Jindong_Shi
202x-xx-xx

# 第 12 节课习题

## 1. 使用 LM 算法估计曲线参数

### 1) 绘制阻尼因子$\mu$随着迭代变化的曲线图

在样例代码中我们使用 Pangolin 来绘制 残差平方和 currentChi 和 阻尼因子 currentLambda 随着迭代步骤 iter 变化的曲线图。我们在 backend/problem.cc 中 Problem 类内写入 draw_curve 函数:

```
void Problem::draw_curve(const vector<double, std::allocator<double>> &currentValue_vec, string name){
    // 参考 Pangolin Tutorial: https://github.com/yuntianli91/pangolin_tutorial/blob/master/task5/main.cpp
    // Create OpenGL window in single line
    pangolin::CreateWindowAndBind( window_title: name, w: 640, h: 480);
    // Data logger object
    pangolin::DataLog log;
    // Optionally add named labels
    std::vector<std::string> labels;
    labels.push_back(std::string(name));
    log.SetLabels(labels);
    // OpenGL 'view' of data. We might have many views of the same data.
    auto maxValue :iterator<...> = max_element( first: currentValue_vec.begin(), last: currentValue_vec.end());
    pangolin::Plotter plotter( default_log: &log, left: 0, right: currentValue_vec.size() - 1, bottom: 0, top: *maxValue, tickx: 1, ticky: *maxValue/20);
    plotter.SetBounds( bottom: 0.0, top: 1.0, left: 0.0, right: 1.0);
    // plotter.Track("$i");//坐标轴自动滚动
    pangolin::DisplayBase().AddDisplay( &: plotter);
    // Default hooks for exiting (Esc) and fullscreen (tab).
    unsigned iter = 0;
    while( !pangolin::ShouldQuit() )
    {
        glClear( mask: GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        if(iter < currentValue_vec.size()){
            log.Log( v: currentValue_vec[iter]);
        }
        iter++;

        // Render graph, Swap frames and Process Events
        pangolin::FinishFrame();
    }
}
```

并在 bool Problem::Solve(int iterations)函数内创建变量 currentChi_vec 和 currentLambda_vec 来存储所有迭代步骤的残差 currentChi_和阻尼因子 currentLambda_:

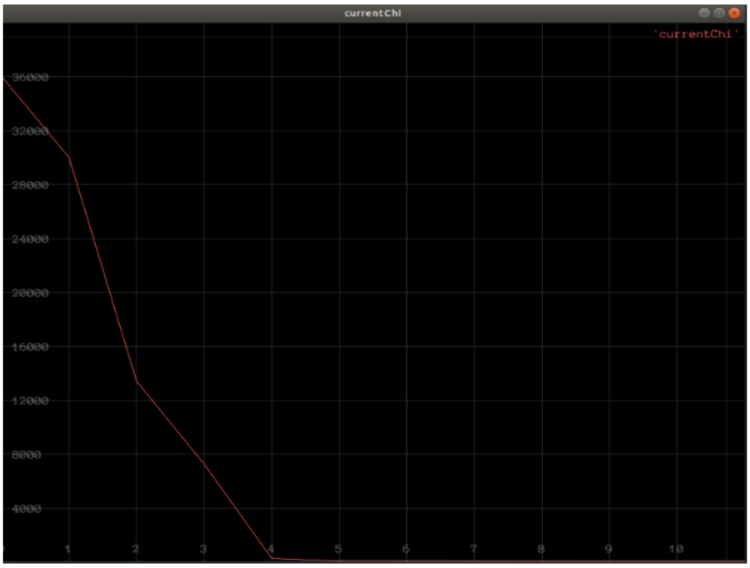
```
    vector<double, std::allocator<double>> currentChi_vec;
    vector<double, std::allocator<double>> currentLambda_vec;

    while (!stop && (iter < iterations)) {
        currentChi_vec.push_back(currentChi_);
        currentLambda_vec.push_back(currentLambda_);
```

并且在 bool Problem::Solve(int iterations)函数末尾调用 draw_curve 函数来绘制残差和阻尼因子随着迭代步骤 iter 变化的曲线图:

