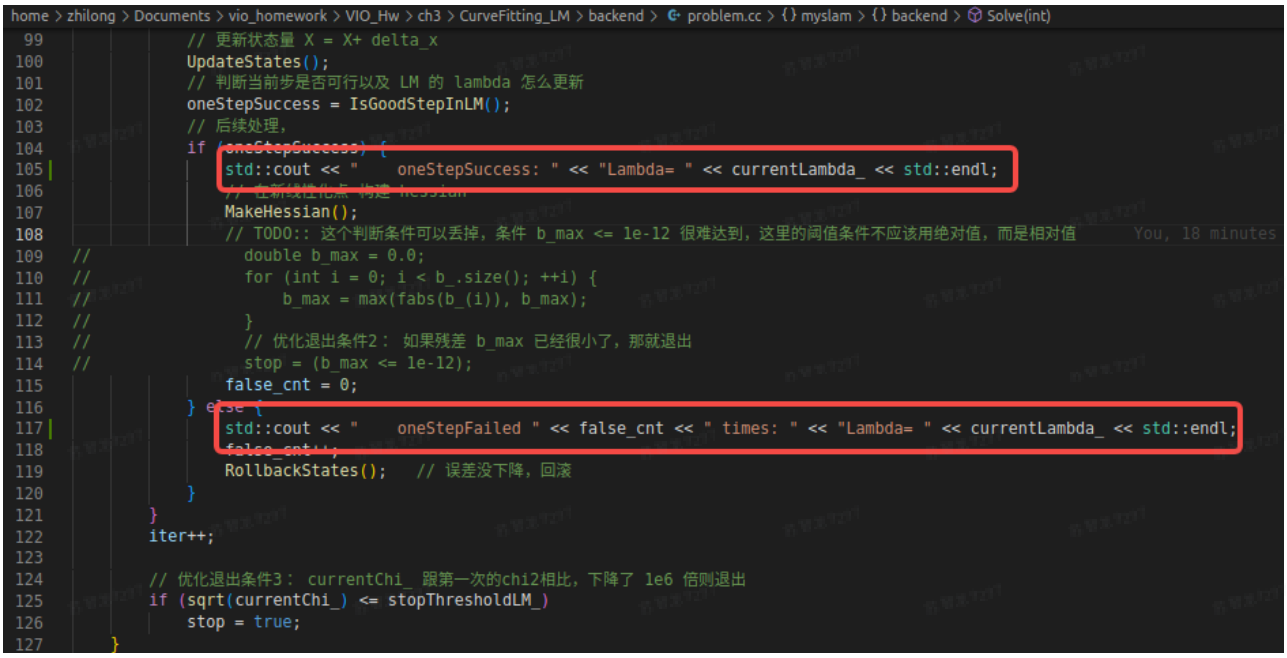
SOUSIC-第 3 章作业

1. LM 算法估计曲线

1.1 绘制样例代码中 LM 阻尼因子 μ 随着迭代变化的曲线图

- 修改 `/backend/problem.cc/Problem::Solve` 中的代码，增加两行输出 `IsGoodStepInLM()` 的结果



```
home > zhilong > Documents > vio_homework > VIO_Hw > ch3 > CurveFitting_LM > backend > problem.cc > {} myslam > {} backend > Solve(int)
            // 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) {
                std::cout << "    oneStepSuccess: " << "Lambda= " << currentLambda_ << std::endl;
                // 在新线性化点 构建 hessian
                MakeHessian();
                // TODO:: 这个判断条件可以丢掉，条件 b_max <= 1e-12 很难达到，这里的阈值条件不应该用绝对值，而是相对值
//                double b_max = 0.0;
//                for (int i = 0; i < b_.size(); ++i) {
//                    b_max = max(fabs(b_(i)), b_max);
//                }
//                // 优化退出条件2： 如果残差 b_max 已经很小了，那就退出
//                stop = (b_max <= 1e-12);
                false_cnt = 0;
            } else {
                std::cout << "    oneStepFailed " << false_cnt << " times: " << "Lambda= " << currentLambda_ << std::endl;
                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;
    }
```

