1、

（1）绘制阻尼因子随迭代变化的曲线图

首先跑通样例代码，运行结果如下：

```
● ctx@ubuntu:~/VIO_homework/HW3/CurveFitting_LM/build/app$ ./testCurveFitting

Test CurveFitting start...
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: 0.334855 ms
   makeHessian cost: 0.143635 ms
-------After optimization, we got these parameters :
0.941939  2.09453 0.965586
-------ground truth:
1.0,  2.0,  1.0
```

其中，iter 是迭代次数，Lambda 是阻尼因子

chi 是整个系统残差的平方和

因为 edge 类中计算残差的函数里计算的是 J^T*I*J

```cpp
Edge::~Edge() {}

//计算残差的平方和，带协方差矩阵，因为是对曲线参数的估计，不涉及标定，所以设为单位矩阵即可
double Edge::Chi2() {
    // TODO::  we should not Multiply information here, because we have computed Jacobian = sqrt_info * Jacobian
    return residual_.transpose() * information_ * residual_;
//    return residual_.transpose() * residual_;   // 当计算 residual 的时候已经乘以了 sqrt_info, 这里不要再乘
}
```

LM 算法初始化函数

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

LM 算法迭代更新步骤，将实时的残差平方和累加起来，赋值给 currentChi_

```cpp
//根据比例因子rho更新阻尼因子的步骤
bool Problem::IsGoodStepInLM() {
    double scale = 0;
    //对应公式11，计算比例因子的分母部分，L(0)-L(delta_x_lm) L是整个系统代价函数的展开式，由对残差函数的一阶泰勒近似，后推导而得，不同于直接对代价函数的一阶泰勒近似
    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
    double tempChi = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->Chi2();
    }
    //计算比例因子rho, PPT 17页
    //PPT 20页 Nielsen策略
    double 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;
    }
}
```

输出的是 currentChi_成员变量

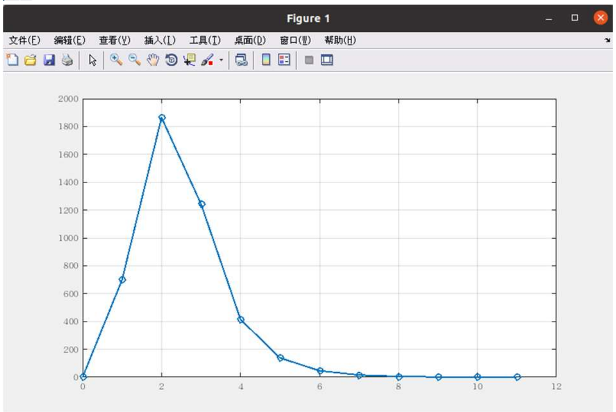
```
while (!stop && (iter < iterations)) {
    std::cout << "iter: " << iter << " , ch = " << currentChi_ << " , Lambda= " << currentLambda_
            << std::endl;
    bool oneStepSuccess = false;
```

在 MATLAB 中画图，横坐标 x 为迭代次数，纵坐标 y 为阻尼因子在各个迭代次数时的值代码及运行结果如下：

编辑器 - /home/ctx/VIO_homework/HW3/curve_plot.m

draw_allan.m | SCRIPT_allan_matparallel.m | SCRIPT_process_results.m | curve_plot.m

```
x=(0:11);%迭代次数
y=[0.001, 699.051, 1864.14, 1242.76, 414.252, 138.084, 46.028, 15.3427, 5.11423, 1.70474, 0.568247, 0.378832];%阻尼因子序列
plot (x,y,'-o','LineWidth',1.5);
grid on;
```



（2）修改曲线函数，完成曲线参数估计

首先修改生成测量值的函数：

```
    // 构造 N 次观测
    for (int i = 0; i < N; ++i) {

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

Main 函数中调用 Problem::Solve 函数，Solve 函数调用 MakeHessian 函数，MakeHessian 函数中调用计算残差和雅可比的函数，对这两个函数进行修改

```
// 遍历每个残差，并计算他们的雅克比，得到最后的 H = J^T * J
for (auto &edge: edges_) {

    edge.second->ComputeResidual();
    edge.second->ComputeJacobians();
```

样例代码估计曲线 $y=e^{ax^2+bx+c}$ 的参数. 测量值 x_m y_m

残差 $e^{ax_m^2+bx_m+c}-y_m$

对a求导 $e^{ax_m^2+bx_m+c}\times x_m^2$

对b求导 $e^{ax_m^2+bx_m+c}\times x_m$

对c求导 $e^{ax_m^2+bx_m+c}$

改成 $y=ax^2+bx+c$

残差 $ax_m^2+bx_m+c-y_m$

对a求导 x_m^2

对b求导 x_m

对c求导 1

修改残差计算:

