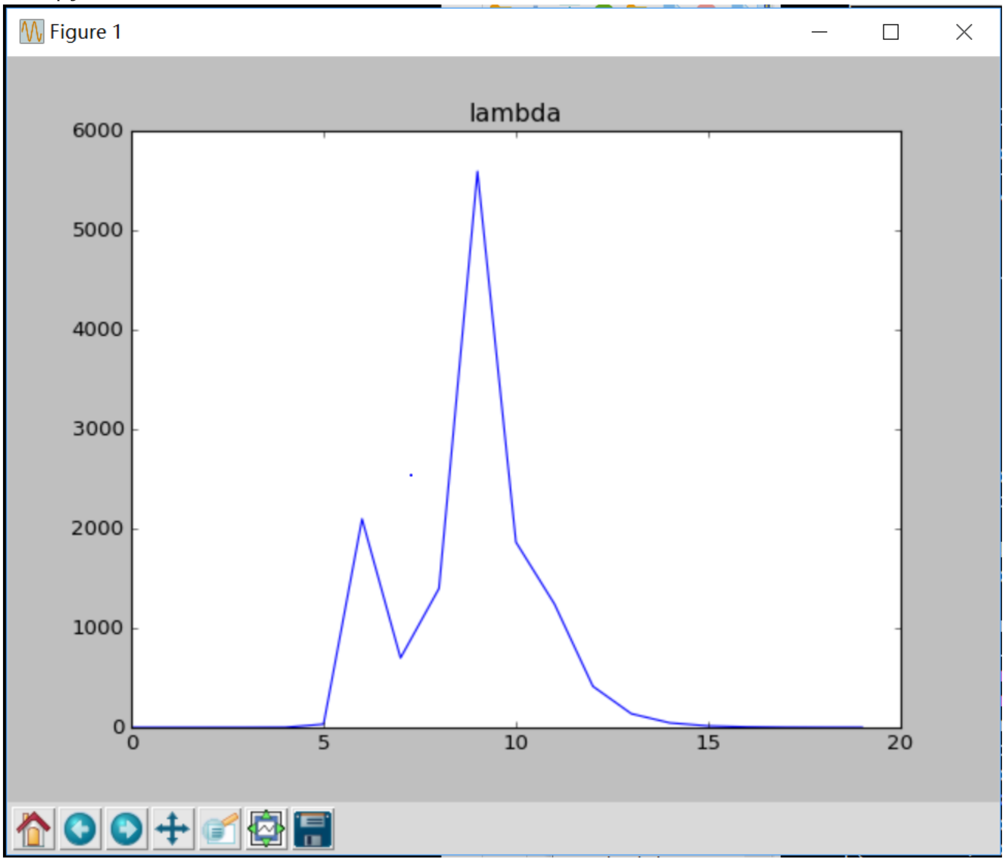
```
    Test CurveFitting start...
iter: 0 , chi= 36048.3 , Lambda= 0.001
    saved currentLambda_: 0.001
    saved currentLambda_: 0.002
    saved currentLambda_: 0.008
    saved currentLambda_: 0.064
    saved currentLambda_: 1.024
    saved currentLambda_: 32.768
saved currentLambda_: 2097.15
    iter: 1 , chi= 30015.5 , Lambda= 699.051
    saved currentLambda_: 699.051
    saved currentLambda_: 1398.1
    saved currentLambda_: 5592.41
iter: 2 , chi= 13421.2 , Lambda= 1864.14
    saved currentLambda_: 1864.14
iter: 3 , chi= 7273.96 , Lambda= 1242.76
    saved currentLambda_: 1242.76
iter: 4 , chi= 269.255 , Lambda= 414.252
    saved currentLambda_: 414.252
iter: 5 , chi= 105.473 , Lambda= 138.084
    saved currentLambda_: 138.084
iter: 6 , chi= 100.845 , Lambda= 46.028
    saved currentLambda_: 46.028
iter: 7 , chi= 95.9439 , Lambda= 15.3427
    saved currentLambda_: 15.3427
iter: 8 , chi= 92.3017 , Lambda= 5.11423
    saved currentLambda_: 5.11423
iter: 9 , chi= 91.442 , Lambda= 1.70474
    saved currentLambda_: 1.70474
iter: 10 , chi= 91.3963 , Lambda= 0.568247
    saved currentLambda_: 0.568247
iter: 11 , chi= 91.3959 , Lambda= 0.378832
  saved currentLambda_: 0.378832
problem solve cost: 2.33059 ms
     makeHessian cost: 0.794637 ms
  -------After optimization, we got these parameters :
    0.941939  2.09453 0.965586
  -------ground truth:
  1.0,  2.0,  1.0
```

