第5章 特征值

在线性代数中，一个矩阵 $\boldsymbol{A}\in\mathbb{C}^{n\times n}$，一个数$\lambda\in\mathbb{C}$和 n 维非零向量 $\boldsymbol{x}\in\mathbb{C}^n$ 满足

$$\boldsymbol{A}\boldsymbol{x}=\lambda\boldsymbol{x} \tag{5-1}$$

称λ为$\boldsymbol{A}$的特征值(Eigen Value)，$\boldsymbol{x}$称为属于特征值λ的特征向量(Eigen Vector)。

特征值表示矩阵在特定方向上的“拉伸”或“压缩”程度，最大特征值对应矩阵的主作用方向。所有特征值都是正数的矩阵称为正定(Positive Definite)矩阵，所有特征值都是非负的矩阵称为半正定(Positive Semidefinite)矩阵，所有特征值都是负数的矩阵称为负定(Negative Definite)矩阵。

特征值和特征向量是AI数值算法中最重要的概念，在人工智能领域，特征值被广泛地应用于降维、图像处理(特征脸)、优化分析与深度学习、图结构学习、自然语言处理语义分析等，是理解AI数据结构和算法行为的重要工具。

首先数据降维可以说是人工智能算法的基础和核心之一。在人工智能计算中，数据往往具有很高的维度，这会增加计算成本和模型的复杂性，同时可能会导致过拟合，因此需要作降维处理。例如主成分分析(PCA)是一种常用的数据降维技术，它利用特征值和特征向量来找到数据的主要成分。具体来讲，首先计算数据的协方差矩阵，然后求出协方差矩阵的特征值和特征向量。特征值表示对应特征向量方向上的数据方差，方差越大意味着该方向上的数据变化越大，包含的信息越多。通过选择具有较大特征值的特征向量，可以构建一个新的低维空间，将原始数据投影到这个空间中，从而实现数据降维。这样在保留大部分重要信息的同时，降低了数据的维度，提高了模型的训练效率和泛化能力。

在图像识别与处理中，图像可以表示为一个矩阵，对图像的处理和分析可以转换为对矩阵特征值的操作，例如，在人脸识别中，通过对大量人脸图像的特征值和特征向量进行分析，可以提取出人脸的关键特征。首先对人脸图像进行灰度化和归一化处理，然后将其转换为一个向量。通过计算这些向量协方差矩阵的特征值和特征向量，可以得到人脸图像的特征脸。特征脸实际上是人脸图像空间中的一组基向量，它们代表了人脸图像的主要变化模式；通过将新的人脸图像投影到特征脸空间中，可以得到该图像的特征向量表示，从而实现对人脸的识别和分类。

在自然语言处理中，词袋模型是一种常用的文本表示方法，它将文本表示为一个向量，

向量的每个维度对应一个单词在文本中出现的频率。然而,这种表示方法往往会导致高维稀疏向量,计算成本高昂。这时可以通过对文本数据的协方差矩阵或相似性矩阵进行特征值和特征向量分析,找到文本数据的主要特征,例如,潜在语义分析(LSA)就是利用奇异值分解(SVD,与特征值分解密切相关)来发现文本中的潜在语义结构。通过将文本矩阵分解为奇异值矩阵和左右奇异向量矩阵,将文本投影到一个低维的语义空间中,从而实现文本的降维和语义表示。在信息检索中,这种方法可以帮助我们找到与查询文本语义相似的文档,从而提高检索的准确性和效率。

在 K-means 等聚类算法中,有时利用特征值和特征向量来初始化聚类中心,以提高聚类的效果和稳定性,例如,通过对数据的协方差矩阵进行特征值分解,选择具有较大特征值的特征向量所对应的点作为初始聚类中心,可以使聚类结果更符合数据的内在结构。在支持向量机(SVM)中,核函数的选择和计算与特征值和特征向量有关;核函数将原始数据映射到高维特征空间,以便在高维空间中进行线性分类。通过对核矩阵进行特征值分解,可以更好地理解核函数的性质和作用,选择合适的核函数和参数,从而提高 SVM 的分类性能。

在神经网络中,特征值和特征向量常常可以用于分析网络的权重矩阵,例如,通过对权重矩阵进行特征值分解,可以研究网络的稳定性和收敛性。此外,在一些算法的优化过程中,也会利用特征值和特征向量信息来调整学习率和更新权重,以加速模型的训练和提高模型的性能。在数据集的异常检测中,异常点通常具有与其他数据点不同的特征;通过对数据的协方差矩阵或距离矩阵进行特征值和特征向量分析,可以发现数据中的异常模式,例如,如果某个数据点在某个特征向量方向上的投影值与其他数据点相差较大,并且该特征向量对应的特征值较小,则说明该数据点在这个方向上的变化与其他数据点不一致,可能是一个异常点。这种方法可以用于检测信用卡欺诈、网络入侵等异常行为。在推荐系统中,常用矩阵分解技术来预测用户对物品的偏好,例如,将用户-物品评分矩阵分解为用户特征矩阵和物品特征矩阵的乘积,这个过程可以通过奇异值分解或其他基于特征值和特征向量的方法来实现;通过分析特征值和特征向量,可以发现用户和物品的潜在特征,从而为用户提供个性化的推荐,例如,具有较大特征值的特征向量可能对应着用户或物品的重要特征维度,根据这些特征可以对用户和物品进行聚类和分类,进而为用户推荐与其兴趣相似的其他喜欢的物品。

特征值常用计算方法主要包括以下几种。

(1) 幂法(或称幂迭代法,Power Iteration):简单,但仅能求最大特征值,特别适用于稀疏矩阵。

(2) 雅可比法(Jacobi Method):该方法主要通过一系列正交相似变换,将对称矩阵逐步对角化,对角线元素即为特征值,但是雅可比法仅适用于对称矩阵,精度高,但计算量较大。

(3) QR 算法:高效稳定,适合中小型稠密矩阵,结合位移技术等方法可加速收敛。

(4) 兰乔斯(Lanczos)法:将大型稀疏矩阵转换为三对角矩阵后求解特征值,主要适用于高维稀疏矩阵,例如机器学习中的大规模数据。

5.1 幂法和反幂法

幂法是通过迭代来计算矩阵的主特征值与其对应特征向量的方法，这种方法特别适合于求解大型稀疏矩阵的特征值问题。反幂法(Inverse Power Method)是幂法的变体，它主要用于计算矩阵的最小绝对值特征值。

5.1.1 幂法简介

幂法主要用于求矩阵按模最大的特征值和相应的特征向量。设矩阵 $\boldsymbol{A}$ 的 n 个特征值 $\lambda_i(i=1,2,\cdots,n)$满足

$$|\lambda_1|>|\lambda_2|\geqslant|\lambda_3|\geqslant\cdots\geqslant|\lambda_n| \tag{5-2}$$

和

$$\boldsymbol{A}\boldsymbol{v}_i=\lambda_i\boldsymbol{v}_i \quad (i=1,2,\cdots,n) \tag{5-3}$$

给定初始向量 $\boldsymbol{x}^{(0)}\in\mathbb{C}^n$，构造一个向量序列 $\{\boldsymbol{x}^{(k)}\}$

$$\boldsymbol{x}^{(0)},\boldsymbol{x}^{(1)}=\boldsymbol{A}\boldsymbol{x}^{(0)},\cdots,\boldsymbol{x}^{(k)}=\boldsymbol{A}\boldsymbol{x}^{(k-1)},\cdots$$

即

$$\boldsymbol{x}^{(k)}=\boldsymbol{A}\boldsymbol{x}^{(k-1)} \quad (k=0,1,2,\cdots) \tag{5-4}$$

分析这一序列可以从中找出计算 λ_1 和相应的特征向量的方法。

由于 n 维向量组 $\boldsymbol{v}_1,\boldsymbol{v}_2,\cdots,\boldsymbol{v}_n$ 线性无关，所以必存在不全为 0 的一组数 $a_1,a_2,\cdots,a_n$，使

$$\boldsymbol{x}^{(0)}=a_1\boldsymbol{v}_1+a_2\boldsymbol{v}_2+\cdots+a_n\boldsymbol{v}_n$$

由式(5-3)和式(5-4)得

$$\begin{cases}\boldsymbol{x}^{(1)}=\boldsymbol{A}\boldsymbol{x}^{(0)}=a_1\lambda_1\boldsymbol{v}_1+a_2\lambda_2\boldsymbol{v}_2+\cdots+a_n\lambda_n\boldsymbol{v}_n\\ \boldsymbol{x}^{(2)}=\boldsymbol{A}\boldsymbol{x}^{(1)}=a_1\lambda_1^2\boldsymbol{v}_1+a_2\lambda_2^2\boldsymbol{v}_2+\cdots+a_n\lambda_n^2\boldsymbol{v}_n\\ \vdots\\ \boldsymbol{x}^{(k)}=\boldsymbol{A}\boldsymbol{x}^{(k-1)}=a_1\lambda_1^k\boldsymbol{v}_1+a_2\lambda_2^k\boldsymbol{v}_2+\cdots+a_n\lambda_n^k\boldsymbol{v}_n\end{cases}$$

如果模最大特征值是实数且为单根，那么

$$\boldsymbol{x}^{(k)}=\lambda_1^k\left(a_1\boldsymbol{v}_1+a_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\boldsymbol{v}_2+\cdots+a_n\left(\frac{\lambda_n}{\lambda_1}\right)^k\boldsymbol{v}_n\right) \tag{5-5}$$

则当 k 很大时，由于

$$\left|\frac{\lambda_i}{\lambda_1}\right|<1 \quad (i=2,3,\cdots,n)$$

当 $a_1\neq0$ 时，有

$$\boldsymbol{x}^{(k)}=\lambda_1^k \quad (a_1\boldsymbol{v}_1+\varepsilon_k) \tag{5-6}$$

这里

$$\varepsilon_k = \sum_{i=2}^{n} a_i \frac{\lambda_i}{\lambda_1} \boldsymbol{x}_i \to 0$$

即有

$$\boldsymbol{x}^{(k)} \approx \lambda_1^k a_1 \boldsymbol{v}_1 \approx \lambda_1 \boldsymbol{x}^{(k-1)} \tag{5-7}$$

$$\boldsymbol{x}^{(k+1)} \approx \lambda_1^{k+1} a_1 \boldsymbol{v}_1 \approx \lambda_1 \boldsymbol{x}^{(k)} \tag{5-8}$$

式(5-7)表明，当 k 充分大时，向量 $\boldsymbol{x}^{(k)}$ 可近似地作为 $\boldsymbol{A}$ 的属于 λ_1 的特征向量[3]，而式(5-8)则说明当 k 充分大时，$\boldsymbol{x}^{(k+1)}$ 与 $\boldsymbol{x}^{(k)}$ 近似地只差一个倍数，这个倍数便是模最大的特征值 λ_1（注意这是两个向量相除，它们近似差一个倍数，但实际不一定，应用的时候可以分别取两个向量分量最大的相除）。

$$\lambda_1 \approx \frac{\boldsymbol{x}_i^{(k+1)}}{\boldsymbol{x}_i^{(k)}} \tag{5-9}$$

式(5-7)实际就是计算

$$\boldsymbol{x}^{(k)} = \boldsymbol{A}^k \boldsymbol{x}^{(0)} \quad (k = 1, 2, \cdots) \tag{5-10}$$

称这种求按模最大的特征值及对应的特征向量的迭代法为幂法。

这一迭代的收敛速度显然与 $|\lambda_2/\lambda_1|$ 有关，如果初始值选取不好，则会出现 $a_1 = 0$，这时可以换初始值，但实际上因为有舍入误差，所以每次迭代会增加在 $\boldsymbol{v}_1$ 方向上的微小分量，随着次数增大，这些微小分量会成为主导。

此外，如果选取的 $\boldsymbol{x}^{(0)}$ 使 $a_1 = 0$，由于有舍入误差的影响，所以必然会在迭代中产生这样的 $\boldsymbol{x}^{*(k)}$，它在 $\boldsymbol{v}^{(1)}$ 方向上的分量不为 0。这样就相当于以 $\boldsymbol{x}^{*(k)}$ 为初始向量重新开始迭代。另外，幂法的主要缺点是，当 $|\lambda_1| > 1$ 或 $|\lambda_1| < 1$ 时，由式(5-5)可知，$\boldsymbol{x}^{(k)}$ 会发生上溢或下溢，因此需要采取其他措施。

为了克服这一缺点，避免出现较大与较小数参与运算，通常采用的方法是，每次迭代对向量 $\boldsymbol{x}^{(k)}$ 作归一化处理

$$\begin{cases} \boldsymbol{y}^{(k)} = \dfrac{\boldsymbol{x}^{(k)}}{\| \boldsymbol{x}^{(k)} \|_\infty} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{A}\boldsymbol{y}^{(k)} \end{cases} \quad (k = 0, 1, 2, \cdots) \tag{5-11}$$

$$\boldsymbol{y}^{(k)} \approx \boldsymbol{v}_1, \quad \max_{1 \leqslant i \leqslant n} | x_i^{(k)} | \approx | \lambda_1 | \tag{5-12}$$

其中，$\boldsymbol{y}$ 为特征向量，其最大分量为 1；对它作用 $\boldsymbol{A}$ 得到 $\boldsymbol{x}$，相当于作用了 λ_1，$\boldsymbol{x}$ 最大分量为 λ_1。

因此，实际使用的迭代公式是

$$\begin{cases} \boldsymbol{y}^{(k)} = \dfrac{\boldsymbol{x}^{(k)}}{x_{j_k}^{(k)}} \\ \boldsymbol{x}^{(k+1)} = \boldsymbol{A}\boldsymbol{y}^{(k)} \end{cases} \quad (k = 0, 1, 2, \cdots) \tag{5-13}$$

幂法在实际计算中有很多加速方法，例如 Rayleigh 商加速，以及原点平移法等，读者可

以参阅相关文献。

5.1.2 幂法算法

输入：维数 n，矩阵 $\boldsymbol{A}$，非零初始向量 $\boldsymbol{x}^{(0)}$，允许误差 ε_0 及最大迭代次数。

输出：矩阵 $\boldsymbol{A}$ 的按模最大的近似特征值 λ_1 及相应的近似特征向量 $\boldsymbol{y}^{(k)}$ 或相关的错误信息。

步骤 1，当 $k=1,2,\cdots$ 时，迭代计算步骤 2～步骤 7。

步骤 2，计算 $\boldsymbol{x}^{(k)}=\boldsymbol{A}\boldsymbol{y}^{(k-1)}$。

步骤 3，找最大分量值

$$|x_{j_k}^{(k)}|=\max_{1\leqslant j\leqslant n}|x_j^{(k)}|$$

的最小下标 j_k。$\max X=x_{j_k}^{(k)}$。

步骤 4，如果 $x_{j_k}^{(k)}=0$，则输出“A 有零特征值，请选择新的初始向量重新开始计算”，并停止迭代，否则继续。

步骤 5，规范化

$$\boldsymbol{y}^{(k)}=\frac{\boldsymbol{x}^{(k)}}{x_{j_k}^{(k)}}$$

步骤 6，计算容差 $\varepsilon_1=\max\limits_{1\leqslant j\leqslant n}|y_j^{(k)}-y_j^{(k-1)}|$，若 $\varepsilon_1<\varepsilon_0$，则输出 $\lambda_1\approx\max X$，相应的近似特征向量为 $\boldsymbol{y}^{(k)}$，并停止迭代，否则继续。

步骤 7，如果超过最大迭代数，则输出错误信息“迭代不收敛！”，停止迭代。

5.1.3 幂法 C++实现

创建幂法类，成员变量有矩阵阶次 n、最大特征值 maxEigen、矩阵 mtx、特征向量 my、向量量 mx。成员函数有构造函数 PowerEigen()、求解函数 SolvePower()和输出结果函数 PrintResult()。

头文件 PowerEigen.h 中的代码如下：

```
//第 5 章/PowerEigen/PowerEigen.h
/**
 ************************************************************
 * @brief 幂法，反幂法的示范程序中 powerEigen 类的头文件
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 * ----------------------------------------------------------
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 ************************************************************
```

```
 * 以下是类注释
 * ------------------------------------------------------------
 * PowerEigen 类,幂法求特征值
 * @param n          矩阵的阶数
 * @param maxEigen   最大特征值
 * @param my         特征向量
 * @param mtx        矩阵
 * @param mx         中间向量
 * ------------------------------------------------------------
 * 以下是各函数的功能注释
 * ------------------------------------------------------------
 * bool SolvePower(const double tol, const int maxCount)
 * @brief 幂法求解矩阵最大特征值
 * @param[in] tol 收敛容差
 * @param[in] maxCount 最大迭代次数
 * ------------------------------------------------------------
 * void PrintResult()
 * @brief 输出结果
 ************************************************************
 */
#pragma once
#include<iostream>
#include<vector>
using namespace std;
class PowerEigen
{
private:
    size_t n;
    double maxEigen;
    vector<double> max;
    vector<double> my;
    vector<vector<double>> mtx;
public:
    PowerEigen(vector<vector<double>>& A, int n);
    bool SolvePower(const double tol, const int maxCount);
    void PrintResult();
};
```

类实现源文件 PowerEigen.cpp,代码如下:

```
//第 5 章/PowerEigen/PowerEigen.cpp
#include "PowerEigen.h"
PowerEigen::PowerEigen(vector<vector<double>>& A, int n):mtx(A),n(n)
{
    max.resize(n, 1.0);
    my.resize(n, 1.0);
}

bool PowerEigen::SolvePower(const double tol, const int maxCount)
{
    vector<double> Ynext(n, 1.0);
    bool flag = 0;
```

```
int iterateNum = 0;
double sum;

do
{
    iterateNum++;
    my = Ynext;

    //步骤 2
    for (int i = 0; i < n; ++i)
    {
        sum = 0;
        for (int j = 0; j < n; ++j) sum += mtx[i][j] * my[j];
        maX[i] = sum;
    }

    //步骤 3 find max xj
    maxEigen = 0;
    for (int i = 0; i < n; ++i)
    {
        if (fabs(maxEigen) < fabs(maX[i])) maxEigen = maX[i];
        //步骤 4
        if (fabs(maxEigen) < 1.0e - 30)
        {
            cout << "有零特征值,请选择新的初始向量重新开始计算" << endl;
            exit(1);
        }
    }
    //步骤 5
    for (int i = 0; i < n; ++i) Ynext[i] = maX[i] / maxEigen;

    //步骤 6 compute residual
    sum = 0;
    for (int i = 0; i < n; ++i)
    {
        sum += fabs(my[i] - Ynext[i]);
    }

    if (iterateNum < maxCount)
    {
        if (fabs(sum) < tol)
        {
            cout << "tolerance = " << sum << endl;
            flag = 1;
        }
    }
    else
    {
        cout << "iterater num >" << iterateNum <<", 迭代不收敛!" << endl;
        flag = 1;
    }

} while (!flag);
```

```
        my = Ynext;
        return true;
    }

    void PowerEigen::PrintResult()
    {
        cout << "max Eigen value = " << maxEigen << endl;
        cout << "对应的特征向量: ";
        for (int i = 0; i < n; ++i)
            cout << my[i]<<"\t";
        cout << endl;

        //归一化向量
        double norm = 0.0;
        for (int j = 0; j < n; ++j) {
            norm += my[j] * my[j];
        }
        norm = std::sqrt(norm);
        vector < double > v(my.size());
        cout << "归一化特征向量: \n";
        for (int j = 0; j < n; ++j) {
            v[j] = my[j]/norm;
            cout << v[j] << endl;
        }
        return;
    }
```

【例 5-1】 求矩阵 $\boldsymbol{A}$ 的最大特征值及其特征向量

$$\boldsymbol{A}=\begin{pmatrix} 3 & -4 & 3.1 \\ -4 & 6 & 3 \\ 3 & 3 & 1 \end{pmatrix}$$

求其按模最大的特征值和对应的特征向量,要求容差小于 10^{-7}。

主程序代码如下:

```
//第 5 章/PowerEigen/PowerMain.cpp
#include "powerEigen.h"
int main()
{
    vector < vector < double >> A{ {3.0, - 4.0,3.1},
                  { - 4.0,6.0,3.0},{3.0,3.0,1.0} };
    double tol = 1e - 3;
    int maxnum = 100;
    PowerEigen p2(A, A.size());
    p2.SolvePower(tol, maxnum);
    p2.PrintResult();
    return 0;
}
```

计算结果如下:

```
max Eigen value = 8.86175
对应的特征向量: - 0.601318      1      0.152379
归一化特征向量:
- 0.510987
0.849779
0.129489
```

5.1.4 反幂法

反幂法用于计算非奇异矩阵 $\boldsymbol{A}$ 的按模最小特征值和特征向量。根据幂法可以求得最大特征值，那么矩阵 $\boldsymbol{A}$ 的逆阵 $\boldsymbol{A}^{-1}$ 的按模最大特征值的倒数就是 $\boldsymbol{A}$ 的按模最小的特征值。

这样就把求矩阵 $\boldsymbol{A}$ 的按模最小特征值转换为求 $\boldsymbol{A}^{-1}$ 的按模最大特征值 $1/\lambda_n$，特征向量不变。

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{A}^{-1}\boldsymbol{x}^{(k)} \quad (k=0,1,2,\cdots) \tag{5-14}$$

然而，在实际技术中求逆矩阵比较困难，而且运算大，因此通常情况下变为求解下面的方程组

$$\boldsymbol{A}\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)} \quad (k=0,1,2,\cdots) \tag{5-15}$$

与幂法一样，需要进行归一化

$$\begin{cases}\boldsymbol{y}^{(k)}=\dfrac{\boldsymbol{x}^{(k)}}{\|\boldsymbol{x}^{(k)}\|_\infty} \quad (k=0,1,2,\cdots)\\ \boldsymbol{A}\boldsymbol{x}^{(k+1)}=\boldsymbol{y}^{(k)}\end{cases} \tag{5-16}$$

也可以采用移位法加速计算

$$\begin{cases}\boldsymbol{y}^{(k)}=\dfrac{\boldsymbol{x}^{(k)}}{\|\boldsymbol{x}^{(k)}\|_\infty} \quad (k=0,1,2,\cdots)\\ (\boldsymbol{A}-\lambda_0\boldsymbol{I})\boldsymbol{x}^{(k+1)}=\boldsymbol{y}^{(k)}\end{cases} \tag{5-17}$$

因为矩阵不变，所以为节省计算量，可以先将矩阵 LU 分解，这样可以大大节省计算量。

5.1.5 反幂法算法

输入：维数 n，矩阵 $\boldsymbol{A}$，非零初始向量 $\boldsymbol{x}^{(0)}$，允许误差 ε_0 及最大迭代次数。

输出：矩阵 $\boldsymbol{A}$ 的按模最大的近似特征值 λ_1 及相应的近似特征向量 $\boldsymbol{y}^{(k)}$ 或错误信息“迭代不收敛!”