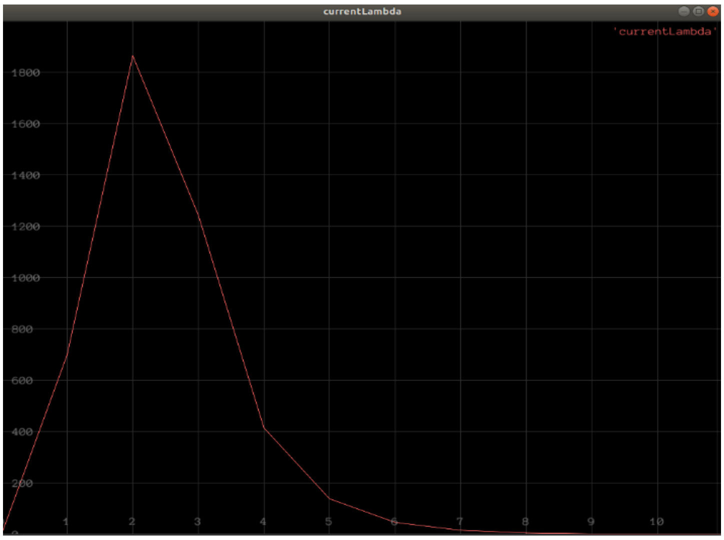
```
    // draw the curve
    draw_curve( currentValue_vec: currentChi_vec,  name: "currentChi");
    draw_curve( currentValue_vec: currentLambda_vec,  name: "currentLambda");
```

曲线绘制结果如下:

**残差平方和变化:**



**阻尼因子变化:**




我们看到在总共的 11 步迭代步骤中，残差一直在下降，这是因为在算法中不满足更新策略的更新都被删除了，导致我们采用的更新都是使得残差缩小的。同时我们看到一开始阻尼因子很小，后来算法逐步增大阻尼因子，来缩小迭代步长，此时 LM 算法逐步趋近于最速下降算法。后来阻尼因子逐渐减小，LM 算法趋近于高斯牛顿。

**2) 将曲线函数改成$y = ax^2 + bx + c$**

我们改变残差函数构建和残差对变量的雅可比，代码更改如下:

```
// 误差模型 模板参数：观测值维度，类型，连接顶点类型
class CurveFittingEdge: public Edge
{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    CurveFittingEdge( double x, double y ): Edge( residual_dimension: 1, num_verticies: 1,  verticies_types: std::vector<std::string>{"abc"}) {
        x_ = x;
        y_ = y;
    }
    // 计算曲线模型误差
    virtual void ComputeResidual() override
    {
        Vec3 abc = verticies_[0]->Parameters();  // 估计的参数
        // residual_(0) = std::exp( abc(0)*x_*x_ + abc(1)*x_ + abc(2) ) - y_;  // 构建残差
        residual_( index: 0) = abc( index: 0)*x_*x_ + abc( index: 1)*x_ + abc( index: 2) - y_;  // 构建残差
    }

    // 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;
        jaco_abc << x_ * x_, x_, 1;

        jacobians_[0] = jaco_abc;
    }
```

参数估计如下:

```
Test CurveFitting start...
iter: 0 , chi= 719.475 , Lambda= 0.001
iter: 1 , chi= 91.395 , Lambda= 0.000333333
problem solve cost: 2.68313 ms
   makeHessian cost: 2.0453 ms
-------After optimization, we got these parameters :
 1.61039  1.61853 0.995178
-------ground truth:
1.0,  2.0,  1.0
```

我们发现只有两步迭代，迭代结束后的残差仍然较大，得到的曲线参数离真实值误差较大。

**3) 实现更优秀的阻尼因子策略，参考文献 [1] 的 4.1.1 节**

新建变量 update_strategy_用来存放用户想要输入的 LM 算法参数更新方法，可输入 1, 2 或 3:

```cpp
bool Problem::Solve(int iterations) {

    cout << "Please enter a LM parameter update strategy (1, 2 or 3):" << endl;
    cin >> update_strategy_;
```

