大作业

1.采用更优的LM策略

参考The Levenvberg-Marquardt algorithm fot nonlinear squares curve-fitting problems：LM优化算法的差异在与如何计算高斯牛顿步骤以及如何在梯度下降和高斯牛顿之间做选择。高斯牛顿更新步骤有以下两种方式，

$$\left[\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J}+\lambda\boldsymbol{I}\right]\boldsymbol{h}_{\mathsf{lm}}=\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}(\boldsymbol{y}-\hat{\boldsymbol{y}})\ , \tag{12}$$

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

不同的初始化参数和更新步长策略有：

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. $\lambda_0=\lambda_{\mathsf{o}}$; 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. $\lambda_0=\lambda_{\mathsf{o}}\max\left[\mathsf{diag}[\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J}]\right]$; λ_{o} is user-specified.
   use eq'n (12) for $\boldsymbol{h}_{\mathsf{lm}}$ and eq'n (15) for ρ
   $\alpha=\left(\left(\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}(\boldsymbol{y}-\hat{\boldsymbol{y}}(\boldsymbol{p}))\right)^{\mathsf{T}}\boldsymbol{h}\right)/\left((\chi^2(\boldsymbol{p}+\boldsymbol{h})-\chi^2(\boldsymbol{p}))/2+2\left(\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}(\boldsymbol{y}-\hat{\boldsymbol{y}}(\boldsymbol{p}))\right)^{\mathsf{T}}\boldsymbol{h}\right)$;
   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. $\lambda_0=\lambda_{\mathsf{o}}\max\left[\mathsf{diag}[\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J}]\right]$; 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: $\lambda_{i+1}=\lambda_i\nu_i$; $\quad\nu_{i+1}=2\nu_i$;

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

其中增益计算方式如下：

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

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

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

其中策略１的λ初始值为0.01,λ更新策略关键代码如下：

```
// 参考The Levenvberg-Marquardt algorithm fot nonlinear squares curve-fitting problems
// section 4
bool Problem::IsGoodStepInLM2() {
    double scale = 0;
    // section 4 公式(16)
    scale = delta_x_.transpose() * (currentLambda_ * Hessian_.diagonal().asDiagonal() * delta_x_ + b_);
    scale += 1e-6;    // make sure it's non-zero

    // recompute residuals after update state
    double tempChi = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->RobustChi2();
    }
    if (err_prior_.size() > 0)
        tempChi += err_prior_.norm();
    tempChi *= 0.5;          // 1/2 * err^2

    double rho = (currentChi_ - tempChi) / scale; // gain ratio
    if (rho > 0 && isfinite(tempChi))   // last step was good, 误差在下降
    {
        currentLambda_ = std::max(currentLambda_ / 9, 1e-7);
        currentChi_ = tempChi;
        return true;
    } else {
        currentLambda_ = std::min(currentLambda_ * 11, 1e7);
        return false;
    }
}
```

策略２的λ初始值和策略３相同都是海森矩阵对角线的最大值：

```cpp
void Problem::ComputeLambdaInitLM() {
    ni_ = 2.;
    currentLambda_ = -1.;
    currentChi_ = 0.0;

    for (auto edge: edges_) {
        currentChi_ += edge.second->RobustChi2();
    }
    if (err_prior_.rows() > 0)
        currentChi_ += err_prior_.norm();
    currentChi_ *= 0.5;

    stopThresholdLM_ = 1e-10 * currentChi_;          // 迭代条件为 误差下降 1e-6 倍


#ifdef USE_LM_2
    currentLambda_ = 0.01; //初始值
#else
    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); // max{J^TwJ}
    }

    maxDiagonal = std::min(5e10, maxDiagonal);
    double tau = 1e-5;  // 1e-5
    currentLambda_ = tau * maxDiagonal; // 初始值
#endif
    // std::cout << "currentLamba_: "<<maxDiagonal<<" "<<currentLambda_<<std::endl;
}
```

策略 2 的λ更新策略关键代码如下，注意计算出$alpha$后，状态需要回滚，再利用$alpha$状态更新或者回滚,其中更新策略：

```cpp
// 参考The Levenvberg-Marquardt algorithm fot nonlinear squares curve-fitting problems
// section 4
bool Problem::IsGoodStepInLM3() {
    // recompute residuals after update state
    double tempChi = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi += edge.second->RobustChi2();
    }
    if (err_prior_.size() > 0)
        tempChi += err_prior_.norm();
    tempChi *= 0.5;          // 1/2 * err^2
    // rollback for update x + alpha * delta_x
    RollbackStates();

    // section 4 strategy 2
    double beta = b_.transpose() * delta_x_;
    alpha_ = beta / (0.5 * (tempChi - currentChi_) + 2 * beta);

    // update x + alpha * delta_x
    for (auto vertex: verticies_) {
        ulong idx = vertex.second->OrderingId();
        ulong dim = vertex.second->LocalDimension();
        VecX delta = delta_x_.segment(idx, dim);
        // 所有的参数 x 叠加一个增量  x_{k+1} = x_{k} + alpha*delta_x
        vertex.second->Plus(alpha_*delta);
    }
    // recompute residuals after update state
    double tempChi2 = 0.0;
    for (auto edge: edges_) {
        edge.second->ComputeResidual();
        tempChi2 += edge.second->RobustChi2();
    }
    if (err_prior_.size() > 0)
        tempChi2 += err_prior_.norm();
    tempChi2 *= 0.5;          // 1/2 * err^2

    double scale = delta_x_.transpose() * (currentLambda_ * alpha_ * delta_x_ + b_);

    double rho = (currentChi_ - tempChi2) / scale;
    //cout << "rho = " << rho << endl;
    if (rho > 0 && isfinite(tempChi))   // last step was good, 误差在下降
    {
        double alpha = currentLambda_ / (1 + alpha_);
        currentLambda_= std::max(alpha, 1e-7);
        currentChi_ = tempChi2;
        return true;
    } else {
        currentLambda_ += std::abs(tempChi2 - currentChi_) / (2*alpha_);
        return false;
    }
}
```

回滚策略：

```cpp
#ifdef USE_LM_3
                // 回退状态
                for (auto vertex: verticies_) {
                    ulong idx = vertex.second->OrderingId();
                    ulong dim = vertex.second->LocalDimension();
                    VecX delta = delta_x_.segment(idx, dim);

                    // x_{k} = x_{k+1} - alpha*delta_x
                    vertex.second->Plus(-alpha_*delta);
                }
#else
                RollbackStates();   // 误差没下降，回滚
#endif
```