步骤 1，当 $k=1,2,\cdots$ 时，迭代计算步骤 2～步骤 7。

步骤 2，取初始向量 $\boldsymbol{u}^{(0)}$（例如取 $\boldsymbol{u}^{(0)}=(1,1,\cdots,1)^{\mathrm{T}}$），置精度要求为 ε。

步骤 3，对 $\boldsymbol{A}$ 作 LU 分解，即 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{U}$。

步骤 4，解线性方程组 $\boldsymbol{L}\boldsymbol{y}^{(k)}=\boldsymbol{u}^{(k-1)}$，$\boldsymbol{U}\boldsymbol{v}^{(k)}=\boldsymbol{y}^{(k)}$。

步骤 5，正则化

$$m_k=\max(\boldsymbol{v}^{(k)})$$

$$\boldsymbol{u}^{(k)}=\boldsymbol{v}^{(k)}/m_k$$

步骤 6，若$|m_k-m_{k-1}|<\varepsilon$，则停止计算（$1/m_k$ 作为绝对值最小特征值 λ_n，$\boldsymbol{u}^{(k)}$ 作为相应的特征向量）。

步骤 7，如果达到最大迭代数，则输出错误信息“迭代不收敛!”，停止迭代。

5.1.6 反幂法 C++算法实现

（1）使用 InvPower 类，成员变量包括阶次 n、向量 mx、向量 my、系数矩阵 mtx、最小特征值 minEigen。

（2）成员函数有带参构造函数、求解函数 solveInvPower()、输出结果函数 PrintResult()。

（3）在求解过程中使用 LU 分解法计算线性方程组，所以包含 LUD.h 头文件。

代码如下：

```
//第 5 章/InvPower/InvPower.h
/**
 ************************************************************
 * @brief 反幂法类的头文件
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 * ----------------------------------------------------------
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 ************************
 * 以下是类注释
 * ----------------------------------------------------------------
 * InvPower 类，反幂法求特征值
 * @param n          矩阵的阶数
 * @param minEigen   最小特征值
 * @param my         特征向量
 * @param mtx        矩阵
 * @param mx         中间向量
 * ----------------------------------------------------------------
 * 以下是各函数的功能注释
 * ----------------------------------------------------------
 * bool SolveInvPower(const double tol, const int maxCount)
 * @brief 反幂法求解矩阵最小特征值
 * @param[in] tol 收敛容差
 * @param[in] maxCount 最大迭代次数
 * ----------------------------------------------------------
 * void PrintResult()
 * @brief 输出结果
 ************************************************************
 */
#pragma once
#include<iostream>
#include<vector>
```

```
#include "LUD.h"
using namespace std;

class InvPower
{
public:
    size_t n;
    vector<double> mx;
    vector<double> my;
    vector<vector<double>> mtx;
    double minEigen;
public:
    InvPower(const vector<vector<double>>&, size_t size);
    ~InvPower() {};
    void SolveInvPower(const double tol, const int maxCount);
    void PrintResult();
};


InvPower::InvPower(const vector<vector<double>>& Amtx, size_t size) {
    this->n = size;
    this->mtx = Amtx;
    mx.resize(n, 1.0);
    my.resize(n, 1.0);
}

void InvPower::SolveInvPower(const double tol, const int maxCount)
{
    bool flag = 0;
    int itNum = 0;
    //步骤 1,LU decomposition
    LUD Lud(mtx);
    Lud.LUdec();

    //步骤 2,iterator
    while (!flag)
    {
        itNum ++;
        my = mx;

        //步骤 3, solve LU equations
        mx = Lud.solveEqn(my);
        //步骤 4, solve LU equations
        minEigen = 0;              //找最小
        for (int i = 0; i < n; ++i)
        {
            if (fabs(minEigen) < fabs(mx[i]))
            {
                minEigen = mx[i];
            }
        }

        //步骤 5,正则化
```

```
            for (int i = 0; i < n; ++i) mx[i] /= minEigen;

           //步骤 6,convergence?
            double sum = 0;
            for (int i = 0; i < n; ++i)
            {
                sum += fabs(my[i] - mx[i]);
            }

            if (itNum < maxCount)
            {
                if (fabs(sum) < tol)
                {
                    cout << "tol = " << sum << endl;
                    flag = 1;
                }
            }
            else
            {
                cout << "iterate number >" << maxCount << "迭代不收敛,quit" << endl;
                flag = 1;
            }
        }
    }
    void InvPower::PrintResult() {
        cout << "min eigen = " << 1.0 / minEigen << endl;
        cout << "对应的特征向量" << endl;
        for (auto xx : mx) cout << xx << endl;
    }
```

【例 5-2】 计算矩阵 $\boldsymbol{A}$ 的最小特征值与特征向量

$$\boldsymbol{A}=\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

求其按模最小的特征值和对应的特征向量,要求容差小于 10^{-7}。

主程序代码如下:

```
//第 5 章/InvPower/InvPowerMain.h
#include < iostream >
#include < vector >
#include "InversePowerEigen.h"
using namespace std;
int main()
{
    vector < vector < double > > A{ { 2.0, -1.0, 0},
                                    {-1.0,2.0, -1.0},
                                    {0, -1.0, 2.0}};
    InversePowerEigen p1(A,A.size());
    double epsilon = 1.0e-6;
```

```
    int maxIterator = 1000;
    p1.SolveInvPower (epsilon,maxIterator);
    p1.PrintResult ();
    return 0;
}
```

计算结果如下：

```
min eigen = 0.585786
对应的特征向量
0.707107
1.000000
0.707107
```

5.2 吉文斯变换

吉文斯(Givens)变换是一种将 n 维向量在第(i,k)两个维度确定的坐标平面进行旋转(将其中一个元素化 0)的变换，因此它又叫平面旋转变换。

5.2.1 原理

如果有二阶实对称矩阵

$$\boldsymbol{A}=\begin{pmatrix}a_{11} & a_{21}\\ a_{21} & a_{22}\end{pmatrix}$$

选取矩阵

$$\boldsymbol{G}(\theta)=\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}$$

利用 $\boldsymbol{G}(\theta)$对二阶实对称矩阵 $\boldsymbol{A}$ 进行变换，得到

$$\boldsymbol{G}^{\mathrm{T}}(\theta)\boldsymbol{A}\boldsymbol{G}(\theta)=\begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}a_{11} & a_{21}\\ a_{21} & a_{22}\end{pmatrix}\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}$$

$$=\begin{pmatrix}a_{11}\cos^2\theta+a_{22}\sin^2\theta+a_{21}\sin2\theta & \frac{1}{2}(a_{22}-a_{11})\sin2\theta+a_{21}\cos2\theta\\ \frac{1}{2}(a_{22}-a_{11})\sin2\theta+a_{21}\cos2\theta & a_{11}\sin^2\theta+a_{22}\cos^2\theta-a_{21}\sin2\theta\end{pmatrix}$$

只要选取 θ 满足$\frac{1}{2}(a_{22}-a_{11})\sin2\theta+a_{21}\cos2\theta=0$，即满足

$$\tan2\theta=\frac{2a_{21}}{a_{11}-a_{22}}$$

$\boldsymbol{G}^{\mathrm{T}}(\theta)\boldsymbol{A}\boldsymbol{G}(\theta)$ 就变为对角阵

$$\begin{pmatrix} a_{11}\cos^2\theta + a_{22}\sin^2\theta + a_{21}\sin 2\theta & 0 \\ 0 & a_{11}\sin^2\theta + a_{22}\cos^2\theta - a_{21}\sin 2\theta \end{pmatrix}$$

可以看出 $\boldsymbol{G}(\theta)$ 可以把矩阵的一些元素变换为 0。

现在看一般情况，设 $\boldsymbol{A}$ 为 n 阶（$n \geqslant 3$）实对称矩阵，若 $\boldsymbol{A}^{k-1}$ 的非对角元中按模最大元素是 a_{pq}（$p<q$），取 $\boldsymbol{G}(p,q,\theta)$ 具有以下形式

$$\boldsymbol{G}(p,q,\theta) = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & \cos\theta & & & & \sin\theta & \leftarrow p\text{ 行} \\ & & & 1 & & & & \\ & & & & \ddots & & & \\ & & & & & 1 & & \\ & & -\sin\theta & & & & \cos\theta & \leftarrow q\text{ 行} \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{pmatrix}$$

记 $\boldsymbol{G}=\boldsymbol{G}(p,q,\theta)$，$\boldsymbol{G}$ 的主对角线元素除第 p 行和第 q 行为 cos 外，其余都是 1，第 q 行第 p 列元素为 $-\sin$，第 p 行第 q 列元素为 sin，G 的其他元素都是零。

可以验证 $\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}=\boldsymbol{I}$，即 $\boldsymbol{G}$ 是正交阵，则称正交矩阵 $\boldsymbol{G}$ 为吉文斯（Givens）变换阵或平面旋转矩阵。作变换

$$\boldsymbol{A}^k = \boldsymbol{G}\boldsymbol{A}^{k-1}\boldsymbol{G}^{\mathrm{T}} \tag{5-18(a)}$$

由 $\boldsymbol{A}^{k-1}$ 到 $\boldsymbol{A}^k$，只有 $\boldsymbol{A}^{k-1}$ 的 p 行 p 列、q 行 q 列元素发生变化，其他元素保持不变，这种变换方法称为 Givens 变换（或 Givens 旋转）。

$\boldsymbol{A}$ 经正交相似变换以后，就将事先选定的非对角线元素变成了 0（$a_{ij}^k=0$）。

注意：不要以为对每个非零元素进行一次这样的变换就可以得到对角矩阵，因为如果 a_{ik} 或 a_{jk} 为 0，经变换以后往往又不是零了，但每经一次这样的变换，非对角线元素总是“向零接近一步”。

采用吉文斯旋转变换能够消去给定向量的某个分量（使其为 0），对于已经有很多零元素的稀疏向量、稀疏矩阵，吉文斯旋转变换非常有效。

对式(5-18(a))直接进行矩阵计算可得，$\boldsymbol{A}^k$ 的元素与 $\boldsymbol{A}^{k-1}$ 的元素之间有以下关系

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \quad (i,j=1,2,\cdots,n;\ i,j \neq p,q) \tag{5-18(b)}$$

而 $\boldsymbol{A}^k$ 的 p,q 行和 p,q 列元素是

$$\begin{cases} a_{ip}^{(k)} = a_{pi}^{(k)} = a_{ip}^{(k-1)}\cos\theta + a_{iq}^{(k-1)}\sin\theta \\ a_{iq}^{(k)} = a_{qi}^{(k)} = -a_{ip}^{(k-1)}\sin\theta + a_{iq}^{(k-1)}\cos\theta \\ a_{pq}^{(k)} = a_{qp}^{(k)} = (a_{qq}^{(k-1)} - a_{pp}^{(k-1)})\sin\theta\cos\theta + a_{pq}^{(k-1)}(\cos^2\theta - \sin^2\theta) \end{cases} \tag{5-19(a)}$$

和

$$\begin{cases} a_{pp}^{(k)} = a_{pp}^{(k-1)}\cos^2\theta + 2a_{pq}^{(k-1)}\sin\theta\cos\theta + a_{qq}^{(k-1)}\sin^2\theta \\ a_{qq}^{(k)} = a_{pp}^{(k-1)}\sin^2\theta - 2a_{pq}^{(k-1)}\sin\theta\cos\theta + a_{qq}^{(k-1)}\cos^2\theta \end{cases} \tag{5-19(b)}$$

为使 p，q 两行元素有变化，其他元素不变，即取

$$a_{pq}^{(k-1)} = a_{qp}^{(k-1)} = 0$$

计算可知，只需

$$\tan(2\theta) = \frac{2a_{pq}^{(k-1)}}{a_{pp}^{k-1} - a_{qq}^{k-1}} \tag{5-20}$$

成立。

在进行实际计算时，为减少计算量，避免有效数字丧失，通常将 θ 限制在 $\left[-\frac{\pi}{4},\frac{\pi}{4}\right]$ 范围内，若 $a_{pp}^{(k-1)} - a_{qq}^{(k-1)} = 0$，则取

$$\theta = \text{sign}\left(a_{pq}^{(k-1)}\right)\frac{\pi}{4} \tag{5-21}$$

在计算中通常并不直接计算 θ，而是利用三角函数关系计算 $\sin\theta$、$\cos\theta$、$\sin2\theta$ 和 $\cos2\theta$。

矩阵 $\boldsymbol{A}$ 按式(5-19)作正交相似变换，元素除 p、q 两行，p、q 两列有变化外，其余元素保持不变，并且 $\boldsymbol{A}^k$ 保持对称性。通过选择 $|a_{pq}| = \max\limits_{i<j}|a_{ij}|$，这样每次变换后，非主对角线元素平方和衰减速度最快。

5.2.2 吉文斯变换算法

输入：矩阵维数 n，矩阵 $\boldsymbol{A}$ 的元素 a_{ij}，允许误差 ε 及最大迭代次数 maxNum。

输出：迭代后矩阵 $\boldsymbol{A}$ 或错误信息“迭代次数超限!”。

步骤 1，当 $k=0,1,\cdots$ 时，迭代计算步骤 2～步骤 7。

步骤 2，从 $\boldsymbol{A}^k$ 中找出绝对值最大的非对角线元素。

$$\max E = |a_{pq}| = \max_{i<j}|a_{i,j}| \quad (i,j=1,\cdots,n)$$

步骤 3，若 $|a_{pq}^{(k)}|<\varepsilon$，则 $\boldsymbol{A}^k$ 近似为一对角矩阵。输出 $\boldsymbol{A}^k$，停止计算，否则继续。

步骤 4，计算旋转角度 θ。

当 $a_{pp}^{(k-1)} - a_{qq}^{(k-1)} = 0$ 时，计算

$$\theta = \begin{cases} -\frac{\pi}{4}, & a_{pq}^{(k-1)} < 0 \\ \frac{\pi}{4}, & a_{pq}^{(k-1)} > 0 \end{cases}$$

否则计算角度

$$\tan2\theta = 2a_{pq}^{(k-1)}/(a_{pp}^{(k-1)} - a_{qq}^{(k-1)})$$

步骤 5，令 sn=$\sin\theta$，cs=$\cos\theta$，根据三角关系，计算 sn、cs、$\sin2\theta$ 和 $\cos2\theta$。

步骤 6，计算各元素

$$\begin{cases} a_{pp} = a_{pp} * \mathrm{cs}^2 + a_{qq} * \mathrm{sn}^2 + a_{pq} * \sin 2\theta; \\ a_{qq} = a_{pp} * \mathrm{sn}^2 + a_{qq} * \mathrm{cs}^2 - a_{pq} * \sin 2\theta; \\ a_{pq} = a_{qp} = 0.5 * (a_{qq} - a_{pp}) * \sin 2\theta + a_{pq} * \cos 2\theta; \\ a_{ip} = a_{pi} = \mathrm{cs} * a_{pi} + \mathrm{sn} * a_{qi}; \\ a_{iq} = a_{qi} = \mathrm{cs} * a_{qi} - \mathrm{sn} * a_{pi}; \\ (i, j \neq p, q) \end{cases}$$

步骤 7，判断迭代次数，如果超限，则输出错误信息“迭代次数超限!”，停止计算。

5.2.3 吉文斯变换算法 C++实现

编程时的考量，为让程序结构清晰，可以采取下面的措施：

(1) 把绝对值最大的非对角线元素独立出来。

(2) 使用 inline 定义取符号函数。

(3) 同上面的方法，可以把吉文斯变换直接简化为下面简单的两步：

第 1 步，寻找最大非对角元素，并判断元素是否趋于 0。

第 2 步，计算变换后的元素，即进行吉文斯旋转。

(4) 旋转矩阵的计算，拆分三部分：

第 1 步，计算旋转角度。

第 2 步，计算 a_{pp}，a_{qq}，a_{pq}。

第 3 步，计算其他元素。

吉文斯旋转类的成员变量有矩阵阶数、实对称矩阵、容差和最大迭代次数。成员函数有带参数构造函数、把矩阵转换为对角阵函数 GivensMtx()、旋转后元素计算函数 Rotation()、寻找非对角元素最大值所在行和列及最大值函数 MaxElement()、打印输出函数 printMtx() 及取符号函数 getSign()。在程序中的一些设计是为 5.3 节雅可比算法考量的，例如，函数 Rotation(const int p,const int q,double& sn,double& cs)，后面 sn 和 cs 变量在本节中就不需要在参数中出现，这些读者需要了解。