4. 编写一个python脚本 `draw.py` 用于绘制 `points.txt` 的点的折线图, 内容如下所示:

```
#!/usr/bin/python
# coding: utf-8
```

```python
# 绘图库
import matplotlib.pyplot as plt

# lambda点的保存文件路径
# 文件内容格式: 每一行一个点
filename =
"/home/hadoop/Documents/CurveFitting_LM/build/app/points.txt"

# 点列表
points = []
# 读取points
with open(filename, 'r') as f:
    lines = f.readlines()
    for line in lines:
        points.append(line)

# 绘图
plt.plot(points)
plt.title('lambda')
plt.show()
```

5. 执行python脚本, lambda随迭代的变化图形如下:



- **目标2: 将曲线函数改成 $y = ax^2 + bx + c$, 请修改样例代码中残差计算,雅可比计算等函数, 完成曲线参数估计**
    - 1. 修改 `CurveFitting.cpp` 代码的 `ComputeReisdual()` 函数:

```
    // 计算曲线模型误差
    virtual void ComputeResidual() override
    {
        Vec3 abc = verticies_[0]->Parameters();  // 估计的参数
         // 构建残差
        //residual_(0) = std::exp( abc(0)*x_*x_ + abc(1)*x_ + abc(2) )
- y_;
        // 残差函数为:  ( ax^2 + bx + c ) - y_
        residual_(0) = abc(0)*x_*x_  + abc(1)*x_ + abc(2) - y_;
    }
```

- 2. 修改`CurveFitting.cpp`代码的`ComputeJacobians()`函数:

```
    // 计算残差对变量的雅克比
    virtual void ComputeJacobians() override
    {
        Vec3 abc = verticies_[0]->Parameters();
        double exp_y = std::exp( abc(0)*x_*x_ + abc(1)*x_ + abc(2) );

        Eigen::Matrix<double, 1, 3> jaco_abc;  // 误差为1维，状态量 3 个，
所以是 1x3 的雅克比矩阵
        //jaco_abc << x_ * x_ * exp_y, x_ * exp_y , 1 * exp_y;
        // ax^2 + bx + c 分别对 a, b, c求偏导
        jaco_abc << x_*x_,  x_,  1;
        jacobians_[0] = jaco_abc;
    }
```

- 3. 修改`CurveFitting.cpp`代码的`main()`函数如下:

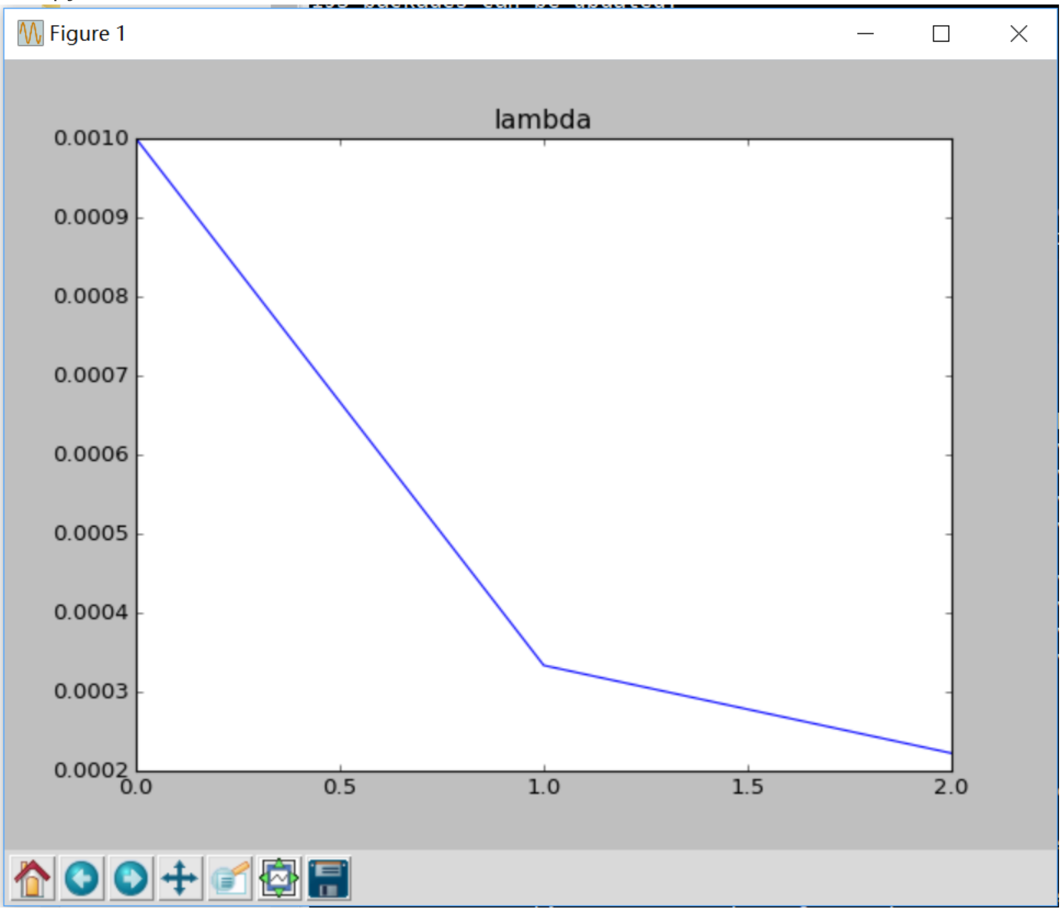
```
    int main()
    {
        double a=1.0, b=2.0, c=1.0;         // 真实参数值
        int N = 100;                        // 数据点
      double w_sigma= 1.;                 // 噪声Sigma值

        std::default_random_engine generator;
      std::normal_distribution<double> noise(0.,w_sigma);

        // 构建 problem
        Problem problem(Problem::ProblemType::GENERIC_PROBLEM);
      shared_ptr< CurveFittingVertex > vertex(new
CurveFittingVertex());

        // 设定待估计参数 a, b, c初始值
        vertex->SetParameters(Eigen::Vector3d (0.,0.,0.));
        // 将待估计的参数加入最小二乘问题
      problem.AddVertex(vertex);

        // 构造 N 次观测
        for (int i = 0; i < N; ++i) {
            double x = i/100.;
            double n = noise(generator);
            // -----  新增代码 Start -------
            // 观测 y
            //double y = std::exp( a*x*x + b*x + c ) + n;
            double y = a*x*x  + b*x  + c  + n*0.1;
```

```
        // -----  新增代码 End -------

          // 每个观测对应的残差函数
          shared_ptr< CurveFittingEdge > edge(new
CurveFittingEdge(x,y));
          std::vector<std::shared_ptr<Vertex>> edge_vertex;
          edge_vertex.push_back(vertex);
        edge->SetVertex(edge_vertex);

          // 把这个残差添加到最小二乘问题
          problem.AddEdge(edge);
    }

      std::cout<<"\nTest CurveFitting start..."<<std::endl;
      /// 使用 LM 求解
    problem.Solve(30);

      std::cout << "-------After optimization, we got these
parameters :" << std::endl;
      std::cout << vertex->Parameters().transpose() << std::endl;
      std::cout << "-------ground truth: " << std::endl;
    std::cout << "1.0,  2.0,  1.0" << std::endl;

      // std
      return 0;
  }
```