1.1 curve_fitting测试

在第三章curve_fitting中，测试结果如下：三者优化后参数精度基本一致，区别在于策略１和策略３的收敛速度明显快于策略２，其中收敛速度：策略１>策略３>策略２（代码原本的）。

策略１需要４次迭代便收敛，耗时0.40ms：

```
Test CurveFitting start...
iter: 0 , chi= 614.937 , Lambda= 0.01
iter: 1 , chi= 0.958059 , Lambda= 0.00111111
iter: 2 , chi= 0.914282 , Lambda= 0.000123457
iter: 3 , chi= 0.91395 , Lambda= 1.37174e-05
Writing lambda to csv file succeed!
problem solve cost: 0.40621 ms
   makeHessian cost: 0.206919 ms
-------After optimization, we got these parameters :
 1.06137  1.96153 0.999573
-------ground truth:
1.0,  2.0,  1.0
```

策略２需要９次迭代才收敛，耗时2.36ms：

```
Test CurveFitting start...
iter: 0 , chi= 614.937 , Lambda= 0.001
iter: 1 , chi= 69.1405 , Lambda= 0.000599999
iter: 2 , chi= 8.49479 , Lambda= 0.000359999
iter: 3 , chi= 1.75627 , Lambda= 0.000215999
iter: 4 , chi= 1.00754 , Lambda= 0.0001296
iter: 5 , chi= 0.924349 , Lambda= 7.77598e-05
iter: 6 , chi= 0.915106 , Lambda= 4.66559e-05
iter: 7 , chi= 0.914079 , Lambda= 2.79935e-05
iter: 8 , chi= 0.913965 , Lambda= 1.67961e-05
Writing lambda to csv file succeed!
problem solve cost: 2.35551 ms
   makeHessian cost: 1.29979 ms
-------After optimization, we got these parameters :
 1.06091  1.96154 0.999365
-------ground truth:
1.0,  2.0,  1.0
```

策略３仅需３次便收敛，耗时0.465ms：

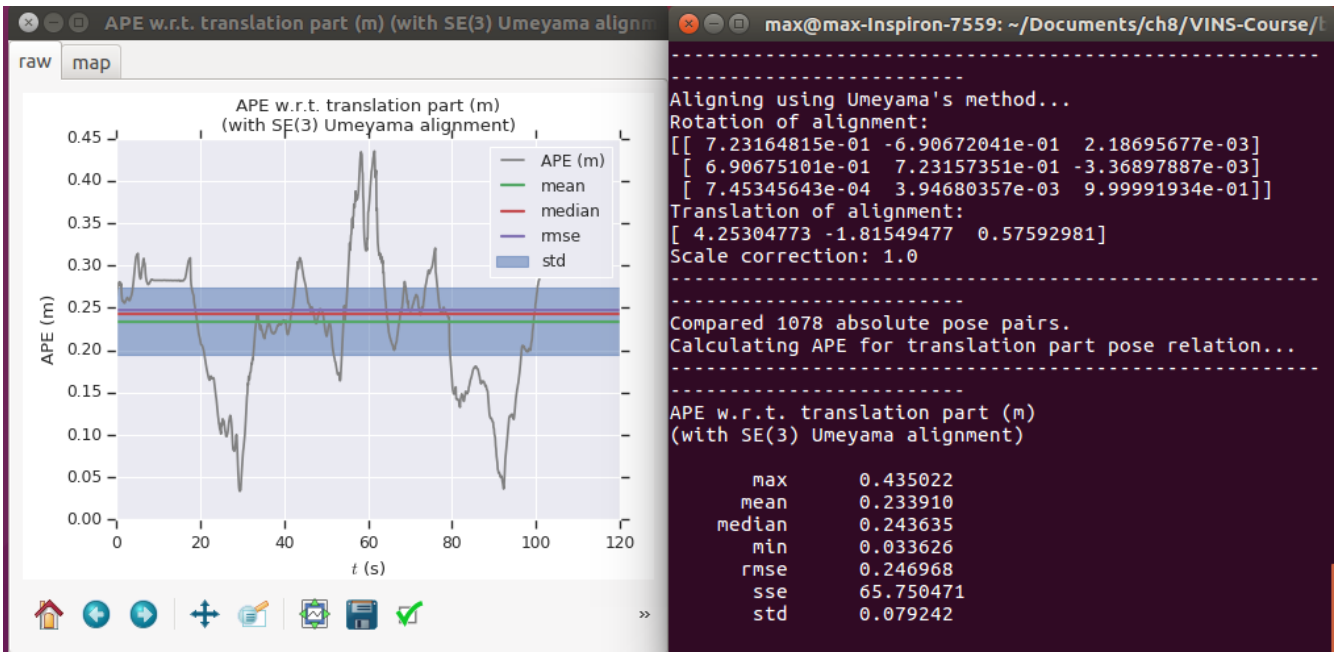
```
Test CurveFitting start...
iter: 0 , chi= 614.937 , Lambda= 0.001
iter: 1 , chi= 0.913952 , Lambda= 0.000333333
iter: 2 , chi= 0.91395 , Lambda= 0.000222222
Writing lambda to csv file succeed!
problem solve cost: 0.464671 ms
   makeHessian cost: 0.266704 ms
-------After optimization, we got these parameters :
 1.06107  1.96183 0.999517
-------ground truth:
1.0,  2.0,  1.0
```

