****

数值分析大作业

计算实习题（1）

|  |  |
| --- | --- |
| 院（系）名称 | 宇航学院 |
| 学号 | ZY2115107 |
| 学生姓名 | 段诗阳 |

2021年10月26日

## 题目要求：

1. 求出矩阵所有特征值中最小的特征值和最大的特征值
2. 求出所有特征值中模最小的特征值
3. 求出靠近的一组数
4. 求出矩阵的条件数
5. 求解

## 解题思路：

1.利用幂法求出矩阵的模最大的特征值，其一定是或之一。然后再利用带原点平移的幂法，将矩阵变成，求出的特征值离最远，故也是或之一。将两个特征值比较大小；大的是，小的是。

2.利用反幂法求出矩阵的模最小的特征值。

3.是最接近的特征值，因此可以利用带原点平移的反幂法，将矩阵变成，求出模最小的特征值，即是。

4. 谱范数条件数是模最小的特征值和模最大的特征值的绝对值比值，由（1）问结果可求得。

5.将矩阵进行LU分解，，得到。为上三角矩阵，为对角线元素的乘积。

## 算法流程：

1.幂法求解模最大的特征值

a.初始化迭代向量。

b.开始迭代

c.计算上一次迭代得到的的2范数，并将归一化得到。计算矩阵与归一化迭代向量的乘积，即为。计算和的内积。

d.终止迭代：如果，则跳出循环；作为所求的特征值，作为对应的特征向量。如果不满足终止条件，则返回上一步，重新迭代。

2.反幂法求解模最小的特征值

a.初始化迭代向量。

b.对矩阵进行LU分解，得到上三角矩阵和下三角矩阵。

c.开始迭代

d.计算上一次迭代得到的的2范数，并将归一化得到。

e.求解：利用LU分解的结果，首先求解,然后求解，得到的即为，计算和的内积。

f.终止迭代：如果，则跳出循环；作为所求的特征值，作为对应的特征向量。如果不满足终止条件，则返回上一步，重新迭代。

## 计算程序

// Question1\_new.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。

// 编程环境：VS2019

#include <iostream>

#include <math.h>

#include <iomanip>

using namespace std;

constexpr auto N = 501; // 矩阵阶数;;

double epsilon = 1e-12; // 精度水平=1e-12

double A[5][N] = { 0 }; // 定义对称矩阵A

double Temp[5][N] = { 0 };

// 函数声明

double PowerMethod(double T[][N]);

double AntiPowerMethod(double T[][N]);

void LUresolve(double T[][N]);

int GetMin(int i, int j);

int GetMax(int i, int j);

int GetMin(int i, int j) {

if (i <= j) return i;

else return j;

}

int GetMax(int i, int j) {

if (i > j) return i;

else return j;

}

// 幂法的实现

double PowerMethod(double T[][N]) {

int i, j, m, n;

double u[N] = { 0 }; // 迭代向量

double y[N] = { 0 }; // 归一化迭代向量

double beta\_last = 0; // k-1时的β

double beta\_now = 0; // k时的β

for (i = 0; i < N; i++)

u[i] = 1; // 初始化迭代向量

while (1) {// 开始迭代

double Sum = 0;

for (i = 0; i < N; i++)

Sum += u[i] \* u[i];

double Mo = sqrt(Sum); // 得到u(k-1)的模

// 归一化迭代向量，得到y(k-1)

for (i = 0; i < N; i++)

y[i] = u[i] / Mo;

for (i = 0; i < N; i++)

u[i] = 0;

// 计算u(k)：A与y(k-1)的乘积

for (i = 2; i < N + 2; i++) {

m = GetMin(i, 4);

n = GetMin(i, N);

for (j = i - m; j < n; j++)

u[i - 2] = u[i - 2] + T[i - j][j] \* y[j];

}

// 计算k-1时的β：yT(k-1)和u(k)的乘积

for (i = 0; i < N; i++)

beta\_now = beta\_now + y[i] \* u[i];

if (fabs(beta\_now - beta\_last) / fabs(beta\_now) <= epsilon)

return beta\_now; // 迭代结束，得到主特征值

else {

beta\_last = beta\_now;

beta\_now = 0;

// cout << "beta\_last = " << beta\_last << endl;

}

}

}

// 反幂法的实现

double AntiPowerMethod(double T[][N]) {

int i, j, k, m, n;

double u[N] = { 0 }; // 迭代向量

double y[N] = { 0 }; // 归一化的迭代向量

double Temp\_1[N] = { 0 };

double beta\_last = 0;

double beta\_now = 0;

for (i = 0; i < N; i++)

u[i] = 1; // 初始化迭代向量

// 对T作LU分解

LUresolve(T);

while (1) {

double Sum = 0;

for (i = 0; i < N; i++)

Sum += u[i] \* u[i];

double Mo = sqrt(Sum); // 得到u(k-1)的模

// cout << "Mo = " << Mo << endl;

// 归一化迭代向量，得到y(k-1)

for (i = 0; i < N; i++)

y[i] = u[i] / Mo;

for (i = 0; i < N; i++)

u[i] = 0;

// 求解Ly=b

Temp\_1[0] = y[0];

for (i = 1; i < N; i++) {

double sum = 0;

for (m = GetMax(1, i - 1); m <= i; m++)

sum += T[i - m + 3][m - 1] \* Temp\_1[m - 1];

Temp\_1[i] = y[i] - sum;