- 4. 输出如下, 可以看到优化后的参数和真实值差别很小, 在1.e-2级别:

```
Test CurveFitting start...
iter: 0 , chi= 614.937 , Lambda= 0.001
  saved currentLambda_: 0.001
iter: 1 , chi= 0.913952 , Lambda= 0.000333333
  saved currentLambda_: 0.000333333
iter: 2 , chi= 0.91395 , Lambda= 0.000222222
  saved currentLambda_: 0.000222222
problem solve cost: 0.429004 ms
   makeHessian cost: 0.18227 ms
-------After optimization, we got these parameters :
 1.06107  1.96183 0.999517
-------ground truth:
1.0,  2.0,  1.0
```

- 5. 执行python脚本, lambda随迭代的变化图形如下:



- **目标3: 如果有实现其他阻尼因子更新策略可加分(选做).**
    - 1. 修改`problom.cc`中`Problem::ComputeLambdaInitLM()`函数, 使得μ具备两种初始化方式:
        1. 一种是使用Hession矩阵对角线最大值作为初始值,
        2. 一种则是直接使用固定值为初始值

      修改的局部代码如下:

```
double tau = 1e-5;
// 初值策略1: 使用Hessian矩阵对角线上的最大值作为初始值
currentLambda_ = tau * maxDiagonal;

// 初值策略2: 使用固定值作为初始值
//currentLambda_ = tau*0.1;
```

    - 2. 修改`problom.cc`中`Problem::IsGoodStepInLM()`函数, 实现如下4种μ的更新方式:
        - 策略1: Nielsen 阻尼因子更新策略 , 即课件公式13

```
// rho > 0,  lambda = lambda * max(1/3, 1-(2*rho - 1)^3); nu = 2
// rho <=0,  lambda = lambda * nu, nu = 2*nu;w
```

        - 策略2: Marquardt 阻尼因子更新策略, 课件公式12

```
// rho < 0.25 , lambda = lambda*2.0
// rho > 0.75 , lambda = lambda/3.0
```

- 策略3: Quadratic 阻尼因子更新策略, 参考 "The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems, Henri P. Gavin"

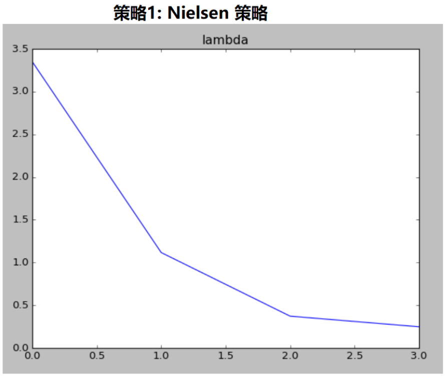
```
// h =  delta_x_*b
// diff = currentChi_ - tempChi;
// alpha = h / (0.5*diff + h )
// rho > 0, lambda = max( lambde/(1+alpha), 1.e-7)
// rho <=0, lambda = lambda + abs(diff*0.5/alpha)
```

- 策略4: 另外一种变形后的Levenberg 阻尼因子更新策略 , 参考 "http://www.duke.edu/~hpgavin/lm.m"

```
// 9.0和11都是经验值, 控制lambda的增长/减少速度
// rho > 0, lambda = max( lambda/9.0, 1.e-7 )
// rho <=0, lambda = min( lambda*11,  1e7 )
```

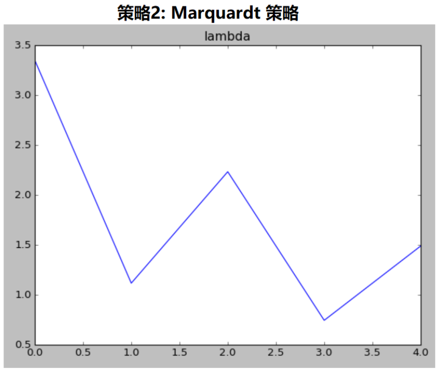
3. 四种策略执行结果与Lambda随迭代变化曲线图比较如下: 程序中其他参数: int N = 700; 且 lambda 使用Hessian矩阵对角线上的最大值作为初始值

策略1: Nielsen 策略



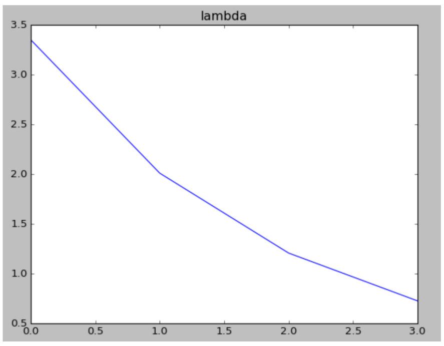

```
-------After optimization, we got these parameters :
1.00191 1.98871 0.98923
-------ground truth:
1.0,  2.0,  1.0
```