1.2 MH_05数据集场景测试

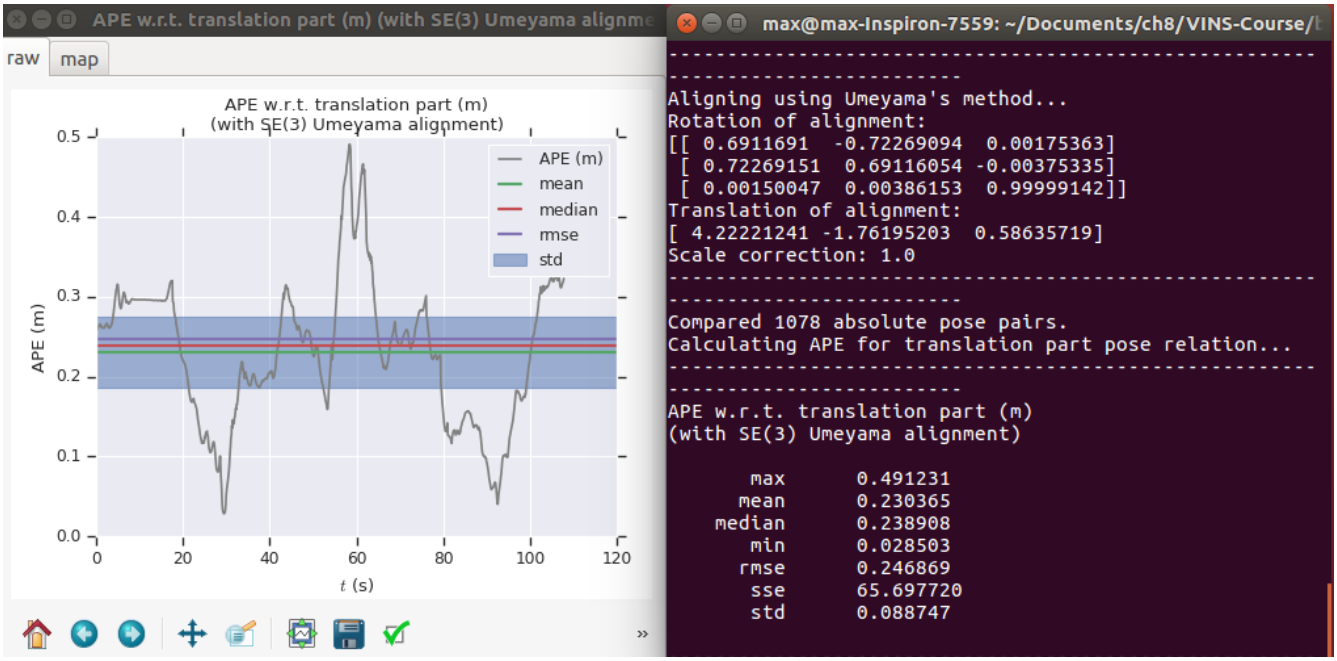
策略２在MH_05上不能很好工作，跑一半轨迹就发散了，因此在精度和收敛速度上，策略２完败。

下面给出策略１和原来课程LM策略３精度对比：

策略１跟ground_truth对比的绝对位姿误差：



策略３(课程代码)跟ground_truth对比的绝对位姿误差：



1.3 对比结论

综上两个实验测试分析，根据rmse均方差可以看到策略１和策略３(课程代码)的精度是一致的，但是误差最大值和标准差方面，说明策略１比策略３能更好地拟合数据，这方面性能略有优势。综合两个实验，策略１在时间和精度上略优于策略３，远优于策略２。

２．采用dogleg算法替代LM算法

dogleg算法也是高斯牛顿和最速下降法的组合策略，是一种完全受信赖域控制的无约束优化算法。信赖域的半径对于迭代的成功至关重要。其中信赖域半径的策略跟准确的高斯牛顿步骤密切相关。具体细节可参考以下两篇文章<>
<>，由于LM算法在一次迭代失败后，LM算法要求再次求解阻尼项增加后的正则方程。换句话说，阻尼项的每次更新都需要一个新的解决方案，因此，在增广方程中，失败的步骤会导致无效的努力。但是dogleg算法中，一旦高斯牛顿步骤被确定下来，可以通过各种信赖域半径的变化去求解一个约束的２次子问题，无需重复求解正则方程。因此dogleg算法相比LM算法更节省计算资源。

作业中实现的代码框架参考第一篇论文，对于高斯牛顿步骤的求解，采用了ceres中的策略去准确地求解高斯牛顿步骤，即采用hessian矩阵的对角线元素去规范正则方程，求解器使用的是QR求解，直接用ldlt求解会导致误差函数不下降。关键代码如图所示，小节末尾会给出dogleg完整迭代代码：

```
while (!stop && (iter < iterations)) {
    std::cout << "iter: " << iter << " , chi= " << currentChi_ << " , radius = " << radius_ << std::endl;
    file << fixed << currentChi_ << endl;
    // step 1: alpha * -gradient is the Cauchy point.
    // The Cauchy point is the global minimizer of the quadratic model
    // along the one-dimensional subspace spanned by the gradient.
    // VecX scaled_gradient = (gradient_.array() / diagonal_.array()).matrix();
    // alpha_ = gradient_.squaredNorm() / (scaled_gradient.transpose() * Hessian_ * scaled_gradient);

    alpha_ = gradient_.squaredNorm() / (gradient_.transpose() * Hessian_ * gradient_);

    // solve gauss newton step exactly
    // step 2: gauss newton step
    while (mu_ < max_mu_)
    {
        lm_diagonal_ = diagonal_ * std::sqrt(mu_);
        assert(Hessian_.cols() == lm_diagonal_.size());
        const int cols = Hessian_.cols();
        Hessian_ += lm_diagonal_.asDiagonal();

        InvalidateArray(cols, gauss_newton_step_.data());
        gauss_newton_step_ = Hessian_.householderQr().solve(b_);
        // gauss_newton_step_ = Hessian_.ldlt().solve(b_);

        if (!IsArrayValid(cols, gauss_newton_step_.data()))
        {
            mu_ *= mu_increase_factor_;
            cout << "Increasing mu " << mu_ << endl;
            continue;
        } else {
            cout << "valid Gauss Newton Step !" << endl;
            break;
        }
    }

    // 这种方式求得的高斯牛顿不好，会导致误差函数不下降
    // gauss_newton_step_ = Hessian_.ldlt().solve(b_);
```