代码如下：

```
//第 5 章//GivensRotation/GivensRotation.h
/**
 ****************************************************************
 * Copyright (C) 2025 SunKunfeng
 * @brief Givens 变换
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025 - 02 - 01
 * @version V1.0 20250201
 * ---------------------------------------------------------------
 * License - Identifier: LGPL - 3.0 - or - later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 ************************************************************************
 * 以下是功能注释
 * ----------------------------------------------------------------------
 * Givens 类,把非对角元素变换为 0
 * @param nDim                  矩阵的阶数
 * @param Eps                   容差
 * @param MaxIterateNumber      最大迭代次数
 * @param a                     二维 n * n 数组,存放实对称矩阵
 * @param eigenVec              特征向量
 * ----------------------------------------------------------------------
 */
#pragma once
#include < vector >
#include < iostream >
#include < iomanip >
#include < cmath >
using namespace std;

//取符号函数
inline int getSign(double num) {
    if (num > 0) return 1;
    else if (num < 0) return -1;
    else return 0;
}

class GivensRot
{
public:
    double Eps;
    int MaxNum;
    int nDim;
    vector < vector < double >> a;
    vector < vector < double >> eigenVec;
public:
    GivensRot(const vector < vector < double >> & A, double eps, int maxN);
    void GivensMtx();
    void Rotation(const int p, const int q, double& sn, double& cs);
    void printMtx(const vector < vector < double >> & arr);
    void MaxElement(int&, int&, double&);
};

//构造函数
GivensRot::GivensRot(const vector < vector < double >> & rhs, double eps, int maxN)
{
    this->a = rhs;
    this->Eps = eps;
    this->MaxNum = maxN;
    this->nDim = rhs.size();
    this->eigenVec.resize(nDim);
    for (int i = 0; i < nDim; i++)
    {
        eigenVec[i].resize(nDim, 0.0);
```

```
            eigenVec[i][i] = 1.0;
        }
    }

    //打印输出矩阵函数
    void GivensRot::printMtx(const vector<vector<double>>& arr)
    {
        for (int i = 0; i < arr.size(); i++) {
            for (int j = 0; j < arr[0].size(); j++) {
                std::cout << setw(12)<< arr[i][j];
            }
            std::cout << '\n';
        }
    }

    //Givens 旋转
    void GivensRot::Rotation(const int p,const int q,double& sn, double&cs)
    {
        //计算旋转角度
        double theta, sn2th, cs2th;

        if ((a[p][p] - a[q][q]) < 1e-20) {          //角度等于 pi/4, -pi/4
            sn = getSign(a[p][q]) / sqrt(2);        //使用 sign 函数
            cs = 1.0 / sqrt(2);
            sn2th = getSign(a[p][q]);
            cs2th = 0;
        }
        else {
            theta = 0.5 * atan2(2 * a[p][q], (a[p][p] - a[q][q]));
            sn = sin(theta);
            cs = cos(theta);
            sn2th = sin(2 * theta);
            cs2th = cos(2 * theta);
        }
        //计算 A 矩阵各元素
        double app = a[p][p];
        double aqq = a[q][q];
        double apq = a[p][q];
        a[p][p] = app * cs * cs + aqq * sn * sn + apq * sn2th;
        a[q][q] = app * sn * sn + aqq * cs * cs - apq * sn2th;
        a[p][q] = 0.5 * (aqq - app) * sn2th + apq * cs2th;
        a[q][p] = a[p][q];

        for (int i = 0; i < nDim; i++)
        {
            if ((i != p) && (i != q))
            {
                double api = a[p][i];
                double aqi = a[q][i];

                a[p][i] = cs * api + sn * aqi;
                a[q][i] = cs * aqi - sn * api;
                a[i][p] = a[p][i];
```

```
                a[i][q] = a[q][i];
            }
        }

        //计算特征向量
        for (int k = 0; k < nDim; k++)
        {
            double Ekp = eigenVec[k][p];
            double Ekq = eigenVec[k][q];

            eigenVec[k][p] = Ekp * cs + Ekq * sn;
            eigenVec[k][q] = Ekq * cs - Ekp * sn;
        }
    }

    //寻找非对角元素最大值所在行和列
    void GivensRot::MaxElement(int& row, int& col, double& Max)
    {
        Max = fabs(a[0][1]);
        row = 0;
        col = 1;
        for (int i = 0; i < nDim; i++)
        {
            for (int j = 0; j < nDim; j++)
            {
                double d = fabs(a[i][j]);
                if ((d > Max) && (i != j))    //不在对角
                {
                    Max = d;
                    row = i;
                    col = j;
                }
            }
        }
    }

    //把矩阵转换为对角阵
    void GivensRot::GivensMtx()
    {
        int nCount = 0;
        double MaxE = fabs(a[0][1]);
        int row, col;
        double sn, cs;

        while (nCount < MaxNum)
        {
            MaxElement(row, col, MaxE);
            if (MaxE < Eps) break;
            Rotation(row, col,sn,cs);
            nCount++;
            cout << "iterate number = " << nCount << endl;
        }
```

```
    if(nCount == MaxNumcout) {
        cout << "超过最大迭代次数不收敛!" << endl;
        exit(1);
    }
}
```

【例 5-3】 使用吉文斯变换矩阵将 $\boldsymbol{A}$ 变为对角阵

$$\boldsymbol{A}=\begin{pmatrix} 0 & 12 & 16 & -15 \\ 12 & 288 & 309 & 185 \\ 16 & 309 & 312 & 80 \\ -15 & 185 & 80 & -600 \end{pmatrix}$$

计算主程序如下：

```
//第 5 章/GivensRotation/GRmain.cpp
#include < iostream >
#include < iomanip >
#include < vector >
#include "GivensRot.h"
using namespace std;
int main()
{
    cout.precision(6);
    cout.setf(ios::fixed);
    vector < vector < double > > a{ {0, 12, 16, -15},
            {12, 288, 309, 185},
            {16, 309, 312, 80},
            {-15,185, 80, -600} };
    double eps = 1e-15;
    double MaxNum = 100;
    GivensRot Giv(a,eps,MaxNum);
    Giv.GivensMtx();
    Giv.printMtx(Giv.a);
}
```

计算结果如下：

```
iterate number = 19
  637.848726    -0.000000    -0.000000     0.000000
   -0.000000     4.899281    -0.000000     0.000000
   -0.000000    -0.000000    -4.899281    -0.000000
    0.000000     0.000000    -0.000000  -637.848726
```

5.3 雅可比法

雅可比法是数学家雅可比(Jacobi)于 1846 年提出的，经典 Jacobi 方法用于求解实对称矩阵的特征值。它的核心思想是采用一些平面旋转矩阵将对称阵变为对角阵。

5.3.1 雅可比法原理

前面介绍的幂法和反幂法仅能求解矩阵最大、最小特征值，而雅可比法可以求实对称矩阵的全部特征值和特征向量。

根据线性代数相关知识知道，若 $\boldsymbol{A}$ 为 n 阶对角阵

$$\boldsymbol{A}=\begin{pmatrix} a_1 & & & \\ & a_2 & & \\ & & \ddots & \\ & & & a_n \end{pmatrix} \tag{5-22}$$

则 $a_1,a_2,\cdots,a_n$ 就是 $\boldsymbol{A}$ 的 n 个特征值。

定理 5-1 设 $\boldsymbol{A}=(a_{ij})$ 是任一 n 阶实对称矩阵，则必存在一个直交相似变换矩阵 $\boldsymbol{U}$ 将它化为一个对角阵，即

$$\boldsymbol{U}\boldsymbol{A}\boldsymbol{U}^{\mathrm{T}}=\boldsymbol{D}=\begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix} \tag{5-23}$$

其中，$\lambda_1,\lambda_2,\cdots,\lambda_n$ 就是 $\boldsymbol{A}$ 的 n 个特征值，而 $\boldsymbol{U}$ 的第 j 列向量是与 λ_j 相应的特征向量；称 $\boldsymbol{A}$ 与 $\boldsymbol{U}^{-1}\boldsymbol{A}\boldsymbol{U}$ 相似，相似矩阵具有相同的特征值。

在正交相似变换下，矩阵元素的特征值保持不变，因此寻找这样的正交相似变换，使对称矩阵 $\boldsymbol{A}$ 经过变换之后，变为式(5-23)形式的矩阵，这就是雅可比算法的基本想法。

通过 Givens 旋转可以把一个矩阵变换为对角阵，如果把 $\boldsymbol{G}$ 阵代替 $\boldsymbol{U}$ 阵，也就是经典雅可比法（简称雅可比法）。一般来讲，必须通过一系列的直交相似变换 $\boldsymbol{G}_1,\boldsymbol{G}_2,\cdots,\boldsymbol{G}_k,\cdots$，才能把矩阵 $\boldsymbol{A}$ 化为一个对角阵

$$\boldsymbol{G}_k\cdots\boldsymbol{G}_2\boldsymbol{G}_1\boldsymbol{A}\boldsymbol{G}_1^{\mathrm{T}}\boldsymbol{G}_2^{\mathrm{T}}\cdots\boldsymbol{G}_k^{\mathrm{T}}\rightarrow\boldsymbol{D}$$

5.3.2 特征向量计算

假设经过 N 次 Givens 变换，有 $\boldsymbol{A}_{N+1}\approx\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n)$，则

$$\boldsymbol{A}_{N+1}=\boldsymbol{G}_N\cdots\boldsymbol{G}_2\boldsymbol{G}_1\boldsymbol{A}\boldsymbol{G}_1^{\mathrm{T}}\boldsymbol{G}_2^{\mathrm{T}}\cdots\boldsymbol{G}_N^{\mathrm{T}}$$

其中，记 $\boldsymbol{R}_N=\boldsymbol{G}_1^{\mathrm{T}}\boldsymbol{G}_2^{\mathrm{T}}\cdots\boldsymbol{G}_N^{\mathrm{T}}$，于是有

$$\boldsymbol{A}\boldsymbol{R}_N=\boldsymbol{R}_N\boldsymbol{A}_{N+1}\approx\boldsymbol{R}_N\,\mathrm{diag}(\lambda_1,\lambda_2,\cdots,\lambda_n) \tag{5-24}$$

式(5-24)表明，$\boldsymbol{R}_N$ 的各列向量就是矩阵 $\boldsymbol{A}$ 的特征值相应的特征向量的近似值。

在进行具体计算时，并不需要保留旋转矩阵 $\boldsymbol{G}_k$，而 $\boldsymbol{R}_N$ 是逐次形成的。首先令 $\boldsymbol{R}=\boldsymbol{I}$，然后每迭代一次作一次变换

$$\boldsymbol{R}\leftarrow\boldsymbol{R}\boldsymbol{G}^{\mathrm{T}}(p_k,q_k,\theta_k)\quad(k=1,2,\cdots,n)$$

由 Givens 变换知，对 $\boldsymbol{C}=\boldsymbol{A}\boldsymbol{G}^{\mathrm{T}}(p,q,\theta)$，则 $\boldsymbol{C}=(c_{ij})_{n\times n}$ 的元素为

$$\begin{cases} c_{ij} = a_{ij} \quad (j \neq p, q) \\ c_{ip} = a_{ip}\cos\theta + a_{iq}\sin\theta \quad (i = 1, 2, \cdots, n) \\ c_{iq} = -a_{ip}\sin\theta + a_{iq}\cos\theta \end{cases} \tag{5-25}$$

即 p,q 两列元素有变化,其余元素不变,因此只需计算 $\boldsymbol{R}$ 中 p,q 两列元素,即

$$\begin{cases} r_{ip} = r_{ip}\cos\theta_k + r_{iq}\sin\theta_k \\ r_{iq} = -r_{ip}\cos\theta_k + r_{iq}\sin\theta_k \end{cases} \quad (i = 1, 2, \cdots, n) \tag{5-26}$$

就可以得到特征向量。

雅可比法的优点是算法简单,有较强的数值稳定性,且不论矩阵 $\boldsymbol{A}$ 的特征值分布如何,雅可比法总是收敛的,它适合于矩阵 $\boldsymbol{A}$ 阶数 n 不太大时求全部特值和特征向量,此外,雅可比法有利于并行算法实现。

经典雅可比法的缺点是不能利用原有矩阵的特性,如稀疏性、带状性;当矩阵 $\boldsymbol{A}$ 阶数 n 较大时,收敛速度较慢。常见的改进方法有循环雅可比法和过关雅可比法。

5.3.3 雅可比法求矩阵特征值及特征向量算法

输入:矩阵维数 n,对称方阵 $\boldsymbol{A}$,容差 ε,最大迭代次数 maxNum。

输出:所有特征值 λ_i,以及对应的特征向量或错误提示。

步骤 1,吉文斯变换为对角阵,各对角线元素即为特征值。

步骤 2,计算特征向量

(1) 取初始 $\boldsymbol{V}_0 = \boldsymbol{I}$ 为单位矩阵,计算

$$\boldsymbol{R} = \boldsymbol{V}_0\boldsymbol{V}_1\cdots\boldsymbol{V}_m$$

(2) $\boldsymbol{R}$ 的第 i 列对应 λ_i 特征向量

$$\begin{cases} r_{k,p} = r_{kp} \cdot \mathrm{cs} + r_{kq} \cdot \mathrm{sn} \\ r_{k,q} = r_{kq} \cdot \mathrm{cs} - r_{kp} \cdot \mathrm{sn} \end{cases}$$

步骤 3,如果迭代次数大于最大迭代数,则停止迭代,输出“超过最大迭代次数不收敛!”。

步骤 4,输出特征值 $\lambda_i = a_{i,i}(i = 1, 2, \cdots, n)$ 和特征向量。

步骤 5,如果需要,则对特征值进行排序。

5.3.4 STL map 用法

在 C++中,STL(Standard Template Library,标准模板库)的 map 是一种关联容器,用于存储键-值对(key-value pairs),并且键是唯一的。

1) 包含< map >

要使用 map,需要包含头文件< map >。

2) 定义格式

```
std::map < KeyType, ValueType > mapName;
```

其中，KeyType 是键的类型，ValueType 是值的类型，代码如下：

```
std::map<int, std::string> myMap;                    //键为 int,值为 std::string
```

3）初始化

初始化可以使用以下几种方式，代码如下：

```
std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}, {3, "three"}};
//使用列表初始化
std::map<int, std::string> myMap2(myMap);    //使用另一个 map 进行复制初始化
```

4）插入元素

插入元素使用 insert 方法，代码如下：

```
myMap.insert(std::make_pair(4, "four"));      //插入键-值对
```

如果键已存在，则 insert 不会覆盖原有值，而是忽略插入操作。也可以使用下标操作符[]进行插入，代码如下：

```
myMap[5] = "five";                            //如果键不存在,则会自动插入键-值对
```

如果键已存在，则[]会覆盖原有值。

5）访问元素

访问元素一般通过下标操作符[]实现，代码如下：

```
std::string value = myMap[1];                 //获取键为 1 的值
```

如果键不存在，则会自动插入一个默认值（ValueType 的默认构造值），可以通过 at 方法访问元素，代码如下：

```
std::string value = myMap.at(1);              //获取键为 1 的值
```

如果键不存在，则 at 会抛出 std::out_of_range 异常。

6）查找元素

查找元素有 find 和 count 方法，代码如下：

```
//使用 find 方法
auto it = myMap.find(2);                      //返回指向键为 2 的迭代器
//使用 count 方法
if (myMap.count(2)) {
    std::cout << "Key 2 exists" << std::endl;
}
```

7）遍历 map

可以使用两种方法进行遍历，示例代码如下：

```
//可以通过迭代器遍历
for (auto it = myMap.begin(); it != myMap.end(); ++it) {
    std::cout << it->first << " -> " << it->second << std::endl;
}
//使用 for 循环遍历
for (const auto& pair : myMap) {
    std::cout << pair.first << " -> " << pair.second << std::endl;
}
```

8）删除

删除单个元素的方法 erase，代码如下：

```
myMap.erase(2);                          //删除键为 2 的元素
```

清空所有元素的方法 clear，格式如下：

```
myMap.clear();
```

map 内部是基于红黑树实现的，键会自动排序。map 的键是唯一的，如果需要允许重复键，则可以使用 std::multimap。如果需要快速查找且不关心顺序，则可以使用 std::unordered_map。总之，map 是 C++STL 中非常强大的容器之一，适用于需要快速查找和管理键-值对的场景。

5.3.5 雅可比法 C++实现

Jacobi 类作为 Givens 类的派生类，构造函数重载了 GivensRot 类的构造。Jacobi 类继承了 GivensRot 类的成员变量容差 Eps、最大迭代次数 MaxNum、阶次 nDim、系数矩阵 a。这里将矩阵变量命名为 a 仅仅是为了与算法对应和便于理解，在实际编码中这种方式是严重不推荐的，最好使用 matrixA 或者_mtxA 这种有实际含义且不易混淆的命名方式。

从算法上看，一旦完成吉文斯变换，对角线就是特征值。下面只需计算特征向量，并对特征值进行排序，这时就可以使用 map 把特征值和特征向量对应起来，并直接排序，代码如下：

```
//第 5 章/Jacobi/Jacobi.h
/**
 ******************************************************************
 * Copyright (C) 2025 SunKunfeng, skf0558@sina.com
 * @brief 雅可比法求实对称矩阵的特征值及特征向量
 * @author SunKunfeng
 * @date 2025-02-01
 * @version V1.0 20250201
 ******************************************************************
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
 * ------------------------------------------------------------------
 * 以下是类注释
 * ------------------------------------------------------------------
 * Jacobi 类,继承吉文斯旋转类,求实对称矩阵的特征值及特征向量的雅可比法
 * 利用雅可比(Jacobi)方法求实对称矩阵的全部特征值及特征向量
 * @param eigenVec          特征向量(按列存储)
 * @param EigenValue        特征值数组
 * @param a                 二维 n * n 数组,存放实对称矩阵
 * @param nDim              矩阵的阶数
 * @param Eps               容差
 * @param MaxIterateNumber 最大迭代次数
 * ------------------------------------------------------------------
 * 以下是功能注释
 * ------------------------------------------------------------------
 * void calEigenVec()
 * @brief 求解特征向量
 * ------------------------------------------------------------------
 * void Solve()
 * @brief 雅可比计算过程函数
 * ------------------------------------------------------------------
 *void sortEigen()
 * @brief 排序特征值函数
 * ------------------------------------------------------------------
 */

#pragma once
#include<map>
#include "GivensRot.h"
using namespace std;

class Jacobi:public GivensRot
{
private:
    vector<double> EigenValue;
    vector<vector<double>> eigenVec;
public:
    Jacobi(const vector<vector<double>>& A, double eps, int maxN);
    void calEigenVec(const int p, const int q, double sn, double cs);
    void Solve();
    void outputResult();
    vector<double> getEigenValue() { return EigenValue; };
    vector<vector<double>> getEigenVec() { return EigenVec; };
    void sortEigen();
    vector<vector<double>> Transpose(vector<vector<double>>& a, const int row, const
int col);
};

//构造函数
Jacobi::Jacobi(const vector<vector<double>>& A, double eps, int maxN):
    GivensRot(A, eps, maxN)
{
    this->nDim = A.size();
```

```
    this->EigenValue.resize(A.size());
    this->eigenVec.resize(nDim);
    for (int i = 0; i < nDim; i++)
    {
        eigenVec[i].resize(nDim, 0.0);
        eigenVec[i][i] = 1.0;
    }
}

//打印输出函数
void Jacobi::outputResult()
{
    cout << "eigen value:\n";
    for (int i = 0; i < nDim; i++)
    {
        cout << EigenValue[i] << " -- 对应的 vector:\n";
        for (int j = 0; j < nDim; j++)
        {
            cout << eigenVec[i][j] << "\n";
        }
        cout << endl;
    }
}

//计算特征向量
void Jacobi::calEigenVec(const int p, const int q, double sn, double cs)
{
    for (int k = 0; k < nDim; k++)
    {
        double Ekp = eigenVec[k][p];
        double Ekq = eigenVec[k][q];
        eigenVec[k][p] = Ekp * cs + Ekq * sn;
        eigenVec[k][q] = Ekq * cs - Ekp * sn;
    }
}

//求解计算过程函数
void Jacobi::Solve()
{
    for (int i = 0; i < nDim; i++)
    {
        for (int j = 0; j < nDim; j++)
        {
            if (i != j) {
                if (a[i][j] != a[j][i])
                {
                    cout << "matrix is not symmtry\n";
                    exit(1);
                }
            }
        }
    }
```

```
    int nCount = 0;
    double MaxE = fabs(a[0][1]);
    int row, col;
    double sn, cs;

    while (nCount < MaxNum)
    {
        MaxElement(row, col, MaxE);
        if (MaxE < Eps) break;
        Rotation(row, col, sn, cs);
        calEigenVec(row, col, sn, cs);
        nCount++;
cout << "iterate number = " << nCount << endl;
}
if(nCount == MaxNum)
    {
        cout << "超过最大迭代次数不收敛!" << endl;
        exit(1);
    }
    //将得到的特征向量转置为行向量
    eigenVec = Transpose(eigenVec, nDim, nDim);
for (int i = 0; i < nDim; i++) EigenValue[i] = this->a[i][i];


}

//矩阵转置函数
vector<vector<double>> Jacobi::Transpose(vector<vector<double>> & a, const int row,
const int col)
{
    vector<vector<double>> tmp(col, vector<double>(row, 0));

    for (int i = 0; i < col; i++)
    {
        for (int j = 0; j < row; j++)
        {
            tmp[i][j] = a[j][i];
        }
    }
    return tmp;
}

inline void printVec(const vector<double> &x)
{
    for (int i = 0; i < x.size(); i++)
        cout << x[i] << endl;
}

//排序特征值函数
void Jacobi::sortEigen()
{
    std::map<double, vector<double>> StEigen;
    for (int i = 0; i < nDim; i++)
```

```
            StEigen.insert(make_pair(EigenValue[i], EigenVec[i]));

        for (auto it = StEigen.begin(); it != StEigen.end(); ++it)
        {
            std::cout << it->first << " -> " << std::endl;
            printVec(it->second);
        }
    }
```