**方法一:**

1. $\lambda_0 = \lambda_o$; $\lambda_o$ is user-specified [5].
   use eq'n (13) for $\mathbf{h}_{\mathsf{lm}}$ and eq'n (16) for $\rho$
   if $\rho_i(\mathbf{h}) > \epsilon_4$: $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{h}$; $\lambda_{i+1} = \max[\lambda_i / L_\downarrow, 10^{-7}]$;
   otherwise: $\lambda_{i+1} = \min[\lambda_i L_\uparrow, 10^7]$;

为了实现该参数更新方法，源代码中要改变的代码如下。

首先是往 Hessian 矩阵中加入阻尼因子 $\lambda$ 和去除 $\lambda$, 根据参数更新方法不同，有不同的做法:

```cpp
void Problem::AddLambdatoHessianLM() {
    ulong size = Hessian_.cols();
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    for (ulong i = 0; i < size; ++i) {
        if(update_strategy_ == 1)
            // Hessian_(i, i) += currentLambda_ * Hessian_(i, i);
            Hessian_( row: i,  col: i) *= (1.+currentLambda_);
        else if(update_strategy_ == 2 || update_strategy_ == 3)
            Hessian_( row: i,  col: i) += currentLambda_;
        else{}
    }
}

void Problem::RemoveLambdaHessianLM() {
    ulong size = Hessian_.cols();
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    // TODO:: 这里不应该减去一个，数值的反复加减容易造成数值精度出问题？而应该保存叠加lambda前的值，在这里直接赋值
    for (ulong i = 0; i < size; ++i) {
        if(update_strategy_ ==1)
            // Hessian_(i, i) -= currentLambda_ * Hessian_(i, i);     // false remove
            Hessian_( row: i,  col: i) /= (1.+currentLambda_);
        else if(update_strategy_ == 2 || update_strategy_ == 3)
            Hessian_( row: i,  col: i) -= currentLambda_;
        else{}
    }
}
```

相对应的，我们在 bool Problem::IsGoodStepInLM()中给出方法 1 的阻尼因子 $\lambda$的更新方法, 其中参数 $L_\uparrow$ 和 $L_\downarrow$ 根据文献可设为 $L_\uparrow = 11$, $L_\downarrow = 9$:

```cpp
bool Problem::IsGoodStepInLM() {
    double scale = 0;
    double rho = 0;

    // recompute residuals after update state
    // 统计所有的残差
    double tempChi = 0.0;
    for (auto edge :pair<...> : edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->Chi2();
    }

    if(update_strategy_ ==1)
    {
        unsigned Hessian_size = Hessian_.rows();
        MatXX diag_Hessian = MatXX::Zero( rows: Hessian_size,  cols: Hessian_size);
        for (ulong i = 0; i < Hessian_size; ++i) {
            diag_Hessian( row: i,  col: i) = Hessian_( row: i,  col: i);
        }
        scale = delta_x_.transpose() * (currentLambda_ * diag_Hessian * delta_x_ + b_);
        scale += 1e-3;    // make sure it's non-zero :)
        rho = (currentChi_ - tempChi) / scale;

        double L_up = 11.0;
        double L_down = 9.0;
        if (rho > 0 && isfinite( x: tempChi))   // last step was good, 误差在下降
        {
            currentLambda_ = (std::max)(currentLambda_/L_down, 1e-7);
            currentChi_ = tempChi;
            return true;
        } else {
            currentLambda_ = (std::min)(currentLambda_ * L_up, 1e7);
            return false;
        }
    }
```

在函数 void Problem::ComputeLambdaInitLM() 中我们规定阻尼因子的初始值，三个方法的策略都是一样的:

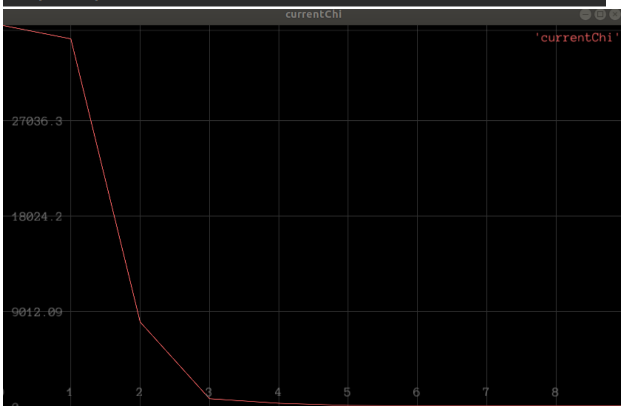
$$\mu_0 = \tau \cdot \max\left\{ \left( \mathbf{J}^\top \mathbf{J} \right)_{ii} \right\}$$