在确定一个准确的gauss newton step后，通过cauchy step和gauss newton step步长确定dogleg的迭代步骤和预计误差函数的下降：

```
// step 3: compute dog leg step
while (rho <= 0)
{
    //TicToc t_linearsolver;
    // dog leg step
    const double gradient_norm = gradient_.norm();
    const double gauss_newton_norm = gauss_newton_step_.norm();

    if (gradient_norm * alpha_ >= radius_)
    {
        // Case 2. The Cauchy point and the Gauss-Newton steps lie outside
        // the trust region. Rescale the Cauchy point to the trust region
        dogleg_step_ = -(radius_ / gradient_norm) * gradient_;
        dogleg_step_norm_ = radius_;
        model_cost_change_ = radius_ * (2 * (alpha_ * gradient_).norm() - radius_) / (2 * alpha_);
        // cout << "Cauchy step size: " << dogleg_step_norm_  << " radius: " << radius_ << endl;
    } else  {
        // Case 1. The Gauss-Newton step lies inside the trust region, and
        // is therefore the optimal solution to the trust-region problem.
        if (gauss_newton_norm <= radius_) {
            dogleg_step_ = gauss_newton_step_;
            dogleg_step_norm_ = gauss_newton_norm;
            model_cost_change_ = currentChi_; // L(0)-L(h_dl)= F(x)
            // cout << "GaussNewton step size: " << dogleg_step_norm_ << " radius: " << radius_ << endl;
        } else {
            // Case 3. The Cauchy point is inside the trust region and the Gauss-Newton step is outside.
            // Compute the line joining the two points and the point on it which intersects the trust region boundary.

            // a = alpha * -gradient
            // b = gauss_newton_step
            const double b_dot_a = -alpha_ * gradient_.dot(gauss_newton_step_);
            const double a_squared_norm = pow(alpha_*gradient_norm, 2.0);
            const double b_minus_a_squared_norm =
                a_squared_norm - 2 * b_dot_a + pow(gauss_newton_norm, 2);

            // c = a' (b - a)
            const double c = b_dot_a - a_squared_norm;
            const double d = sqrt(c * c + b_minus_a_squared_norm *
                                  (pow(radius_, 2.0) - a_squared_norm));

            double beta =
                (c <= 0)
                ? (d - c) /  b_minus_a_squared_norm
                : (radius_ * radius_ - a_squared_norm) / (d + c);
            dogleg_step_ = (-alpha_ * (1.0 - beta)) * gradient_ + beta * gauss_newton_step_;
            dogleg_step_norm_ = dogleg_step_.norm();
            model_cost_change_ = 0.5 * alpha_ * pow(1 - beta, 2) * pow(gradient_norm, 2) + beta * (2 - beta) * currentChi_;
            // cout << "Dogleg step size: " << dogleg_step_norm_  << " radius: " << radius_ << endl;
        }
    }
    delta_x_ = dogleg_step_;
```

有了dogleg step后，判断这一次迭代的有效性，计算增益率，更新信赖域半径等，直到一次有效的迭代或者迭代终止：

```
        // step 4: update states
        UpdateStates();

        // recompute residuals after update state F(x_new)
        double tempChi = 0.0;
        for (auto edge: edges_) {
            edge.second->ComputeResidual();
            tempChi += edge.second->RobustChi2();
        }
        if (err_prior_.size() > 0)
            tempChi += err_prior_.norm();
        tempChi *= 0.5;          // 1/2 * err^2

        //  compute gain ratio
        rho = (currentChi_ - tempChi) / model_cost_change_;
        cout << "gain ratio: " << rho << endl;

        // 更新误差函数和求解高斯牛顿需要的对角元素
        if (rho > 0) {
            // update F(x) = F(x_new)
            currentChi_ = tempChi;

            // Reduce the regularization multiplier, in the hope that whatever
            // was causing the rank deficiency has gone away and we can return
            // to doing a pure Gauss-Newton solve.
            mu_ = std::max(min_mu_, 2.0 * mu_ / mu_increase_factor_);

        } else {
            // don't update state
            RollbackStates();
        }

        // update radius (keep unchanged,if 0.25 <= rho <= 0.75)
        if (rho > increase_threshold_) {
            radius_ = std::max(radius_, 3.0 * dogleg_step_norm_);
        } else if (rho < decrease_threshold_) {
            radius_ *= 0.5;
            // stop on radius
            for (auto vertex : verticies_)
            {
                ulong idx = vertex.second->OrderingId();
                ulong dim = vertex.second->Dimension();
                x.segment(idx, dim).noalias() += vertex.second->Parameters();
                // cout << "idx = " << idx << ", dim = " << dim  << ", param size = " << vertex.second->Parameters().size() << endl;
            }
            stop = radius_ <= eps2 * x.norm() ? true : false;
            if (stop)
            {
                cout << "stop on radius is true !" << endl;
                break;
            }
```

整个过程的关键在于减少了make hessian 和gauss newton step的求解。先给出dogleg初始化参数：

```
void Problem::SetDogLegParameters() {
    radius_ = 1e4; // 初始半径
    eps1 = eps2 = 1e-3; // 1e-4 rmse比lm小 但是平均耗时高
    increase_threshold_ = 0.75;
    decrease_threshold_ = 0.25;

    min_diagonal_= 1e-6;
    max_diagonal_ = 1e32;

    mu_ = 1e-8;
    min_mu_ = 1e-8;
    max_mu_ = 1.0;
    mu_increase_factor_ = 10.;

    diagonal_ = Hessian_.diagonal();
    ulong size = Hessian_.cols();
    for (int i = 0; i < size; ++i) {
        diagonal_(i) = std::min(std::max(diagonal_(i), min_diagonal_),
                                max_diagonal_);
    }
    diagonal_ = diagonal_.array().sqrt();
```

```
    dogleg_step_ = VecX::Zero(size);
    model_cost_change_ = 0;
    dogleg_step_norm_ = 0.;

    //计算初始的误差
    currentChi_ = 0;
    for (auto edge: edges_) {
        currentChi_ += edge.second->RobustChi2();
    }
    if (err_prior_.rows() > 0)
        currentChi_ += err_prior_.norm();

    currentChi_ *= 0.5;
}
```