【例 5-4】 用雅可比方法求下列矩阵的所有特征值 λ_i 及对应的单位特征向量：

$$\boldsymbol{A}=\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

要求非对角线的元素的绝对值不超过 10^{-7}。

主程序代码如下：

```
//第 5 章/Jacobi/JacobiMain.cpp
#include "Jacobi.h"
int main()
{
    cout.precision(6);
    cout.setf(ios::fixed);
    vector<vector<double>> a{ {2, -1, 0},{-1, 2, -1} ,{0, -1, 2} };
    double eps = 1e-7;
    double MaxNum = 100;
    Jacobi Ja(a,eps,MaxNum);
    Ja.Solve();
    Ja.outputResult();
    return 0;
}
```

计算结果如下：

```
iterate number = 9
eigen value:
3.414214 -- 对应的 vector:
0.500000
-0.707107
0.500000
2.000000 -- 对应的 vector:
0.707107
-0.000000
-0.707107
0.585786 -- 对应的 vector:
0.500000
0.707107
0.500000
```

5.4 豪斯霍尔德变换

豪斯霍尔德(Householder)变换是于1958年Householder提出的，又称镜像映射，该方法比吉文斯旋转方法工作量少，精度高，缺点是算法和程序比较复杂。

5.4.1 豪斯霍尔德变换原理

设 $\boldsymbol{w}\in\mathbb{R}^n$，并且 $\|\boldsymbol{w}\|_2=1$，则初等矩阵

$$\boldsymbol{H}(\boldsymbol{w})=\boldsymbol{I}-2\boldsymbol{w}\boldsymbol{w}^{\mathrm{T}} \tag{5-27(a)}$$

称为豪斯霍尔德变换矩阵，也称为初等镜面反射矩阵[17]。

豪斯霍尔德矩阵 $\boldsymbol{H}(\boldsymbol{w})$有以下基本性质：

(1) 矩阵 $\boldsymbol{H}$ 为对称正交矩阵，即 $\boldsymbol{H}=\boldsymbol{H}^{\mathrm{T}}=\boldsymbol{H}^{-1}$。

(2) 设 $\boldsymbol{x}\in\mathbb{R}^n$，$\boldsymbol{y}=\boldsymbol{H}\boldsymbol{x}$，则 $\|\boldsymbol{y}\|_2=\|\boldsymbol{x}\|_2$。

(3) 设 $\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^n$，$\boldsymbol{x}\neq\boldsymbol{y}$，并且 $\|\boldsymbol{x}\|_2=\|\boldsymbol{y}\|_2$，则由向量 $\boldsymbol{w}=\dfrac{\boldsymbol{x}-\boldsymbol{y}}{\|\boldsymbol{x}-\boldsymbol{y}\|_2}$ 确定的豪斯霍尔德矩阵 $\boldsymbol{H}(\boldsymbol{w})$，使 $\boldsymbol{H}\boldsymbol{x}=\boldsymbol{y}$。

豪斯霍尔德变换性质(2)的意义是任意向量 $\boldsymbol{x}\in\mathbb{R}^n$，在豪斯霍尔德变换作用下，欧几里得长度不变。性质(3)表明对任意两个非零向量 $\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^n(\boldsymbol{x}\neq\boldsymbol{y})$，总可以选择单位法向量 $\boldsymbol{w}=\dfrac{\boldsymbol{x}-\boldsymbol{y}}{\|\boldsymbol{x}-\boldsymbol{y}\|_2}$ 所确定的 $\boldsymbol{H}(\boldsymbol{w})$，使 $\boldsymbol{y}=\boldsymbol{H}\boldsymbol{x}$ 与向量 $\boldsymbol{x}$ 关于以 $\boldsymbol{w}$ 为法向量的超平面是对称的，这也是 $\boldsymbol{H}(\boldsymbol{w})$称为镜面反射矩阵的原因。

豪斯霍尔德矩阵的一个重要应用是，对任意向量 $\boldsymbol{x}\in\mathbb{R}^n$，并且分量 $x_2,x_3,\cdots,x_n$ 不全为0，总可求得 $\boldsymbol{H}$ 矩阵，使

$$\boldsymbol{H}\boldsymbol{x}=\sigma\mathbf{e}_1 \tag{5-28}$$

其中，$\mathbf{e}_1=(1,0,\cdots,0)^{\mathrm{T}}$，$\sigma=\pm\|\boldsymbol{x}\|_2$。

令 $\boldsymbol{y}=\sigma\mathbf{e}_1$，则 $\|\boldsymbol{y}\|_2=\|\boldsymbol{x}\|_2$ 且 $\boldsymbol{x}\neq\boldsymbol{y}$。由向量

$$\boldsymbol{w}=\frac{\boldsymbol{x}-\boldsymbol{y}}{\|\boldsymbol{x}-\boldsymbol{y}\|_2}\triangleq\frac{\boldsymbol{u}}{\|\boldsymbol{u}\|_2}$$

知

$$\boldsymbol{H}=\boldsymbol{I}-2\boldsymbol{w}\boldsymbol{w}^{\mathrm{T}}=\boldsymbol{I}-2\frac{\boldsymbol{u}\boldsymbol{u}^{\mathrm{T}}}{\|\boldsymbol{u}\|_2^2} \tag{5-27(b)}$$

必有 $\boldsymbol{H}\boldsymbol{x}=\boldsymbol{y}$。$\boldsymbol{H}$ 也可以写作

$$\boldsymbol{H}=\boldsymbol{I}-\beta\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} \tag{5-27(c)}$$

其中，$\beta=1/\rho=2/\boldsymbol{v}^{\mathrm{T}}\boldsymbol{v}$。

综上，Householder 变换矩阵的计算公式为

$$\begin{cases} \boldsymbol{H}=\boldsymbol{I}-\beta\boldsymbol{u}\boldsymbol{u}^{\mathrm{T}}=\boldsymbol{I}-\dfrac{\boldsymbol{u}\boldsymbol{u}^{\mathrm{T}}}{\rho} \\ \sigma=\operatorname{sgn}(x_1)(\|\boldsymbol{x}\|_2)^{\frac{1}{2}} \\ \boldsymbol{u}=\boldsymbol{x}-\sigma\mathbf{e}_1 \\ \rho=\dfrac{1}{\beta}=\dfrac{1}{2}\|\boldsymbol{u}\|_2^2=\sigma(\sigma+x_1) \end{cases} \tag{5-29(a)}$$

针对矩阵计算的分量形式为

$$\begin{cases} \sigma_k=\operatorname{sgn}(a_{k+1,k})\left(\sum\limits_{j=k+1}^{n}a_{i,k}^2\right)^{\frac{1}{2}} \\ u_k^{\mathrm{T}}=(0,\cdots,0,a_{k+1,k}^{(j)}+\sigma_k,a_{k+2,k}^{(k)},\cdots,a_{n,k}^{(k)}) \\ \rho_k=a_{k+1,k}^{(k)}\sigma^{\frac{1}{2}}+\sigma_k=\dfrac{1}{\beta} \\ H_k=I-\dfrac{u_k u_k^{\mathrm{T}}}{\rho_k} \end{cases} \tag{5-29(b)}$$

在进行实际计算时，为了避免计算 u_k 时丢失有效数字，采用参考文献[8]和[9]给出的计算公式

$$u_k=\frac{a_k-\sigma}{\sqrt{2\sigma(\sigma-a_k)}}$$

则 ρ 的计算公式改写为

$$\rho=\sqrt{2\sigma(\sigma-a_k)} \tag{5-30}$$

5.4.2 豪斯霍尔德变换算法

为了避免计算中可能出现溢出问题，先正则化，使用 $\boldsymbol{x}/\|\boldsymbol{x}\|_\infty$ 代替 $\boldsymbol{x}$ 构造 $\boldsymbol{u}$。

输入：向量 $\boldsymbol{u}$ 的阶数 n，$\boldsymbol{u}$ 的元素。

输出：σ 和变换后的 $\boldsymbol{u}$。

步骤 1，当 $k=1,\cdots,n-1$ 时，迭代计算步骤 2～步骤 6。

步骤 2，寻找 $\eta=\max|u_i|(i=k,\cdots,n)$ 最大元素。

步骤 3，计算 $\sigma=\operatorname{sign}(u_k)\sqrt{\sum\limits_{i=1}^{n}\left(\dfrac{u_i}{\eta}\right)^2}$。

步骤 4，计算 $\rho=\sqrt{2\sigma(\sigma-u_k)}$。

步骤 5，计算 $u_k=(u_k-\sigma)$。

步骤 6，当 $i=k+1,\cdots,m$ 时，计算 $u_i=u_i/\rho$。

步骤 7，输出结果。

5.4.3 豪斯霍尔德变换 C++实现

编程时考虑便于以后扩展 QR 算法的使用，豪斯霍尔德变换改为列向量变换，代码如下：

```
//第 5 章/Housholder/Householder.h
/**
 ****************************************************************
 * Copyright (C) 2025
 * @brief 豪斯霍尔德变换
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 ****************************************************************
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * ---------------------------------------------------------------
 * 以下是功能注释
 * ---------------------------------------------------------------
 *列向量 Householder 变换
 * @param u,          列向量
 * @param k,          向量默认 = 0,对矩阵的变换有用
 * @param sigma,      在矩阵 QR 分解中使用
 * ---------------------------------------------------------------
 */
class hqr
{
private:
    int m_size;
    vector<vector<double>> m_elements;
public:
    hqr(const vector<vector<double>>& mtx, int dim);
    ~hqr() {};
    int getSize() const { return this->m_size; };
    double element(int row, int column)const { return m_elements[row][column]; };
    void printMtx(const vector<vector<double>>& mtx);
    void dswap(double& a, double& b);
    void Householder(vector<double>&u,double& sigma, int k = 0);
};

hqr::hqr(const vector<vector<double>>& mtx, int dim): m_size(dim), m_elements(mtx){}

//以下几个略
//void hqr::dswap(double& a, double& b)
//void hqr::printMtx(const vector<vector<double>>& mtx);

//Householder 列向量变换
void hqr::Householder(vector<double>& u,int k,double& sigma)
{
```

```
        int n = m_size;
        double eta = 0.0;

        //find max val
        for (int i = k; i < n; ++i)
        {
            if (fabs(u[i]) > eta) eta = fabs(u[i]);
        }

        //2.sigma,正则化
        sigma = 0;
        for (int i = k; i < n; ++i) sigma += u[i] * u[i] / (eta * eta);
        if (u[k] > 0.0) eta = -eta;
        sigma = eta * sqrt(sigma);

        //3.rho,u_k->a_kk,u_i->a_ik
        double rho = sqrt(2.0 * sigma * (sigma - u[k]));

        if ((rho + 1.0) != 1.0)
        {
            u[k] = (u[k] - sigma) / rho;
            for (int i = k + 1; i < n; ++i) u[i] /= rho;
        }
    }
```

5.5 QR 算法

QR 算法是数值分析的一颗明珠,该算法被评为 20 世纪十大算法之一。QR 算法最早在 1961 年由 J. G. Francis 提出,而后由一些学者进行了深入讨论并做出了卓有成效的改进与发展。

5.5.1 QR 算法原理

QR 算法是目前求一般矩阵全部特征值和特征向量行之有效的一种方法,对称矩阵和非对称矩阵都适用。这里将介绍最简单形式的 QR 算法,这种算法也可以看作计算矩阵幂 $\boldsymbol{A},\boldsymbol{A}^2,\boldsymbol{A}^3,\cdots$ 的一种稳定方法。

定理 5-2 设矩阵 $\boldsymbol{A}\in\mathbb{R}^{n\times n}$ 且非奇异,则一定存在正交矩阵 $\boldsymbol{Q}$ 和上三角矩阵 $\boldsymbol{R}$,使

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{R} \tag{5-31}$$

且当 $\boldsymbol{R}$ 的主对角元素均为正数时,则分解式(5-31)是唯一的。

令 $\boldsymbol{A}^{(1)}=\boldsymbol{A}$,正交分解为一个正交矩阵 $\boldsymbol{Q}_1$ 和一个上三角矩阵 $\boldsymbol{R}_1$ 的乘积:

$$\boldsymbol{A}^{(1)}=\boldsymbol{Q}_1\boldsymbol{R}_1$$

交换 $\boldsymbol{Q}_1$ 和 $\boldsymbol{R}_1$ 的次序得到

$$\boldsymbol{A}^{(2)}=\boldsymbol{R}_1\boldsymbol{Q}_1$$

因 $\boldsymbol{Q}_1$ 为正交矩阵,所以 $\boldsymbol{Q}_1^{\mathrm{T}}=\boldsymbol{Q}_1^{-1}$,于是

$$\boldsymbol{A}^{(2)}=\boldsymbol{Q}_1^{\mathrm{T}}\boldsymbol{A}^{(1)}\boldsymbol{Q}_1$$

即 $\boldsymbol{A}^{(2)}$ 与 $\boldsymbol{A}^{(1)}$ 正交相似。

用 $\boldsymbol{A}^{(2)}$ 代替 $\boldsymbol{A}^{(1)}$，重复上述步骤，一般地，若已知矩阵 $\boldsymbol{A}^{(k)}$ 具有分解式

$$\boldsymbol{A}^{(k)}=\boldsymbol{Q}_k\boldsymbol{R}_k$$

其中，$\boldsymbol{Q}_k$ 为正交矩阵；$\boldsymbol{R}_k$ 为上三角矩阵。将 $\boldsymbol{Q}_k$ 与 $\boldsymbol{R}_k$ 换序之后，得

$$\boldsymbol{A}^{(k+1)}=\boldsymbol{R}_k\boldsymbol{Q}_k=\boldsymbol{Q}_k^{\mathrm{T}}\boldsymbol{A}^{(k)}\boldsymbol{Q}_k$$

于是得到一个迭代序列 $\{\boldsymbol{A}^{(k)}\}$，这就是基本的 QR 算法。

QR 算法的理论基础是，若对矩阵 $\boldsymbol{A}$ 进行相似变换 $\boldsymbol{B}=\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{C}$，则变换后的矩阵 $\boldsymbol{B}$ 的特征值与 $\boldsymbol{A}$ 一致，这里简单予以说明。

若 $\boldsymbol{A}\boldsymbol{v}=\lambda\boldsymbol{v}$，令 $\boldsymbol{u}=\boldsymbol{C}^{-1}\boldsymbol{v}$，则有

$$\boldsymbol{A}\boldsymbol{v}=\boldsymbol{A}\boldsymbol{C}\boldsymbol{u}=\lambda\boldsymbol{C}\boldsymbol{u}$$

进一步得

$$\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{C}\boldsymbol{u}=\lambda\boldsymbol{u}$$

因此，λ 也是 $\boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{C}$ 的特征值，因此 QR 算法对 $\boldsymbol{A}$ 进行一系列的正交相似变换，就可以求出矩阵 $\boldsymbol{A}$ 的全部特征值和相应的特征向量。

基本 QR 算法就可以写为[8]

（1）分解：$\boldsymbol{A}_k=\boldsymbol{Q}_k\boldsymbol{R}_k$。

（2）构造：$\boldsymbol{A}_{k+1}=\boldsymbol{Q}_k^{\mathrm{T}}\boldsymbol{A}_k\boldsymbol{Q}_k=\boldsymbol{R}_k\boldsymbol{Q}_k$　$(k=1,2,3,\cdots)$。

这里，$\boldsymbol{Q}_k$ 为正交矩阵，$\boldsymbol{R}_k$ 为上三角形矩阵，并且当 $\boldsymbol{R}_k$ 主对角元均为正数时，基本 QR 算法正交三角分解唯一。QR 方法是幂法的推广，它可以求任意矩阵的全部特征值，该算法重要的一步是求上三角形矩阵，采用豪斯霍尔德变换是一种比较有效的方法。

豪斯霍尔德 QR 分解的过程，也就是先找到 $n-1$ 个初等反射矩阵 $\boldsymbol{H}$，然后左乘 $\boldsymbol{A}$ 矩阵，使对角线以下的列向量全部化成只有首元素不为 0 的单位向量（或其倍数），写成公式就是：

$$\boldsymbol{H}_{n-1}\cdots\boldsymbol{H}_2\boldsymbol{H}_1\boldsymbol{A}=\boldsymbol{R} \tag{5-32}$$

$$\boldsymbol{A}=(\boldsymbol{H}_{n-1}\cdots\boldsymbol{H}_2\boldsymbol{H}_1)^{\mathrm{T}}\boldsymbol{A}\boldsymbol{R} \tag{5-33}$$

其中，$\boldsymbol{H}$ 是初等反射矩阵。

由矩阵 $\boldsymbol{A}$ 的非奇异性及正交变换矩阵的性质知，一定可构造 $n-1$ 个 $\boldsymbol{H}$ 矩阵，$\boldsymbol{H}_1,\boldsymbol{H}_2,\cdots,\boldsymbol{H}_{n-1}$，使

$$\boldsymbol{A}_{k+1}=\boldsymbol{H}_k\boldsymbol{A}_k\quad(k=1,2,\cdots,n-1) \tag{5-34}$$

其中，$\boldsymbol{A}_1=\boldsymbol{A}$，到第 $n-1$ 步后，得到上三角矩阵

$$\boldsymbol{A}_n=\begin{pmatrix}\sigma_1 & a_{12}^{(n)} & \cdots & a_{1n}^{(n)}\\ & \sigma_2 & \cdots & a_{2n}^{(n)}\\ & \ddots & \vdots & \vdots\\ & & \sigma_{n-1} & a_{n-1n}^{(n)}\\ & & & \sigma_n\end{pmatrix}\triangleq\boldsymbol{R} \tag{5-35}$$

对角线元素即为特征值，其中

$$\boldsymbol{H}_{n-1}\boldsymbol{H}_{n-2}\cdots\boldsymbol{H}_2\boldsymbol{H}_1\boldsymbol{A}=\boldsymbol{R} \tag{5-36}$$

因为 $\boldsymbol{Q}=\boldsymbol{H}_1\boldsymbol{H}_2\cdots\boldsymbol{H}_{n-1}$ 为正交矩阵，即有

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{R}$$

若 $\boldsymbol{A}$ 非奇异，则上三角阵 $\boldsymbol{R}$ 也非奇异，从而 $\boldsymbol{R}$ 主对角线元素不为 0。

定理 5-3 设 $\boldsymbol{A}\in\mathbb{R}^{n\times n}$ 的特征值 λ_i 满足条件 $|\lambda_1|>|\lambda_2|>\cdots>|\lambda_n|>0$，$\boldsymbol{X}\in\mathbb{R}^{n\times n}$ 是以 $\boldsymbol{A}$ 的特征向量为列组成的矩阵，并且 $\boldsymbol{Y}=\boldsymbol{X}^{-1}$ 有 LU 分解，则 $\{\boldsymbol{A}^{(k)}\}$ 本质收敛或基本收敛于上三角矩阵

$$\boldsymbol{A}^{(k)}\xrightarrow{\text{本质收敛}}\begin{pmatrix}\lambda_1 & * & \cdots & *\\ & \lambda_2 & \ddots & \vdots\\ & & \ddots & *\\ & & & \lambda_n\end{pmatrix}\quad(k\to\infty)$$

定理 5-4 对任意 $\boldsymbol{A}\in\mathbb{R}^{n\times n}$，由 QR 算法产生的序列 $\{\boldsymbol{A}^{(k)}\}$，具有

(1) $\boldsymbol{A}^{(k)}$ 与 $\boldsymbol{A}$ 正交相似，从而 $\boldsymbol{A}^{(k)}$ 与 $\boldsymbol{A}$ 有相同的特征值。

(2) 由 QR 算法产生的 $\{\boldsymbol{A}^{(k)}\}$，如果当 $k\to\infty$ 时 $\boldsymbol{A}^{(k)}$ 收敛于上三角矩阵，则 QR 算法是收敛的。