- 运行结果：可以看出，第 0 次迭代失败了 6 次，第 1 次迭代失败了 2 次

```
root@zhilong-ubuntu:/home/zhilong/Documents/vio_homework/VIO_Hw/ch3/CurveFitting_LM/build# ./app/testCurveFitting

Test CurveFitting start...
iter: 0 , chi= 36048.3 , Lambda= 0.001
    oneStepFailed 0 times: Lambda= 0.002
    oneStepFailed 1 times: Lambda= 0.008
    oneStepFailed 2 times: Lambda= 0.064
    oneStepFailed 3 times: Lambda= 1.024
    oneStepFailed 4 times: Lambda= 32.768
    oneStepFailed 5 times: Lambda= 2097.15
    oneStepSuccess: Lambda= 699.051
iter: 1 , chi= 30015.5 , Lambda= 699.051
    oneStepFailed 0 times: Lambda= 1398.1
    oneStepFailed 1 times: Lambda= 5592.41
    oneStepSuccess: Lambda= 1864.14
iter: 2 , chi= 13421.2 , Lambda= 1864.14
    oneStepSuccess: Lambda= 1242.76
iter: 3 , chi= 7273.96 , Lambda= 1242.76
    oneStepSuccess: Lambda= 414.252
iter: 4 , chi= 269.255 , Lambda= 414.252
    oneStepSuccess: Lambda= 138.084
iter: 5 , chi= 105.473 , Lambda= 138.084
    oneStepSuccess: Lambda= 46.028
iter: 6 , chi= 100.845 , Lambda= 46.028
    oneStepSuccess: Lambda= 15.3427
iter: 7 , chi= 95.9439 , Lambda= 15.3427
    oneStepSuccess: Lambda= 5.11423
iter: 8 , chi= 92.3017 , Lambda= 5.11423
    oneStepSuccess: Lambda= 1.70474
iter: 9 , chi= 91.442 , Lambda= 1.70474
    oneStepSuccess: Lambda= 0.568247
iter: 10 , chi= 91.3963 , Lambda= 0.568247
    oneStepSuccess: Lambda= 0.378832
iter: 11 , chi= 91.3959 , Lambda= 0.378832
problem solve cost: 0.425025 ms
makeHessian cost: 0.188902 ms
-------After optimization, we got these parameters :
0.941939  2.09453 0.965586
-------ground truth:
1.0,  2.0,  1.0
```

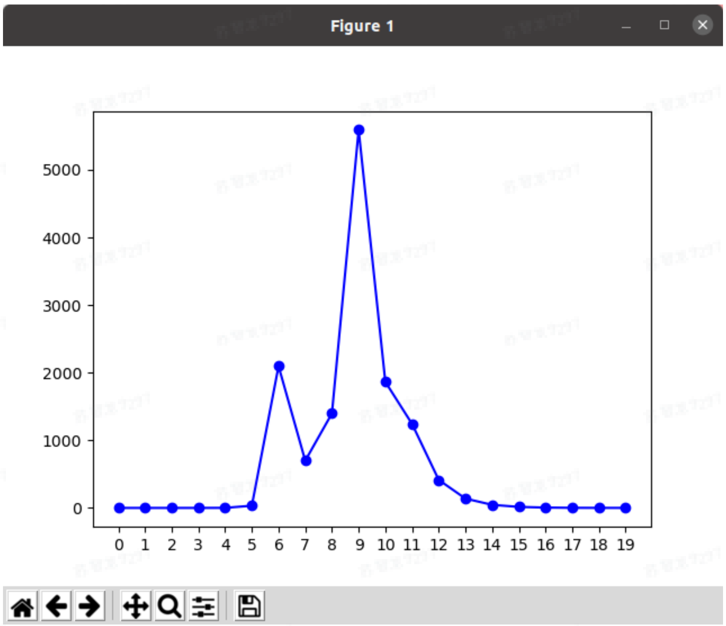
- 画曲线 python 代码

```python
import numpy as np;
import matplotlib.pyplot as plt

def draw_damp(data):
    x_data = [i for i in range(0, len(data))]
    plt.plot(x_data, data, linestyle='-', marker='o', color='blue')
    # plt.xlim(0, len(data) - 1)
    plt.xticks(x_data)
    plt.show()


if __name__ == "__main__":
    data = [0.001, 0.002, 0.008, 0.064, 1.024, 32.768, 2097.15,
        699.051, 1398.1, 5592.41,
        1864.14, 1242.76,
        414.252,
        138.084,
        46.028,
        15.3427,
        5.11423,
        1.70474,
        0.568247,
        0.378832]
    draw_damp(data)
```