策略2: Marquardt 策略



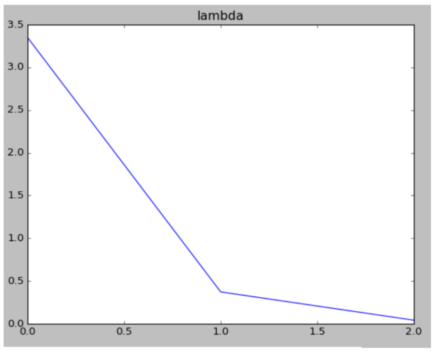

```
-------After optimization, we got these parameters :
 1.00191  1.98869 0.989258
-------ground truth:
1.0,  2.0,  1.0
```

策略3: Qudratic 策略




```
-------After optimization, we got these parameters :
 1.00191  1.98871 0.989234
-------ground truth:
1.0,  2.0,  1.0
```

策略4: Levenberg 策略




```
-------After optimization, we got these parameters :
 1.00191  1.98869 0.989271
-------ground truth:
1.0,  2.0,  1.0
```

4. 四种更新方式的代码如下所示:

```
bool Problem::IsGoodStepInLM() {
    double scale = 0;
    // 对应课件公式11, 得到 scale = L(0) - L(delta_x)
    //  currentLambda_ 即课件中的 μ
    scale = delta_x_.transpose() * (currentLambda_ * delta_x_ + b_);
  scale += 1e-3;    // make sure it's non-zero :)

    // recompute residuals after update state
    // 统计所有的残差, 即公式10的分子部分的  F(x+delta_x)
```

```
    double tempChi = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->Chi2();
  }

    // 计算比例因子, 公式10 的rho
    double rho = (currentChi_ - tempChi) / scale;
    // 阻尼因子更新策略选择器 strategy ,有多种策略课选择
    int strategy = 4;
    switch ( strategy ) {
        case 1:  // Nielsen 阻尼因子更新策略 , 即课件公式13
            // rho > 0,  lambda = lambda * max(1/3, 1-(2*rho - 1)^3);
nu = 2
            // rho <=0,  lambda = lambda * nu, nu = 2*nu;W
            if (rho > 0 && isfinite(tempChi))   // last step was good,
误差在下降
            {
                double alpha = 1. - pow((2 * rho - 1), 3);
                alpha = std::min(alpha, 2. / 3.);
                double scaleFactor = (std::max)(1. / 3., alpha);
                currentLambda_ *= scaleFactor;
                ni_ = 2;
                currentChi_ = tempChi;
                return true;
            } else {
                currentLambda_ *= ni_;
                ni_ *= 2;
                return false;
            }
            break;
        case 2: // Marquardt的阻尼策略, 课件公式12
            // rho < 0.25 , lambda = lambda*2.0
            // rho > 0.75 , lambda = lambda/3.0
            // 备注: 在本例中, 此方法无法优化得到合适的a,b,c的值
            if ( rho < 0.25 && isfinite(tempChi)) {
                currentLambda_ *= 2.0;
                currentChi_ = tempChi;
                return true;
            }else if ( rho > 0.75 && isfinite(tempChi) ) {
                currentLambda_ /= 3.0;
                currentChi_ = tempChi;
                return false;
            } else {
                // do nothing
                return false;
            }
            break;
        case 3:  // Quadratic策略
            //参见论文"The Levenberg-Marquardt algorithm for nonlinear
least squares curve-fitting problems, Henri P. Gavin"
            // h =  delta_x_*b
            // diff = currentChi_ - tempChi;
            // alpha = h / (0.5*diff + h )
            // rho > 0, lambda = max( lambde/(1+alpha), 1.e-7)
            // rho <=0, lambda = lambda + abs(diff*0.5/alpha)
            { // 代码块, 避免编译出现错误提示(本case块初始化变量进入下一个case)
                double  h = delta_x_.transpose() * b_;
```

```
                double  diff = currentChi_ - tempChi;
                double  alpha_ = h / (0.5*diff + h);
                if ( rho > 0 && isfinite(tempChi) ){
                    currentLambda_ =
std::max(currentLambda_/(1+alpha_), 1.e-7 );
                    currentChi_ = tempChi;
                    return true;
                }else if( rho <=0 && isfinite(tempChi) ){
                    currentLambda_ = currentLambda_ +
std::abs(diff*0.5/alpha_);
                    currentChi_ = tempChi;
                    return false;
                }
            }
            break;
        case 4: //Levenberg 策略
          // 参考"http://www.duke.edu/~hpgavin/lm.m"
          // 9.0和11都是经验值, 控制lambda的增长/减少速度
            // rho > 0, lambda = max( lambda/9.0, 1.e-7 )
            // rho <=0, lambda = min( lambda*11,  1e7 )
            if ( rho > 0 && isfinite(tempChi)) {
                currentLambda_ = std::max(currentLambda_/9.0, 1.e-7);
                return true;
            }else{
                currentLambda_ = std::min(currentLambda_*11, 1.e7);
                return false;
            }
            break;
    }
}
```

