Lab03_Report

申长硕 大数据学院 PB22020518

问题引入

1. 二次型在数学很多分支中都频繁出现，而且在优化、概率图论、统计、机器学习、信号处理等领域随处可见。二次型的机制问题通常转化为最大最小特征值问题进行求解。
2. 针对计算图形学的四面体网格生成领域，一个常见问题是四面体中向量场的生成，可以在每个四面体中定义一组四元数$q_i = (q^w, q^x, q^y, q^z)^T$来表示四面体上的场，我么希望让向量场光滑，所以目标优化：

$$q_{otp} = min_q \Sigma_{i,j} w_{ij} ||q_i - q_j||^2 \ ||q_i|| = 1$$

   a. 其中$w_{ij} > 0$为权重，将能量放松到全局单位约束$||q|| = 1$，将问题转化为：

$$q_{opt} = min_{||q||=1} \Sigma_{i,j} w_{ij} ||q_i - q_j||^2$$

   b. 该问题能够被作为特征值问题进行求解

$$L\ q_{opt} = \lambda_0\ q_{opt}$$

   c. 其中λ_0是矩阵L的最小特征值，L关于w_{ij}的系数矩阵，可以通过反幂法近进行求解

3. 实验要求实现带有规范方法的反幂法，求得给定矩阵的按模最小特征值

数学分析

LU分解–Doolittle分解

1. Doolittle分解是LU分解的一种，它将一个矩阵 A 分解为一个单位下三角矩阵 L 和一个上三角矩阵 U 的乘积。给定矩阵 (A) 和分解矩阵 (L)、(U)，其中：

$$A = LU \tag{1.1}$$

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix} \tag{1.2}$$

$$\begin{bmatrix} \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \qquad \begin{bmatrix} \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

2. 那么 L 和 U的元素 l_{ij} 和 u_{ij} 可以通过下面的公式得到：

$$l_{ij} = \begin{cases} a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & \text{if } i \leq j \\ 0, & \text{if } i > j \end{cases} \tag{1.3}$$

$$u_{ij} = \begin{cases} \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), & \text{if } i \leq j \\ 0, & \text{if } i > j \end{cases} \tag{1.4}$$

3. Doolittle分解是一种经典的LU分解方法之一，在数值计算中被广泛应用。

反幂法Inverse-Power-Method

反幂法是一种用于求解特征值问题的迭代数值方法。它通常用于找到矩阵 A 的按模最小特征值。该方法的基本思想是通过迭代来寻找 A 的逆矩阵的最大特征值，迭代过程中使用规范化技术将结果无穷范数规范为1。

1. 过程：
    a. **初始化**：选择一个初始向量 $x^{(0)}$ 和一个容许误差 ϵ。
    b. **求解线性方程**：计算 $Ax^{(k)} = x^{(k-1)}$ 的解，其中A 是待求特征向量的矩阵，$x^{(k-1)}$是上一步得到的向量。
    c. **规范化**：使用无穷范数将得到的新向量 $x^{(k)}$ 规范化为$y^{(k)}$。
    d. **检查收敛**：计算计算前后向量的比值$\lambda_{k+1} = \frac{||x^{(k+1)}||}{||y^{(k)}||}$是否满足 $|\lambda_{k+1} - \lambda_k| < \epsilon$，如果满足则停止迭代，否则返回步骤b。
2. 原理：
    a. 反幂法的核心原理是基于特征值分解的特性。对于矩阵 A，如果 λ 是其特征值，x 是对应的特征向量，则满足 $Ax = \lambda x$。换言之，特征向量 x 是矩阵 A 的一个不动点，即x在 Ax 下的缩放因子是特征值 λ。因此，通过求解 $Ax = x'$ 的线性方程来找到特征向量，我们可以得到 x' 在特征向量x下的缩放因子，即$\frac{1}{\lambda'}$。

