1 基础算法

1.1 反幂法

设矩阵 A 有按摸最小特征值 λ，则有：

$$\boldsymbol{AX} = \lambda \boldsymbol{X}$$
$$\Longrightarrow \boldsymbol{A}^{-1}\boldsymbol{X} = \frac{1}{\lambda}\boldsymbol{X}$$

若记 $\mu = \dfrac{1}{\lambda}$，则 μ 为 $\boldsymbol{A}^{-1}$ 的按摸最大特征值。以此类推，若要计算矩阵 A 的按摸最小特征值，只需使用幂法计算其逆矩阵的按摸最大特征值：

$$\begin{cases} \boldsymbol{Y}^{(k)} = \dfrac{\boldsymbol{X}^{(k)}}{\|X\|_\infty} \\ \boldsymbol{X}^{(k+1)} = \boldsymbol{A^{-1}Y}^{(k)} \end{cases}$$

如此规范化迭代计算，会由于矩阵求逆运算而在实际计算中并不稳定，因此采用下面的变换：

$$\begin{cases} \boldsymbol{Y}^{(k)} = \dfrac{\boldsymbol{X}^{(k)}}{\|X\|_\infty} \\ \boldsymbol{AX}^{(k+1)} = \boldsymbol{Y}^{(k)} \end{cases}$$

通过求解方程组 $\boldsymbol{AX}^{(k+1} = \boldsymbol{Y}^{(k)}$ 来代替矩阵求逆并迭代。

1.2 Doolittle 分解

将矩阵 $\boldsymbol{A}$ 分解成下三角矩阵 $\boldsymbol{L}$ 和上三角矩阵 $\boldsymbol{U}$ 的乘积：

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

然后分别解方程组：

$$\begin{cases} \boldsymbol{LM} = \boldsymbol{Y}^{(k)} \\ \boldsymbol{UX}^{(k+1)} = \boldsymbol{M} \end{cases}$$

迭代求解。

2 代码实现

2.1 数据结构

```
typedef struct Matrix{
    int Dimension;
    double **OMat;
    double **LMat;
    double **UMat;
}matrix;
```

其中 Dimension 为矩阵维度、OMat 为矩阵的元素数组、LMat 为矩阵做 Doolittle 分解后的下三角矩阵 $\boldsymbol{L}$，UMat 为矩阵做 Doolittle 分解后的上三角矩阵 $\boldsymbol{U}$，如此设计数据结构使得算法具有普适性，能够对不同维度的矩阵做处理。

2.2 Doollittle 分解

```
void Doolittle(matrix Mat){
    for (int k = 0; k < Mat.Dimension; k++){
        Mat.LMat[k][k] = 1; // 对角线上的元素为1
        // 计算 U 矩阵的第 k 行
        for (int j = k; j < Mat.Dimension; j++){
            Mat.UMat[k][j] = Mat.OMat[k][j];
            for (int p = 0; p < k; p++) {
                Mat.UMat[k][j] -= Mat.LMat[k][p] * Mat.UMat[p][j];
            }
        }
        // 计算 L 矩阵的第 k 列
        for (int i = k + 1; i < Mat.Dimension; i++){
            Mat.LMat[i][k] = Mat.OMat[i][k];
            for (int p = 0; p < k; p++) {
                Mat.LMat[i][k] -= Mat.LMat[i][p] * Mat.UMat[p][k];
            }
            Mat.LMat[i][k] /= Mat.UMat[k][k];
        }
    }
}
```

以上算法将对矩阵 Mat 做 Doolittle 分解，以便接下来的求解。

2.3 利用 Doolittle 分解的结果求解方程组

```
void SolveFunction(matrix Mat, double *Bias, double *Solution){
    double *y = (double *)malloc(Mat.Dimension * sizeof(double));
    // 前代法求解 Ly = b
    for (int i = 0; i < Mat.Dimension; ++i) {
        double sum = 0;
        for (int j = 0; j < i; ++j) {
            sum += Mat.LMat[i][j] * y[j];
        }
        y[i] = Bias[i] - sum;
    }
    // 回代法求解 Ux = y
    for (int i = Mat.Dimension - 1; i >= 0; --i) {
        double sum = 0;
        for (int j = i + 1; j < Mat.Dimension; ++j) {
            sum += Mat.UMat[i][j] * Solution[j];
        }
        Solution[i] = (y[i] - sum) / Mat.UMat[i][i];
    }
    // 释放内存
    free(y);
}
```

