Lab 5 - 非线性最小二乘

学号：3210106034
姓名：王伟杰

> 实验环境：
>
> | | |
> |---|---|
> | 计算机名: | LHMD-LAPTOP |
> | 操作系统: | Windows 10 专业版 64 位 (10.0，内部版本 19045) |
> | 语言: | 中文(简体) (区域设置: 中文(简体)) |
> | 系统制造商: | ASUSTeK COMPUTER INC. |
> | 系统型号: | ROG Zephyrus G14 GA401QM_GA401QM |
> | BIOS: | GA401QM.415 |
> | 处理器: | AMD Ryzen 9 5900HS with Radeon Graphics (16 CPUs), ~3.3GHz |
> | 内存: | 32768MB RAM |
>
> ```
> Microsoft Visual Studio Professional 2017
> 版本 15.9.59
> ```

实验内容

- Gauss Newton 求解最小二乘问题

理论分析

非线性最小二乘

与线性最小二乘问题不同，非线性最小二乘问题中，函数模型与参数之间存在非线性关系。

一般来说，非线性最小二乘问题的形式如下：

给定数据集 (x_i, y_i)，$i = 1, 2, \ldots, n$，以及一个参数化的函数模型 $f(x_i, \theta)$，其中 θ 表示模型的参数。我们的目标是找到最优的参数 $\hat{\theta}$，使得残差平方和最小化，即：

$$\hat{\theta} = \arg\min_{\theta} \sum_{i=1}^{n} [y_i - f(x_i, \theta)]^2$$

这里的 $[y_i - f(x_i, \theta)]$ 即为残差，而我们的目标是使所有残差的平方和最小。

Gauss Newton 法

对于一个非线性最小二乘问题

$$x = \arg\min_{x} \frac{1}{2} \parallel f(x) \parallel^2.$$

高斯牛顿的思想是把$f(x)$利用泰勒展开，取一阶线性项近似。

$$f(x + \Delta x) = f(x) + f'(x)\Delta x = f(x) + J(x)\Delta x.$$

得到

$$\frac{1}{2} \parallel f(x + \Delta x) \parallel^2 = \frac{1}{2}\{f(x)^T f(x) + 2f(x)^T . J(x)\Delta x + \Delta x^T J(x)^T . J(x)\Delta x\}.$$

对上式求导，令导数为0

$$J(x)^T J(x)\Delta x = -J(x)^T f(x).$$

这一标准方程对应于求$\mathrm{J_R \Delta x = -R}$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。

再进行线性搜索求出合适的步长，更新x即可。

算法框架如下:

- $\mathrm{x \leftarrow x_0}$
- $\mathrm{n \leftarrow 0}$
- $\mathrm{while\ n < n_{max}}$:
  - $\mathrm{\Delta x \leftarrow Solution\ of\ J_R \Delta x = -R}$:
    - $\mathrm{Conjugate\ Gradient\ or\ Other}$
  - $\mathrm{if\ \|R\|_\infty \le \varepsilon_r \vee \|\Delta x\|_\infty \le \varepsilon_g\ return\ x}$
  - $\mathrm{\alpha \leftarrow \arg\min_\alpha \{x + \alpha \Delta x\}}$
  - $\mathrm{x \leftarrow x + \alpha \Delta x}$
  - $\mathrm{n \leftarrow n + 1}$

实验细节

ResidualFunction

继承接口类 `ResidualFunction` ，定义实现 `Ellipse` 类。

在类中我们定义一个私有成员变量 `data` ，它是一个 `cv::Mat` 类型的矩阵，用于存储椭圆模型的数据。这个数据将由 `main` 函数调用 `readData` 填入。

`nR()` 和 `nX()` 这两个方法分别返回残差向量的长度和参数向量的长度，这里它们分别代表了数据矩阵的行数和列数。

在 `eval` 方法的实现中：

- 首先，将传入的指针参数转换为OpenCV的矩阵类型。
- 然后，计算残差向量，其形式为 $1-\frac{x^2}{A^2}-\frac{y^2}{B^2}-\frac{z^2}{C^2}$，其中$x,y,z$分别表示参数向量$X$的三个元素，$A,B,C$分别表示数据矩阵 `data` 的三列。
- 接着，计算雅可比矩阵，其计算式为$\frac{2x^2}{A^3}+\frac{2y^2}{B^3}+\frac{2z^2}{C^3}$。

函数实现如下：

```cpp
int Ellipse::nR() const {
    return data.rows;
}

int Ellipse::nX() const {
    return data.cols;
}

void Ellipse::eval(double *R, double *J, double *X) {
    cv::Mat MR(nR(), 1, CV_64F, R);
    cv::Mat MX(nX(), 1, CV_64F, X);
    cv::Mat MJ(nR(), nX(), CV_64F, J);
    cv::Mat TDATA(nR(), nX(), CV_64F);

    cv::Mat TX(nR(), nX(), CV_64F);
    cv::repeat(MX.t(), nR(), 1, TX);

    TDATA = data.mul(data);
    // 1 - \frac{x^2}{A^2} - \frac{y^2}{B^2} - \frac{z^2}{C^2}
    cv::reduce(TDATA / TX.mul(TX), MR, 1, cv::REDUCE_SUM);
    MR = 1 - MR;
```

```

    // \frac{2x^2}{A^3} + \frac{2y^2}{B^3} + \frac{2z^2}{C^3}
    MJ = 2 * TDATA / TX.mul(TX).mul(TX);
}
```