下面给出完整的dogleg代码：

```
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 矩阵
    MakeHessian();
    // DL 初始参数
    SetDogLegParameters();
    // DL 算法迭代求解
    bool stop = false;
    int iter = 0;

    if (gradient_.norm() <= eps1) {
        return true;
    }

    std::ofstream file;
    file.open("./dogleg_chi.txt", fstream::app | fstream::out);
    if(!file.is_open())
    {
        cerr << "file is not open" << endl;
    }

    while (!stop && (iter < iterations)) {
        std::cout << "iter: " << iter << " , chi= " << currentChi_ << " , radius = " <<
radius_ << std::endl;
        file << fixed << currentChi_ << endl;
        // step 1: alpha * -gradient is the Cauchy point.
        // The Cauchy point is the global minimizer of the quadratic model
```

```
        // along the one-dimensional subspace spanned by the gradient.
        // VecX scaled_gradient = (gradient_.array() / diagonal_.array()).matrix();
        // alpha_ = gradient_.squaredNorm() / (scaled_gradient.transpose() * Hessian_ *
scaled_gradient);

        alpha_ = gradient_.squaredNorm() / (gradient_.transpose() * Hessian_ * gradient_);

        // solve gauss newton step exactly
        // step 2: gauss newton step
        while (mu_ < max_mu_)
        {
            lm_diagonal_ = diagonal_ * std::sqrt(mu_);
            assert(Hessian_.cols() == lm_diagonal_.size());
            const int cols = Hessian_.cols();
            Hessian_ += lm_diagonal_.asDiagonal();

            InvalidateArray(cols, gauss_newton_step_.data());
            gauss_newton_step_ = Hessian_.householderQr().solve(b_);
            // gauss_newton_step_ = Hessian_.ldlt().solve(b_);

            if (!IsArrayValid(cols, gauss_newton_step_.data()))
            {
                mu_ *= mu_increase_factor_;
                cout << "Increasing mu " << mu_ << endl;
                continue;
            } else {
                cout << "valid Gauss Newton Step !" << endl;
                break;
            }
        }

        // 这种方式求得的高斯牛顿不好，会导致误差函数不下降
        // gauss_newton_step_ = Hessian_.ldlt().solve(b_);

        double rho = 0.;

        // step 3: compute dog leg step
        while (rho <= 0)
        {
            //TicToc t_linearsolver;
            // dog leg step
            const double gradient_norm = gradient_.norm();
            const double gauss_newton_norm = gauss_newton_step_.norm();

            if (gradient_norm * alpha_ >= radius_)
            {
                // Case 2. The Cauchy point and the Gauss-Newton steps lie outside
                // the trust region. Rescale the Cauchy point to the trust region
                dogleg_step_ = -(radius_ / gradient_norm) * gradient_;
                dogleg_step_norm_ = radius_;
                model_cost_change_ = radius_ * (2 * (alpha_ * gradient_).norm() - radius_)
/ (2 * alpha_);
```