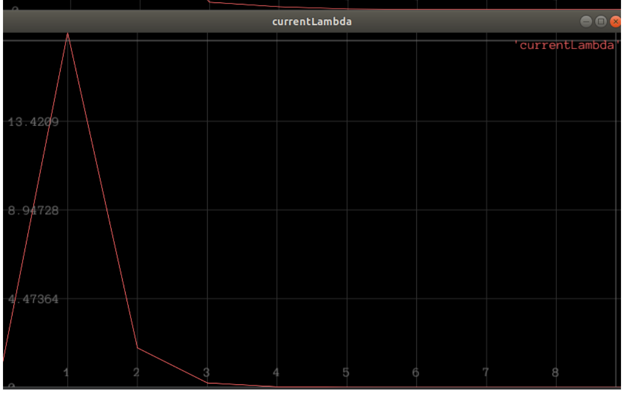
通常，按需设定$\tau \sim [10^{-8}, 1]$，在原代码中 $\tau = 10^{-5}$：

```
/// LM
void Problem::ComputeLambdaInitLM() {
    ni_ = 2.;
    currentLambda_ = -1.;
    currentChi_ = 0.0;
    // TODO:: robust cost chi2
    for (auto edge : edges_) {
        currentChi_ += edge.second->Chi2();
    }
    if (err_prior_.rows() > 0)
        currentChi_ += err_prior_.norm();

    stopThresholdLM_ = 1e-6 * currentChi_;          // 迭代条件为 误差下降 1e-6 倍

    double maxDiagonal = 0;
    ulong size = Hessian_.cols();
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    for (ulong i = 0; i < size; ++i) {
        maxDiagonal = std::max(fabs(Hessian_(i, i)), maxDiagonal);
    }
    double tau = 1e-5;
    currentLambda_ = tau * maxDiagonal;
}
```

方法 1 计算得到的参数计算以及绘制如下:

```
Test CurveFitting start...
Please enter a LM parameter update strategy (1, 2 or 3):
1
iter: 0 , chi= 36048.3 , Lambda= 0.001
iter: 1 , chi= 34760.2 , Lambda= 17.8946
iter: 2 , chi= 8020.58 , Lambda= 1.98828
iter: 3 , chi= 779.997 , Lambda= 0.22092
iter: 4 , chi= 348.805 , Lambda= 0.0245467
iter: 5 , chi= 145.33 , Lambda= 0.00272741
iter: 6 , chi= 101 , Lambda= 0.000303046
iter: 7 , chi= 92.3181 , Lambda= 3.36718e-05
iter: 8 , chi= 91.3999 , Lambda= 3.74131e-06
iter: 9 , chi= 91.3959 , Lambda= 4.15701e-07
problem solve cost: 22.0139 ms
   makeHessian cost: 14.5564 ms
-------After optimization, we got these parameters :
0.941955   2.0945   0.9656
-------ground truth:
1.0,  2.0,  1.0
```



