计算摄影学作业报告
Lab 4 —— 非线性最小二乘

Jessie Peng

2019/04/8

1 实验内容

本实验需要实现 Gauss Newton 求解最小二乘问题。

1.1 接口要求

本实验需要实现继承自接口类 GaussNewtonSolver 的优化器类，接口定义如下：

```
class GaussNewtonSolver {
public:
    virtual double 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
                ) = 0;
};
```

还需要实现继承自接口类 ResidualFunction 的目标函数类，接口定义如下：

```
class ResidualFunction {
public:
    virtual int nR() const = 0;
    virtual int nX() const = 0;
    virtual void eval(double *R, double *J, double *X) = 0;
};
```

优化参数结构体定义如下：

```
struct GaussNewtonParams{
    GaussNewtonParams() :
       exact_line_search(false),
       gradient_tolerance(1e-5),
       residual_tolerance(1e-5),
       max_iter(1000),
       verbose(false)
    {}
    bool exact_line_search; // 使用精确线性搜索还是近似线性搜索
    double gradient_tolerance; // 梯度阈值，当前梯度小于这个阈值时停止迭代
    double residual_tolerance; // 余项阈值，当前余项小于这个阈值时停止迭代
    int max_iter; // 最大迭代步数
    bool verbose; // 是否打印每步迭代的信息
};
```

优化结果报告结构体定义如下：

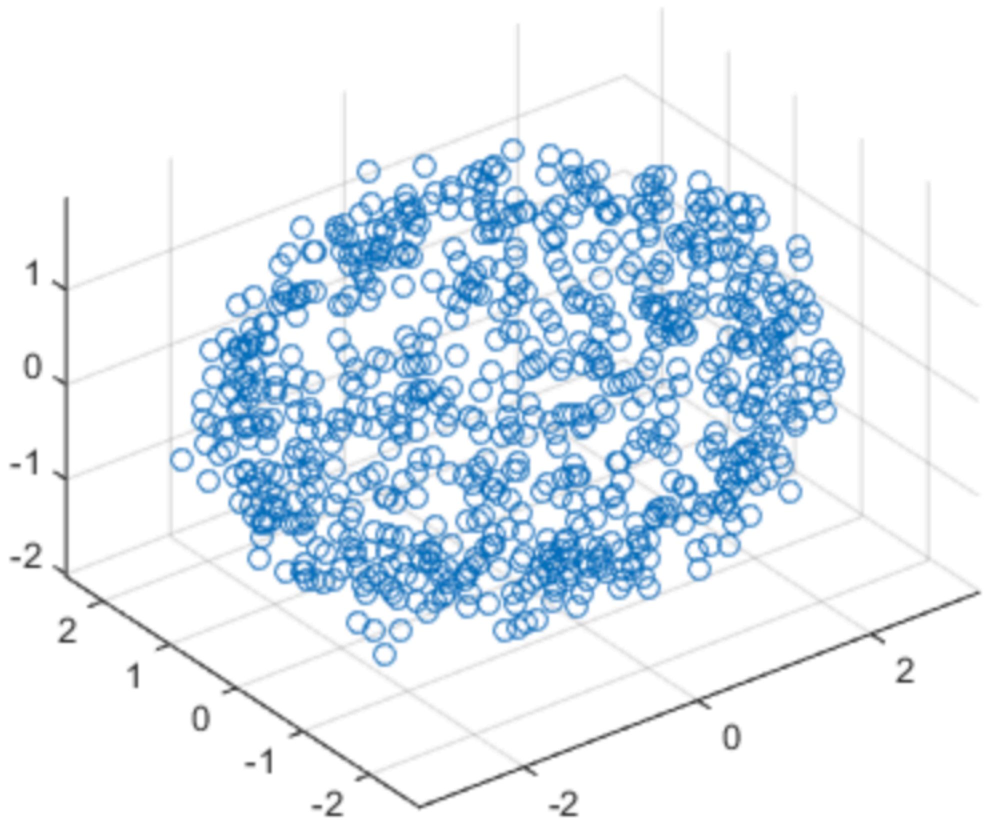
```
struct GaussNewtonReport {
    enum StopType {
        STOP_GRAD_TOL,        // 梯度达到阈值
        STOP_RESIDUAL_TOL,    // 余项达到阈值
        STOP_NO_CONVERGE,     // 不收敛
        STOP_NUMERIC_FAILURE  // 其它数值错误
    };
    StopType stop_type; // 优化终止的原因
    double n_iter;      // 迭代次数
};
```

1.2 测试要求

作为测试，我们求解从三维点云拟合椭球的问题，目标函数如下：

$$\frac{x^2}{A^2}+\frac{y^2}{B^2}+\frac{z^2}{C^2}=1$$

输入的点云包含 753 个含有噪音的点，需要通过这些点恢复出参数 A^2、B^2、C^2，点云图示如下：



2 实验环境

编程语言：C++

开发环境：CLion 2018.3
操作系统：macOS 10.14.1 (18B75)