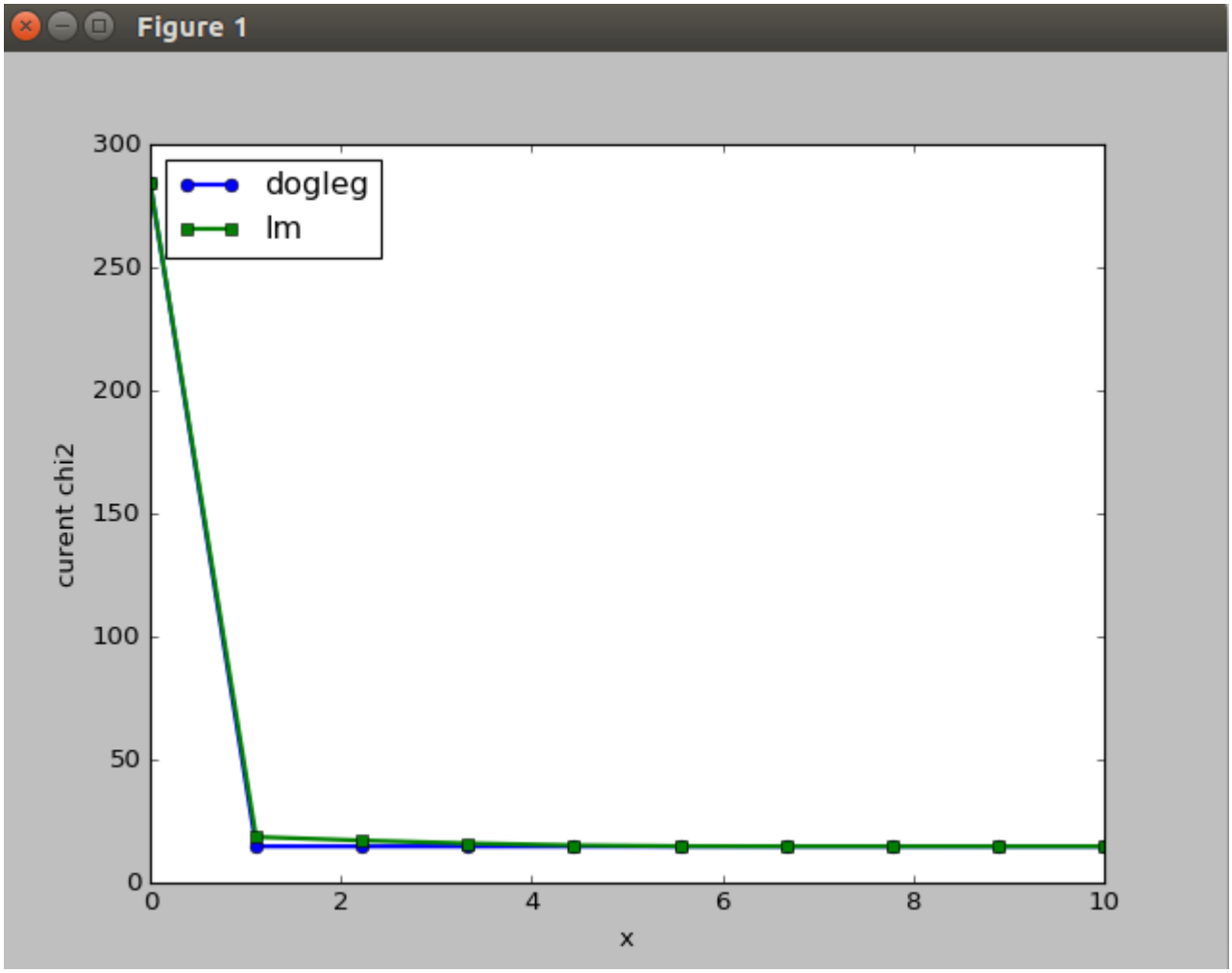
```
                // cout << "Cauchy step size: " << dogleg_step_norm_  << " radius: " <<
radius_ << endl;
            } else  {
                // Case 1. The Gauss-Newton step lies inside the trust region, and
                // is therefore the optimal solution to the trust-region problem.
                if (gauss_newton_norm <= radius_) {
                    dogleg_step_ = gauss_newton_step_;
                    dogleg_step_norm_ = gauss_newton_norm;
                    model_cost_change_ = currentChi_; // L(0)-L(h_dl)= F(x)
                    // cout << "GaussNewton step size: " << dogleg_step_norm_ << " radius:
" << radius_ << endl;
                } else {
                    // Case 3. The Cauchy point is inside the trust region and the Gauss-
Newton step is outside.
                    // Compute the line joining the two points and the point on it which
intersects the trust region boundary.

                    // a = alpha * -gradient
                    // b = gauss_newton_step
                    const double b_dot_a = -alpha_ * gradient_.dot(gauss_newton_step_);
                    const double a_squared_norm = pow(alpha_*gradient_norm, 2.0);
                    const double b_minus_a_squared_norm =
                        a_squared_norm - 2 * b_dot_a + pow(gauss_newton_norm, 2);

                    // c = a' (b - a)
                    const double c = b_dot_a - a_squared_norm;
                    const double d = sqrt(c * c + b_minus_a_squared_norm *
                                          (pow(radius_, 2.0) - a_squared_norm));

                    double beta =
                        (c <= 0)
                        ? (d - c) /  b_minus_a_squared_norm
                        : (radius_ * radius_ - a_squared_norm) / (d + c);
                    dogleg_step_ = (-alpha_ * (1.0 - beta)) * gradient_ + beta *
gauss_newton_step_;
                    dogleg_step_norm_ = dogleg_step_.norm();
                    model_cost_change_ = 0.5 * alpha_ * pow(1 - beta, 2) *
pow(gradient_norm, 2) + beta * (2 - beta) * currentChi_;
                    // cout << "Dogleg step size: " << dogleg_step_norm_  << " radius: " <<
radius_ << endl;
                }
            }
            delta_x_ = dogleg_step_;

            int size = delta_x_.size();
            VecX x(VecX::Zero(size));

            // stop on dog leg
            for (auto vertex : verticies_)
            {
                ulong idx = vertex.second->OrderingId();
                ulong dim = vertex.second->Dimension();
                x.segment(idx, dim).noalias() += vertex.second->Parameters();
```

```
                // cout << "idx = " << idx << ", dim = " << dim  << ", param size = " <<
vertex.second->Parameters().size() << endl;
            }
            stop = dogleg_step_norm_ <= eps2 * x.norm() ? true : false;

            if (stop){
                cout << "stop on dog leg is true !" << endl;
                break;
            } else {
                // step 4: update states
                UpdateStates();

                // recompute residuals after update state F(x_new)
                double tempChi = 0.0;
                for (auto edge: edges_) {
                    edge.second->ComputeResidual();
                    tempChi += edge.second->RobustChi2();
                }
                if (err_prior_.size() > 0)
                    tempChi += err_prior_.norm();
                tempChi *= 0.5;          // 1/2 * err^2

                //  compute gain ratio
                rho = (currentChi_ - tempChi) / model_cost_change_;
                cout << "gain ratio: " << rho << endl;

                // 更新误差函数和求解高斯牛顿需要的对角元素
                if (rho > 0) {
                    // update F(x) = F(x_new)
                    currentChi_ = tempChi;

                    // Reduce the regularization multiplier, in the hope that whatever
                    // was causing the rank deficiency has gone away and we can return
                    // to doing a pure Gauss-Newton solve.
                    mu_ = std::max(min_mu_, 2.0 * mu_ / mu_increase_factor_);

                } else {
                    // don't update state
                    RollbackStates();
                }

                // update radius (keep unchanged,if 0.25 <= rho <= 0.75)
                if (rho > increase_threshold_) {
                    radius_ = std::max(radius_, 3.0 * dogleg_step_norm_);
                } else if (rho < decrease_threshold_) {
                    radius_ *= 0.5;
                    // stop on radius
                    for (auto vertex : verticies_)
                    {
                        ulong idx = vertex.second->OrderingId();
                        ulong dim = vertex.second->Dimension();
                        x.segment(idx, dim).noalias() += vertex.second->Parameters();
```

```
                        // cout << "idx = " << idx << ", dim = " << dim  << ", param size =
" << vertex.second->Parameters().size() << endl;
                    }
                    stop = radius_ <= eps2 * x.norm() ? true : false;
                    if (stop)
                    {
                        cout << "stop on radius is true !" << endl;
                        break;
                    }
                }
            }
        }
        // 终止后，减少make hessisan的次数
        if (!stop)
        {
            // step 5: 在新线性化点 构建 hessian gradient
            MakeHessian();

            // stop
            if (gradient_.norm() <= eps1) {
                cout << "stop on gradient is true !" << endl;
                stop = true;
                break;
            }

            iter++;
        }
    }
    std::cout << "problem solve cost: " << t_solve.toc() << " ms" << std::endl;
    std::cout << "make Hessian cost: " << t_hessian_cost_ << " ms" << std::endl;
    t_hessian_cost_ = 0.;
    file.close();
    return true;
}
```

