非线性最小二乘

lab 5

姓名： 周炜

学号： 32010103790

非线性最小二乘

线性最小二乘

最小二乘法是一种用于拟合数据和解决最小化误差的优化技术。它的核心思想是通过最小化观测数据与模型预测值之间的残差平方和来找到最佳参数。通常用于线性和非线性回归分析，以及其他数据拟合问题。

在最小二乘法中，我们通常有一组观测数据点(x_i, y_i)，其中x_i是自变量，y_i是对应的因变量。我们要找到一个函数或模型$f(x;\theta)$，其中θ是模型参数，使得该模型的预测值$f(x_i;\theta)$尽可能地接近观测值y_i。通常，这可以通过最小化残差的平方和来实现，即最小化下式：

$$\min \sum_{i=1}^{n} (y_i - f(x_i;\theta))^2$$

其中，n是观测数据点的数量。

对于线性回归问题，模型通常是一个线性函数，例如：

$$f(x;\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_m x_m$$

这里，$\theta_0, \theta_1, \ldots, \theta_m$是线性模型的参数。

对于非线性回归问题，模型可以是多项式、指数函数、对数函数等形式。

最小二乘法的解可以通过各种数值优化算法来求解，其中最常见的是梯度下降法和正规方程法。

- **梯度下降法**：通过迭代更新参数，沿着误差曲面的负梯度方向逐步调整参数，直至达到局部最优解。
- **正规方程法**：通过对残差的平方和关于参数求导，令导数为零，得到参数的解析解。

总的来说，最小二乘法是一种强大的工具，用于估计模型参数，拟合数据，并找到观测数据与模型之间的最佳拟合。

非线性最小二乘

Gauss-Newton（GN）方法是从Newton-Raphson（NR）方法演变而来的。NR方法利用二次型近似来逼近目标函数。为了进行这种近似，NR方法需要计算目标函数的Hessian矩阵，但这通常不是一个容易完成的任务。GN方法利用最小二乘问题本身的特性，通过计算Jacobian矩阵来近似Hessian矩阵。

NR方法使用二阶Taylor展开来近似目标函数：

$$\mathrm{F}(\mathrm{x}+\Delta\mathrm{x}) \approx \mathrm{F}(\mathrm{x}) + \mathrm{J}_\mathrm{F}\Delta\mathrm{x} + \Delta\mathrm{x}^\mathrm{T}\mathrm{H}_\mathrm{F}\Delta\mathrm{x}$$

最小二乘问题具有$\mathrm{F}(\mathrm{x}) = ||\mathrm{R}(\mathrm{x})||_2^2$的形式，对 R 进行一阶 Taylor 展开，可以得到：

$$\begin{aligned}\mathrm{F}(\mathrm{x}+\Delta\mathrm{x}) &= \|\mathrm{R}(\mathrm{x}+\Delta\mathrm{x})\|_2^2 \\ &\approx \|\mathrm{R}(\mathrm{x}) + \mathrm{J}_\mathrm{R}\Delta\mathrm{x}\|_2^2 \\ &= \|\mathrm{R}(\mathrm{x})\|_2^2 + 2\mathrm{R}^\mathrm{T}\mathrm{J}_\mathrm{R}\Delta\mathrm{x} + \Delta\mathrm{x}^\mathrm{T}\mathrm{J}_\mathrm{R}^\mathrm{T}\mathrm{J}_\mathrm{R}\Delta\mathrm{x} \\ &= \mathrm{F}(\mathrm{x}) + \mathrm{J}_\mathrm{F}\Delta\mathrm{x} + \Delta\mathrm{x}^\mathrm{T}\mathrm{J}_\mathrm{R}^\mathrm{T}\mathrm{J}_\mathrm{R}\Delta\mathrm{x}\end{aligned}$$

将两种展开方式进行对比，可知在最小二乘问题里$\mathrm{H}_\mathrm{F} \approx 2\mathrm{J}_\mathrm{R}^\mathrm{T}\mathrm{J}_\mathrm{R}$

