Homework 6

Problem

1. Implement Gauss-Newton's and LM (Levenberg-Marquardt) for the following models and the given observations:
   (a) Model 1: $\phi(a,b,c,d;x,y,z) = ax + by + cz + d$
   (b) Model 2: $\phi(a,b,c,d;x,y,z) = exp(-[(x-a)^2 + (y-b)^2 + (z-c)^2]/d^2)$
2. You should determine unknown parameters a, b, c and d in the least square sense. Please discuss how Gauss-Newton's and LM methods are going on.

Implementation

1. Implementation
   (a) **LeastSquareMethod**
      i. This is the abstract class to implement Gauss-Newton's method and LM method
      ii. Therefore, this has the utilizing functions
         - converting function: original function to residisual function
         - residual vector calculating function
         - jacobian matrix calculating function

```
#ifndef __LEASTSQUARE_H__
#define __LEASTSQUARE_H__

#include <Eigen/Dense>
#include "multivariate.h"

namespace numerical_optimization {

template<typename vector_t>
class LSM : Method {
public:
    // the coefficients are one more than the variable
    using coeff_t = Eigen::Matrix<typename vector_t::Scalar,
    ↪  vector_t::RowsAtCompileTime+1, vector_t::ColsAtCompileTime>;
    using function_t = std::function<double(const coeff_t&, const vector_t&)>;

    // constructor
    LSM(function_t func):function(func),coefficient(coeff_t::Constant(5)){};
```

```
18      LSM(function_t func, coeff_t coef):function(func),coefficient(coef){};
19
20      // set observation data
21      void set_observation(const std::vector<vector_t>& obs_x, const
    ↪   std::vector<double> obs_f){ observation_x=obs_x;observation_f=obs_f; };
22
23      // get redisidual function
24      double residual_function(const coeff_t& coeff, const vector_t& vars, double
    ↪   f) {
25          return function(coeff, vars) - f;
26      }
27
28      // calculate residual vectors
29      VectorXd calculate_residual(const coeff_t& coef) {
30
31          size_t size = observation_x.size();
32          VectorXd residue(size);
33
34          for(size_t i=0; i<size; i++) {
35              auto var = observation_x[i];
36              auto val = observation_f[i];
37
38              residue[i] = residual_function(coef, var, val);
39          }
40          return residue;
41      }
42
43      MatrixXd calculate_jacobian(coeff_t coef, double eps=1e-6) {
44
45          const size_t     cols = observation_x.size();
46          constexpr size_t rows = coeff_t::RowsAtCompileTime;
47          MatrixXd result(cols, rows);
48
49          for(size_t i=0; i<coeff_t::RowsAtCompileTime; i++) {
50              coeff_t coef_1=coef, coef_2=coef;
51
52              coef_1[i]+=eps; coef_2[i]-=eps;
53              VectorXd diff = (calculate_residual(coef_1) -
    ↪   calculate_residual(coef_2))/(2*eps);
54              result.col(i) = diff;
55          }
56          return result;
57      }
58
```

```
    double loss(const coeff_t& coeff) {
        double inv = 1/2;
        return calculate_residual(coeff).squaredNorm()/2;
    }

protected:
    function_t function;
    coeff_t coefficient;
    std::vector<vector_t> observation_x;
    std::vector<double>  observation_f;
};

////////////////////////////////////////////////////
} /// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__LEASTSQUARE_H__
```

(a) **Gauss-Newton's**

i. $(J^T * J)^{-1} * J$ is a pseudo inverse form of jacobian, and this method is already implemented in Eigen3 library

```
#ifndef __GAUSSNEWTONS__
#define __GAUSSNEWTONS__

#include "lsm.h"

namespace numerical_optimization {

template<typename vector_t>
class GaussNewtons : public LSM<vector_t> {
public:
    using Base = LSM<vector_t>;
    using Base::Base;
    using Base::function;
    using Base::coefficient;
    using Base::calculate_residual;
    using Base::calculate_jacobian;
    using Base::loss;

    using coeff_t = typename Base::coeff_t;
    using function_t = std::function<double(const coeff_t&, const vector_t&)>;