## 方法二:

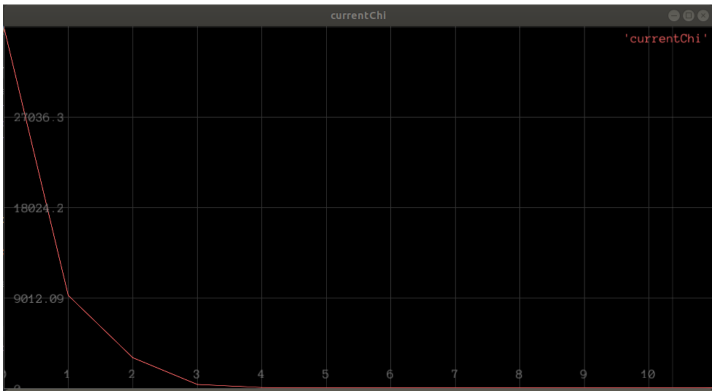
2. $\lambda_0 = \lambda_o \max\left[\text{diag}[\mathbf{J}^\mathsf{T}\mathbf{W}\mathbf{J}]\right]$; $\lambda_o$ is user-specified.
   use eq'n (12) for $\mathbf{h}_{\text{lm}}$ and eq'n (15) for $\rho$
   $\alpha = \left(\left(\mathbf{J}^\mathsf{T}\mathbf{W}(\mathbf{y}-\hat{\mathbf{y}}(\mathbf{p}))\right)^\mathsf{T}\mathbf{h}\right) / \left(\left(\chi^2(\mathbf{p}+\mathbf{h})-\chi^2(\mathbf{p})\right)/2 + 2\left(\mathbf{J}^\mathsf{T}\mathbf{W}(\mathbf{y}-\hat{\mathbf{y}}(\mathbf{p}))\right)^\mathsf{T}\mathbf{h}\right)$;
   if $\rho_i(\alpha\mathbf{h}) > \epsilon_4$: $\mathbf{p} \leftarrow \mathbf{p}+\alpha\mathbf{h}$; $\lambda_{i+1} = \max\left[\lambda_i/(1+\alpha), 10^{-7}\right]$;
   otherwise: $\lambda_{i+1} = \lambda_i + |\chi^2(\mathbf{p}+\alpha\mathbf{h})-\chi^2(\mathbf{p})|/(2\alpha)$;

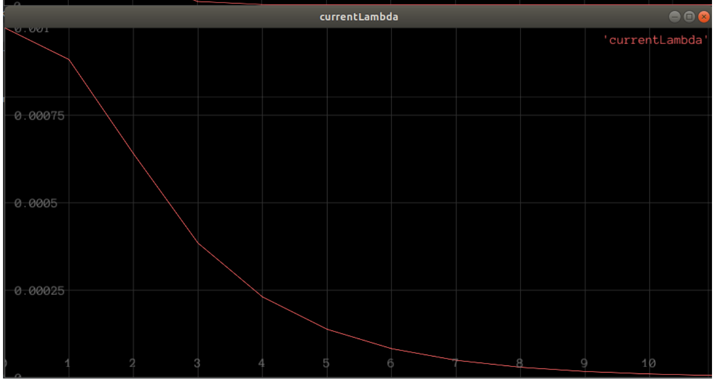
方法二相较方法一更为复杂一些，我们需要先回滚$\Delta x$, 然后再更新$\alpha \cdot \Delta x$ 来计算残差和，然后根据计算出的 rho 和 alpha 来更新的阻尼因子 $\lambda$:

```cpp
else if(update_strategy_ == 2)
{
    double alpha = double(b_.transpose() * delta_x_)/((tempChi - currentChi_)/2.0 + 2.0 * b_.transpose() * delta_x_);
    alpha = std::max(alpha, 1e-1);
    // cout << "alpha_factor = " << alpha_factor << endl;

    // 回滚delta_x_
    RollbackStates();
    // 回滚delta_x_之后，更新状态量 X = X + alpha * delta_x
    delta_x_ *= alpha;
    UpdateStates();

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

    scale = delta_x_.transpose() * (currentLambda_ * delta_x_ + b_);
    scale += 1e-3;    // make sure it's non-zero :)

    rho = (currentChi_ - tempChi) / scale;

    if (rho > 0 && isfinite(tempChi))   // last step was good, 误差在下降
    {
        currentLambda_ = (std::max)(currentLambda_/(1 + alpha), 1e-7);
        currentChi_ = tempChi;
        return true;
    }
    else {
        currentLambda_ += abs(tempChi - currentChi_)/(2 * alpha);
        return false;
    }
}
```