上面的算法利用 Doolittle 分解的结果求解出 $\boldsymbol{X}^{(k+1)}$ 以便后续反幂法迭代的计算。

2.4 反幂法迭代

```
double InversePowerMethod(matrix Mat, double *InitialX, double *
   Vector, double epsilon){
    double *XK = (double *)malloc(Mat.Dimension * sizeof(double));
    double *NextXK = (double *)malloc(Mat.Dimension * sizeof(double
       ));
    double Max, Change; int flag = 0;

    for (int i = 0; i < Mat.Dimension; i++){XK[i] = InitialX[i];}
    do {
        Max = 0, Change = 0;
```

```
        for (int i = 0; i < Mat.Dimension; i++){Max = (fabs(XK[i])
           > Max) ? fabs(XK[i]) : Max;}
        for (int i = 0; i < Mat.Dimension; i++){Vector[i] = XK[i] /
            Max;}
        SolveFunction(Mat, Vector, NextXK);
        for (int i = 0; i < Mat.Dimension; i++){
            Change = (fabs(NextXK[i] - XK[i]) > Change) ? fabs(
               NextXK[i] - XK[i]) : Change;
        }
        for (int i = 0; i < Mat.Dimension; i++){XK[i] = NextXK[i];}
    }while(Change > epsilon);
    double lambda; Max = 0;
    for (int i = 0; i < Mat.Dimension; i++){
        Max = (fabs(NextXK[i]) > Max) ? fabs(NextXK[i]) : Max;
    }
    lambda = 1 / Max; free(XK), free(NextXK);
    return lambda;
}
```

3 实验结果

3.1 矩阵 1

k	$\boldsymbol{X}^{(k+1)}$					$\boldsymbol{Y}^{(k)}$					λ^{-1}
0	630	-1120	630	-120	5	1	1	1	1	1	1120
1	146253	-297849	196175	-45114	2377	0.6	-1	0.6	-0.1	0.004	297849
2	149113	-304047	200595	-46245	2447	0.5	-1	0.7	-0.2	0.008	304047
3	149157	-304142	200661	-46261	2447	0.5	-1.0	0.7	-0.2	0.01	304142
4	149158	-304143	200662	-46261	2448	0.5	-1.0	0.7	-0.2	0.01	304143
5	149158	-304143	200662	-46261	2448	0.5	-1.0	0.7	-0.2	0.01	304143
6	149158	-304143	200662	-46261	2448	0.5	-1.0	0.7	-0.2	0.01	304143
7	149158	-304143	200662	-46261	2448	0.5	-1.0	0.7	-0.2	0.01	304143

表 1: 矩阵 1 的迭代

由于数位较多，因此在此展示社区较多小数部分，计算得出 $\lambda = 0.000003285$，按摸最小特征向量 $\boldsymbol{X} = (0.49042, -1.00000, 0.65976, -0.15210, 0.00805)^{\mathrm{T}}$。

3.2 矩阵 2

k	$\boldsymbol{X}^{(k+1)}$				$\boldsymbol{Y}^{(k)}$				λ^{-1}
0	0.00	2.00	-0.00	1.00	1.00	1.00	1.00	1.00	2.00
1	-0.62	5.62	-2.38	3.50	0.00	1.00	-0.00	0.50	5.62
2	-0.93	8.08	-3.43	5.04	-0.11	1.00	-0.42	0.62	8.08
3	-0.94	8.09	-3.44	5.05	-0.12	1.00	-0.43	0.62	8.09
4	-0.94	8.09	-3.45	5.06	-0.12	1.00	-0.43	0.62	8.09
5	-0.94	8.09	-3.45	5.06	-0.12	1.00	-0.43	0.62	8.09
6	-0.94	8.09	-3.45	5.06	-0.12	1.00	-0.43	0.62	8.09

表 2: 矩阵 2 的迭代

由于数位较多，因此在此展示社区较多小数部分，计算得出按摸最小特征值 $\lambda = 0.12355$，对应的特征向量为 $\boldsymbol{X} = (-0.11573, 1.00000, -0.42569, 0.62477)^{\mathrm{T}}$。

4 结果分析

a. 从实验结果来看，矩阵 $\boldsymbol{A}_1$ 的按摸最小特征值更接近于 0，但是迭代次数却比矩阵 $\boldsymbol{A}_2$ 要多一次，这并不符合预期。但是通过观察迭代表格，发现矩阵 1 从第 2 次计算开始就离最终结果比较接近了，而矩阵 2 要更多一次，所以可能因为矩阵 1 计算结果的数值更大，相比矩阵 2 更难达到指定精度，因此迭代次数更多，但收敛速度更快；

b. 在实验中，我们尝试估计每次迭代的特征值，但在计算过程中可能会遇到数值问题。特别是在计算 $\boldsymbol{X}^{(k+1)}/\boldsymbol{Y}^{(k)}$ 时，由于分母中的特征向量可能包含非常小的数值，这可能导致数值不稳定性。为了解决这个问题，我们可以采取一些数值稳定性的技巧，比如在计算中避免除以接近零的数值，或者采用其他数值稳定的方法来估计特征值。