5.5.2 一般实矩阵简单 QR 算法

输入：矩阵 $\boldsymbol{A}$，矩阵阶数 n，容差 ε。

输出：特征值或者错误信息。

步骤 1，当 $k=1,2,\cdots$ 时，迭代计算步骤 3～步骤 5。

步骤 2，对 $\boldsymbol{A}$ 做 QR 分解得到 $\boldsymbol{Q}$ 和 $\boldsymbol{R}$ 阵。

步骤 3，如果容差 tol$<\varepsilon$，则退出，否则转步骤 4。

步骤 4，计算 $\boldsymbol{A}_k=\boldsymbol{Q}_k\boldsymbol{R}_k$。

步骤 5，计算 $\boldsymbol{A}_{k+1}=\boldsymbol{R}_k\boldsymbol{Q}_k$。

步骤 6，输出 $\boldsymbol{A}$ 的对角线元素即为特征值，或者输出迭代次数超限信息。

5.5.3 豪斯霍尔德 QR 算法

找到 $n-1$ 个初等反射阵 $\boldsymbol{H}$ 把 $\boldsymbol{A}$ 对角线以下的列向量全部化为单位向量(或其倍数，方向由原向量中的最大元素的符号确定)，这样乘一个 $\boldsymbol{H}$ 矩阵就把一列对角线以下元素化为 0，如果这个 $\boldsymbol{A}$ 阵有 n 列，就要 $n-1$ 个 $\boldsymbol{H}$ 矩阵，然后不断地左乘 $\boldsymbol{A}$，直到最后变换为一个上三角阵 $\boldsymbol{R}$。

步骤 1，计算 Householder 矩阵

$$\boldsymbol{H}_k=\begin{pmatrix}\boldsymbol{I}_{k-1} & \boldsymbol{0}\\ \boldsymbol{0} & \hat{\boldsymbol{H}}_{m-k+1}\end{pmatrix} \tag{5-37}$$

其中

$$\hat{\boldsymbol{H}}_{m-k+1}=\begin{pmatrix}1-2u_k^2 & -2u_ku_{k+1} & \cdots & -2u_ku_m\\ -2u_{k+1}u_k & 1-2u_{k+1}^2 & \cdots & -2u_{k+1}u_m\\ \vdots & \vdots & \ddots & \vdots\\ -2u_mu_k & -2u_mu_{k+1} & \cdots & 1-u_m^2\end{pmatrix} \tag{5-38}$$

矩阵 $\hat{\boldsymbol{H}}_{m-k+1}$ 中 $u_i(i=k,k+1,\cdots,m)$的计算可以参见 Householder 变换。

步骤 2，计算 $\boldsymbol{Q}$。

$$\boldsymbol{H}_k\boldsymbol{Q}\Rightarrow\boldsymbol{Q} \tag{5-39}$$

分量计算形式可表示为

$$\begin{cases}t=\sum_{l=k}^{m}a_{lk}q_{lj}\\ q_{ij}-2a_{ik}t\Rightarrow q_{ij}\\ j=1,2,\cdots,m;\ i=k,k+1,\cdots,m\end{cases} \tag{5-40}$$

步骤 3，计算 $\boldsymbol{R}$。

$$\boldsymbol{H}_k\boldsymbol{A}\Rightarrow\boldsymbol{A} \tag{5-41}$$

分量形式为

$$\begin{cases}t=\sum_{l=k}^{m}a_{lk}a_{lj}\\ a_{ij}-2a_{ik}t\Rightarrow a_{ij}\\ j=1,2,\cdots,m;\ i=k,k+1,\cdots,m\end{cases} \tag{5-42}$$

最后取 $\boldsymbol{A}$ 的右上三角部分即为 $\boldsymbol{R}$，而 $\boldsymbol{Q}^{\mathrm{T}}\Rightarrow\boldsymbol{Q}$。

5.5.4 基本 QR 算法 C++实现

为了方便理解和使代码简明，先写一个向量运算的头文件 mtxVec.h，包括矩阵乘法、向量加减、向量与常量乘法、点积运算等，代码如下：

```
//第 5 章/HHQR4Eigen/mtxVec.h
/**
 ****************************************************************
 * Copyright (C)
 * @brief QR 分解
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 ****************************************************************
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
 * ----------------------------------------------------------------
 */
#pragma once
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

//打印矩阵函数
inline void printMatrix(const std::vector<std::vector<double>>& matrix) {
    for (const auto& row : matrix) {
        for (double val : row) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
}

//矩阵乘法函数
inline std::vector<std::vector<double>> matrixMultiply(const std::vector<std::vector
<double>>& A, const std::vector<std::vector<double>>& B) {
    int rowsA = A.size();
    int colsA = A[0].size();
    int colsB = B[0].size();
    std::vector<std::vector<double>> result(rowsA, std::vector<double>(colsB, 0.0));
    for (int i = 0; i < rowsA; ++i) {
        for (int j = 0; j < colsB; ++j) {
            for (int k = 0; k < colsA; ++k) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

//向量减法函数
inline std::vector<double> vectorSubtract(const std::vector<double>& a, const std::
vector<double>& b) {
    int n = a.size();
    std::vector<double> result(n);
    for (int i = 0; i < n; ++i) {
        result[i] = a[i] - b[i];
    }
    return result;
}

//向量点积函数
inline double dotVec(const std::vector<double>& a, const std::vector<double>& b) {
    int n = a.size();
    double result = 0.0;
    for (int i = 0; i < n; ++i) {
        result += a[i] * b[i];
    }
```

```
    return result;
}

//向量数乘函数
inline std:: vector < double > scalarMultiply ( const std:: vector < double > & a, double
scalar) {
    int n = a.size();
    std::vector < double > result(n);
    for (int i = 0; i < n; ++i) {
        result[i] = a[i] * scalar;
    }
    return result;
}

//计算向量的 2 - 范数
inline double norm(const std::vector < double > & a) {
    return std::sqrt(dotVec(a, a));
}
```

将豪斯霍尔德变换程序略微改造一下，另外也是为了适用 QR 分解。豪斯霍尔德变换类增加了 Q、A、QT 成员变量，同时增加了 getR()、getQ()等成员函数，这是为了便于进行 QR 分解时调用 Q、R 矩阵。qrDecomposition()函数用于 QR 分解，qrHouseholder()是豪斯霍尔德变换程序，程序同时使用矩阵和向量运算函数，简化了代码：

```
//第 5 章/HHQR4Eigen/HHQR.h
/**
 ************************************************************************
 * Copyright (C)
 * @brief QR 分解
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025 - 02 - 01
 * @version V1.0 20250201
 ************************************************************************
 * License - Identifier: LGPL - 3.0 - or - later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * ----------------------------------------------------------------------
 * 以下是功能注释
 * ----------------------------------------------------------------------
 * QR 分解类
 * @brief QR 分解类成员变量
 * @param m_size,      矩阵阶数
 * @param Q,           矩阵 Q
 * @param A,           矩阵 A
 * @param QT,          矩阵 Q 转置
 *  ---------------------------------------------------------------------
 * 成员函数 qrHouseholder();
 * @brief 豪斯霍尔德变换
 * @param[in] x,       待变换的向量
 * @param[out] H,      矩阵 H
```

```
 * ------------------------------------------------------------------
 * 成员函数 void qrDecomposition();
 * @brief QR 分解,将矩阵 A 分解为 Q 和 R 阵
 * ------------------------------------------------------------------
 */

#pragma once
#include "mtxVec.h"
class hhqr
{
private:
    int m_size;
    vector<vector<double>> Q;
    vector<vector<double>> A;
    vector<vector<double>> QT;
public:
    hhqr(const vector<vector<double>> & mtx, int dim);
    ~hhqr() {};
    int getSize() const { return this->m_size; };
    double element(int row, int column)const { return A[row][column]; };
    vector<vector<double>> getR() { return A; };
    vector<vector<double>> getQ() { return Q; };
    vector<vector<double>> getQT() { return QT; };
    vector<vector<double>>> qrHouseholder(const vector<double> & x);
    //QR 分解用矩阵变换
    void qrDecomposition();
};

hhqr::hhqr(const vector<vector<double>> & mtx, int dim) :m_size(dim),A(mtx)
{
    this->Q.resize(m_size, vector<double>(m_size));
    this->QT.resize(m_size, vector<double>(m_size));
}

//构造 Householder 变换矩阵
vector<vector<double>>> hhqr::qrHouseholder(const vector<double> & x)
{
    int n = x.size();
    std::vector<double> e(n, 0.0);
    e[0] = 1.0;
    double alpha = (x[0] > 0) ? -norm(x) : norm(x);
    std::vector<double> v = vectorSubtract(x, scalarMultiply(e, alpha));
    double beta = 2.0 / dotVec(v, v);
    std::vector<std::vector<double>>> H(n, std::vector<double>(n, 0.0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) {
                H[i][j] = 1.0 - beta * v[i] * v[j];
            }
            else {
                H[i][j] = -beta * v[i] * v[j];
            }
        }
```

```
        }
        return H;
    }

    //QR 分解函数
    void hhqr::qrDecomposition()
    {
        int m = A.size();
        int n = A[0].size();
        vector<vector<double>> R = A;
        Q.assign(m, std::vector<double>(m, 0.0));
        for (int i = 0; i < m; ++i) {
            Q[i][i] = 1.0;
        }

        for (int k = 0; k < n; ++k) {
            vector<double> x(m - k);
            for (int i = 0; i < m - k; ++i) {
                x[i] = R[k + i][k];
            }
            vector<vector<double>> Hk = qrHouseholder(x);
            vector<vector<double>> H(m,vector<double>(m, 0.0));
            for (int i = 0; i < m; ++i) {
                for (int j = 0; j < m; ++j) {
                    if (i < k || j < k) {
                        H[i][j] = (i == j) ? 1.0 : 0.0;
                    }
                    else {
                        H[i][j] = Hk[i - k][j - k];
                    }
                }
            }
            R = matrixMultiply(H, R);
            A = R;
            Q = matrixMultiply(Q, H);
        }
    }
```

QREigen()函数实现 QR 算法，这个算法实现起来非常简单，所有的难点都在豪斯霍尔德变换，代码如下：

```
//第 5 章/HHQR4Eigen/QREigen.h
#pragma once
#include<vector>
#include "HHQR.h"

void QREigen(const vector<vector<double>>& Amtx, double tol, int maxItNum)
{
    int n = Amtx.size();
    vector<vector<double>> A = Amtx;
    vector<double> Adiag(n, 0);
```

```
    double tolerance = 1.0;
    int itNum = 0;
      vector < vector < double >> QT(n,vector < double >(n,0));
for (int i = 0; i < n; ++i)QT[i][i] = 1.0;      //单位阵
    do
    {
        hhqr hq(A, n);
        hq.QRdecom();                           //QR 分解
        A = MxM(hq.getR(), hq.getQ());

       //计算特征向量
        QT = MxM(QT,hq.getQ());
        vector < double > Dsum(n, 0);
        for (int i = 0; i < n; i++)
            Dsum[i] = fabs(Adiag[i] - A[i][i]);

        //计算容差
        tolerance = * max_element(Dsum.begin(), Dsum.end());

        //对角元素即特征值
        for (int i = 0; i < n; ++i)
            Adiag[i] = A[i][i];
        itNum++;
    } while (tolerance > tol && itNum < maxItNum);

    cout << "tol = " << tolerance << " \t iteration num = " << itNum << endl;
    cout << "eigen value" << endl;
    for (int i = 0; i < n; ++i)
    {
            cout << Adiag[i] << endl;
    }
    cout << " eigen value = \n";
    printMtx(QT);
}
```

另外一个可以讨论的是，如果矩阵很大，则对于矩阵乘法可以有很多提高计算效率的方法。由于 C++语言的 vector 存储行优先，对二维数据的乘法中的 b[k][j]按列访问，可能会造成缓存频繁失效，其次可以将 a[i][k]提取到内层循环外，减少冗余读取。如果要进一步优化，则可以考虑把二维 vector 转换为一维 vector 存储。优化后的代码如下：

```
inline vector < vector < double >> mtxMultiply(
    const vector < vector < double >> & A, const vector < vector < double >> & B)
{
    size_t rows = A.size();
    size_t cols = B[0].size();
    size_t inner = A[0].size();
    vector < vector < double >> product(rows, vector < double >(cols));
    if (A[0].size() != B.size())
      {
std::cerr << "error: dimension of matrix not match" << endl;
```

```
    }
    for (size_t i = 0; i < rows; ++i)
        {
            for (size_t k = 0; k < inner; ++k)
            {
                double aik = A[i][k]; //提取公共项
                for (size_t j = 0; j < cols; ++j)
                {
                    product[i][j] += aik * B[k][j];
                }
            }
        }
        return product;
    }
```

【例 5-5】 使用 QR 迭代求矩阵

$$\boldsymbol{A}=\begin{pmatrix}1 & 2 & 0\\ 2 & -1 & 1\\ 0 & 1 & 3\end{pmatrix}$$

的全部特征值。

主程序代码如下：

```
//第 5 章/HHQR4Eigen/QRmain.cpp
#include < iomanip >
#include "QREig.hpp"
int main()
{
    cout.precision(6);
    cout.setf(ios::fixed);
    vector < vector < double > > b{{1, 2, 0},
                                   {2, -1,1},{0, 1, 3}};
     double err = 1.0e-3;
    int max = 200;
    QREigen(b, err, max);
    return 0;
}
```

计算结果如下：

```
Tol = 0.000923
iter num = 17
eigen value
3.372164
-2.370188
1.998024
```

本题精确解为

$$\lambda_1=\frac{1}{2}(1+\sqrt{33})\approx 3.37228132,\quad \lambda_2=2,\quad \lambda_3=\frac{1}{2}(1-\sqrt{33})\approx -2.37228132。$$

可以看出 QR 算法精度较高。

5.6 兰乔斯法

兰乔斯(Lanczos)法是 20 世纪计算数学方向最有影响的方法之一,并且已经在工程中得到了广泛应用。

5.6.1 兰乔斯迭代原理

兰乔斯方法是求解高阶对称正定广义特征值问题部分大的(或小的)特征值和相应的特征向量的有效方法,常用于求解稀疏矩阵。

若能迭代求得三对角阵 $\boldsymbol{T}$ 和正交阵 $\boldsymbol{Q}$,使

$$\boldsymbol{Q}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{Q}=\boldsymbol{T} \tag{5-43}$$

即

$$\boldsymbol{A}\boldsymbol{Q}=\boldsymbol{Q}\boldsymbol{T} \tag{5-44}$$

则对称矩阵 $\boldsymbol{A}$ 有三对角分解

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^{\mathrm{T}} \tag{5-45}$$

其中,$\boldsymbol{Q}=[\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_n]$ 且 $\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{q}_j=\delta_{ij}$。

$$\boldsymbol{T}=\begin{pmatrix}\alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \beta_2 & \\ & \ddots & \ddots & \beta_{n-1}\\ & & \beta_{n-1} & \alpha_n\end{pmatrix} \tag{5-46}$$

通常称 $\boldsymbol{T}$ 为 n 阶兰乔斯矩阵。

比较 $\boldsymbol{A}\boldsymbol{Q}=\boldsymbol{Q}\boldsymbol{T}$,即

$$\boldsymbol{A}(\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_n)=(\boldsymbol{q}_1,\boldsymbol{q}_2,\cdots,\boldsymbol{q}_n)\boldsymbol{T} \tag{5-47}$$

可得[23-25]

$$\boldsymbol{A}\boldsymbol{q}_j=\beta_{j-1}\boldsymbol{q}_{j-1}+\alpha_j\boldsymbol{q}_j+\beta_j\boldsymbol{q}_{j+1}\quad(j=1,2,\cdots,n-1) \tag{5-48}$$

由于 $\beta_0,\beta_n,\boldsymbol{q}_0,\boldsymbol{q}_{n+1}$ 未定义,补充定义 $\beta_0\boldsymbol{q}_0=\beta_n\boldsymbol{q}_{n+1}=0$。在式(5-48)的两边乘以 $\boldsymbol{q}_j^{\mathrm{T}}$ 有

$$\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j=\beta_{j-1}\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{q}_{j-1}+\alpha_j\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{q}_j+\beta_j\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{q}_{j+1} \tag{5-49}$$

由于 $\boldsymbol{Q}$ 正交,所以得

$$\alpha_j=\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j \tag{5-50}$$

如果

$$A\boldsymbol{q}_j-\alpha_j\boldsymbol{q}_j-\beta_{j-1}\boldsymbol{q}_{j-1}=\boldsymbol{r}_i\neq 0 \tag{5-51}$$

则有式(5-48)知

$$\boldsymbol{r}_j=\beta_j\boldsymbol{q}_{j+1} \tag{5-52}$$

即

$$\boldsymbol{q}_{j+1}=\frac{1}{\beta_j}\boldsymbol{r}_j \tag{5-53}$$

对式(5-52)两边取2-范数，得

$$\beta_j=\|\boldsymbol{r}_j\|_2 \tag{5-54}$$

从而对于给定的规范向量 $\boldsymbol{q}_1$，可以通过迭代计算后直接得到 $\boldsymbol{A}$ 的舒尔(Schur)分解，也就是将一个矩阵分解成酉矩阵和上三角矩阵，这种迭代称为兰乔斯迭代。

由此，就可以给出计算方法。只要给定任意的 $\boldsymbol{q}_1\in\mathbb{R}^n$ 且 $\|\boldsymbol{q}_1\|_2=1$，就能够利用递推关系得到全部 $\boldsymbol{q}_i$，α_i，β_i，可得迭代算法如下。

首先初始化

$$\alpha_1=\boldsymbol{q}_1^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_1 \tag{5-55}$$

然后迭代计算

$$\begin{cases}\boldsymbol{r}_i=\boldsymbol{A}\boldsymbol{q}_i-\alpha_i\boldsymbol{q}_i-\beta_{i-1}\boldsymbol{q}_{i-1}\\ \beta_i=\|\boldsymbol{r}_i\|_2\\ \boldsymbol{q}_{i+1}=\dfrac{\boldsymbol{r}_i}{\beta_1}\quad(\beta_i\neq 0)\\ \alpha_{i+1}=\boldsymbol{q}_{i+1}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_{i+1}\end{cases}\qquad(i=1,2,\cdots,n-1) \tag{5-56}$$

此即兰乔斯迭代，其中，$\boldsymbol{q}_i$ 称为兰乔斯向量。在第 j 步产生的矩阵 $\boldsymbol{T}_j$ 称为 j 阶兰乔斯矩阵，其特征值是 $\boldsymbol{A}$ 的部分特征值的近似。

兰乔斯迭代的优点是，不但在计算过程中不破坏矩阵的稀疏性，而且在未完成矩阵三对角化前可以得到某些极端特征值。但也要看到，有时迭代得到的向量序列的正交性较差。

5.6.2 兰乔斯迭代算法

输入：实对称阵 $\boldsymbol{A}$。

输出：三对角阵 $\boldsymbol{T}$。

步骤1，初始化。

$$\boldsymbol{r}_0=\boldsymbol{q}_1\quad(\|\boldsymbol{q}_1\|_2=1),\quad\beta_0=1,\quad\boldsymbol{q}_0=0\quad(j=0)$$

步骤2，当 $j=1,2,\cdots,n-1$ 时，计算步骤3和步骤4。

步骤3，按下面的公式进行计算。

$$\begin{cases}\alpha_j=\boldsymbol{q}_j^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_j\\ \boldsymbol{r}_j=\boldsymbol{A}\boldsymbol{q}_j-\alpha_j\boldsymbol{q}_j-\beta_{j-1}\boldsymbol{q}_{j-1}\\ \beta_i=\|\boldsymbol{r}_j\|_2\end{cases}$$

步骤4，当 $\beta_j=0$ 时，停止计算，否则计算 $\boldsymbol{q}_{j-1}=\boldsymbol{r}_j/\beta_j$。

步骤5，输出结果 $\boldsymbol{T}_n=\boldsymbol{q}_n^{\mathrm{T}}\boldsymbol{A}\boldsymbol{q}_n$，停止计算。