继续依照 NR 法的思路，每一步迭代中，我们求解如下的标准方程：

$$\mathrm{J}_\mathrm{R}^\mathrm{T}\mathrm{J}_\mathrm{R}\Delta x + \mathrm{J}_\mathrm{R}R = 0$$

这一标准方程对应于求$J_R\Delta x = -R$的（线性）最小二乘解，可以采用共轭梯度法求解。本次实验中，可以直接采用 OpenCV 提供的矩阵算法完成。

得到 Δx 后，以它作为下降方向，我们进行线性搜索求出合适的步长α，更新 x：

$$x + \alpha\Delta x \to x$$

这样就得到了 GN 法，GN 法的伪代码：

- $\mathrm{x} \leftarrow \mathrm{x}_0$
- $\mathrm{n} \leftarrow 0$
- while $\mathrm{n} < \mathrm{n}_{\max}$:
    - $\Delta\mathrm{x} \leftarrow$ Solution of $\mathrm{J}_\mathrm{R}\Delta\mathrm{x} = -\mathrm{R}$:
        - Conjugate Gradient or Other
    - if $\|\mathrm{R}\|_\infty \leq \varepsilon_\mathrm{r} \vee \|\Delta\mathrm{x}\|_\infty \leq \varepsilon_\mathrm{g}$ return x
    - $\alpha \leftarrow \arg\min_\alpha\{\mathrm{x} + \alpha\Delta\mathrm{x}\}$
    - $\mathrm{x} \leftarrow \mathrm{x} + \alpha\Delta\mathrm{x}$
    - $\mathrm{n} \leftarrow \mathrm{n} + 1$

代码实现

Optimizer

`Optimizer`的类，继承自接口类`ResidualFunction`。该类用于执行高斯牛顿法进行优化。

在`Optimizer`类中，首先实现了一个`readDataFromFile`函数，用于从文件中读取数据，并将数据存储在`x`、`y`和`z`数组中。这些数组分别代表了三维空间中的点的坐标。

接着，实现了一个`cal_value`函数，用于计算目标函数的值，并在`eval`函数中计算了残差和雅可比矩阵。在`eval`函数中，你使用了目标函数和雅可比矩阵的计算结果来填充传入的`R`和`J`数组。

```
class Optimizer : public ResidualFunction {
public :
    double *x,*y,*z;
    int dimension = 3;
    int size;

    Optimizer() {
        readDataFromFile("ellipse753.txt");
    }

    ~Optimizer() {
        delete[] x; delete[] y; delete[] z;
    }

    void readDataFromFile(const string& filename) {
        ifstream file(filename);
        string line;
        vector<vector<double>> points;

        while (getline(file, line)) {
            stringstream ss(line);
            double val;
            vector<double> point;
            while (ss >> val) {
                point.push_back(val);
            }
            if(point.size() == 3)
                points.push_back(point);
        }

        size = points.size();
        x = new double[size];
        y = new double[size];
        z = new double[size];

        for (int i = 0; i < size; i++) {
            x[i] = points[i][0];
            y[i] = points[i][1];
            z[i] = points[i][2];
        }
    }

    double cal_value(double *coefficient, int i) {
```

```
        double A = coefficient[0];
        double B = coefficient[1];
        double C = coefficient[2];
        double result = 1 - 1.0 / POW(A)*POW(x[i]) - 1.0 / POW(B)*POW(y[i]) - 1.0
/ POW(C)*POW(z[i]); // 目标函数
        return result;
    }


    virtual int nR() const {
        return size;
    }

    virtual int nX() const {
        return dimension;
    }

    virtual void eval(double *R, double *J, double *coefficient) {
        for (int i = 0; i < size; i++) {
            R[i] = cal_value(coefficient, i);
            J[i * 3 + 0] = -2 * POW(x[i])*POW_n3(coefficient[0]);
            J[i * 3 + 1] = -2 * POW(y[i])*POW_n3(coefficient[1]);
            J[i * 3 + 2] = -2 * POW(z[i])*POW_n3(coefficient[2]);
        }
    }
};
```