算法设计及代码实现

本实验主要实现的函数有：

- `void LU_decomposition(const Matrix &A, Matrix &L, Matrix &U)`实现矩阵的LU分解
- `Vector solveEquation(const Matrix &L, const Matrix &U, const Vector &b)`计算$LUx = b$的方程，防止为防止相同的矩阵多次分解所以直接使用的就是L,U矩阵
- `Vector inversePowerMethod(const Matrix &A, const Vector &initialGuess, double tolerance)`反幂法具体实现，通过不断求解$Ax^{(k+1)} = x^{(k)}$求解，通过两次的λ判断是否已经收敛

1. 主函数

```
typedef std::vector<std::vector<double>> Matrix;
typedef std::vector<double> Vector;


int main() {
    Matrix A = {{1.0/9, 1.0/8, 1.0/7, 1.0/6, 1.0/5},
                {1.0/8, 1.0/7, 1.0/6, 1.0/5, 1.0/4},
                {1.0/7, 1.0/6, 1.0/5, 1.0/4, 1.0/3},
                {1.0/6, 1.0/5, 1.0/4, 1.0/3, 1.0/2},
                {1.0/5, 1.0/4, 1.0/3, 1.0/2, 1.0/1}};

    printMatrix(A);

    std::cout<<std::endl;
    Matrix B = {{4.0, -1.0, 1.0, 3.0},
                {16.0, -2.0, -2.0, 5.0},
                {16.0, -3.0, -1.0, 7.0},
                {6.0, -4.0, 2.0, 9.0}};
    printMatrix(B);
    std::cout<<std::endl;
    Vector initialVector_A = {1, 1, 1, 1, 1};
    Vector initialVector_B = {1, 1, 1, 1};
    // 允许误差
    double tolerance = 1e-6;

    Vector eigenVector_A = inversePowerMethod(A, initialVector_A,
tolerance);
    Vector eigenVector_B = inversePowerMethod(B, initialVector_B,
tolerance);

    std::cout<<"A'a eigen vector: ";
```

```
    printVector(eigenVector_A);
    std::cout<<"B's eigen vector: ";
    printVector(eigenVector_B);
    return 0;
}
```

2. LU分解函数

```
// 执行 LU 分解
void LU_decomposition(const Matrix &A, Matrix &L, Matrix &U) {
    int n = A.size();
    L = Matrix(n, Vector(n, 0));
    U = Matrix(n, Vector(n, 0));

    // 先初始化L，对角元都是1
    for (int i = 0; i < n; ++i)
    {
        L[i][i] = 1.0;
    }

    // Perform LU decomposition
    for (int i = 0; i < n; ++i)
    {
        // Compute U matrix
        for (int k = i; k < n; ++k)
        {
            double sum = 0.0;
            for (int j = 0; j < i; ++j)
                sum += L[i][j] * U[j][k];
            U[i][k] = A[i][k] - sum;
        }

        // Compute L matrix
        for (int k = i + 1; k < n; ++k)
        {
            double sum = 0.0;
            for (int j = 0; j < i; ++j)
                sum += L[k][j] * U[j][i];
            L[k][i] = (A[k][i] - sum) / U[i][i];
        }
    }
    std::cout<<"LU decomposition finished"<<std::endl;
```

```
    std::cout<<"L: "<<std::endl;
    printMatrix(L);
    std::cout<<"U: "<<std::endl;
    printMatrix(U);
}
```

1. 通过LU解方程

```
// 使用分解好的LU计算Ax = b，防止没必要的多次分解
Vector solveEquation(const Matrix &L, const Matrix &U, const Vector
&b) {
    int n = L.size();

    // Solve Ly = b
    Vector y(n);
    for (int i = 0; i < n; ++i)
    {
        double sum = 0.0;
        for (int j = 0; j < i; ++j)
            sum += L[i][j] * y[j];
        y[i] = b[i] - sum;
    }

    // Solve Ux = y
    Vector x(n);
    for (int i = n - 1; i >= 0; --i)
    {
        double sum = 0.0;
        for (int j = i + 1; j < n; ++j)
            sum += U[i][j] * x[j];
        x[i] = (y[i] - sum) / U[i][i];
    }

    return x;
}
```