- 阻尼因子变化图



1.2 将曲线函数改成 y = ax^2 + bx + c，修改样例代码中残差计算、雅克比计算等函数，完成曲线参数估计

- 修改 `CurveFitting.cpp` 中的 `ComputeResidual()` 和 `ComputeJacobians()`

home > zhilong > Documents > vio_homework > VIO_Hw > ch3 > CurveFitting_LM > app > CurveFitting.cpp > CurveFittingEdge > C

```cpp
    }
    // 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_(0) = abc(0)*x_*x_ + abc(1)*x_ + abc(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;
    }
```

- 修改 `CurveFitting.cpp/main()` 中的观测 y

```
home > zhilong > Documents > vio_homework > VIO_Hw > ch3 > CurveFitting_LM > app > CurveFitting.cpp >
    for (int i = 0; i < N; ++i) {

        double x = i/100.;
        double n = noise(generator);
        // 观测 y
        // double y = std::exp( a*x*x + b*x + c ) + n;
//      double y = std::exp( a*x*x + b*x + c );
        double y = a*x*x + b*x + c + n;

        // 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);
```

- 结果：若采用原始参数，拟合结果较差，可通过以下操作进行改进：
  - 增加数据点数，如 N = 1000（原始N=100）
  - 增大步长以增大数据范围，如 x = i/10（原始 x = i/100）
  - 减小噪声均方差，如 w_sigma = 0.01（原始 w_sigma = 0.1）
- 数据点 N = 100 时，拟合效果一般

```
root@zhilong-ubuntu:/home/zhilong/Documents/vio_homework/VIO_Hw/ch3/CurveFitt

Test CurveFitting start...
iter: 0 , chi= 719.475 , Lambda= 0.001
    oneStepSuccess: Lambda= 0.000333333
iter: 1 , chi= 91.395 , Lambda= 0.000333333
problem solve cost: 0.063685 ms
makeHessian cost: 0.030656 ms
-------After optimization, we got these parameters :
 1.61039  1.61853 0.995178
-------ground truth:
1.0,  2.0,  1.0
```

- 数据点 N = 1000 时，拟合效果较好

```
root@zhilong-ubuntu:/home/zhilong/Documents/vio_homewor

Test CurveFitting start...
iter: 0 , chi= 3.21386e+06 , Lambda= 19.95
    oneStepSuccess: Lambda= 6.65001
iter: 1 , chi= 974.658 , Lambda= 6.65001
    oneStepSuccess: Lambda= 2.21667
iter: 2 , chi= 973.881 , Lambda= 2.21667
    oneStepSuccess: Lambda= 1.47778
iter: 3 , chi= 973.88 , Lambda= 1.47778
problem solve cost: 0.877852 ms
makeHessian cost: 0.728292 ms
-------After optimization, we got these parameters :
0.999588   2.0063 0.968786
-------ground truth:
1.0,  2.0,  1.0
```