Solver3790

继承自接口类`GaussNewtonSolver`。在`Solver3790`类中，你实现了`solve`函数，用于执行高斯牛顿法

在该函数中，首先获取了目标函数的残差和雅可比矩阵，然后使用OpenCV的`cv::solve`函数来解线性方程组以计算参数的增量。

接着，检查了残差和参数增量的最大值是否满足收敛条件，并根据结果设置了优化报告中的相关信息。最后更新了参数并重复迭代过程，直到达到最大迭代次数或满足收敛条件

```
class Sover3790 : public GaussNewtonSolver {
    public:
        virtual double solve(
            ResidualFunction *f,
            double *X,
            GaussNewtonParams param = GaussNewtonParams(),
            GaussNewtonReport *report = nullptr
        ) override {
            double *x = X;
            int n = 0;
            double step = 1;
            int nR = f->nR();
            int nX = f->nX();
            double *J = new double[nR*nX];
            double *R = new double[nR];

            while (n < param.max_iter) {
                f->eval(R, J, x);
```

```
                Mat mat_R(nR, 1,CV_64FC1, R);
                Mat mat_J(nR, nX, CV_64FC1, J);
                Mat mat_Delta(nX, 1, CV_64FC1);
                cv::solve(mat_J, mat_R, mat_Delta, DECOMP_SVD);

                double max_R = getMaxAbs(mat_R);
                double max_mat_Delta = getMaxAbs(mat_Delta);

                if (checkConvergence(max_R, max_mat_Delta, param, report, n))
                    return 0; // Converged

                updateX(x, mat_Delta, step, nX);
                n++;
            }

            setNoConverge(report, n);
            return 1; // No Convergence
        }

    private:
        double getMaxAbs(const Mat &mat) {
            double max_val = -1;
            for (int i = 0; i < mat.rows; i++) {
                double val = abs(mat.at<double>(i, 0));
                if (val > max_val) {
                    max_val = val;
                }
            }
            return max_val;
        }

        bool checkConvergence(double max_R, double max_mat_Delta,
GaussNewtonParams &param, GaussNewtonReport *report, int n) {
            if (max_R <= param.residual_tolerance || max_mat_Delta <=
param.gradient_tolerance) {
                report->stop_type = (max_R <= param.residual_tolerance) ? report-
>STOP_RESIDUAL_TOL : report->STOP_RESIDUAL_TOL;
                report->n_iter = n;
                return true;
            }
            return false;
        }

        void updateX(double *x, const Mat &mat_Delta, double step, int nX) {
            for (int i = 0; i < nX; i++) {
                x[i] += step * mat_Delta.at<double>(i, 0);
            }
        }

        void setNoConverge(GaussNewtonReport *report, int n) {
            report->stop_type = report->STOP_NO_CONVERGE;
            report->n_iter = n;
        }
};
```

运行环境与实验结果

我的运行环境为windows10

我编写了`main.py`调用`hw3_gn.h`求解问题，并且最后输出迭代的次数，迭代停止的原因和最后迭代的结果

我把 A, B, C的初始值都定为同一值，如下表所示，发现影响并不是很大所求结果基本一直

0.5	1	2	3
Num of iteration: 8 A 2.94405 B 2.30504 C 1.79783 [Stop]: The remainder reaches the threshold	Num of iteration: 6 A 2.94404 B 2.30504 C 1.79783 [Stop]: The remainder reaches the threshold	Num of iteration: 4 A 2.94404 B 2.30504 C 1.79783 [Stop]: The remainder reaches the threshold	Num of iteration: 9 A 2.94405 B 2.30504 C 1.79783 [Stop]: The remainder reaches the threshold

我用`draw/draw_gif.py`绘制了动图展示拟合效果，保存为`plot.gif`,见于文件夹中， 可以看出结果比较准确