方法 2 计算得到的参数计算以及绘制如下:

```
Test CurveFitting start...
Please enter a LM parameter update strategy (1, 2 or 3):
2
iter: 0 , chi= 36048.3 , Lambda= 0.001
iter: 1 , chi= 9294.65 , Lambda= 0.000909091
iter: 2 , chi= 3097.71 , Lambda= 0.000640061
iter: 3 , chi= 436.783 , Lambda= 0.000384749
iter: 4 , chi= 132.226 , Lambda= 0.000230966
iter: 5 , chi= 96.0453 , Lambda= 0.000138626
iter: 6 , chi= 91.9154 , Lambda= 8.32001e-05
iter: 7 , chi= 91.4535 , Lambda= 4.99341e-05
iter: 8 , chi= 91.4022 , Lambda= 2.99687e-05
iter: 9 , chi= 91.3966 , Lambda= 1.7986e-05
iter: 10 , chi= 91.3959 , Lambda= 1.07944e-05
iter: 11 , chi= 91.3959 , Lambda= 6.47825e-06
problem solve cost: 21.1229 ms
   makeHessian cost: 14.1695 ms
-------After optimization, we got these parameters :
0.942115  2.09435 0.965614
-------ground truth:
1.0,  2.0,  1.0
```




## 方法三:

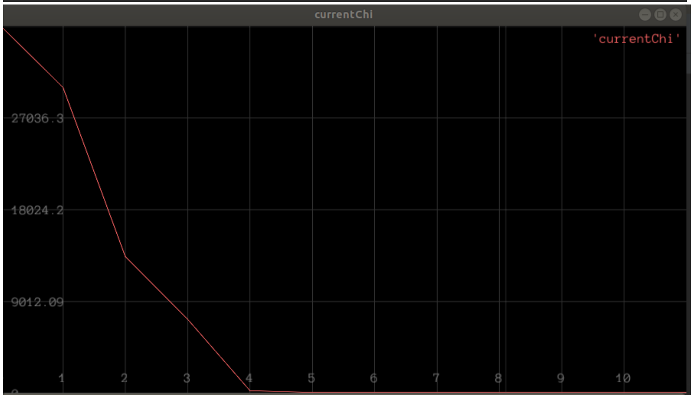
3. $\lambda_0 = \lambda_o \max\left[\text{diag}[\mathbf{J}^\mathsf{T}\mathbf{W}\mathbf{J}]\right]$; $\lambda_o$ is user-specified [6].
   use eq'n (12) for $\mathbf{h}_{\text{lm}}$ and eq'n (15) for $\rho$
   if $\rho_i(\mathbf{h}) > \epsilon_4$: $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{h}$; $\lambda_{i+1} = \lambda_i \max\left[1/3, 1 - (2\rho_i - 1)^3\right]; \nu_i = 2$;
   otherwise: $\lambda_{i+1} = \lambda_i \nu_i$;  $\nu_{i+1} = 2\nu_i$;

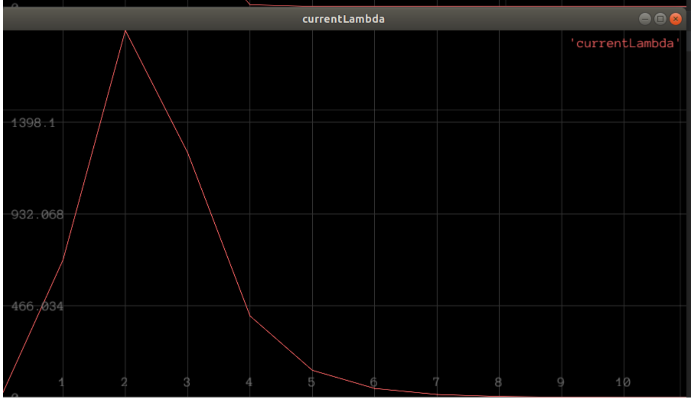
方法 3 是课上介绍的 Nielsen 策略，也是给出的原代码中实现的参数更新方法, 代码如下:

```
else if(update_strategy_ == 3) {
    scale = delta_x_.transpose() * (currentLambda_ * delta_x_ + b_);
    scale += 1e-3;    // make sure it's non-zero :)
    rho = (currentChi_ - tempChi) / scale;

    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;
    }
}
else{
    cout << "Can not find suitable LM update methods!" << endl;
    return false;
}
```