1.3 实现其他更优秀的阻尼因子策略，并给出实验对比

> Henri Gavin. “The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems

- 论文中其他阻尼因子，第 3 种就是代码中原来的实现

1. $\lambda_0 = \lambda_o$; λ_o is user-specified [8].
   use eq'n (13) for $\boldsymbol{h}_{\mathsf{lm}}$ and eq'n (16) for ρ
   if $\rho_i(\boldsymbol{h}) > \epsilon_4$: $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{h}$; $\lambda_{i+1} = \max[\lambda_i / L_{\downarrow}, 10^{-7}]$;
   otherwise: $\lambda_{i+1} = \min[\lambda_i L_{\uparrow}, 10^7]$;

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

3. $\lambda_0 = \lambda_o \max\left[\mathsf{diag}[\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J}]\right]$; λ_o is user-specified [9].
   use eq'n (12) for $\boldsymbol{h}_{\mathsf{lm}}$ and eq'n (15) for ρ
   if $\rho_i(\boldsymbol{h}) > \epsilon_4$: $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{h}$; $\lambda_{i+1} = \lambda_i \max[1/3, 1 - (2\rho_i - 1)^3]$; $\nu_i = 2$;
   otherwise: $\lambda_{i+1} = \lambda_i \nu_i$; $\quad \nu_{i+1} = 2\nu_i$;

1. 第一种策略代码修改和结果

home > zhilong > Documents > vio_homework > VIO_Hw > ch3 > CurveFitting_LM > backend > problem.cc > {} m

```cpp
bool Problem::IsGoodStepInLM1() {
    ulong size = Hessian_.cols();
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    MatXX DiagH(MatXX::Zero(size, size));
    for (ulong i = 0; i < size; ++i) {
        DiagH(i, i) = std::fabs(Hessian_(i, i));
    }

    double scale = 0;
    scale = delta_x_.transpose() * (currentLambda_ * DiagH * delta_x_ + b_);
    scale += 1e-3;    // make sure it's non-zero :)

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

    double rho = (currentChi_ - tempChi) / scale;
    if (rho > 0 && isfinite(tempChi))   // last step was good, 误差在下降
    {
        currentLambda_ = std::min(currentLambda_ / 9.0, 1.0e-7);
        currentChi_ = tempChi;
        return true;
    } else {
        currentLambda_ = std::min(currentLambda_ * 11.0, 1.0e-7);
        return false;
    }
}
```

```
root@zhilong-ubuntu:/home/zhilong/Documents/vio_homework/

Test CurveFitting start...
iter: 0 , chi= 3.21386e+06 , Lambda= 19.95
    oneStepSuccess: Lambda= 1e-07
iter: 1 , chi= 974.658 , Lambda= 1e-07
    oneStepSuccess: Lambda= 1.11111e-08
iter: 2 , chi= 973.88 , Lambda= 1.11111e-08
problem solve cost: 0.579454 ms
makeHessian cost: 0.442093 ms
-------After optimization, we got these parameters :
0.999589  2.00628 0.968821
-------ground truth:
1.0,  2.0,  1.0
```