GaussNewtonSolver

`Solver` 接受四个参数:

- `ResidualFunction *f` : 指向目标函数的指针。
- `double *X` : 指向参数向量的指针，作为输入初值，并在函数执行完成后保存结果。
- `GaussNewtonParams param` : Gauss-Newton优化算法的参数。
- `GaussNewtonReport *report` : 用于存储优化结果报告的指针。

在实现中:

- 首先，通过 `new` 操作符动态分配了一些用于存储中间结果的数组，包括残差向量 `R` 、雅可比矩阵 `J` 、参数更新量 `delta` ，以及一个临时的参数向量 `X_tmp` 。
- 将输入参数 `X` 转换为OpenCV的矩阵类型 `X_mat` 。
- 在一个迭代循环中，执行Gauss-Newton算法的核心步骤：计算残差和雅可比矩阵，解线性方程组以获得参数更新量，根据一定的步长策略更新参数，并根据收敛条件判断是否终止迭代。
- 在迭代结束后，根据收敛结果计算并返回优化的目标函数值。

最后，释放了动态分配的内存空间，并返回优化的目标函数值。

`solve` 函数实现如下:

```
double Solver6034::solve(
                ResidualFunction *f, // 目标函数
                double *X,           // 输入作为初值，输出作为结果
                GaussNewtonParams param = GaussNewtonParams(), // 优化参数
                GaussNewtonReport *report = nullptr // 优化结果报告
                ) {
                 double *R = new double[f->nR()];
                 double *J = new double[f->nR() * f->nX()];
                 double *delta = new double[f->nX()];
                 double *X_tmp = new double[f->nX()];
                 cv::Mat X_mat = cv::Mat(f->nX(), 1, CV_64F, X);
                 cv::Mat J_mat = cv::Mat(f->nR(), f->nX(), CV_64F, J);
                 cv::Mat R_mat = cv::Mat(f->nR(), 1, CV_64F, R);
                 cv::Mat delta_mat = cv::Mat(f->nX(), 1, CV_64F, delta);
```