我们同样给出参数计算以及绘制情况:

```
Test CurveFitting start...
Please enter a LM parameter update strategy (1, 2 or 3):
3
iter: 0 , chi= 36048.3 , Lambda= 0.001
iter: 1 , chi= 30015.5 , Lambda= 699.051
iter: 2 , chi= 13421.2 , Lambda= 1864.14
iter: 3 , chi= 7273.96 , Lambda= 1242.76
iter: 4 , chi= 269.255 , Lambda= 414.252
iter: 5 , chi= 105.473 , Lambda= 138.084
iter: 6 , chi= 100.845 , Lambda= 46.028
iter: 7 , chi= 95.9439 , Lambda= 15.3427
iter: 8 , chi= 92.3017 , Lambda= 5.11423
iter: 9 , chi= 91.442 , Lambda= 1.70474
iter: 10 , chi= 91.3963 , Lambda= 0.568247
iter: 11 , chi= 91.3959 , Lambda= 0.378832
problem solve cost: 20.7231 ms
   makeHessian cost: 14.7291 ms
-------After optimization, we got these parameters :
0.941939  2.09453 0.965586
-------ground truth:
1.0,  2.0,  1.0
```




**总结:**

比较以上三种方法，仅从我们有限的实验中，我们发现三种方法估计出的曲线参数相差不大，最后的残差和都比较接近，且迭代步骤都在 10 步左右。三种方法都是可行的 LM 参数更新方法。其中方法二最为复杂，它的特别之处在于根据比例因子来计算缩放因子$\alpha$，改变更新步长为$\alpha \cdot \Delta x$。从绘制的阻尼因子变化曲线来看，方法二的阻尼因子逐渐减小，变化最为平滑，方法三阻尼因子变化最为剧烈，所以从这个角度来看，方法二好于方法一，方法一好于方法三。

当然可能在不同拟合参数的情况下，情况又会有所不同。

**2. 公式推导雅可比 F, G 中的两项**

位置预积分量 $\alpha$ 的递推计算形式：

$$\alpha_{b_ib_{k+1}} = \alpha_{b_ib_k} + \beta_{b_ib_k}\delta t + \frac{1}{2} a \delta t^2$$

$$= \alpha_{b_ib_k} + \beta_{b_ib_k}\delta t + \frac{1}{4}\left( q_{b_ib_k}(a^{b_k} - b_k^a) + q_{b_ib_{k+1}}(a^{b_{k+1}} - b_k^a)\right)\delta t^2$$

位置预积分量对 $k$ 时刻角速度 bias 的 Jacobian：

$$\alpha_{b_ib_{k+1}} = \alpha_{b_ib_k} + \beta_{b_ib_k}\delta t + \frac{1}{4}\left( q_{b_ib_k}(a^{b_k} - b_k^a) + q_{b_ib_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\omega\delta t \end{bmatrix} (a^{b_{k+1}} - b_k^a)\right)\delta t^2$$

$$f_{15} = \frac{\partial \delta\alpha_{b_ib_{k+1}}}{\partial \delta b_k^g} = \frac{1}{4}\frac{\partial q_{b_ib_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}(\omega - \delta b_k^g)\delta t \end{bmatrix} (a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta b_k^g}$$

$$= \frac{1}{4}\frac{\partial R_{b_ib_k}\exp([(\omega - \delta b_k^g)\delta t]_\times)(a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta b_k^g}$$

$$\approx \frac{1}{4}\frac{\partial R_{b_ib_k}(I + [(\omega - \delta b_k^g)\delta t]_\times)(a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta b_k^g}$$

$$= \frac{1}{4}\frac{-\partial R_{b_ib_k}[(a^{b_{k+1}} - b_k^a)]_\times \delta t^2 \cdot [-\delta b_k^g \delta t]}{\partial \delta b_k^g}$$

$$= -\frac{1}{4}\left(R_{b_ib_k}[(a^{b_{k+1}} - b_k^a)]_\times \delta t^2\right)(-\delta t)$$