3 实验原理

3.1 Newton Rapson 法

Gauss Newton 法来源于 Newton Rapson 法，后者利用二次型逼近目标函数 。采用二阶泰勒展开来近似目标函数：

$$F(x+\Delta x) \approx F(x) + J_F \Delta x + \frac{1}{2}\Delta x^T H_F \Delta x$$

这样的二次逼近，需要计算目标函数的 Hessian 矩阵，但是这并不容易。

3.2 Gauss Newton 法

针对最小二乘问题本身的特点，即具有如下形式：

$$F(x) = \|R(x)\|_2^2$$

Gauss Newton 法可以通过计算 Jacobian 矩阵来近似 Hessian 矩阵。

推导过程如下，对 R 进行一阶泰勒展开：

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

对比发现，在最小二乘问题里有：

$$H_F \approx 2 J_R^T J_R$$

于是就可以这样来近似计算 Hessian 矩阵。

依照 NR 法的思路，在每一步迭代中需要求解如下的标准方程：

$$J_R^T J_R \Delta x + J_R^T R = 0$$

其对应于求如下方程的线性最小二乘解：

$$J_R \Delta x = -R$$

得到梯度方向作为下降方向，再进行线性搜索求得下降的步长，然后进行迭代更新。

3.3 GN 法伪代码

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

4 实现过程与代码分析

4.1 优化器类 Solverxxxx

优化器类继承自接口类，声明如下：

```
class Solverxxxx: public GaussNewtonSolver {
public:
    double solve(
            ResidualFunction *f,
            double *X,
            GaussNewtonParams param,
            GaussNewtonReport *report
            ) override;
};
```

下面分析 solve 函数的实现：

首先是为所需的矩阵（J、R、deltaX）分配好空间：

```
int nX = f->nX();
int nR = f->nR();
int nJ = nR * nX;
double R[nR];
double J[nJ];
Mat_<double> deltaX(nX, 1);
Mat_<double> mJ(nR, nX, J);
Mat_<double> mR(nR, 1, R);
```

然后进入迭代循环：

```
int n = 0;
double step = 0;
```

```
while (n < param.max_iter)
{
    n++;

    ...
}
```

在每次循环中，先计算 R、J 矩阵：

```
// calculate R, J
f->eval(R, J, X);
```

然后使用 OpenCV 提供的 solve 函数用最小二乘求解 deltaX：

```
// solve deltaX
Mat_<double> mJT(mJ.t());
if (!cv::solve(mJT * mJ, mJT * (-mR), deltaX, CV_SVD)) // JT * J * deltaX = JT * (-
R)
{
    if (report != nullptr) // ERROR solving deltaX, return
    {
        report->n_iter = n;
        report->stop_type = report->STOP_NUMERIC_FAILURE;
    }
    return norm(mR, NORM_L2);
}
```

求解完成后，判断是否达到了（梯度足够小或者残差足够小）条件可以终止迭代：

```
// decide if stop iteration
if (norm(mR, NORM_INF) <= param.residual_tolerance)
{
    if (report != nullptr) // reach residual tolerance, return
    {
        report->n_iter = n;
        report->stop_type = report->STOP_RESIDUAL_TOL;
    }
    return norm(mR, NORM_L2);
}
if (norm(deltaX, NORM_INF) <= param.gradient_tolerance)
{
    if (report != nullptr) // reach grad tolerance, return
    {
        report->n_iter = n;
        report->stop_type = report->STOP_GRAD_TOL;
    }
    return norm(mR, NORM_L2);
}
```

如果没有终止迭代，则线性搜索决定步长，然后更新 X 的值（这里为了简便，并未实现步长的搜索算法，只是简单地规定一个固定的步长）：

```
// update X
for (int i = 0; i < nX; i++)
{
    X[i] += step * deltaX(i, 0);
}
if (param.verbose)
{
    cout << "X = ";
```

```
    for (int i = 0; i < nX; i++)
    {
        cout << X[i] << " ";
    }
    cout << endl;
}
```

更新完后则一次迭代结束。所有循环结束后，生成结果报告并返回 R 矩阵的二范数：

```
if (report != nullptr)
{
    report->n_iter = n;
    report->stop_type = report->STOP_NO_CONVERGE;
}
return norm(mR, NORM_L2);
```

4.2 目标函数类 ResidualFunctionxxxx

目标函数类继承自接口类，但增添了一个二维数组成员变量用以储存输入的点云数据，其声明如下：

```
// x^2 / a^2 + y^2 / b^2 + z^2 / c^2 = 1
class ResidualFunctionxxxx: public ResidualFunction {
public:
    int nR() const override;
    int nX() const override;
    void eval(double *R, double *J, double *X) override;
    ResidualFunctionxxxx();

private:
    double testData[TEST_DATA_NUM][VARIABLE_NUM];
};
```

nR()返回 R 的大小，即数据量：

```
int ResidualFunctionxxxx::nR() const
{
    return TEST_DATA_NUM;
}
```

nX()返回 X 的大小，即变量数：

```
int ResidualFunctionxxxx::nX() const
{
    return VARIABLE_NUM; // a^2, b^2, c^2
}
```

eval()函数用于计算 R、J 两个矩阵（这里把 X 当作$[A^2, B^2, C^2]^T$）：

```
void ResidualFunctionxxxx::eval(double *R, double *J, double *X)
{
    // x^2 / a^2 + y^2 / b^2 + z^2 / c^2 = 1
    for (int i = 0; i < TEST_DATA_NUM; i++)
    {
        R[i] = -1;
        for (int j = 0; j < VARIABLE_NUM; j++)
        {
            R[i] += testData[i][j] * testData[i][j] / X[j];
        }
```

```
    }
    for (int i = 0; i < TEST_DATA_NUM; i++)
    {
        for (int j = 0; j < VARIABLE_NUM; j++)
        {
            J[i*VARIABLE_NUM+j] = (-1) * testData[i][j] * testData[i][j] / X[j] /
X[j];
        }
    }
}
```