```
    // 计算曲线模型误差
    //只算一个点?
    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

    }
```

修改雅可比计算:

```
    // 计算残差对变量的雅克比
    //手写笔记中记录雅可比计算的推导过程
    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_ , x_ , 1 ;
        jacobians_[0] = jaco_abc;
    }
```

运行结果:

```
ctx@ubuntu:~/VIO_homework/HW3/CurveFitting_LM/build/app$ ./testCurveFitting

Test CurveFitting start...
iter: 0 , chi= 719.475 , Lambda= 0.001
iter: 1 , chi= 91.395 , Lambda= 0.000333333
problem solve cost: 0.099697 ms
   makeHessian cost: 0.025581 ms
-------After optimization, we got these parameters :
 1.61039  1.61853 0.995178
-------ground truth:
1.0,  2.0,  1.0
```

对 a 和 b 的估计结果不太理想，而且只迭代了两次，所以增加观测点数量，运行结果如下：

```
● ctx@ubuntu:~/VIO_homework/HW3/CurveFitting_LM/build/app$ ./testCurveFitting

  Test CurveFitting start...
  iter: 0 , chi= 3.21386e+06 , Lambda= 19.95
  iter: 1 , chi= 974.658 , Lambda= 6.65001
  iter: 2 , chi= 973.881 , Lambda= 2.21667
  iter: 3 , chi= 973.88 , Lambda= 1.47778
  problem solve cost: 0.616518 ms
     makeHessian cost: 0.456167 ms
  -------After optimization, we got these parameters :
  0.999588   2.0063 0.968786
  -------ground truth:
  1.0,  2.0,  1.0
```

对参数的估计比较准确

（3）其他阻尼因子策略

提供的论文中 4.1.1 节内容：

4.1.1 Initialization and update of the L-M parameter, λ, and the parameters $\mathbf{p}$

In `lm.m` users may select one of three methods for initializing and updating λ and $\boldsymbol{p}$.

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: $\lambda_{i+1} = \min\left[\lambda_i L_\uparrow, 10^{7}\right]$;

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 $\rho_i(\alpha\boldsymbol{h}) > \epsilon_4$: $\boldsymbol{p} \leftarrow p + \alpha\boldsymbol{h}$; $\lambda_{i+1} = \max\left[\lambda_i/(1+\alpha), 10^{-7}\right]$;
   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 $\rho_i(\boldsymbol{h}) > \epsilon_4$: $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{h}$; $\lambda_{i+1} = \lambda_i \max\left[1/3, 1 - (2\rho_i - 1)^3\right]; \nu_i = 2$;
   otherwise: $\lambda_{i+1} = \lambda_i \nu_i$; $\nu_{i+1} = 2\nu_i$;

For the examples in section 4.4, method 1 [8] with $L_\uparrow \approx 11$ and $L_\downarrow \approx 9$ exhibits good convergence properties.

第三种是 PPT20 页中提到的 Nielsen 策略

```cpp
    //计算比例因子rho, PPT 17页
    //PPT 20页 Nielsen策略
    double 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;
    }
```

修改阻尼因子策略，有四个函数需要注意：

一、LM 算法初始化函数 ComputeLambdaInitLM 函数

二、LM 算法迭代更新步骤函数 IsGoodStepInLM 函数

三、将阻尼因子添加到 J^T*J 上的函数：AddLambdatoHessianLM 函数和 RemoveLambdaHessianLM 函数

为了进行不同阻尼因子策略的对比，估计曲线选用 y=exp(ax^2+bx+c)，因为迭代次数比 y=ax^2+bx+c 多一些，数据点 N 设为 100 个，通过第一小问运行结果可以看出 Nielsen 策略中阻尼因子初始化值为 0.001，所以我们这里的实现也初始化为 0.001

第一种阻尼因子策略的实现：

论文中的 h 就是 PPT 中的Δx

1. $\lambda_0 = \lambda_o$; λ_o is user-specified [8].
   use eq'n (13) for $\boldsymbol{h}_{\text{lm}}$ 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: $\lambda_{i+1} = \min[\lambda_i L_\uparrow, 10^7]$;

For the examples in section 4.4, method 1 [8] with $L_\uparrow \approx 11$ and $L_\downarrow \approx 9$ exhibits good convergence properties.

公式 13、16 如下：

In Marquardt's update relationship [8], the damping parameter λ is scaled by the diagonal of the Hessian $\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J}$ for each parameter.

$$\left[\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J} + \lambda \,\mathsf{diag}(\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J})\right] \boldsymbol{h}_{\text{lm}} = \boldsymbol{J}^\mathsf{T}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}) \,, \tag{13}$$

$$\rho_i(\boldsymbol{h}_{\text{lm}}) = \frac{\chi^2(\boldsymbol{p}) - \chi^2(\boldsymbol{p} + \boldsymbol{h}_{\text{lm}})}{|(\boldsymbol{y} - \hat{\boldsymbol{y}})^\mathsf{T}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}) - (\boldsymbol{y} - \hat{\boldsymbol{y}} - \boldsymbol{J}\boldsymbol{h}_{\text{lm}})^\mathsf{T}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}} - \boldsymbol{J}\boldsymbol{h}_{\text{lm}})|} \tag{14}$$

$$= \frac{\chi^2(\boldsymbol{p}) - \chi^2(\boldsymbol{p} + \boldsymbol{h}_{\text{lm}})}{|\boldsymbol{h}_{\text{lm}}^\mathsf{T}(\lambda_i \boldsymbol{h}_{\text{lm}} + \boldsymbol{J}^\mathsf{T}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}(\boldsymbol{p})))|} \quad \text{if using eq'n (12) for } \boldsymbol{h}_{\text{lm}} \tag{15}$$

$$= \frac{\chi^2(\boldsymbol{p}) - \chi^2(\boldsymbol{p} + \boldsymbol{h}_{\text{lm}})}{|\boldsymbol{h}_{\text{lm}}^\mathsf{T}(\lambda_i \mathsf{diag}(\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J})\boldsymbol{h}_{\text{lm}} + \boldsymbol{J}^\mathsf{T}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}(\boldsymbol{p})))|} \quad \text{if using eq'n (13) for } \boldsymbol{h}_{\text{lm}} \tag{16}$$

修改 ComputeLambdaInitLM 函数:

```
//论文中第一种阻尼因子策略实现
void Problem::ComputeLambdaInitLM() {
    // 计算残差部分要保留
    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
    //用户设定初始阻尼因子
    currentLambda_ = 1e-3;
}
```

修改 IsGoodStepInLM 函数:

```cpp
//论文中第一种阻尼因子策略的实现
bool Problem::IsGoodStepInLM() {
    // 统计所有的残差
    double tempChi = 0.0;
    for (auto edge : edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->Chi2();
    }
    // compute rho
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    ulong size = Hessian_.cols();
    //取Hessian矩阵的对角线元素，用于计算论文公式16
    MatXX diag_hessian(MatXX::Zero(size, size));
    for (ulong i = 0; i < size; ++i) {
        diag_hessian(i, i) = Hessian_(i, i);
    }
    //scale是比例因子rho的分母
    double scale = delta_x_.transpose() * (currentLambda_ * diag_hessian * delta_x_ + b_);
    //计算比例因子rho
    double rho = (currentChi_ - tempChi) / scale;
    // update currentLambda_
    double epsilon = 0.0;
```

```cpp
    //论文说L_down和L_up约等于9和11
    double L_down = 9.0;
    double L_up = 11.0;
    if (rho > epsilon && isfinite(tempChi)) {
        currentLambda_ = std::max(currentLambda_ / L_down, 1e-7);
        currentChi_ = tempChi;
        return true;
    }
    else {
        currentLambda_ = std::min(currentLambda_ * L_up, 1e7);
        return false;
    }
}
```

修改 AddLambdatoHessianLM 函数:

```cpp
//论文中第一种阻尼因子策略实现
void Problem::AddLambdatoHessianLM() {
    ulong size = Hessian_.cols();
    assert(Hessian_.rows() == Hessian_.cols() && "Hessian is not square");
    for (ulong i = 0; i < size; ++i) {
        Hessian_(i, i) += currentLambda_ * Hessian_(i, i);
    }
}
```

修改 RemoveLambdaHessianLM 函数:

```cpp
//论文中第一种阻尼因子策略实现
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) {
        //注意只处理对角线上的元素即可
        Hessian_(i, i) /= 1.0 + currentLambda_;
    }
}
```

运行结果:

```
ctx@ubuntu:~/VIO_homework/HW3/CurveFitting_LM/build/app$ ./testCurveFitting

Test CurveFitting start...
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: 0.284379 ms
   makeHessian cost: 0.118116 ms
-------After optimization, we got these parameters :
0.941955   2.0945   0.9656
-------ground truth:
1.0,  2.0,  1.0
```

和 Nielsen 策略对比迭代次数少两次，体现不出啥东西，残差函数平方和都从最初的 36048.3 找到了最小为 91.3959，阻尼因子和 Nielsen 策略相比整体小很多，不过因为这里实现用到的论文公式 13 与 Nielsen 用到的公式 12 已经导致阻尼因子在问题求解中的地位不太一样了，不是直接加到原来 Hessian 矩阵（不是真正的 Hessian 矩阵，实际是 J^T * W * J），而是先乘以对应 Hessian 矩阵中对应位置的对角线元素再加上去。

第二题和第三题在手写笔记的后四页中