计算位置预积分量对 $k$ 时刻角速度高斯白噪声的 Jacobian：

$$\omega = \frac{1}{2}\left((\bar{\omega}^{b_k} + n_k^g - b_k^g) + (\bar{\omega}^{b_{k+1}} + n_{k+1}^g - b_k^g)\right) = \underbrace{\frac{1}{2}(\bar{\omega}^{b_k} + \bar{\omega}^{b_{k+1}})}_{=\bar{\omega}} + \frac{1}{2}n_k^g + \frac{1}{2}n_{k+1}^g - b_k^g$$

$$\alpha_{b_ib_{k+1}} = \alpha_{b_ib_k} + \beta_{b_ib_k}\delta t + \frac{1}{4}\left( q_{b_ib_k}(a^{b_k} - b_k^a) + q_{b_ib_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}(\bar{\omega} + \frac{1}{2}n_k^g + \frac{1}{2}n_{k+1}^g - b_k^g) \end{bmatrix} (a^{b_{k+1}} - b_k^a)\right)\delta t^2$$

$$g_{12} = \frac{\partial \delta\alpha_{b_ib_{k+1}}}{\partial \delta n_k^g} = \frac{1}{4}\frac{\partial q_{b_ib_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}(\bar{\omega} + \frac{1}{2}(n_k^g + \delta n_k^g) + \frac{1}{2}n_{k+1}^g - b_k^g)\delta t \end{bmatrix} (a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta n_k^g}$$

$$= \frac{1}{4}\frac{\partial R_{b_ib_k}\exp([\frac{1}{2}\delta n_k^g \delta t]_\times)(a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta n_k^g}$$

$$= \frac{1}{4}\frac{\partial R_{b_ib_k}(I + [\frac{1}{2}\delta n_k^g \delta t]_\times)(a^{b_{k+1}} - b_k^a)\delta t^2}{\partial \delta n_k^g}$$

$$= \frac{1}{4}\frac{\partial R_{b_ib_k}[(a^{b_{k+1}} - b_k^a)]_\times \delta t^2 (\frac{1}{2}\delta n_k^g \delta t)}{\partial \delta n_k^g}$$

$$= \frac{1}{4}R_{b_ib_k}[(a^{b_{k+1}} - b_k^a)]_\times \delta t^2 (\frac{1}{2}\delta t)$$

同理，可计算位置预积分对 $k+1$ 时刻角速度高斯白噪声的 Jacobian：

$$g_{14} = \frac{\partial \delta\alpha_{b_ib_{k+1}}}{\partial \delta n_{k+1}^g} = g_{12}$$

**3. 证明式(9)**

证明式(9)：

已知半正定信息矩阵 $J^TJ$ 特征值 $\{\lambda_j\}$ 和对应特征向量为 $\{V_j\}$

证明 LM 方法中 $(J^TJ+\mu I)\Delta x_{lm}=-J^Tf$ 中更新量 $\Delta x_{lm}=-\sum_{j=1}^{n}\frac{V_j^TF'^T}{\lambda_j+\mu}V_j$

证：

由
$$F(x+\Delta x)\approx L(\Delta x)=\frac{1}{2}l(\Delta x)^Tl(\Delta x)$$
$$=\frac{1}{2}(f(x)+J\Delta x)^T(f(x)+J\Delta x)$$
$$=\frac{1}{2}f^Tf+\Delta x^TJ^Tf+\frac{1}{2}\Delta x^TJ^TJ\Delta x$$
$$=F(x)+(J^Tf)^T\Delta x+\frac{1}{2}\Delta x^TJ^TJ\Delta x$$

可知：
$$F'(x)\approx(J^Tf)^T,\quad F''(x)\approx J^TJ$$

将 $J^TJ=V\Lambda V^T$ 代入正规方程：
$$(V\Lambda V^T+\mu I)\Delta x_{lm}=-J^Tf$$

由于特征相量矩阵为正交矩阵：$VV^T=I$，将其代入，得
$$(V\Lambda V^T+\mu VV^T)\Delta x_{lm}=-J^Tf$$
$$\Longrightarrow\quad V(\Lambda+\mu I)V^T\Delta x_{lm}=-F'^T$$

由于特征相量矩阵一定可逆，且 $V^T=V^{-1}$，可得：
$$\Delta x_{lm}=-V(\Lambda+\mu I)^{-1}V^TF'^T$$
$$\Rightarrow\quad \Delta x_{lm}=-\sum_{j=1}^{n}\frac{V_j^TV_j}{\lambda_j+\mu}F'^T$$

**参考文献:**

**[1] The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems, Henri P. Gavin, Department of Civil and Environmental Engineering, Duke University, 2016**