2. 第二种策略代码修改和结果

```cpp
home > zhilong > Documents > vio_homework > VIO_Hw > ch3 > CurveFitting_LM > backend > problem.cc > {} myslam > {} bac
bool Problem::IsGoodStepInLM2() {
    // recompute residuals after update state
    // 统计所有的残差
    double tempChi = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->Chi2();
    }

    // compute alpha
    double alpha = b_.transpose() * delta_x_;
    alpha = alpha / ((tempChi - currentChi_) / 2.0 + 2.0 * b_.transpose() * delta_x_);
    alpha = std::max(alpha, 0.1);

    // reupdate
    RollbackStates();
    delta_x_ *= alpha;
    UpdateStates();

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

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

    double rho = (currentChi_ - tempChi) / scale;
    if (rho > 0 && isfinite(tempChi))   // last step was good, 误差在下降
    {
        currentLambda_ = std::min(currentLambda_ / (1 + alpha), 1.0e-7);
        currentChi_ = tempChi;
        return true;
    } else {
        currentLambda_ += std::abs(currentChi_ - tempChi) / (2 * alpha);
        return false;
    }
}
```

```
root@zhilong-ubuntu:/home/zhilong/Documents/vio_homewo

Test CurveFitting start...
iter: 0 , chi= 3.21386e+06 , Lambda= 19.95
    oneStepSuccess: Lambda= 1e-07
iter: 1 , chi= 357996 , Lambda= 1e-07
    oneStepSuccess: Lambda= 6e-08
iter: 2 , chi= 40643 , Lambda= 6e-08
    oneStepSuccess: Lambda= 3.6e-08
iter: 3 , chi= 5381.56 , Lambda= 3.6e-08
    oneStepSuccess: Lambda= 2.16e-08
iter: 4 , chi= 1463.62 , Lambda= 2.16e-08
    oneStepSuccess: Lambda= 1.296e-08
iter: 5 , chi= 1028.3 , Lambda= 1.296e-08
    oneStepSuccess: Lambda= 7.776e-09
iter: 6 , chi= 979.927 , Lambda= 7.776e-09
    oneStepSuccess: Lambda= 4.6656e-09
iter: 7 , chi= 974.552 , Lambda= 4.6656e-09
    oneStepSuccess: Lambda= 2.79936e-09
iter: 8 , chi= 973.955 , Lambda= 2.79936e-09
problem solve cost: 1.71642 ms
makeHessian cost: 1.23661 ms
-------After optimization, we got these parameters :
0.999437  2.00598 0.968664
-------ground truth:
1.0,  2.0,  1.0
```

- 实验结论：第一种策略耗时 0.57ms，第二种策略耗时 1.71ms，第三种策略耗时 0.87ms。三种精度都差不多，第一种策略耗时最少。

2. 公式推导

- f15 推导

$$\boldsymbol{\omega} = \frac{1}{2}\left(\omega^{b_k} + \omega^{b_{k+1}}\right) - \mathbf{b}_k^g$$

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

- g12 推导

$$
\omega = \frac{1}{2} \left(\left(\bar{\omega}^{b_k} + \mathbf{n}_k^g - \mathbf{b}_k^g\right) + \left(\bar{\omega}^{b_{k+1}} + \mathbf{n}_{k+1}^g - \mathbf{b}_k^g\right)\right)
$$

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

3. 证明 $\Delta \mathbf{x}_{\mathrm{lm}} = -\sum_{j=1}^{n} \frac{\mathbf{v}_j^T \mathbf{F}'^T}{\lambda_j + \mu} \mathbf{v}_j$

$$
\left(J^T J + \mu I\right) \Delta x_{lm} = \left(V \Lambda V^T + \mu I\right) \Delta x_{lm} = \left(V(\Lambda + \mu I) V^T\right) \Delta x_{lm} = -J^T f = -F'^T
$$

$$\Delta x_{lm} = -V(\Lambda + \mu I)^{-1} V^T F'^T = -\begin{bmatrix} v_1 v_2 & \cdots & v_3 \end{bmatrix} \begin{bmatrix} \frac{1}{\lambda_1+\mu} & & \cdots & \\ & \frac{1}{\lambda_2+\mu} & \cdots & \\ & & \ddots & \\ & & & \frac{1}{\lambda_n+\mu} \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} F'^T$$

$$= -\begin{bmatrix} v_1 v_2 & \cdots & v_3 \end{bmatrix} \begin{bmatrix} \frac{v_1^T F'^T}{\lambda_1+\mu} \\ \frac{v_2^T F'^T}{\lambda_2+\mu} \\ \vdots \\ \frac{v_n^T F'^T}{\lambda_n+\mu} \end{bmatrix} = -\left(\frac{v_1^T F'^T}{\lambda_1 + \mu} v_1 + \frac{v_2^T F'^T}{\lambda_2 + \mu} v_2 + \cdots + \frac{v_n^T F'^T}{\lambda_n + \mu} v_n \right) = -\sum_{j=1}^{n} \frac{v_j^T F'^T}{\lambda_j + \mu}$$