2. 公式推导,根据课程知识,完成F,G中如下两项的推导过程:

$$
\begin{aligned}
\mathbf{f}_{15} &= \frac{\partial \boldsymbol{\alpha}_{b_i b_{k+1}}}{\partial \delta \mathbf{b}_k^g} = -\frac{1}{4}\left(\mathbf{R}_{b_i b_{k+1}}\left[\left(\mathbf{a}^{b_k} - \mathbf{b}_k^a\right)\right]_\times \delta t^2\right)(-\delta t) \\
\mathbf{g}_{12} &= \frac{\partial \boldsymbol{\alpha}_{b_i b_{k+1}}}{\partial \mathbf{n}_k^g} = -\frac{1}{4}\left(\mathbf{R}_{b_i b_{k+1}}\left[\left(\mathbf{a}^{b_k} - \mathbf{b}_k^a\right)\right]_\times \delta t^2\right)\left(\frac{1}{2}\delta t\right)
\end{aligned}
$$

回答1: f_{15}推导:

其中分子 $\boldsymbol{\alpha}_{b_i b_{k+1}} = \boldsymbol{\alpha}_{b_i b_k} + \boldsymbol{\beta}_{b_i b_k}\delta t + \frac{1}{2}a\delta t^2$,

由于是对k时刻的δb_k^g求导, 因此前两项均与它无关, 所以可略去.

第三项中的a可表示为: $a = \frac{1}{2}(q_{b_i b_k}(a^{b_k} + n_k^a - b_k^a) + q_{b_i b_{k+1}}(a^{b_{k+1}} + n_{k+1}^a - b_k^a))$, 其中前半部分在对$\delta \mathbf{b}_k^g$求导时无关, 可略去; 后一项中的n_{k+1}^g已经被包含在$a^{b_{k+1}}$中, 因此也略去;

因此f_{15}可变为如下形式, 然后加入右扰动$\begin{bmatrix} 1 \\ -\frac{1}{2}\delta \mathbf{b}_k^g \delta t \end{bmatrix}$并转化为旋转矩阵形式, 展开$\exp$, 并利用叉乘性质交换顺序, 最后得到结果:

$$
\begin{aligned}
\boldsymbol{f}_{15} &= \frac{\partial \boldsymbol{\alpha}_{b_i b_{k+1}}}{\partial \delta \boldsymbol{b}_k^g} \\
&= \frac{1}{4} \frac{\partial \mathbf{q}_{b_i b_{k+1}} \otimes \begin{bmatrix} 1 \\ -\frac{1}{2} \delta \boldsymbol{b}_k^g \delta t \end{bmatrix} \left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \delta t^2}{\partial \delta \boldsymbol{b}_k^g} \\
&= \frac{1}{4} \frac{\partial \mathbf{R}_{b_i b_{k+1}} \exp\left(\left[-\delta \mathbf{b}_k^g \delta t \right]_\times \right) \left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \delta t^2}{\partial \delta \boldsymbol{b}_k^g} \\
&\approx \frac{1}{4} \frac{\partial \mathbf{R}_{b i b_{k+1}} \left(\mathbf{I} + \left[-\delta \mathbf{b}_k^g \delta t \right]_\times \right) \left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \delta t^2}{\partial \delta \boldsymbol{b}_k^g} \\
&= \frac{1}{4} \frac{-\partial \mathbf{R}_{b_i b_{k+1}} \left[\left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \right]_\times \delta t^2 \left(-\delta \mathbf{b}_k^g \delta t \right)}{\partial \delta \boldsymbol{b}_k^g} \\
&= -\frac{1}{4} (\mathbf{R}_{b_i b_{k+1}} \left[\left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \right]_\times \delta t^2) (-\delta t)
\end{aligned} \tag{1}
$$

回答2: g_{12}推导:

其中分子 $\boldsymbol{\alpha}_{b_i b_{k+1}} = \boldsymbol{\alpha}_{b_i b_k} + \boldsymbol{\beta}_{b_i b_k} \delta t + \frac{1}{2} a \delta t^2$,

由于是对k时刻的$\boldsymbol{n}_k^g$求导, 因此前两项均与它无关(和k-1时刻的$\boldsymbol{n}_{k-1}^g$有关), 因此可略去.

第三项中的a可表示为 $a = \frac{1}{2}(q_{b_i b_k}(a^{b_k} + n_k^a - b_k^a) + q_{b_i b_{k+1}}(a^{b_{k+1}} + n_{k+1}^a - b_k^a))$, 其中前半部分在对$\boldsymbol{n}_k^g$求导时无关, 可略去; 后一项中的n_{k+1}^g已经被包含在$a^{b_{k+1}}$中, 因此也略去;

因此g_{12}可变为如下形式, 然后加入右扰动$\begin{bmatrix} 1 \\ \frac{1}{4} \mathbf{n}_k^g \delta t \end{bmatrix}$ 并转化为旋转矩阵形式, 展开$\exp$, 并利用叉乘性质交换顺序, 最后得到结果:

$$
\begin{aligned}
\boldsymbol{g}_{12} &= \frac{\partial \boldsymbol{\alpha}_{b_i b_{k+1}}}{\partial \mathbf{n}_k^g} \\
&= \frac{1}{4} \frac{\partial \mathbf{q}_{b_i b_{k+1}} \otimes \begin{bmatrix} 1 \\ \frac{1}{4} \boldsymbol{n}_k^g \delta t \end{bmatrix} (\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a) \delta t^2}{\partial \mathbf{n}_k^g} \\
&= \frac{1}{4} \frac{\partial \mathbf{R}_{b_i b_{k+1}} \exp\left(\left[\frac{1}{2} \boldsymbol{n}_k^g \delta t \right]_\times \right) \left(\mathbf{a}^{b_{k+1}} - b_k^a \right) \delta t^2}{\partial \mathbf{n}_k^g} \\
&\approx \frac{1}{4} \frac{\partial \mathbf{R}_{b_i b_{k+1}} \left(\mathbf{I} + \left[\frac{1}{2} \mathbf{n}_k^g \delta t \right]_\times \right) \left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \delta t^2}{\partial \mathbf{n}_k^g} \\
&= \frac{1}{4} \frac{\partial \mathbf{R}_{b_i b_{k+1}} \left[\frac{1}{2} \mathbf{n}_k^g \delta t \right]_\times \left(\mathbf{a}^{b_{k+1}} - b_k^a \right) \delta t^2}{\partial \mathbf{n}_k^g} \\
&= -\frac{1}{4} \frac{\partial \mathbf{R}_{b_i b_{k+1}} \left[\left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \right]_\times \delta t^2 \left(\frac{1}{2} \mathbf{n}_k^g \delta t \right)}{\partial \mathbf{n}_k^g} \\
&= -\frac{1}{4} \left(\mathbf{R}_{b_i b_{k+1}} \left[\left(\mathbf{a}^{b_{k+1}} - \mathbf{b}_k^a \right) \right]_\times \delta t^2 \right) \left(\frac{1}{2} \delta t \right)
\end{aligned} \tag{2}
$$

3. 证明式(9).