最后，在构造函数中，需要从给定的文件 ellipse753.txt 中读取点云数据，并存进数组中：

```
ResidualFunctionxxxx::ResidualFunctionxxxx()
{
    fstream file("ellipse753.txt", ios::in);
    if (!file.is_open())
    {
        cout << "Error opening file" << endl;
        exit(1);
    }
    for (auto &i : testData)
    {
        for (double &j : i)
        {
            if (!(file >> j))
            {
                cout << "Error reading file" << endl;
                exit(1);
            }
        }
    }
}
```

5 结果分析与实验总结

5.1 GN 法求解测试问题

用如下代码求解测试问题：

```
    double X[3] = {4, 4, 1};
    ResidualFunctionxxxx myfunc;
    Solverxxxx mysolver;
    GaussNewtonParams param;
//    param.verbose = true;
    GaussNewtonReport report;

    double res = mysolver.solve(&myfunc, X, param, &report);

    cout << "res = " << res << endl;
    cout << "X = (" << X[0] << ", " << X[1] << ", " << X[2] << ")" << endl;
    cout << report.n_iter << " iterations" << endl;
    cout << "stop type = " << report.stop_type << endl;
```

结果如下：

```
res = 4.76812
X = (8.66741, 5.31319, 3.23218)
116 iterations
stop type = 0
```

恢复出的参数 $(A^2, B^2, C^2) = (8.66741, 5.31319, 3.23218)$

5.2 线性最小二乘求解验证

测试问题中如果把待求解的参数当作 $(1/A^2, 1/B^2, 1/C^2)$，那么是可以用线性最小二乘来求解的，因此可以线性形式再求解一次（直接使用 OpenCV 提供的 solve 函数）用来验证上述答案。

编写验证程序如下：

```
/* linear least square solution, for verification */
void verify(double *X)
{
    // read data
    double testData[TEST_DATA_NUM][VARIABLE_NUM];
    fstream file("ellipse753.txt", ios::in);
    if (!file.is_open())
    {
        cout << "Error opening file" << endl;
        exit(1);
    }
    for (auto &i : testData)
    {
        for (double &j : i)
        {
            if (!(file >> j))
            {
                cout << "Error reading file" << endl;
                exit(1);
            }
            j *= j; // squared
        }
    }

    // solve linear equations for X = (1/a^2, 1/b^2, 1/c^2)
    Mat_<double> mX(VARIABLE_NUM, 1, X);
    Mat_<double> mA(TEST_DATA_NUM, VARIABLE_NUM, (double*)testData);
    cout << mA(1, 2) << endl;
    Mat_<double> mb = Mat_<double>::ones(TEST_DATA_NUM, 1);
    if (!cv::solve(mA, mb, mX, CV_SVD)) // AX = b
    {
        cout << "Error solving AX=b" << endl;
        exit(1);
    }
}
```

结果如下：

```
(verify)1/X = (0.115375, 0.188211, 0.309389)
(verify)X = (8.66741, 5.31319, 3.23218)
```

恢复出的参数 $(1/A^2, 1/B^2, 1/C^2) = (0.115375, 0.188211, 3.23218)$
即 $(A^2, B^2, C^2) = (8.66741, 5.31319, 3.23218)$，与上述答案相符合，验证成功。

5.3 问题与解决

本次作业遇到的问题还是与 Mat 的内存有关。由于要求实现的接口中矩阵参数是以数组的形式存在的，而在函数实现中需要用到 OpenCV 的一些运算和函数，因此要转换为 Mat 的形式。我在一开始使用数组初始化 Mat 对象的方式进行数组到 Mat 的转换，比如 Mat_<double> mJ(nR, nX, J)通过数组 J 来构造 Mat mJ，但却没有考虑到这样的拷贝方式其实是浅拷贝，与用一个 Mat 拷贝构造另一个 Mat 的情形类似，所创建出来的 Mat 与原来的数组是共用一块内存的，刚开始忽略了这个问题造成了一些 bug。后来找到问题的根源后，发现其实这样使代码更加简洁，函数内部对 Mat 进行各种运算都能自动反映到原来的数组中，不用再手动转换一次类型，其实更方便了。

6 参考文献

课程网站：http://www.cad.zju.edu.cn/home/gfzhang/course/computational-photography/lab5-gauss-newton/gauss-newton.html
OpenCV 的 solve 函数解析：https://blog.csdn.net/u014652390/article/details/52789591