2.1 收敛速度

利用第三章仿真的数据集，分别得到VINS初始化完成过程中，利用LM和dogleg算法的误差下降曲线，绘制如下：

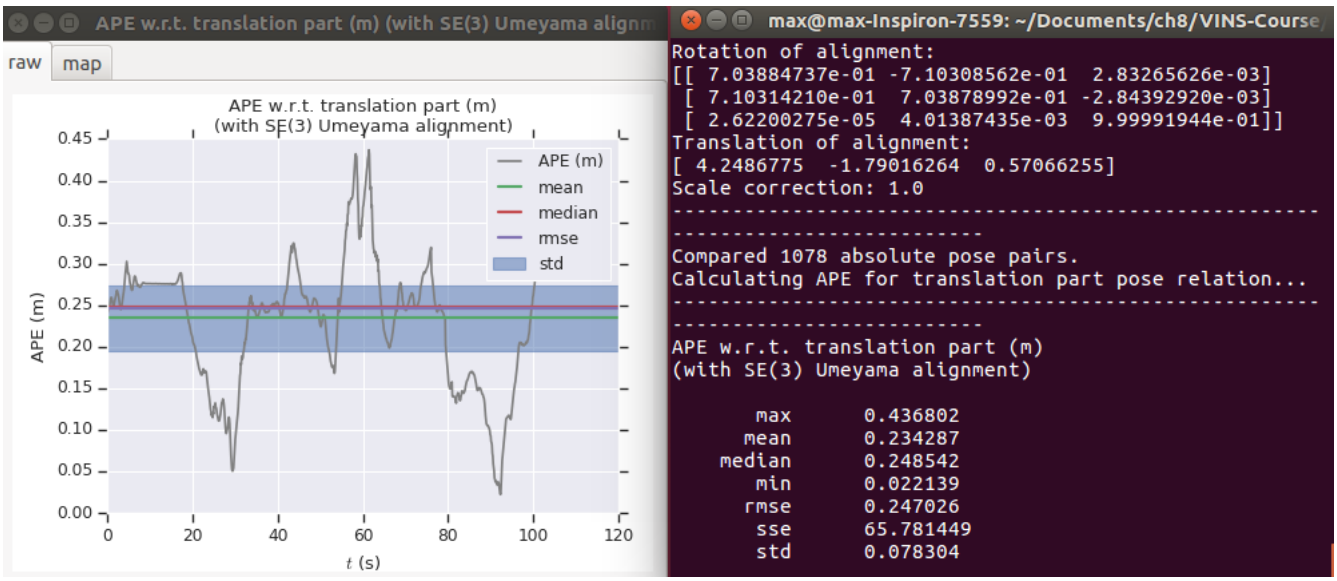


可以看到，LM算法要经过 5 次迭代才能收敛到dogleg经过 1 次迭代误差下降的状态，这个过程无疑dogleg收敛更快。

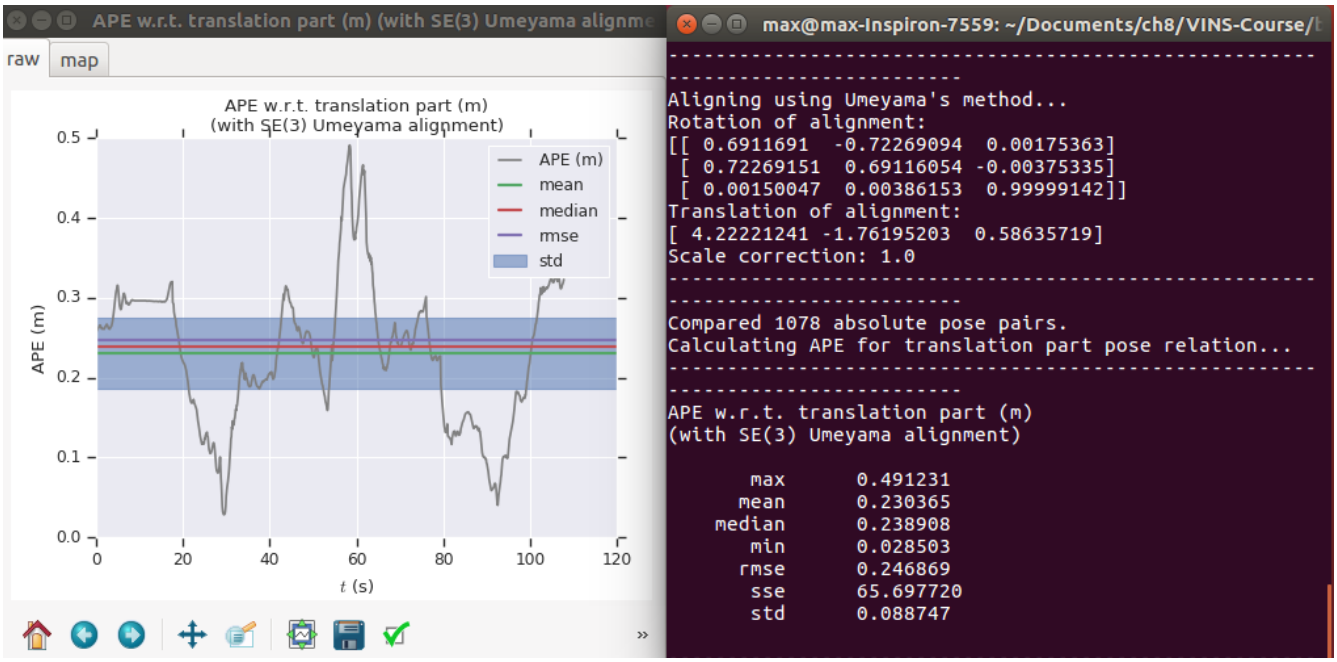
2.2 优化精度

在MH_05数据集上，分别利用原始的LM算法和dogleg算法跑完整个数据集，得到输出轨迹，并利用evo_ape评价两种轨迹与ground truth的绝对位姿误差：

dogleg算法与ground truth对比的绝对位姿误差：



LM策略 3 (课程代码)跟ground_truth对比的绝对位姿误差：



在精度对比上，通过rmse可以看到LM算法和dogleg算法基本完全一致，平均误差LM比dogleg小１cm，但dogleg算法在最大误差以及误差分布方面比LM算法略有优势。

2.3 对比结论

dogleg算法在精度方面跟LM算法基本一致的情况下，收敛速度优势明显，对于计算资源有限和实时性要求较高的场景有重要的意义，这个在下一小节后端求解器性能对比也会详细给出。