5.6.3 兰乔斯迭代C++代码实现

这个程序使用模板类，LanczosIterator类成员是矩阵 m_A，成员函数只有一个带参数

构造。代码里有很多矩阵和向量计算，使用向量操作的头函数 vectorOperator.h，该函数采用模板编程，主要包括加法、减法、点乘、叉乘，向量和常量相乘，矩阵和向量相乘，打印输出函数。读者也可以调用第 4 章中的 VectorOpt.h 文件来对比一下模板和 inline()函数的使用，代码如下：

```
//第 5 章/Lanczos/vectorOperator.h
/*************************************************************
 * Copyright (C) 2025
 * @brief 向量操作头文件
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 * -----------------------------------------------------------
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *************************************************************
 * 以下是各函数的功能注释
 * -----------------------------------------------------------
 * double dot(const Tv&x, const Tv&y)
 * @brief 向量点乘
 * @param[in] x     向量 1
 * @param[in] y     向量 2
 * @param[out]      点乘积
 * -----------------------------------------------------------
 * double norm2(const Tv&a)
 * @brief 2         范数
 * @param[in] a     向量
 * @param[out]   2 范数
 * -----------------------------------------------------------
 * VectorTimes(const Tv& x, const double C)
 * @brief           向量与常量乘积
 * @param[in] x     向量 1
 * @param[in] C     常量
 * @param[out]      乘积
 * -----------------------------------------------------------
 * Tv VectorSubstract(const Tv&x, const Tv&y)
 * @brief           向量相减
 * @param[in] x     向量 1
 * @param[in] y     向量 2
 * @param[out]      向量差
 * -----------------------------------------------------------
 * Tv VectorAdd(const Tv&x, const Tv&y)
 * @brief           向量相加
 * @param[in] x     向量 1
 * @param[in] y     向量 2
 * @param[out]      向量和
 * -----------------------------------------------------------
 * Tv MatrixMultiVector(const vector<Tv>&A, const Tv&x)
 * @brief 二维向量与一维向量相乘
```

```
 * @param[in] A      二维向量
 * @param[in] x      向量
 * @param[out]       乘积
 * ------------------------------------------------------
 * vector<vector<double>> Transpose(const vector<Tv> &rhs)
 * @brief            二维向量转置
 * @param[in] rhs    二维向量
 * @param[out]       转置结果
 * ------------------------------------------------------
 * vector<Tv> vecCrossMultiply(const Tv& x, const Tv& y)
 * @brief            向量叉乘
 * @param[in] x      向量 1
 * @param[in] y      向量 2
 * @param[out]       叉乘积
 ************************************************************
 */
#pragma once
#include<vector>
#include<iostream>
using namespace std;

template<class Tv = vector<double>>
inline double dot(const Tv&x, const Tv&y)
{
    double sum = 0;
    if (x.size() != y.size())
    {
        cout << "size not match\n";
        exit(1);
    }
    else {
        for (int i = 0; i < x.size(); i++)
        {
            sum += x[i] * y[i];
        }
    }
    return sum;
}

template<class Tv = vector<double>>
inline double norm2(const Tv&a)
{
    return sqrt(dot(a, a));
}

template<class Tv = vector<double>>
inline Tv VectorTimes(const Tv& x, const double C)
{
    Tv res;
    for (int i = 0; i < x.size(); i++) res.push_back(C * x[i]);
    return res;
}
```

```
template < class Tv = vector < double > >
inline Tv VectorSubstract(const Tv&x, const Tv&y)
{
    Tv res;
    if (x.size() != y.size())
    {
        cout << "size not match\n";
        exit(1);
    }
    else
    {
        for (int i = 0; i < x.size(); i++) res.push_back(x[i] - y[i]);
    }
    return res;
}

template < class Tv = vector < double > >
inline Tv VectorAdd(const Tv&x, const Tv&y)
{
    Tv res;
    if (x.size() != y.size())
    {
        cout << "size not match\n";
        exit(1);
    }
    else
    {
        for (int i = 0; i < x.size(); i++)
        res.push_back(x[i] + y[i]);
    }
    return res;
}


template < class Tv = vector < double > >
inline Tv MatrixMultiVector(const vector < Tv > &A, const Tv&x)
{
    Tv tm;
    if (A[0].size() != x.size())
    {
        cout << "size not match\n";
        exit(1);
    }
    else
    {
        for (int i = 0; i < A.size(); i++)
        {
            double sum = 0;
            for (int j = 0; j < A[0].size(); j++)
            {
                sum += A[i][j] * x[j];
                //tm[i] = tm[i] + A[i][j] * x[j];
```

```
                }
                tm.push_back(sum);
            }
        }
        return tm;
    }

    template < class Tv = vector < double > >
    inline void print2Dvec(char * name, const vector < Tv > &m)
    {
        cout << name << endl;
        for (int i = 0; i < m.size(); i++)
        {
            for (int j = 0; j < m[0].size(); j++)
            {
                cout << m[i][j] << "\t";
            }
            cout << endl;
        }
    }

    template < class Tv = vector < double > >
    inline void printVector(const Tv&x)
    {
        for (size_t j = 0; j < x.size(); j++)
        {
            cout << x[j] << "\t";
        }
        cout << endl;
    }

    template < class Tv = vector < double > >
    inline vector < vector < double >> Transpose(const vector < Tv > &rhs)
    {
        size_t n = rhs.size();
        vector < vector < double >> tmp(n, vector < double >(n,0));

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                tmp[i][j] = rhs[j][i];
            }
        }
        return tmp;
    }

    template < typename Tv = vector < double >>
    inline vector < Tv > vecCrossMultiply(const Tv& x, const Tv& y)
    {
        size_t n = x.size();
        vector < Tv > tmp(n, vector < double >(n, 0));
        if (x.size() != y.size())
```

```
    {
        cout << "size not match!!\n";
        exit(1);
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            tmp[i][j] = x[i] * y[j];
        }
    }
    return tmp;
}
```

LanczosIterator 类的头文件，主要就是兰乔斯迭代类，采用模板形式，成员变量为矩阵 m_A。成员函数只有一个 LanczosIterator()，代码如下：

```
//第 5 章/Lanczos/Lanczos.h
/*************************************************************
 * Copyright (C) 2025 SunKunfeng, skf0558@sina.com
 * @brief 兰乔斯类的头文件
 * @author SunKunfeng
 * @date 2025-02-01
 * @version V1.0 20250201
 * -----------------------------------------------------------
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *************************************************************
 * 以下是功能注释
 * ------------------------------------------------------------------------
 * LanczosIterator(const double tol, const int maxCount)
 *@brief                    将矩阵三角化
 *@param[in] m_A            给定矩阵
 *@param[in] tol            容差
 *@param[in] maxCount       最大迭代次数
 *************************************************************
 */
#pragma once
#include "vectorOperator.h"

template<typename Tv = vector<double>>
class LanczosIterator
{
private:
    vector<Tv> m_A;
public:
    LanczosIterator(const vector<Tv>& rhs, const size_t n)
    {
```

```
            m_A = rhs;
            Tv q0(n);
            q0[0] = 1;
            Tv q = VectorTimes(q0, 1 / norm2(q0)); //q1 = q0/||q0||_2
            vector<Tv> r(n, Tv(n, 0));
            vector<Tv> T(n, Tv(n, 0.0));          //存储三对角矩阵 T
            r[0] = q;
            Tv qpre(n), beta(n), alpha(n);
            beta[0] = 1;

            for (int k = 0; k < n; k++)
            {
                r[k] = MatrixMultiVector(m_A, q);
                alpha[k] = dot(q, r[k]);
                Tv aq = VectorTimes(q, alpha[k]);
                r[k] = VectorSubstract(r[k], aq);

                Tv bq(n, 0);
                if (k > 0)
                {
                    bq = VectorTimes(qpre, beta[k - 1]);
                }

                r[k] = VectorSubstract(r[k], bq);
                beta[k] = norm2(r[k]);
                qpre = q;
                q = VectorTimes(r[k], 1 / beta[k]);
            }

            T[0][0] = alpha[0];
            for (int i = 1; i < n; ++i)
            {
                T[i][i] = alpha[i];
                T[i - 1][i] = beta[i - 1];
                T[i][i - 1] = beta[i - 1];

            }
            cout << "T = \n";
            for (int i = 0; i < n; ++i)
            {
                for (int j = 0; j < n; ++j)
                {
                    cout << T[i][j] << "\t";
                }
                cout << endl;
            }
            cout << "alpha = \n";
            printVector(alpha);
            cout << "beta = \n";
            printVector(beta);
        }
    };
```

【例 5-6】 用 Lanczos 法将对称矩阵 $\boldsymbol{A}$ 化为三对角阵

$$\boldsymbol{A}=\begin{pmatrix}1 & 0 & 2\\ 0 & 1 & 2\\ 2 & 2 & -1\end{pmatrix}$$

主程序代码如下：

```
//第 5 章/Lanczos/main.cpp
#include "LanczosIterator.h"
int main()
{
    vector<vector<double>> a = { {1,0,2},
                                 {0,1, 2},
                                 {2, 2, -1} };
    LanczosIterator<vector<double>> Li(a, a.size());
}
```

计算结果如下：

```
T =
1   2    0
2   -1   2
0   2    1
alpha =
1   -1   1
beta =
2   2    0
```

5.6.4 带原点位移 QR 法求特征值原理

基本的 QR 算法与乘幂法一样是线性收敛的，该方法的计算量为 $O(n^3)$，为加快其收敛速度，可使用原点位移方法进行加速，这种方法被称为带原点位移的 QR 算法。

记 $\boldsymbol{A}_1=\boldsymbol{A}$，选取一个适当的实数 t_1，对矩阵 $\boldsymbol{A}_1-t_1\boldsymbol{I}$ 作 QR 分解

$$\boldsymbol{A}_1-t_1\boldsymbol{I}=\boldsymbol{Q}_1\boldsymbol{R}_2 \tag{5-57}$$

令

$$\boldsymbol{A}_2=\boldsymbol{R}_1\boldsymbol{Q}_1+t_1\boldsymbol{I} \tag{5-58}$$

然后选取适当的实数 t_2，对矩阵 $\boldsymbol{A}_2-t_2\boldsymbol{I}$ 作 QR 分解

$$\boldsymbol{A}_2-t_2\boldsymbol{I}=\boldsymbol{Q}_2\boldsymbol{R}_2 \tag{5-59}$$

进而，令

$$\boldsymbol{A}_3=\boldsymbol{R}_2\boldsymbol{Q}_2+t_2\boldsymbol{I} \tag{5-60}$$

一般地，设已得到 $\boldsymbol{A}_m$，则选取适当的实数 t_m，对 $\boldsymbol{A}_m-t_m\boldsymbol{I}$ 作 QR 分解

$$\boldsymbol{A}_m-t_m\boldsymbol{I}=\boldsymbol{Q}_m\boldsymbol{R}_m \tag{5-61}$$

令

$$\boldsymbol{A}_{m+1}=\boldsymbol{R}_m\boldsymbol{Q}_m+t_m\boldsymbol{I} \tag{5-62}$$

以此类推，称 t_m 为第 m 次迭代的原点平移量。据式(5-61)和式(5-62)，有

$$\boldsymbol{A}_{m+1}=\boldsymbol{Q}_m^{\mathrm{T}}(\boldsymbol{A}_m-t_m\boldsymbol{I})\boldsymbol{Q}_m+t_m\boldsymbol{I}=\boldsymbol{Q}_m^{\mathrm{T}}\boldsymbol{A}_m\boldsymbol{Q}_m \tag{5-63}$$

即 $\boldsymbol{A}_{m+1}$ 与 $\boldsymbol{A}_m$ 相似，因此序列 $\{\boldsymbol{A}_m\}$ 中的每个矩阵都与原矩阵相似，从而它们的特征值相同。一般地，$\boldsymbol{A}_m$ 的主对角线以下的元素都趋于零。适当选取平移量 t_m，可使 $\boldsymbol{A}_m$ 最后一行的非对角元素更迅速地趋于零。

如果使用 Lanczos 迭代得到 $\boldsymbol{A}$ 的三对角矩阵 $\boldsymbol{T}$ 和正交阵 $\boldsymbol{Q}_n$

$$\boldsymbol{Q}_n^{\mathrm{T}}\boldsymbol{A}\boldsymbol{Q}_n=\boldsymbol{T} \tag{5-64}$$

对 $\boldsymbol{T}$ 使用 QR 迭代，即

$$\begin{cases}\boldsymbol{A}\boldsymbol{Q}_n=\boldsymbol{Q}_n\boldsymbol{T}^k\\ \boldsymbol{T}^{k+1}=\boldsymbol{R}\boldsymbol{Q}\end{cases} \tag{5-65}$$

可以求得特征值。为了加快收敛速度，可以选用带位移的 QR 迭代。

当 $k=1,2,\cdots$ 时，执行

$$\begin{cases}\boldsymbol{T}^k-t\boldsymbol{I}=\boldsymbol{Q}\boldsymbol{R}\\ \boldsymbol{T}^{k+1}=\boldsymbol{R}\boldsymbol{Q}+t\boldsymbol{I}\end{cases} \tag{5-66}$$

位移 t 的选取方法有两种[18-19]：

(1) $t=\alpha_n$。

(2) 取 t 为矩阵 $\begin{bmatrix}\alpha_{n-1} & \beta_{n-1}\\ \beta_{n-1} & \alpha_n\end{bmatrix}$ 靠近 α_n 的特征值，计算公式为

$$t=\alpha_n+a-\mathrm{sgn}(a)\sqrt{a^2+\beta_{n-1}^2} \tag{5-67}$$

其中，$a=\dfrac{1}{2}(\alpha_{n-1}-\alpha_n)$。

5.6.5 带原点位移 QR 法实现与算例

为了便于计算位移，将 QREig.hpp 文件中的 void QREigen()修改为 vector<double> QREigen()，将 Adiag 作为返回值，代码如下：

```
//第 5 章/Lanczos_shift/shift_Main.cpp
#include<iomanip>
#include "QREig.hpp"
#include "Lanczos.h"
//移位法
void shiftQR(vector<vector<double>> T1, double err, int max, int n)
{
    vector<double> Eig(n, 0);
    double t = T1[n - 1][n - 1];
    //cout << "t = " << t << endl;
    for (int i = 0; i < n; i++) T1[i][i] -= t; //移位
    Eig = QREigen(T1, err, max);
    cout << "shift QR method Eigen Value = " << endl;
```

```
    for (auto& x : Eig) {
        x += t;
        cout << x << endl;
    }
}
```

【例 5-7】 使用简单 QR 法和带原点位移 QR 法求

$$\boldsymbol{A}=\begin{pmatrix} 5 & 1 & 2 & 2 \\ -2 & 6 & -1 & 3 \\ 1 & 3 & 4 & -1 \\ 2 & -4 & -2 & 6 \end{pmatrix}$$

的全部特征值,要求容差为 10^{-6}。

主程序代码如下:

```
//第 5 章/Lanczos_shift/shift_Main.cpp
//void shiftQR(),略

int main()
{
    cout.precision(6);
    cout.setf(ios::fixed);
    vector < vector < double > > b{{ 5,1, 2, 2},{ -2,6, -1,3},
                                   {1,3,4, -1},{2, -4, -2,6}};

    double err = 1.0e-6;
    int max = 500;
    int n = b.size();
    LanczosIterator < vector < double >> Lan(b, n);
    vector < vector < double > > T = Lan.getT();
    cout << "simple QR method = \n";
    vector < double > Eig = QREigen(T, err, max);
    shiftQR(T, err, max, n);                    //shift QR 计算特征值
    return 0;
}
```

两种 QR 算法的计算结果如下:

```
simple QR method =
tol = 0.000001 iteration num = 213
14.129648
7.699325
-2.834363
2.746902
tol = 0.000001 iteration num = 88
shift QR method Eigen Value =
-2.834372
14.129643
2.746916
7.699325
```

从结果看，迭代次数明显减少，计算效率得到大大提升。

5.7 主成分分析法

人工智能领域的一个最主要挑战就是对海量数据进行处理，在实际应用中，数据集往往包含大量特征（维度），例如，图像数据可能有数百万像素作为特征，基因数据可能包含成千上万的基因表达值。高维度数据会带来“维度灾难”（Curse of Dimensionality），从而导致计算复杂度急剧增加，模型容易过拟合，并且难以解释。另外原始数据中的特征可能是复杂的、不直观的，甚至难以理解它们对数据的贡献程度，例如，在金融数据中，各种经济指标可能相互交织，难以直接判断哪些因素是最重要的。此时就需要对数据进行降维处理。

5.7.1 主成分分析原理

主成分分析（Principal Component Analysis，PCA）是一种经典的统计方法，是一种线性降维技术，它被广泛地应用于数据降维、特征提取和数据可视化等领域。

主成分分析的基本原理是通过正交变换将高维数据转换为低维特征空间，同时最大化保留原始数据的方差（信息），其本质是寻找数据的主成分方向，实现维度压缩。

主成分（PC）是原始变量的线性组合或混合构建的新变量，代表了数据中能够解释最大方差的方向。PCA 通过线性变换将原始数据从 N 维降为 k 维（$N>k$），得到一组 k 个线性无关（正交）的变量，这 k 个变量被称为主成分，按照方差递减的顺序排列，常被命名为 PC1，PC2，…，PCk。

另外还有一些其他种类的降维技术，例如因子分析（Factor Analysis）和独立成分分析（Independent Component Analysis，ICA）[30]。目前在实际应用中 PCA 方法使用最为广泛，它可以大大地降低数据的复杂性，识别出最重要的特征，特别适用于数值型的数据。

PCA 分析通常可分为 5 个步骤。

1）数据标准化

将数据集中的每列（变量）减去其均值后除以标准差，使每个特征的均值为 0，并使标准差为 1。数据标准化首先可以消除尺度影响，确保各个变量的方差相等，保持其对主成分的贡献，这使数据以原点为中心，其次可以消除量纲差异，避免单位大的变量主导分析结果，标准化后协方差矩阵和相关矩阵就等价了。这对 PCA 很重要，因为它旨在找到数据中的重要性轴，如果想从 PCA 编码的数据中恢复未居中的数据，则需要保留此平均值，这样才能将其添加回去。

2）计算协方差矩阵

协方差矩阵反映了原始数据中各个变量之间的相关性，以此来观察各指标之间是否包含冗余信息。

3）计算特征值和特征向量

使用前面介绍的方法计算协方差矩阵的特征值和对应的特征向量，其中特征向量就是

方差最大(信息最多)的轴的方向,称为主成分。特征值反映了每个主成分所携带的方差量。

4) 选择主成分

根据特征值的大小,选择前 k 个特征向量作为主成分。首先计算方差贡献率和累计方差贡献率,然后当贡献率达到一定阈值时来确定保留的主成分数量。

将特征向量先从大到小排序,然后可以使它们成为矩阵的列。这就是用来将数据转换为“PCA 空间”的矩阵——主成分系数矩阵。要降低维度,既可以在此时丢弃右侧的列,也可以等到将数据转换为新空间后再决定丢弃多少列。

5) 恢复数据输出结果

将原始数据投影到所选择的主成分上,在低维空间上观察样本的分布、聚类情况和异常值等,制作各主成分对解释总体方差的贡献条形图等。

5.7.2 计算协方差矩阵

在数学上,两个随机变量的协方差表示它们之间的线性相关程度,其计算公式为

$$\mathrm{Cov}(X,Y)=\frac{\sum_{i=1}^{n}(X_i-\overline{X})(Y_i-\overline{Y})}{(n-1)}$$

对于向量 $\boldsymbol{X}$ 它自身的协方差就是

$$\mathrm{Cov}(X,X)=\frac{\sum_{i=1}^{n}(X_i-\overline{X})(X_i-\overline{X})}{(n-1)}$$

协方差矩阵具有对称性、半正定性,其对角线元素为方差、秩的非负性、特征值的非负性,协方差矩阵的行列式还具有非负等特性。

5.7.3 数据协方差矩阵算法

输入:测试样本集$\{x_1,x_2,\cdots,x_n\}$,低维空间维数 d。

输出:协方差矩阵,中心化矩阵。

步骤 1,数据标准化,计算数据均值

$$\overline{X}=\frac{1}{n}\sum_{i=1}^{n}x_i$$

步骤 2,样本去中心化,计算中心化矩阵

$$Z_i=X_i-\overline{X}\quad(i=1,\cdots,n)$$

步骤 3,计算协方差矩阵ZZ^{T}

$$\mathrm{CovMtx}=\frac{1}{n-1}\sum_{i=1}^{n}Z_iZ_i'$$

5.7.4 协方差矩阵 C++实现

代码中有两个函数,一个是打印输出矩阵函数 printMtx(),另一个是计算协方差矩阵

函数 covMtx()，代码如下：

```
//第 5 章/PCA/covariancMtx.hpp
/**
 ****************************************************************
 * Copyright (C)
 * @brief 计算协方差矩阵
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 ****************************************************************
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * ---------------------------------------------------------------
 */
#pragma once
#include<vector>
#include<iostream>
using namespace std;
void printMtx(const vector<vector<double>>& a)
{
    int n = a.size();
    int m = a[0].size();
    cout << " mtx = " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cout << a[i][j] << "\t";
        }
        cout << endl;
    }
}
void covMtx(vector<vector<double>>& mtx)
{
    size_t m = mtx.size();
    size_t n = mtx[0].size();

    //(1)计算均值向量(以列向量为例)
    vector<double> E(n);
    for (int i = 0; i < n; ++i)
    {
        double sum = 0;
        for (int j = 0; j < m; ++j)
        {
            sum += mtx[j][i];
        }
        E[i] = sum / m;
    }
```

```
        cout << "mean value = \n";
        for (int j = 0; j < n; ++j) cout << E[j]<< "\t";
        cout << endl;

        //(2)样本中心化矩阵
        vector < vector < double >> centrMtx(m, vector < double >(n));
        vector < vector < double >> centrMtxT(n, vector < double >(m));
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                centrMtx[i][j] = mtx[i][j] - E[j]; //z = x_i - x_bar
                centrMtxT[j][i] = centrMtx[i][j];
            }
        }

        //(3)求协方差矩阵,1/(m - 1) * centrMtxT * centrMtx
        vector < vector < double >> covMtx(n, vector < double >(n));
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                for (int k = 0; k < m; k++)
                {
                    covMtx[i][j] += centrMtxT[i][k] * centrMtx[k][j];
                }
                covMtx[i][j] /= (m - 1.0);
            }
        }
    }
```