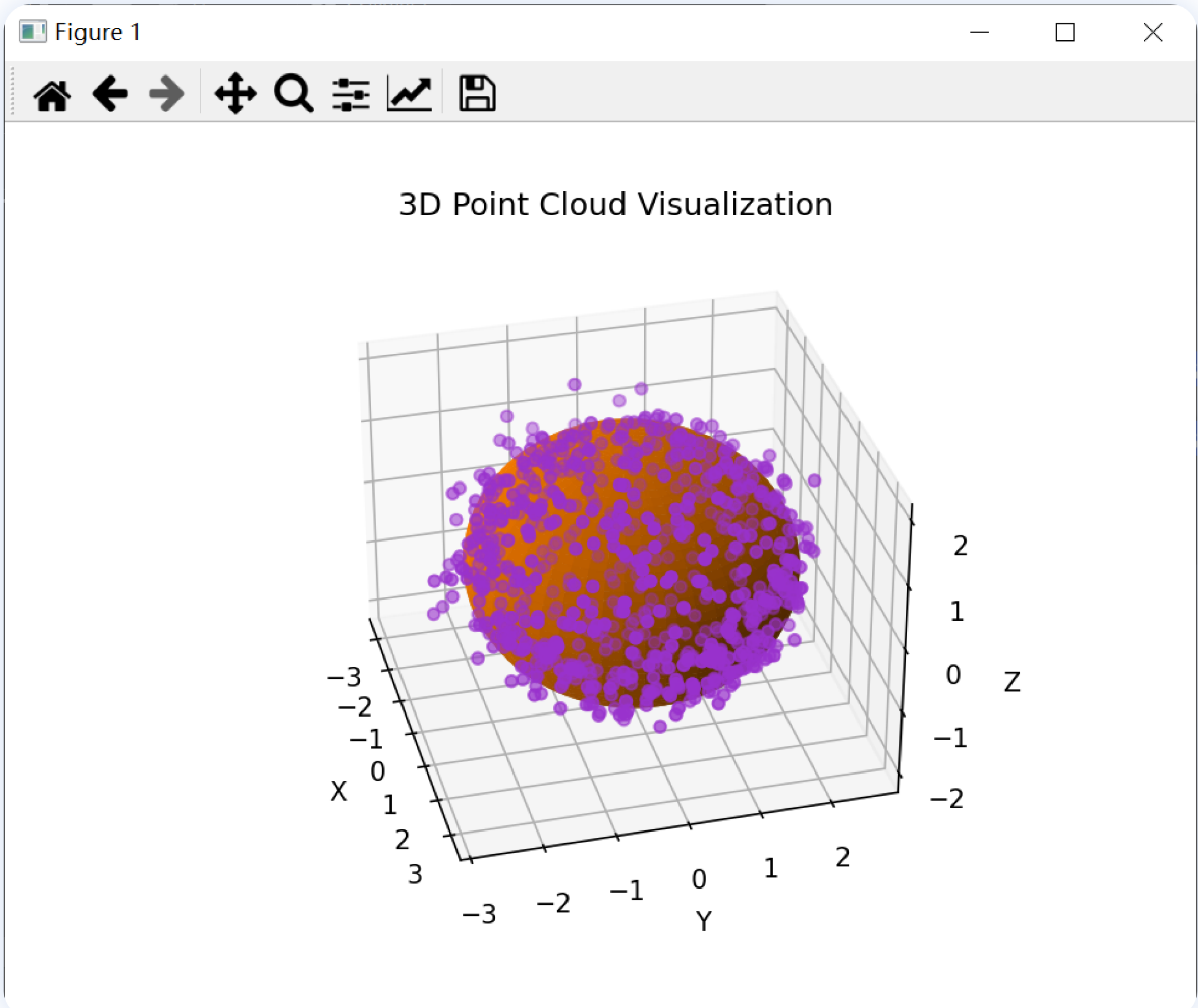
```cpp
                cv::Mat X_tmp_mat = cv::Mat(f->nX(), 1, CV_64F, X_tmp);

                for (int i = 0; i < param.max_iter; ++i) {
                    // std::cout << i << std::endl;
                    f->eval(R, J, X);
                    cv::solve(J_mat, -R_mat, delta_mat, cv::DECOMP_SVD);
                    // std::cout << delta_mat << std::endl;


                    // 无穷范数
                    double norm_delta = cv::norm(delta_mat, cv::NORM_INF);
                    double norm_R = cv::norm(R_mat, cv::NORM_INF);
                    if (norm_delta < param.gradient_tolerance) {
                        report->stop_type = GaussNewtonReport::STOP_GRAD_TOL;
                        report->n_iter = i;
                        break;
                    } else if (norm_R < param.residual_tolerance) {
                        report->stop_type = GaussNewtonReport::STOP_RESIDUAL_TOL;
                        report->n_iter = i;
                        break;
                    } else {
                        double alpha = 1;
                        double beta = 0.1;
                        // 计算最优的alpha使得x = x + alpha * delta
                        // 使得目标函数的值下降
                        double phi0_diff = cv::sum(2 * R_mat.t() * J_mat * delta_mat).val[0] * beta;
                        double phi0 = cv::norm(R_mat, cv::NORM_L2SQR);
                        while (true) {
                            X_tmp_mat = X_mat + alpha * delta_mat;
                            f->eval(R, J, X_tmp);
                            double phi1 = cv::norm(R_mat, cv::NORM_L2SQR);

                            // std::cout << phi1 - phi0 - phi0_diff << std::endl;
                            if (phi1 < phi0 + phi0_diff * alpha) {
                                break;
                            } else {
                                alpha *= beta;
                            }
                        }
                        X_mat = X_mat + alpha * delta_mat;
```

```cpp

            if (param.verbose) {
                std::cout << "iter: " << i << ", norm_delta: " << norm_delta << ", norm_R: " << norm_R << std::endl;
            }
        }

        if (i == param.max_iter - 1) {
            report->stop_type = GaussNewtonReport::STOP_NO_CONVERGE;
            report->n_iter = i;
        }
    }

    double result = cv::norm(R_mat, cv::NORM_L2SQR);
    delete[] R;
    delete[] J;
    delete[] delta;
    return result;

}
```

结果展示

```
❯ .\GaussNewton.exe
X = [2.94404, 2.30504, 1.79783]
```

将得到的参数和初始点云数据可视化：



可以看到椭球和初始点云基本吻合，我们再计算一下有多少点在椭球内部：

```
❯ python main.py
Number of points inside the ellipsoid: 442
Total number of points: 753
Percentage of points inside the ellipsoid: 58.69853917662683 %
```

可以看到有大约58.7%的点在内部，符合常识。

附录：可视化代码

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

```python

point_cloud = np.loadtxt('ellipse753.txt')

x = point_cloud[:, 0]
y = point_cloud[:, 1]
z = point_cloud[:, 2]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, color='#9932CD')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Point Cloud Visualization')

# 椭球参数
a = 2.94405
b = 2.30504
c = 1.79783

# 计算给出点云中有多少点在椭球内
count = 0
for i in range(len(x)):
    if (x[i] ** 2 / a ** 2 + y[i] ** 2 / b ** 2 + z[i] ** 2 / c ** 2) <= 1:
        count += 1

print('Number of points inside the ellipsoid:', count)
print('Total number of points:', len(x))
print('Percentage of points inside the ellipsoid:', count / len(x) * 100, '%')

u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

x = a * np.outer(np.cos(u), np.sin(v))
y = b * np.outer(np.sin(u), np.sin(v))
z = c * np.outer(np.ones(np.size(u)), np.cos(v))

ax.plot_surface(x, y, z, color='#FF7F00', alpha=1)

ax.set_xlabel('X')
ax.set_ylabel('Y')
```

```
ax.set_zlabel('Z')

plt.show()
```