```

```
22      GaussNewtons(function_t func):Base(func){};
23      GaussNewtons(function_t func, coeff_t coef):Base(func, coef){};
24
25      coeff_t fit(size_t max_iter) {
26
27          for(size_t i=0; i<max_iter; i++) {
28
29              VectorXd residual = calculate_residual(coefficient);
30              MatrixXd jacobian = calculate_jacobian(coefficient);
31              MatrixXd pinv =
    ↪   jacobian.completeOrthogonalDecomposition().pseudoInverse();
32
33              auto p = -pinv*residual;
34              coefficient = coefficient + p;
35              if(loss(coefficient)<1e-4) break;
36          }
37
38          return coefficient;
39      }
40
41  };
42  //// ////////////////////////////////////////////////
43  }/// the end of namespace numerical_optimization ///
44  ////////////////////////////////////////////////////
45  #endif //__GAUSSNEWTONS__
```

(a) **LM (Levenberg-Marquardt)**

i. λ is alwary initialized as one in the start point of optimization iteration, and adaptively computed newly after calculating the pseudo hessian value

```
#ifndef __LM__
#define __LM__

#include "lsm.h"

namespace numerical_optimization {

template<typename vector_t>
class LM : public LSM<vector_t> {
public:
    using Base = LSM<vector_t>;
    using Base::Base;
    using Base::function;
    using Base::coefficient;
    using Base::calculate_residual;
    using Base::calculate_jacobian;
    using Base::loss;

    using coeff_t = typename Base::coeff_t;
    using function_t = std::function<double(const coeff_t&, const vector_t&)>;

    LM(function_t func):Base(func){};
    LM(function_t func, coeff_t coef):Base(func, coef){};

    inline MatrixXd calculate_hessian(const MatrixXd& jacobian, double lambda) {
        MatrixXd jtj = jacobian.transpose() * jacobian;
        return (jtj + lambda * MatrixXd::Identity(jtj.cols(),
↪   jtj.rows())).inverse() * jacobian.transpose();
    }

    inline bool is_descent(const coeff_t& coefficient, const coeff_t& p) {
        return
↪   calculate_residual(coefficient-p).squaredNorm()<=calculate_residual(coefficient).squaredNo
    }

    coeff_t fit(size_t max_iter) {

        for(size_t i=0; i<max_iter; i++) {

            double lambda = 1;
```

```
39
40              VectorXd residual = calculate_residual(coefficient);
41              MatrixXd jacobian = calculate_jacobian(coefficient);
42              MatrixXd phessian = calculate_hessian(jacobian, lambda);
43              VectorXd p = phessian * residual;
44
45              if(is_descent(coefficient, p)) {
46                  lambda /= 10;
47                  p = calculate_hessian(jacobian, lambda) * residual;
48              } else {
49                  while(!is_descent(coefficient, p)) {
50                      lambda *= 10;
51                      p = calculate_hessian(jacobian, lambda) * residual;
52                  }
53              }
54              coefficient = coefficient - p;
55              if(loss(coefficient)<1e-4) break;
56          }
57
58          return coefficient;
59      }
60
61  };
62  //// ////////////////////////////////////////////////
63  }/// the end of namespace numerical_optimization ///
64  ////////////////////////////////////////////////////
65  #endif //__LM__
```

Homework 6

Result

1. Convergence

   - When the descent condition is satisfied, which means the $the value of \lambda is small$, the two method behavior is almost same. The first function case shows this phenomenon.
   - Depending on the initial point, Gauss-Newton's method can be failed to converge. In constrast to this, LM method shows more stable behavior to initial points. For example, as shown in the table, in the second function case, gauss-newton's method faild to converge, when the initial points are [-2, -2, 2, 2] and [10, 10, 10, 10]

initial	$f(a,b,c,d;t)$	Final point(a,b,c,d)	
		Gauss-Newton's	LM
[-1, -1, 1, 1]	(a)	0.00171, 0.00071, -0.00352, 0.253	0.00171, 0.00071, -0.00352, 0.253
	(b)	10412, -22924.2, 31402.8, 6338.3	5.33133, 5.68737, 5.60708, 9.92438
[-2, -2, 2, 2]	(a)	0.00171, 0.00071, -0.00352, 0.253	0.00171, 0.00071, -0.00352, 0.253
	(b)	63.6425, 27.2601, -46.5018, 5.0931	5.33133, 5.68737, 5.60708, 9.92438
[10,10,10,10]	(a)	0.00171, 0.00071, -0.00352, 0.253	0.00171, 0.00071, -0.00352, 0.253
	(b)	5.33133, 5.68737, 5.60708, 9.92438	5.33133, 5.68737, 5.60708, 9.92438