1. 反幂法

```
// 反幂法
Vector inversePowerMethod(const Matrix &A, const Vector
&initialGuess, double tolerance) {
```

```
    assert(A.size() == A[0].size() && "Matrix A must be square");
    assert(A.size() == initialGuess.size() && "Initial guess vector
must have the same size as A");
    int iteration_cnt = 0;
    Vector eigenVector = initialGuess;
    double lambda = 0.0;
    double prevLambda = 0.0;
    Matrix L, U;
    LU_decomposition(A, L, U);
    do
    {
        prevLambda = lambda;

        // 计算Ax_{k+1} = x_k，也就是LUx = x
        Vector nextEigenVector = solveEquation(L, U, eigenVector);
        std::cout<<"next eigen vector:\t";
        printVector(nextEigenVector);
        // 使用无穷范数规范化
        int max_index = max_elem_index(nextEigenVector);
        // 规范化之前先记录按模最大的元素，在迭代后期就是特征值lambda
        lambda = nextEigenVector[max_index];
        // double norm = std::abs(nextEigenVector[max_index]);
        for (double &element : nextEigenVector)
        {
            element /= lambda;
        }
        std::cout<<"normalized vector:";
        printVector(nextEigenVector);
        std::cout<<"Lambda - prevLambda:\t"<<lambda -
prevLambda<<std::endl;
        std::cout<<"lambda this loop(before processed): "
<<lambda<<"iteration times so far: "<<++iteration_cnt<<std::endl;

        eigenVector = nextEigenVector;

    } while (std::abs(lambda - prevLambda) > tolerance &&
iteration_cnt <= 1000);

    lambda = 1.0 / lambda;

    std::cout<<"Real Lambda: "<<lambda<<std::endl<<"eigenVector:";
    printVector(eigenVector);
```

```
    return eigenVector;
}
```

分析与思考

结果

在代码所在目录输入：

```
g++ -g ./run.cpp -o test && ./test
```

得到结果如下：

注：此处是通过规范话之后，λ直接就是取按模最大值元素的值即可，最后返回的时候需要取倒数

1. 对于矩阵A：
    a. 每次迭代的向量，以及此时的λ

ITERATION	$X^{(k)}$	NORMALIZED	λ	$\lambda^{(k)} - \lambda^{(k-1)}$
1	(630, -1120, 630, -120, 5)	(-0.5625, 1, -0.5625, 0.1071428571, -0.004464285714)	-1120	-1120
2	(-146252.8125, 297848.75, -196174.6875, 45114.375, -2377.254464)	(-0.4910304727, 1, -0.6586386127, 0.1514673975, -0.007981414944)	297848.75	298968.75
3	(-149112.7707, 304047.4062, -200595.2752, 46244.69167, -2446.559496)	(-0.4904260574, 1, -0.6597499966, 0.1520969781, -0.008046638274)	304047.4062	6198.656171
4	(-149157.1052, 304141.8099, -200661.194, 46261.12275, -2447.536172)	(-0.4904196016, 1, -0.6597619515, 0.1521037925, -0.0080473519)	304141.8099	94.40375598

ITERATION	$X^{(k)}$	NORMALIZED	λ	$\lambda^{(k)} - \lambda^{(k-1)}$
5	(-149157.5847, 304142.8306, -200661.9065, 46261.30027, -2447.54672)	(-0.4904195322, 1, -0.65976208, 0.1521038657, -0.008047359574)	304142.8306	1.020658774
6	(-149157.5898, 304142.8416, -200661.9142, 46261.30218, -2447.546833)	(-0.4904195314, 1, -0.6597620813, 0.1521038665, -0.008047359656)	304142.8416	0.01097213337
7	(-149157.5899, 304142.8417, -200661.9143, 46261.3022, -2447.546834)	(-0.4904195314, 1, -0.6597620814, 0.1521038665, -0.008047359657)	304142.8417	0.0001179340179
8	(-149157.5899, 304142.8417, -200661.9143, 46261.3022, -2447.546834)	(-0.4904195314, 1, -0.6597620814, 0.1521038665, -0.008047359657)	304142.8417	1.267646439e-06
9	(-149157.5899, 304142.8417, -200661.9143, 46261.3022, -2447.546834)	(-0.4904195314, 1, -0.6597620814, 0.1521038665, -0.008047359657)	304142.8417	1.356238499e-08

b. 最终:

λ	EIGENVECTOR
3.287928772e-06	(-0.4904195314 1 -0.6597620814 0.1521038665 -0.008047359657)

2. 对于矩阵B：

a.

ITERATION	$X^{(k)}$	NORMALIZED	λ	$\lambda^{(k)} - \lambda^{(k-1)}$
1	(0, 2, 0, 1)	(0, 1, 0, 0.5)	2	2

ITERATION	$X^{(k)}$	NORMALIZED	λ	$\lambda^{(k)} - \lambda^{(k-1)}$
2	(-0.625, 5.625, -2.375, 3.5)	(-0.1111111111, 1, -0.4222222222, 0.6222222222)	3.625	3.625
3	(-0.9333333333, 8.077777778, -3.433333333, 5.044444444)	(-0.1155433287, 1, -0.4250343879, 0.6244841816)	2.452777778	2.452777778
4	(-0.9362104539, 8.089924347, -3.443775791, 5.054332875)	(-0.1157254894, 1, -0.4256870205, 0.6247688678)	8.089924347	0.01214656885
5	(-0.9367122909, 8.09381841, -3.44549, 5.056810695)	(-0.1157318145, 1, -0.4256940082, 0.624774419)	8.09381841	0.003894063181
6	(-0.9367188367, 8.093856123, -3.445512966, 5.05683754)	(-0.115732084, 1, -0.425694862, 0.6247748247)	8.093856123	3.771335444e-05
7	(-0.9367195054, 8.093861209, -3.445515253, 5.0568408)	(-0.1157320939, 1, -0.4256948771, 0.6247748348)	8.093861209	5.085662821e-06
8	(-0.9367195187, 8.093861294, -3.445515299, 5.056840858)	(-0.1157320943, 1, -0.4256948783, 0.6247748354)	8.093861294	8.527408113e-08

b. 最终：

λ	EIGENVECTOR				
0.1235504247	(-0.1157320943	1	-0.4256948783	0.6247748354	)

分析

1. 关于按模最小特征值约接近于0,收敛速度快，这不是绝对的，特别是误差取的是绝对误差而不是相对误差的时候，关于这个问题，本人最初标准输入输出流中由于整数部分比较大返回的小数部分很少甚至没有，导致本人观察A矩阵情况下后面几次迭代好像是没有意义的，因为返回的一直都是一样的，直到使用了`std::setprecision`函数返回小数点后10位，才发现，其实是因为绝对误差还没有满足收敛条件，所以这不是一个绝对的规律
2. 关于λ的确定，直接用两个向量做比是不可行的，或者用模长做比值是效率低的，其实只要正常进行了normalization部分，那么经过计算后的向量的按模最大的元素其实就是λ

反思

主要是编成过程中遇到的一些问题和解决措施：

1. 对于一个整数部分很大的double型数字，最好使用`setprecision`函数多返回小数点后几位，方便观察
2. 在规范化阶段，本人最开始直接使用norm进行规范化，但这有一个问题，就是按模最大的元素是一个负数的时候其实规范化之后会得到-1,那么计算之后的按模最大的元素就不一定是想要的λ而有可能是$-\lambda$，所以，规范化的时候不应该直接使用inf_norm而是应该使用按模最大的元素进行规范化
3. 截至本次实验完成，还没有花时间去了解本次实验的计算机图形学背景，希望之后有时间可以了解一下相关的知识