}

// 求解Ux=y

u[N - 1] = Temp\_1[N - 1] / T[2][N - 1];

for (i = N - 2; i >= 0; i--) {

if (i + 2 >= N) n = N - 1;

else n = i + 2;

double sum = 0;

for (j = i + 1; j <= n; j++)

sum += T[i - j + 2][j] \* u[j];

u[i] = (Temp\_1[i] - sum) / T[2][i];

}

// 计算k-1时的β：yT(k-1)和u(k)的乘积

for (i = 0; i < N; i++)

beta\_now = beta\_now + y[i] \* u[i];

if (fabs(beta\_now - beta\_last) / fabs(beta\_now) <= epsilon)

return 1 / beta\_now; // 迭代结束，得到主特征值

else {

beta\_last = beta\_now;

beta\_now = 0;

// cout << "beta\_last = " << beta\_last << endl;

}

}

}

// LU分解的实现

void LUresolve(double T[][N]) {

int i, j, p, k, t;

for (i = 3; i < 5; i++)

T[i][0] = T[i][0] / T[2][0];

for (k = 2; k <= N; k++) {

p = GetMin(k + 2, N);

for (j = k; j <= p; j++) {// 得到U

double sum = 0;

for (t = GetMax(1, GetMax(k - 2, j - 2)); t < k; t++)

sum += T[k - t + 2][t - 1] \* T[t - j + 2][j - 1];

T[k - j + 2][j - 1] = T[k - j + 2][j - 1] - sum;

}

if (k < N) {

p = GetMin(k + 2, N);

for (i = k + 1; i <= p; i++) {

double sum = 0;

for (t = GetMax(1, GetMax(i - 2, k - 2)); t < k; t++)

sum += T[i - t + 2][t - 1] \* T[t - k + 2][k - 1];

T[i - k + 2][k - 1] = T[i - k + 2][k - 1] - sum;

T[i - k + 2][k - 1] = T[i - k + 2][k - 1] / T[2][k - 1];

}

}

}

}

int main()

{

cout << "Hello World!\n";

int i, j, k;

for (i = 0; i < N; i++) {

// 构造对称矩阵A

k = i + 1;

A[0][i] = -0.064;

A[1][i] = 0.16;

A[2][i] = (1.64 - 0.024 \* k) \* sin(0.2 \* k) - 0.64 \* exp(0.1 / k);

A[3][i] = 0.16;

A[4][i] = -0.064;

}

A[0][0] = A[0][1] = A[1][0] = 0;

A[3][N - 1] = A[4][N - 1] = 0;

A[4][N - 2] = 0;

// 复制矩阵A，用于求解

for (i = 0; i < 5; i++) {

for (j = 0; j < N; j++)

Temp[i][j] = A[i][j];

}

// 幂法求解主特征值

double Lamd1 = PowerMethod(Temp);

// 构造带原点平移的矩阵Temp

for (i = 0; i < 5; i++) {

for (j = 0; j < N; j++) {

if (i == 2) Temp[i][j] = A[i][j] - Lamd1;

else Temp[i][j] = A[i][j];

}

}

// Lamd1模值最大，为最大值最小值之一

// Lamd2离Lamd1最远，故也为最大值最小值之一

double Lamd2 = PowerMethod(Temp) + Lamd1;

cout << setiosflags(ios::scientific); // 输出E型数

// cout << setprecision(12) << "Lamd1 = " << Lamd1 << endl;

// cout << setprecision(12) << "Lamd2 = " << Lamd2 << endl;

double Lamd\_1, Lamd\_501;

if (Lamd1 >= Lamd2) {// 比较特征值大小

Lamd\_1 = Lamd2;

Lamd\_501 = Lamd1;

}

else {

Lamd\_1 = Lamd1;

Lamd\_501 = Lamd2;

}

cout << setprecision(12) << "Lamd\_1 = " << Lamd\_1 << endl;

cout << setprecision(12) << "Lamd\_501 = " << Lamd\_501 << endl;

// 复制矩阵A，用于求解

for (i = 0; i < 5; i++) {

for (j = 0; j < N; j++)

Temp[i][j] = A[i][j];

}

// 反幂法求解模最小的特征值

double Lamd\_s = AntiPowerMethod(Temp);

cout << setprecision(12) << "Lamd\_s = " << Lamd\_s << endl;

// 第二问

for (k = 1; k < 40; k++) {

double U\_k = Lamd\_1 + k \* (Lamd\_501 - Lamd\_1) / 40;

// 构造带位移的矩阵Tem

for (i = 0; i < 5; i++) {

for (j = 0; j < N; j++) {

if (i == 2)

Temp[i][j] = A[i][j] - U\_k;

else

Temp[i][j] = A[i][j];

}

}

double Lam\_ik = AntiPowerMethod(Temp) + U\_k;

cout << "when k = " << k << " " << setprecision(12) << "Lam\_ik=" << Lam\_ik << endl;

}

// 第三问

cout << "cond(A)2 = " << setprecision(12) << fabs(Lamd1 / Lamd\_s) << endl;

for (i = 0; i < 5; i++) {

for (j = 0; j < N; j++)

Temp[i][j] = A[i][j];

}

LUresolve(Temp);

double det = 1;

for (i = 0; i < N; i++) det = det \* Temp[2][i];

cout << "detA = " << setprecision(12) << det << endl;

return 0;

}

## 运行结果

图片包含 文本

描述已自动生成

文本

描述已自动生成

## 结果分析

1.在编程时发现与参考答案偏差较大，经过测试其与迭代向量的初始化有关。而与迭代向量的初始化取值没有明显关系，总能迭代到准确的值。

2.通过将迭代向量取不同的初始值发现对迭代次数（速度）没有明显影响。

3.为了避免初始向量对程序的影响，可以先对A做平移变换再求。