【例 5-8】 求下面数据集

$$\boldsymbol{A}=\begin{pmatrix}4.0 & 2.0 & 0.6\\ 4.2 & 2.1 & 0.59\\ 3.9 & 2.0 & 0.58\\ 4.3 & 2.1 & 0.62\\ 4.1 & 2.2 & 0.63\end{pmatrix}$$

的协方差矩阵。

主程序代码如下：

```
//第 5 章/PCA/Cmtx.cpp
#include "covariancMtx.hpp"
int main()
{
    cout.setf(ios::fixed);
    cout.precision(6);
    vector < vector < double > > aT = {{4.0, 2.0, 0.6},
                                       {4.2, 2.1, 0.59},
                                       {3.9, 2.0, 0.58},
```

```
                                        {4.3, 2.1, 0.62},
                                        {4.1, 2.2, 0.63}};
    covMtx(aT);
    cout << endl << "协方差矩阵: " << endl;
    printMtx(covMtx);
    return 0;
}
```

计算结果如下：

```
协方差矩阵
mtx =
0.025000   0.007500   0.001750
0.007500   0.007000   0.001350
0.001750   0.001350   0.000430
```

5.7.5 PCA 算法

输入：测试样本集$\{x_1,x_2,\cdots,x_n\}$，低维空间维数 d。

输出：降维后的数据。

步骤 1，计算协方差矩阵和中心化矩阵。

步骤 2，计算矩阵特征值和特征向量。

步骤 3，将特征值排序。

步骤 4，保留前 d 个最大特征值和对应的特征向量 $\boldsymbol{W}_i$。

步骤 5，计算投影矩阵 $\boldsymbol{W}=[w_1,w_2,\cdots,w_d]$。

步骤 6，恢复数据，用中心矩阵乘以投影矩阵，得到降维后的数据。

5.7.6 通用算法库和 ranges 库

1. 通用算法库

C++通用算法库(< algorithm >)自 C++98 被正式纳入标准库，C++11、C++17、C++20 不断为其新增了算法和优化。它是 C++标准模板库(STL)的核心组成部分之一，提供了大量通用算法，用于操作容器(如 vector、list)中的数据。这些算法包括多种操作。

(1) 排序：如 std::sort、std::stable_sort。

(2) 查找：如 std::find、std::binary_search、std::lower_bound。

(3) 遍历：如 std::for_each、std::count。

(4) 修改：如 std::transform、std::replace。

(5) 其他：如 std::merge、std::nth_element。

使用< algorithm >库，首先要包含头文件，代码如下：

```
#include <algorithm>      //核心算法头文件
#include <vector>         //示例容器头文件
```

算法通常接收多个参数：起始迭代器(begin())、结束迭代器(end()，左闭右开区间)、可选参数(如排序的比较函数、替换值等)。下面是示例代码：

```
std::vector<int> v = {3, 1, 4, 1, 5};
//排序
std::sort(v.begin(), v.end());                          //升序
//查找元素
auto it = std::find(v.begin(), v.end(), 4);
if (it != v.end()) *it = 10;                            //替换找到的元素
//统计出现次数
int count = std::count(v.begin(), v.end(), 1);          //count = 2
//升序排序
std::sort(v.begin(), v.end());
//降序排序
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
//稳定排序(保持相等元素的相对顺序)
std::stable_sort(v.begin(), v.end());
```

通过合理使用<algorithm>库，可以极大地简化代码，显著提高代码的可读性和效率，建议结合标准容器的迭代器接口灵活运用。

注意

范围正确性：始终确保迭代器不越界(如 end() 不能解引用)。

随机访问迭代器：部分算法(如 std::sort)需要容器支持随机访问(如 vector、array)。

稳定性：某些操作(如 std::stable_sort)会保留原有顺序。

复杂度：例如 std::sort 为 O(n log n)，std::find 为 O(n)。

2. ranges 库

ranges 库是 C++20 标准的一部分，它为 C++标准库引入了一种新的范式，旨在提供更现代、更安全、更高效的方式来处理序列和算法。范围的定义非常灵活，它可以是一个数组、一个标准容器(如 std::vector、std::list 等)、一个字符串。ranges 库中的范围算法是对传统 STL 算法的扩展和改进。它们直接作用于范围(ranges)，而不是迭代器对(Iterator Pairs)，从而使代码更加简洁和易于理解。这些算法涵盖了排序、搜索、变换、统计等多种操作，旨在提供更高效、更安全的序列处理能力。

ranges 库都有统一的接口，接受范围作为参数，这使它们的使用更加一致和直观。许多范围算法支持惰性求值，即只有在需要时才进行计算，这有助于提高性能。通过减少裸迭代器的使用，范围算法降低了出错的风险。

ranges 库支持的常用算法有以下几种。

(1) 排序算法：C++的 ranges 库提供了多种排序算法，使对序列的排序更加方便和高效。ranges::sort 对给定范围内的元素进行排序，默认使用 operator<进行比较；ranges::stable_sort 对范围内的元素进行稳定排序，保持相等元素的相对顺序；ranges::partial_sort 对范围内的一部分元素进行排序，使前 N 个元素是整个范围中最小的 N 个元素。

（2）搜索算法：如 ranges::find 在范围内查找指定的元素。ranges::find_if 在给定范围内查找满足特定条件的元素，条件通过一个谓词函数指定。ranges::binary_search 对已排序的范围进行二分搜索，检查指定的值是否存在。

（3）变换算法：如 ranges::transform，对范围内的每个元素应用给定的函数。

（4）统计算法：如 ranges::count，计算范围内满足特定条件的元素数量。

（5）数值算法：ranges::accumulate 计算给定范围内所有元素的累加值，可以指定一个初始值和一个二元操作函数。ranges::reduce 与 accumulate 类似，但是更适合并行计算。ranges::inner_product 计算两个范围的内积，即对应元素乘积的累加值。ranges::partial_sum 计算给定范围内每个元素的前缀和，并将结果存储在另一个范围中。

下面是一个使用 ranges::sort 对一个 std::vector 进行排序的示例：

```
#include <vector>
#include <iostream>
#include <ranges>
int main() {
    std::vector<int> v = {5, 3, 1, 4, 2};
    std::ranges::sort(v);
    for (int i : v) {
        std::cout << i << " ";
    }
    return 0;
}
```

5.7.7 PCA 算法 C++实现

特征值的排序使用 ranges 库和排序算法，需要注意的是 ranges 库需要 C++20 标准的支持。特征值排序使用 stable_sort()函数，该函数相对于 sort()函数，其值可以重复，代码如下：

```
//第 5 章/PCA_57/pca.h
/**
 ****************************************************************
 * Copyright (C)
 * @brief PCA 算法
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025-02-01
 * @version V1.0 20250201
 ****************************************************************
 * License-Identifier: LGPL-3.0-or-later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * ---------------------------------------------------------------
 */
#pragma once
#include <algorithm>
#include <ranges>//c++20
#include "Matrix.h"
#include "Jacobi.h"
```

```
//用 Jacobi 法求 A 特征值和特征向量
template < typename Tv = Matrix >
class PCA : public Matrix
{
public:
    size_t m, n, r;
    vector < double > eigenVal;
    Tv EigenVec;
    Tv A;
public:
    PCA() {};
    ~PCA() {};
    PCA(const Tv& arr, int row, int col, int MaxIteNum, double eps);
    void sortedEigen();
    size_t rank();
    void recovery(size_t Num, Matrix& Xc); //恢复阶次,协方差中心矩阵
};

template < typename Tv >
size_t PCA < Tv >::rank()
{
    size_t rank = 0;
    for (size_t i = 0; i < eigenVal.size(); i++)
    {
        if (fabs(eigenVal[i]) > 1e - 30) rank++;
    }
    return rank;
}

template < typename Tv >
void PCA < Tv >::recovery(size_t Num, Matrix& Xc)
{
    //取前 N 项特征向量
    vector < vector < double >> v2(n, vector < double >(Num));
    //Matrix V2(n, Num);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < Num; j++)
            v2[i][j] = EigenVec.m_matrix[i][j];

    //恢复数据
    Matrix mtxV2(v2, n, Num);              //转换为矩阵,利用矩阵乘法
    Matrix arr2T = Xc * mtxV2;

    //输出结果
    cout << "pca = " << endl;
    arr2T.printMatrix();
}

template < typename Tv >
PCA < Tv >::PCA(const Tv& arr, int row, int col, int MaxIteNum, double eps)
{
    //各个变量初始化 m,n,A,eigenVal
    m = row;
```

```
        n = col;
        r = min(m, n);
        A = arr;
        eigenVal.assign(n,0);

        //计算 A 特征值和特征向量
        Jacobi Ja(A, eps, MaxIteNum);
        EigenVec = Ja.eigenVec;

        cout << "eigen value :\n";
        for (int i = 0; i < n; i++)
        {
            eigenVal[i] = Ja.a.m_matrix[i][i];
        }

        sortedEigen();
        cout << "sorted eigen value :\n";
        for (int i = 0; i < n; i++)
        {
            eigenVal[i] = Ja.a.m_matrix[i][i];
            cout << eigenVal[i] << endl;
        }
        EigenVec.printMatrix();
    }

    //特征值排序
    template < typename Tv >
    void PCA < Tv >::sortedEigen()
    {
        //创建一个结构体
        struct EigenRecord {
            double val;
            vector < double > vec;
        };

        vector < EigenRecord > data(n);
        for (int i = 0; i < n; i++)
        {
            data[i].val = eigenVal[i];
            data[i].vec = EigenVec.m_matrix[i];
        }

        //排序
        std::ranges::stable_sort(data, std::greater <>{}, & EigenRecord::val);
        Tv tmp(n, n);
        for (int i = 0; i < n; i++)
        {
            eigenVal[i] = data[i].val;
            tmp.m_matrix[i] = data[i].vec;
        }
        EigenVec = tmp;
    }
```

【例 5-9】 二维数据集

$$\boldsymbol{X}=\begin{pmatrix}-1 & -1 & 0 & 2 & 0\\ -2 & 0 & 0 & 1 & 1\end{pmatrix}$$

使用 PCA 算法将这组二维数据降维成一维数据。

主程序代码如下：

```
//第 5 章/PCA_57/PCAmain.cpp
#include < iostream >
#include < vector >
#include "covariancMtx.hpp"
#include "readFileData.hpp"
#include "PCA.h"
using namespace std;
int main()
{
    cout.setf(ios::fixed);
    cout.precision(6);
    vector < vector < double >> data{ { - 1, - 2},{ - 1,0} ,{0,0},{2,1},{0,1} };
            int n = data[0].size();
        int Num = 1;
    vector < vector < double >> covM(n, vector < double >(n)); //协方差矩阵
    pair < vector < vector < double >>, vector < vector < double >>> result = covMtx(data);
    covM = result.first;
    vector < vector < double >> Xc = result.second; //centrMtx
    cout << "协方差矩阵: \n";
    printMtx(covM);
    //求特征值和特征向量
    int maxIt = 500;
    double eps = 1e - 10;
    int row = covM.size();
    int col = covM[0].size();
    Matrix arr(covM, row, col);          //把 vector 转换为 matrix
    Matrix arr2Xc(Xc);                   //由于 PCA 只接受 matrix
    PCA < Matrix > p1(arr, row, col, maxIt, eps);
    p1.recovery(Num,arr2Xc);             //取前 Num 项特征向量
    return 0;
}
```

计算结果如下：

```
协方差矩阵:
  mtx =
1.500000   1.000000
1.000000   1.500000
sorted eigen value :
2.500000
0.500000
pca =
 - 2.121320
 - 0.707107
```

```
0.000000
2.121320
0.707107
```

5.8 特征值算法应用

最早的搜索引擎采用分类目录的方法，即通过人工对网页进行分类并整理出高质量的网站，例如早期的 Yahoo。后来由于网络的高速发展，网页数量多到使用人工分类已经不可能实现了。谷歌创始人拉里·佩奇和谢尔盖·布林，当时他们还是斯坦福大学（Stanford University）的研究生，他们借鉴了评价学术论文重要性的方法——引用次数，于是 PageRank 算法的核心思想诞生了。PageRank 算法是网络分析和信息检索中的重要算法，尽管已有改进版本，但其核心思想仍被广泛地应用。

5.8.1 PageRank 算法

1. 原理

PageRank 算法的结果是由 WWW 链接结构决定的，并且不涉及任何页面的内容。它通过分析网页之间的链接关系来评估网页的权重，首先将整个互联网看作一张有向图，每个网页都有一个权威性得分，称作 PageRank。可以把它当作是投票权，将每个超链接作为一次投票，每个网页的 PageRank 等于所有具有指向该网页超链接的网页的 PageRank 的加权，这些权值等于这些网页各自向外链接数目的倒数，总结起来主要有以下两点：

（1）一个网页被其他高质量网页链接的次数越多，其权重越高。

（2）来自高权重网页的链接比低权重网页的链接贡献更大。

该算法具有非线性和全局性等特点，其网页的权重值依赖于整个网络。要计算 PageRank，必须先假设一个随机上网过程，即每次看完当前网页后，有两种选择：

（1）在当前网页中随机选一个超链接进入下一个网页。

（2）随机新开一个网页。

这在数学上称为马尔可夫过程（Markov Process）。若这样的随机“冲浪”一直进行下去，某个网页被访问的极限概率就是它的 PageRank。PageRank 的计算方法是，预先给定每个页面一个 PageRank（PR）值，从理论上来讲，PR 值是一个网页的访问概率，初始值一般是 $1/n$，n 是 Internet 中所有可访问网页的数目。此外，所有网页的 PR 值的总和应为 1。

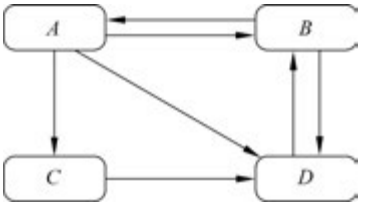

图 5-1 网页链接示意图

下面以一个简单的网络为例说明 PageRank 算法，一个由 4 个网页组成的网络：A、B、C 和 D，如图 5-1 所示。

假设，所有页面都只链接至 A，那么 A 的 PR 值将是 B、C 及 D 的 PR 值之和，即

$$\mathrm{PR}(A)=\mathrm{PR}(B)+\mathrm{PR}(C)+\mathrm{PR}(D) \quad (5\text{-}68)$$

重新假设，B 链接到 A 和 C，C 链接到 A，并且 D 链

接到A、B、C。由于最初一个页面总共只有一票，所以B给A、C每个页面半票。以此类推，D投出的票只有三分之一加到了A的PR值上。

$$\mathrm{PR}(A)=\frac{\mathrm{PR}(B)}{2}+\frac{\mathrm{PR}(C)}{1}+\frac{\mathrm{PR}(D)}{3} \tag{5-69}$$

换句话说，算法将根据每个页面连出的总数$L(x)$平分该页面的PR值，并将其加到所指向的页面

$$\mathrm{PR}(A)=\frac{\mathrm{PR}(B)}{L(B)}+\frac{\mathrm{PR}(C)}{L(C)}+\frac{\mathrm{PR}(D)}{L(D)} \tag{5-70}$$

为了处理那些“没有外部链接的页面”(这些页面就像“黑洞”一样吞噬掉用户继续向下浏览的概率)所带来的问题，此时PR值会是0，而这会导致指向它的页面的PR值的计算结果同样为0。那么假设这类页面链接到集合中所有的网页(不管它们是否相关)，使这类网页的PR值被所有网页均分，称这种情况为残差概率(Residual Probability)，为处理这种网页引入了阻尼系数d，d的大小由一般上网者使用浏览器书签功能的频率的平均值估算得到，通常取$d=0.85$；其意义是：任意时刻，用户访问某页面后继续访问下一个页面的概率，即随机浏览新网页的概率。这样由于“没有向外链接的网页”传递出去的页面，所以就赋给一个最小值$(1-d)/N$。

综上，对于某个页面i，其对应的PR值大小的计算公式如下：

$$\mathrm{PR}(p_i)=\frac{1-d}{N}+d\sum_{p_j\in M(p_i)}\frac{\mathrm{PR}(p_j)}{L(p_j)} \tag{5-71}$$

这里，$p_1,p_2,\cdots,p_N$是目标页面p_i，$M(p_i)$是链入p_i页面的集合，$L(p_j)$是页面p_j链出页面的数量，而N是集合中所有页面的数量。

集合中所有页面的PR值可以组成一个特殊的转置矩阵$\boldsymbol{A}$，$\boldsymbol{A}$由一系列的特征向量$\boldsymbol{x}$表示。

$$\boldsymbol{x}=\begin{pmatrix}\mathrm{PR}(p_1)\\ \mathrm{PR}(p_2)\\ \vdots\\ \mathrm{PR}(p_N)\end{pmatrix} \tag{5-72}$$

计算PageRank的过程可以用数学语言描述，设$x_i^{(k)}(i=1,2,\cdots,n)$，它表示某时刻$k$浏览网页$i$的概率，向量$\boldsymbol{x}^{(k)}$表示当前时刻浏览各网页的概率分布，那么下一时刻浏览到网页i的概率为$\sum_{j=1}^{n}a_{ij}x_j^{(k)}$，此时浏览各网页的概率分布为$\boldsymbol{x}^{(k+1)}=\boldsymbol{A}\boldsymbol{x}^{(k)}$。

这个过程无限进行下去，最后达到极限情况，即网页访问概率$\boldsymbol{x}^{(k)}$收敛到一个极限值，这个极限向量$\boldsymbol{x}$为各网页的PR(PageRank)，它满足$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{x}$，并且$\sum_{i=1}^{n}x_i=1$。

从前面的知识可以知道这个算法类似于求特征值，特别地，对于计算这种规模矩阵的所

有特征值和特征向量是非常困难的(按量级与工作量和储存量)。幸运的是,PageRank 算法仅需要求主特征向量,此时就可以使用幂法求解,更重要的是,幂法并不需要储存整个矩阵。事实上,只要知道 p 和 n,$\boldsymbol{A}$ 的非零元素及每个页面的输出度就足够了。

为了让算法更有普遍性且方便数学表述,首先定义一个 $n\times n$ 的矩阵 $\boldsymbol{G}=(g_{ij})\in\mathbb{R}^{n\times n}$,若网页 j 有一个到网页 i 的链接,则 $g_{ij}=1$,否则 $g_{ij}=0$,称这样的矩阵为网页链接矩阵。网页链接矩阵 $\boldsymbol{G}$ 主要有以下特点[11]:

(1) $\boldsymbol{G}$ 矩阵是大规模稀疏矩阵。

(2) 第 j 列非零元素的位置表示从网页 j 链接出去的所有网页。

(3) 第 i 行非零元素的位置表示所有链接到网页 i 的网页。

(4) $\boldsymbol{G}$ 矩阵中非零元素的数目为整个 Internet 中存在的超链接的数量。

(5) 记 $\boldsymbol{G}$ 矩阵行元素之和 $r_i=\sum_j g_{ij}$,它表示第 i 个网页的“入度”。

(6) 记 $\boldsymbol{G}$ 矩阵列元素之和 $c_j=\sum_i g_{ij}$,它表示第 j 个网页的“出度”。

使用矩阵 $\boldsymbol{D}$ 表示各个网页出度的倒数(若没有出度,则设为 1)构成的 n 阶对角矩阵,$\boldsymbol{e}$ 为全是 1 的 n 维向量,则可以将式(5-71)应用到任意两个网页之间的转移概率,形成一个转置矩阵 $\boldsymbol{A}=(a_{ij})_{n\times n}$。

$$\boldsymbol{A}=d\boldsymbol{G}\boldsymbol{D}+\frac{(1-d)}{n}\boldsymbol{e}\boldsymbol{e}^{\mathrm{T}} \tag{5-73}$$