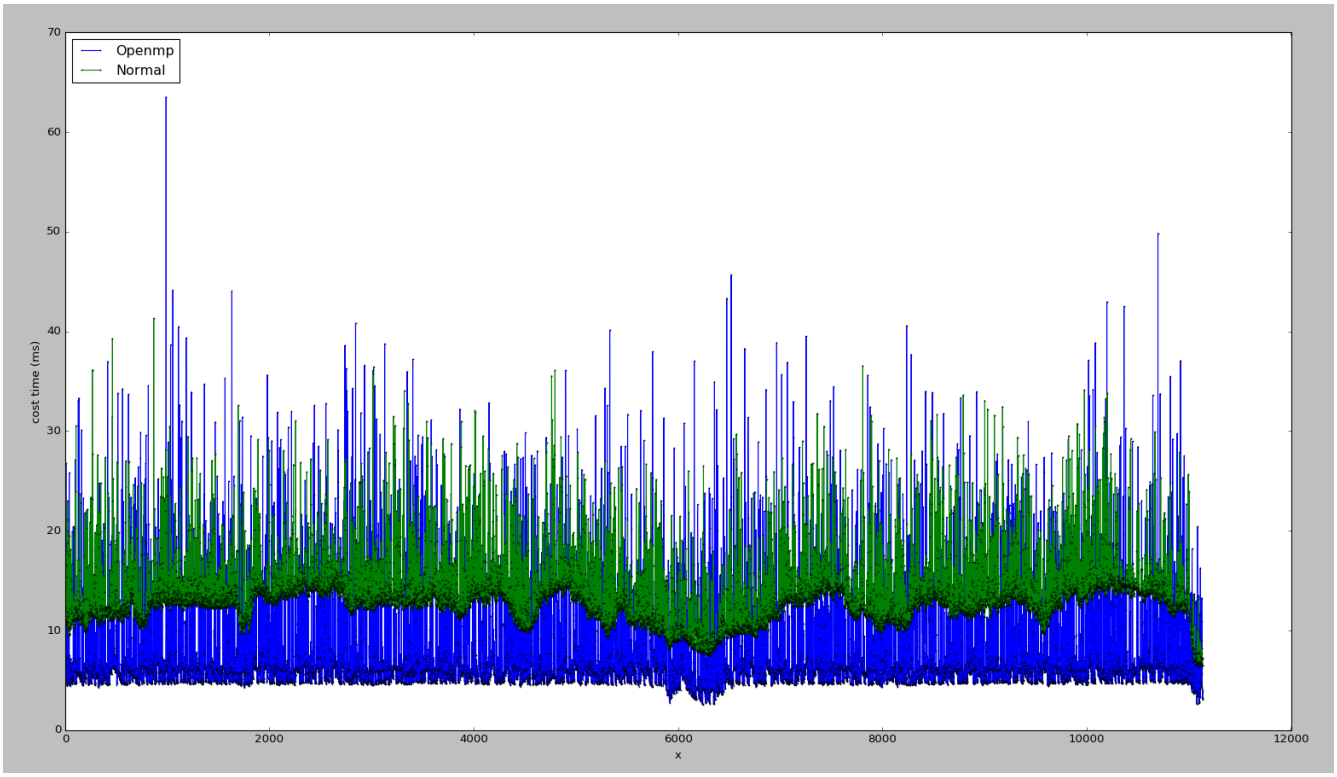
3 . 使用openmp加速make hessian计算

电脑配置为i5 6300HQ, 4核。编译器gcc 5.4, openmp版本 4 .0,主要是openmp 4.0不支持c++11auto range ,for 循环要遵从范式，因此对for循环方式做了更改。另外多线程的时候，存在数据依赖，需要用到reduction, 并对reduction针对eigen matrix做了重载操作，代码如下：

```cpp
#ifdef USE_OPENMP

#pragma omp declare reduction (+: MatXX: omp_out=omp_out+omp_in)\
    initializer(omp_priv=MatXX::Zero(omp_orig.rows(), omp_orig.cols()))

#pragma omp declare reduction (+: VecX: omp_out=omp_out+omp_in)\
    initializer(omp_priv=VecX::Zero(omp_orig.size()))

#pragma omp parallel for  reduction(+: H) reduction(+: b) reduction(+: g)

    for (size_t i = 0; i < edges_.bucket_count(); i++)
    {
        for (auto edge = edges_.begin(i); edge != edges_.end(i); edge++)
        {
            edge->second->ComputeResidual();
            edge->second->ComputeJacobians();

            // TODO:: robust cost
            auto jacobians = edge->second->Jacobians();
            auto verticies = edge->second->Verticies();
            assert(jacobians.size() == verticies.size());
            for (size_t i = 0; i < verticies.size(); ++i) {
                auto v_i = verticies[i];
                if (v_i->IsFixed()) continue;    // Hessian 里不需要添加它的信息，也就是它的雅克比为 0

                auto jacobian_i = jacobians[i];
                ulong index_i = v_i->OrderingId();
                ulong dim_i = v_i->LocalDimension();

                // 鲁棒核函数会修改残差和信息矩阵，如果没有设置 robust cost function，就会返回原来的
                double drho;
                MatXX robustInfo(edge->second->Information().rows(),edge->second->Information().cols());
                edge->second->RobustInfo(drho,robustInfo);

                MatXX JtW = jacobian_i.transpose() * robustInfo;
                for (size_t j = i; j < verticies.size(); ++j) {
                    auto v_j = verticies[j];

                    if (v_j->IsFixed()) continue;

                    auto jacobian_j = jacobians[j];
                    ulong index_j = v_j->OrderingId();
                    ulong dim_j = v_j->LocalDimension();

                    assert(v_j->OrderingId() != -1);
                    MatXX hessian = JtW * jacobian_j;

                    // 所有的信息矩阵叠加起来
                    H.block(index_i, index_j, dim_i, dim_j).noalias() += hessian;
                    if (j != i) {
                        // 对称的下三角
                        H.block(index_j, index_i, dim_j, dim_i).noalias() += hessian.transpose();
                    }
                }
                b.segment(index_i, dim_i).noalias() -= drho * jacobian_i.transpose()* edge->second->Information() * edge->second->Residual();
                g.segment(index_i, dim_i).noalias() += drho * jacobian_i.transpose()* edge->second->Information() * edge->second->Residual();
            }
        }
    }
```

3.1 make hessian加速时间对比

在MH_05数据集上，跑完整个数据集，分别得到每次使用openmp加速make hessian的时间和正常make hessian的时间如下(采用omp_get_wtime获取时间)，其中openmp加速不存在race condition时基本耗时在4ms左右，cpu占用150%，openmp加速的平均耗时7.93ms，cpu占用350%，正常make hessian平均耗时13.52ms，总体提速41.3%（在cpu算力最多提高133%的情况下，cpu资源不够用了）：



3.2 整体后端求解器性能对比

在MH_05数据集上，分别测得LM,LM_OpenMP加速,dogleg,dogleg_OpenMP加速与ground truth对比的精度，以及每次求解的平均耗时：

Method	SolveAvgTime	Rmse	Median	Std	Max	Min
LM	218.77 ms	0.247	0.239	0.089	0.491	0.029
LM_OpenMP	148.73 ms	0.247	0.239	0.089	0.491	0.028
DogLeg	61.42 ms	0.254	0.258	0.077	0.441	0.023
DogLeg_OpenMP	83.40 ms	0.254	0.258	0.077	0.441	0.023

3.3 对比结论

经过上述两组在数据集MH_05上的对比测试分析，dogleg算法在基本不损失精度的情况下，求解时间可以提高２～３倍。但是openmp加速make hessian过程在LM算法基础上求解速度提高了32%左右，在dogleg算法基础上反而平均时间有所增加，应该是cpu占用过多时，由于make hessian过程中的数据依赖，访问共享数据时，需要加锁保护，增加了额外的开销。可以通过开辟局部变量，通过空间换时间的处理，但是这种方式在嵌入式平台上，也不太可行。总体感觉openmp对make hessian加速贡献不如dogleg算法大。