题目:

已知阻尼因子定义:

$$
(\mathbf{J}^\top \mathbf{J} + \mu \mathbf{I}) \Delta \mathbf{x}_{lm} = -\mathbf{J}^\top \mathbf{f} \qquad with \quad \mu \geq 0 \tag{3}
$$

阻尼因子 μ 大小是相对与 $\mathbf{J}^\top\mathbf{J}$ 的元素而言的。 半正定的信息矩阵 $\mathbf{J}^\top\mathbf{J}$ 特征值 λ_j 和对应的特征向量为 $\mathbf{v}_j$ 。对 $\mathbf{J}^\top\mathbf{J}$ 做特征值分解后有： $\mathbf{J}^\top\mathbf{J}=\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$, 可得：

$$\Delta\mathbf{x}_{\mathrm{1m}}=-\sum_{j=1}^{n}\frac{\mathbf{v}_j^\top\mathbf{F}'^\top}{\lambda_j+\mu}\mathbf{v}_j \tag{4}$$

回答:

由 $\mathbf{J}^\top\mathbf{J}=\mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top$, 设 λ_i 为特征值, $\mathbf{V}=(v_1,v_2,\cdots,v_n)$ 其中 v_i 是列向量：

$$\begin{aligned}\Delta\mathbf{x}_{lm}&=(\mathbf{J}^\top\mathbf{J}+\mu\mathbf{I})^{-1}(-\mathbf{J}^\top\mathbf{f})\\&=(\mathbf{V}diag(\lambda_1+\mu\quad\lambda_2+\mu\quad\cdots\quad\lambda_n+\mu)\mathbf{V}^\top)^{-1}(-\mathbf{J}^\top\mathbf{f})\\&=\mathbf{V}^\top diag\left(\frac{1}{\lambda_1+\mu}\quad\frac{1}{\lambda_2+\mu}\quad\cdots\quad\frac{1}{\lambda_n+\mu}\right)\mathbf{V}\left(-\mathbf{J}^\top\mathbf{f}\right)\\&=[v_1\quad v_2\quad\cdots\quad v_n]\begin{bmatrix}\frac{1}{\lambda_1+\mu}&0&\cdots&0\\0&\frac{1}{\lambda_2+\mu}&\cdots&0\\\vdots&\vdots&\ddots&\vdots\\0&\cdots&\cdots&\frac{1}{\lambda_n+\mu}\end{bmatrix}\begin{bmatrix}v_1^\top\\v_2^\top\\\vdots\\v_n^\top\end{bmatrix}\left(-\mathbf{J}^\top\mathbf{f}\right)\\&=-\sum_{j=1}^{n}\frac{v_j^\top\mathbf{F}'^\top}{\lambda_j+\mu}v_j\end{aligned}$$

最后一步推导过程说明：

使用矩阵乘法的结合律，从右向左计算:

- $\mathbf{F}'^\top=\mathbf{J}^\top\mathbf{f}$ 是一个列向量， 而 $v_j^\top$ 是一个行向量， 所以相乘之后的量 $v_j^\top\mathbf{F}'^\top$ 是一个标量，n个标量 $v_j^\top\mathbf{F}'^\top$ 组成的是一个新的列向量 $[v_1^\top\mathbf{F}'^\top\quad v_2^\top\mathbf{F}'^\top\cdots v_n^\top\mathbf{F}'^\top]$;
- 这个列向量再和前面的对角阵相乘后， 结果依然还是一个列向量，列向量的每一项值是 $\frac{v_j^\top\mathbf{F}'^\top}{\lambda_j+\mu}$;
- 这个列向量再和前面的矩阵 $[v_1\quad v_2\quad\cdots v_n]$ 相乘， 乘出来每一项都是 $\frac{v_j^\top\mathbf{F}'^\top}{\lambda_j+\mu}v_j$, 写成求和形式即为：

$\sum_{j=1}^{n}\frac{v_j^\top\mathbf{F}'^\top}{\lambda_j+\mu}v_j$

- 再添上负号即为公式(9)