若 $c_j=0$,则意味着 $g_{ij}=0$,上式改为 $a_{ij}=1/n$。

2. PageRank 算法

输入:网页链接矩阵 $\boldsymbol{G}$ 邻接矩阵 $\boldsymbol{M}$,阻尼因子 d,迭代次数,收敛容差 tol。

输出:PageRank 值向量。

步骤 1,初始化,为每个网页分配相同的初始 PR 值。

步骤 2,计算网页出度倒数 $\boldsymbol{D}$。

步骤 3,使用式(5-73)计算转置矩阵 $\boldsymbol{A}$,即每个网页的 PR 值。

步骤 4,使用幂法求特征值和特征向量。

步骤 5,输出结果 PageRank 值向量。

3. make_pair 用法

由于 PageRank 需要最大特征值和相应的特征向量,为了方便,对幂法的返回进行了改进,这里使用 make_pair。

make_pair 是 C++标准库 <utility>头文件中定义的一个函数模板,其作用是创建一个 std::pair 对象,它的基本语法格式如下:

```
template < class T1, class T2 >
std::pair < T1,T2 > make_pair(T1 t, T2 u);
```

首先它接收两个参数 t 和 u,然后返回一个 std::pair < T1,T2 >对象,其中 T1 和 T2

是根据传入参数的类型自动推导得出的。

借助 std::make_pair 函数来创建 std::pair 对象，其类型会依据传入的参数自动推导。可以通过 .first 和 .second 成员访问 std::pair 中的第 1 个和第 2 个元素，代码如下：

```
//创建一个包含 int 和 string 的 pair
auto pair1 = std::make_pair(10, std::string("hello"));
//输出 pair 中的元素
std::cout << "First element: " << pair1.first << std::endl;
std::cout << "Second element: " << pair1.second << std::endl
```

无须明确指定 std::pair 的类型，编译器会自动推导，这能简化代码编写。相比于直接使用 std::pair 的构造函数，std::make_pair 更简洁。

4. PageRank 算法 C++实现

代码包含打印输出函数 printMtx()、pageRank()函数，以及修改后的幂法函数。幂法函数输出采用了 pair、pair < double，vector < double >> powerMethod()，同时可以返回特征值和特征向量，代码如下：

```
//第 5 章/pagerank/PageRank.cpp
/**
 ******************************************************************
 * Copyright (C) 2025
 * @brief PageRank 算法
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025 - 02 - 01
 * @version V1.0 20250201
 * ----------------------------------------------------------------
 * License - Identifier: LGPL - 3.0 - or - later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 ******************************************************************
 * 以下是函数的功能注释
 * ----------------------------------------------------------------
 * void PageRank()
 * @brief 简单 PageRank 算法
@param[in] 邻接矩阵 M
@param[in] 阻尼因子 d
@param[in] 迭代次数 iterations
@param[in] 容差 tolerance
@param[out] PageRank 值向量
 * ----------------------------------------------------------------
 * void powerMethod ()
 * @brief 幂法求矩阵的最大特征值和对应的特征向量
@param[in] 矩阵 A
@param[in] 初始向量 v
@param[in] 迭代次数 iterations
@param[in] 容差 tolerance
@param[out] 最大特征值和对应的特征向量
```

```
 * ----------------------------------------------------------------
 **/

#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

void printMtx(const vector<vector<double>>& arr)
{
    for (int i = 0; i < arr.size(); i++) {
        for (int j = 0; j < arr[0].size(); j++) {
            std::cout << arr[i][j]<<"\t";
        }
        std::cout << '\n';
    }
}

std::pair<double, std::vector<double>> powerMethod(const std::vector<std::vector
<double>>& A, std::vector<double>& v, int iterations, double tolerance) {
    int n = A.size();
    std::vector<double> prev_v(n);
    double lambda = 0.0;
    for (int i = 0; i < iterations; ++i) {
        prev_v = v;
        //矩阵向量乘法
        for (int j = 0; j < n; ++j) {
            v[j] = 0.0;
            for (int k = 0; k < n; ++k) {
                v[j] += A[j][k] * prev_v[k];
            }
        }
        //归一化向量
        double norm = 0.0;
        for (int j = 0; j < n; ++j) {
            norm += v[j] * v[j];
        }
        norm = std::sqrt(norm);
        for (int j = 0; j < n; ++j) {
            v[j] /= norm;
        }
        //再归一化总和 1
        double sum = 0;
        for (int j = 0; j < n; ++j) {
            sum += v[j];
        }
        for (int j = 0; j < n; ++j) v[j] /= sum;
        //计算特征值
        double new_lambda = 0.0;
        for (int j = 0; j < n; ++j) {
            new_lambda += v[j] * prev_v[j];
        }
        if (std::abs(lambda - new_lambda) < tolerance) {
```

```
            break;
        }
        lambda = new_lambda;
    }
    return std::make_pair(lambda, v);
}

std::vector < double > pageRank ( const std:: vector < std:: vector < double >> & M, int
iterations, double tolerance) {
    int n = M.size();
    double d = 0.85;
    std::vector < double > v(n, 1.0 / n);
    std::vector < std::vector < double >> A(n, std::vector < double >(n, 0.0));
    double dangling_sum = 0.0;
    for (int i = 0; i < n; ++i)
    {
        double row_sum = 0.0;
        for (int j = 0; j < n; ++j)
        {
            row_sum += M[i][j];
        }
        if (row_sum == 0)
        {
            dangling_sum += v[i];
        }
        else
        {
            for (int j = 0; j < n; ++j)
            {
                A[j][i] = M[i][j] / row_sum;
            }
        }
    }
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            A[j][i] = d * A[j][i] + (1 - d) / n;
        }
}
cout << "转置矩阵 A = \n";
    printMtx(A);
    std::pair < double, std::vector < double >> result = powerMethod(A, v, iterations,
tolerance);
    return result.second;
}
```

【例 5-10】 利用 PageRank 随机浏览模型求如图 5-2 所示的网页的 PageRank 值。网页之间的关系见表 5-1。

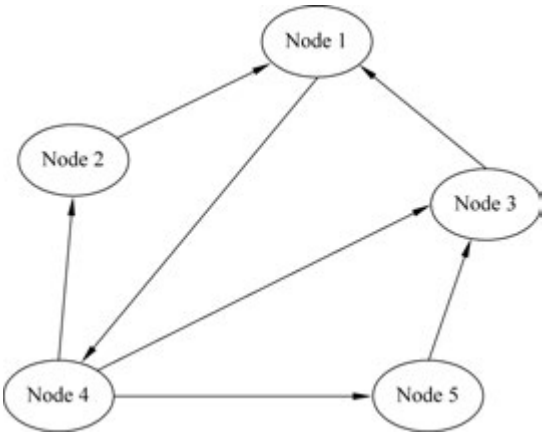

图 5-2　网络示意图

表 5-1　网页间的链接关系表

链接源 Node	1	2	3	4	5
链出目标 Node	4	1	1	2,3,5	3

由网页关系图和表很容易给出网络系统的邻接矩阵 $\boldsymbol{G}$。

$$\boldsymbol{G}=\begin{pmatrix}0&0&0&1&0\\1&0&0&0&0\\1&0&0&0&0\\0&1&1&0&1\\0&0&1&0&0\end{pmatrix}$$

将邻接矩阵输入主程序 main()函数中，代码如下：

```
//第 5 章/pagerank/PageRank.cpp
//同上，略
int main()
{
    std::vector < std::vector < double >> G{ {0, 0, 0,1,0},
                                             {1, 0, 0,0,0},
                                             {1, 0, 0,0,0},
                                             {0, 1, 1,0,1},
                                             {0, 0,1, 0,0}};
    double d = 0.85;
    int iterations = 100;
    int n = G.size();
    double tolerance = 1e-6;
    std::vector < double > ranks = pageRank(G, iterations, tolerance);
    cout << "各节点的 PageRank 值\n";
    for (double rank : ranks) {
        std::cout << rank << " ";
    }
    return 0;
}
```

计算结果如下：

```
转置矩阵 A =
0.03    0.88    0.88    0.03        0.03
0.03    0.03    0.03    0.313333    0.03
0.03    0.03    0.03    0.313333    0.88
0.88    0.03    0.03    0.03        0.03
0.03    0.03    0.03    0.313333    0.03
各节点的 PageRank 值
0.295852 0.109775 0.203048 0.28155 0.109775
```

根据计算结果，可以得到这个小网络的网页名次表，见表 5-2。

表 5-2 网页名次表

网页 Node	1	4	3	2	5
PageRank 值	0.295852	0.28155	0.203048	0.109775	0.109775

从表 5-2 可以看出，一个随机搜索访问页面 1 的概率大约是 30%，而页面 2 的概率大约是 11 %。把这些 PageRank 值与图 5-1 中的网格系统进行比较，页面 1 和页面 3 有相同的输入度和输出度，而页面 1 有更大的 PageRank 值，因为它有比链接页面 3 的页面更重要的(有更大的 PageRank 值)页面的链接。如果搜索访问页面 1(大约 30% 的时间出现)，则 28% 的时间这个搜索将按链接连接到页面 4。从表 5-2 还可以看出：

(1) 被链接个数越多，其 PageRank 值越大；如果被链接个数相同，则发出链接个数越多其 PageRank 值越大。

(2) 页面 1 的 PageRank 值是 0.2958，占据全体接近三分之一，排第 1 位。从图 5-2 与结果汇总表 5-2 可以看出，因为页面 1 是链出链接和链入链接最多的页面，所以可以理解为它是最受欢迎的页面。

5.8.2 鸢尾花分类

1. 问题简介

鸢尾花分类问题是机器学习领域中的一个经典的入门级案例，被广泛地用于演示机器学习算法。

鸢尾花(Iris)是一种常见的花卉，其花朵具有独特的形态特征。鸢尾花分类问题的目标是根据鸢尾花的特征(如花瓣和花萼的长度、宽度等)来判断其所属的种类。这个数据集最初由英国统计学家和生物学家罗纳德·费希尔(Ronald Fisher)在 1936 年的论文中提出。

鸢尾花数据集(Iris Dataset)是机器学习中使用最广泛的数据集之一，该数据集包含 150 个样本，每个样本代表一朵鸢尾花。每个样本有 4 个特征，分别为花萼长度(Sepal Length，单位为厘米)、花萼宽度(Sepal Width，单位为厘米)、花瓣长度(Petal Length，单位为厘米)、花瓣宽度(Petal Width，单位为厘米)。每个样本属于 3 个种类之一，分别为山鸢尾(Iris Setosa)、变色鸢尾(Iris Versicolor)、维吉尼亚鸢尾(Iris Virginica)。每个种类有 50 个样本，并且数据分布较为均匀，详见表 5-3。

表 5-3 鸢尾花各种属性的统计值

属　　性	最大值/cm	最小值/cm	均值/cm	方差/cm
花萼长	7.9	4.3	5.84	0.83
花萼宽	4.4	2.0	3.05	0.43
花瓣长	6.9	1.0	3.76	1.76
花瓣宽	2.5	0.1	1.20	0.76

鸢尾花分类问题的目标是根据给定的 4 个特征（花萼长度、花萼宽度、花瓣长度、花瓣宽度）训练一个分类模型，能够准确地预测鸢尾花的种类。这是一个典型的多分类问题。

尽管鸢尾花分类问题是一个简单的入门案例，但它更具有重要的教学意义。通过这个简单的问题，初学者可以快速地理解机器学习的基本概念，如特征、标签、训练集、测试集等，还可演示数据预处理、模型训练和评估的完整流程，为解决更复杂的问题奠定基础。

此外，鸢尾花分类问题的思路和方法可以推广到其他类似的分类任务，如医学诊断、图像识别、语音识别等领域。

总之，鸢尾花分类问题是机器学习领域的一个经典入门案例，它简单易懂，但涵盖了机器学习的核心概念和流程，是学习机器学习的绝佳起点。

2. C++文件读写操作

要在 C++中进行文件处理，必须在 C++源代码文件中包含头文件 <iostream>和 <fstream>。在 C++中，文件读写操作是通过标准库中的 fstream 类来实现的。fstream 类提供了 3 个用于文件操作的子类：ifstream、ofstream 和 fstream。ifstream 用于从文件读取数据，ofstream 用于向文件写入数据，而 fstream 可以同时用于读取和写入。

1）文件打开模式

当我们使用这些类打开文件时，可以指定不同的文件打开模式，这些模式包括以下几种。

ios::in：以只读方式打开文件。

ios::out：以只写方式打开文件。

ios::app：以追加模式打开文件，写入的数据将添加到文件末尾。

ios::binary：以二进制模式打开文件。

ios::trunc：如果文件已存在，则打开时清空文件内容。

在从文件读取信息或者向文件写入信息之前，必须先打开文件。ofstream 和 fstream 对象都可以用来打开文件进行写操作，如果只需打开文件进行读操作，则使用 ifstream 对象。

2）写入文件

在 C++编程中，我们使用流插入运算符（<<）向文件写入信息，就像使用该运算符将信息输出到屏幕上一样。唯一不同的是，在这里使用的是 ofstream 或 fstream 对象，而不是 cout 对象。

3）读取文件

在 C++编程中，我们使用流提取运算符（>>）从文件读取信息，就像使用该运算符从

键盘输入信息一样。唯一不同的是,在这里使用的是 ifstream 或 fstream 对象,而不是 cin 对象。读取 C++文件的步骤如下。

(1) 创建输入流对象,用于读取文件,示例代码如下:

```
ifstream fin(filename,ios::in);
```

或者

```
ifstream fin(filename);
```

(2) 检查文件是否成功打开,使用 if 判断,示例代码如下:

```
if (!fin.is_open())
```

(3) 读取文件内容,使用>>读取。

(4) 关闭文件,当 C++程序终止时,它会自动关闭刷新所有流,释放所有分配的内存,并关闭所有打开的文件,但程序员应该养成一个好习惯,在程序终止前关闭所有打开的文件。

3. 鸢尾花分类算法

步骤 1,读取数据。

步骤 2,计算数据协方差矩阵。

步骤 3,主成分分析 PCA。

步骤 4,输出结果。

4. 读取 iris 数据程序

读取数据需要包含头文件< fstream >,代码如下:

```
//第 5 章/Iris/readFileData.h
#pragma once
#include < vector >
#include < fstream >
#include < iostream >
using namespace std;

vector < vector < double >> readFile(const char * filename,const int m,const int n)
{
    ifstream fin(filename,ios::in);
    if (!fin.is_open())
    {
        cout << "error open file!" << endl;
        exit(1);
    }

    double d;
    vector < double > V;
    int length = 0;
    while (!fin.eof())
```

```
    {
        fin >> d;
        V.push_back(d);
    }
    fin.close();

    vector < vector < double >> data(m, vector < double >(n));
    for (auto it = V.begin(); it != V.end();)
    {
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                data[i][j] = * it;
                it++;
            }
        }
        break;
    }
    return data;
}
```

5. 求 iris 数据集协方差矩阵

协方差矩阵计算函数 covMtx()，使用 pair 返回两个矩阵，代码如下：

```
//第 5 章/Iris/covariancMtx.h
/**
 ********************************************************************
 * Copyright (C)
 * @brief 求协方差矩阵(pair 方法)
 * @author SunKunfeng, skf0558@sina.com
 * @date 2025 - 02 - 01
 * @version V1.0 20250201
 ********************************************************************
 * License - Identifier: LGPL - 3.0 - or - later
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty
 * of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * ------------------------------------------------------------------
 */

#pragma once
#include < vector >
#include < iostream >
using namespace std;

void printMtx(const vector < vector < double >> & a)
{
    int n = a.size();
    int m = a[0].size();
    cout << " mtx = " << endl;
```

```
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < m; j++)
            {
                cout << a[i][j] << "\t";
            }
            cout << endl;
        }
    }

    //计算协方差矩阵,返回两组值
    pair < vector < vector < double >>, vector < vector < double >> >
    covMtx(vector < vector < double >> & mtx)
    {
        size_t m = mtx.size();
        size_t n = mtx[0].size();

        //1. 计算均值向量(以列向量为例)
        vector < double > E(n);
        for (int i = 0; i < n; ++i)
        {
            double sum = 0;
            for (int j = 0; j < m; ++j)
            {
                sum += mtx[j][i];
            }
            E[i] = sum / m;
        }

        cout << "mean value = \n";
        for (int j = 0; j < n; ++j) cout << E[j]<< "\t";
        cout << endl;

        //2. 样本中心化矩阵
        vector < vector < double >> centrMtx(m, vector < double >(n));
        vector < vector < double >> centrMtxT(n, vector < double >(m));
        for (int i = 0; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                centrMtx[i][j] = mtx[i][j] - E[j]; //z = x_i - x_bar
                centrMtxT[j][i] = centrMtx[i][j];
            }
        }

        //3. 求协方差矩阵,1/(m - 1) * centrMtxT * centrMtx
        vector < vector < double >> covMtx(n, vector < double >(n));
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                for (int k = 0; k < m; k++)
                {
```

```
                covMtx[i][j] += centrMtxT[i][k] * centrMtx[k][j];
            }
            covMtx[i][j] /= (m - 1.0);
        }
    }
    return make_pair(covMtx,centrMtx);
}
```

6. 主成分分析 PCA

计算特征值和特征向量使用雅可比法，调用 PCA.h 文件进行主成分分析计算。

```
//第 5 章/Iris/PCA.h
//同 5.7 节，略
```

7. 计算结果分析

主程序代码如下：

```
//第 5 章/Iris/main.cpp
#include "covariancMtx.h"
#include "readFileData.h"
#include "PCA.h"
int main()
{
    cout.setf(ios::fixed);
    cout.precision(6);
    //鸢尾花例子数据
    int m = 150, n = 4;
    const char* file = "irisData.txt";
    vector<vector<double>> data(m, vector<double>(n));
    data = readFile(file, m, n);

    vector<vector<double>> covM(n, vector<double>(n)); //协方差矩阵
    pair<vector<vector<double>>, vector<vector<double>> >
    result = covMtx(data);
    covM = result.first;
    vector<vector<double>> Xc = result.second;          //centrMtx
    cout << "协方差矩阵：\n";
    printMtx(covM);

    //求特征值和特征向量
    int maxIt = 500;
    double eps = 1e-10;
    int row = covM.size();
    int col = covM[0].size();

    Matrix arr(covM, row, col);                         //把 vector 转换为 matrix
    Matrix arr2Xc(Xc);                                  //由于 PCA 只接受 matrix

    PCA<Matrix> p1(arr, row, col, maxIt, eps);
    int Num = 2;
    p1.recovery(Num,arr2Xc);                            //取前 Num 项特征向量
    return 0;
}
```

数据从四维降到二维后，可以很方便地进行可视化。从散点图中可以直观地看到，降维后的数据较好地保留了原数据的分布信息，计算结果如图 5-3 所示。

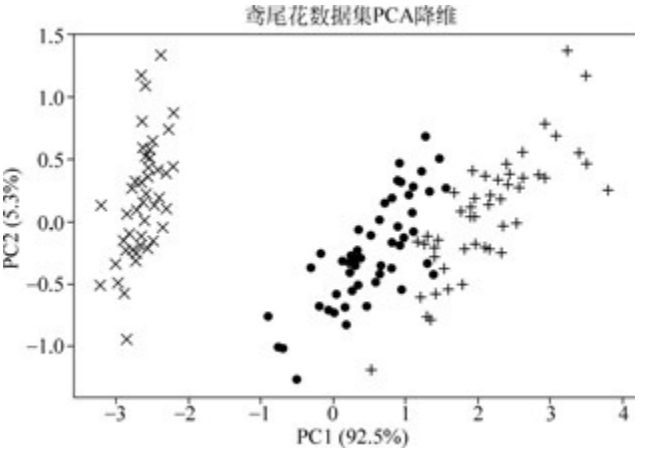

图 5-3 鸢尾花 PCA 分类图