数值分析第一次大作业

傅锡豪, 15091082

2017 年 4 月 12 日

1 算法设计

本程序的算法包括以下四个方面

一、存储： 由于题目的矩阵是稀疏的带状矩阵，所以通过把矩阵逆时针旋转储存，大大减少存储空间。原矩阵和被压缩矩阵之间元素的对应关系是：$a_{i,j} = c_{i-j+3,j}$。原本是 501x501 的矩阵，经过压缩后是 5x501 的矩阵，而且只有 $c_{1,1}$ $c_{1,2}$ $c_{2,1}$ $c_{501,501}$ $c_{501,500}$ $c_{500,501}$ 是不存储数据的，其余空间都存放了原矩阵的非 0 数据。所以后面使用幂法和反幂法求解特征值的时候都是使用针对**带状矩阵**的算法。

二、计算特征值： 对于该题，需要分别求解最大、最小、按模最小、和距离特定 39 个值最近的特征值，分别由以下算法给定：

（1）**求解按模最大特征值**：

直接对矩阵 A 使用幂法，可直接求解出按模最大的特征值 λ_{max}。

（2）**通过平移求出另一特征值**：

用（1）求的 λ_{max} 的绝对值对矩阵 A 做平移，对 $(A - |\lambda_{max}|I)$ 进行反幂法操作得到 $1/\beta$，计算 $\lambda = 1/\beta + |\lambda_{max}|$。通过比较 λ_{max} 和 λ 的值的大小，把数值大的赋给 λ_{501}，数值小的赋给 λ_1。

（3）**求按模最小特征值**

对矩阵 A 用反幂法直接求出按模最小的特征值，即得到 λ_s。

（4）**求和 μ_k 相近的特征值**

进入循环：k: $1 \to 39$，依次执行以下操作：

1. 求解 μ_k：$\mu_k = \lambda_1 + k * (\lambda_{501} - \lambda 1)/40$;

2. 对用 μ_k 的值矩阵 A 平移，对 $(A - \mu_k I)$ 做反幂法操作，得到 $1/\beta_k$；

3. 距离 μ_k 最近的特征值由下式给定：$\lambda_{\mu_k} = \mu_k + 1/\beta_k$

三、A 的条件数

由非奇异对称矩阵的条件数由该式给定：$cond(A)_2 = |\lambda_1/\lambda_n|$，其中 λ_1 和 λ_n 分别是该矩阵的模最大和模最小的特征值。

在该题中，$cond(A)_2 = |\lambda_{max}/\lambda_s|$。

四、A 的行列式

本题中 A 的阶数过大，无法直接求解行列式。采用将 A 进行 Doolittle 分解，则 $|A| = |L||U|$。由于 L 是对角线为 1 的下三角阵，而 U 是三上角阵，所以 $|A| = |L||U| = \Pi_{i=1}^{501} u_{i,i}$。

在本题中，由于使用反幂法时对 A 做了 Doolittle 分解，所以在对矩阵 A 用反幂法求出 λ_s 后，把被处理过的 A 第 3 行（相当于未被压缩矩阵的对角线）各元素相乘，即 $det(A) = \Pi_{i=1}^{501} c_{3,i}$。

2 源程序

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <float.h>
#include <iomanip>
#include <cmath>
#define MEle A->Elements
#define VEle b->Elements
#define SEle Solution->Elements
#define YEle y->Elements
#define UEle u->Elements
#define TEle tp->Elements
#define AREle A_remain->Elements
using namespace std;

int main()
{
    double Epthron = 10e-12;
    double c, b;
    double Miu[39];
    double a[501];
    double DET_A = 1;
    double lambda_s;
    double lambda_max;
    double lambda1;
    double lambda501;
    double cond;
    int s = 2;
    Matrix* A = new Matrix(5, 501);
    //用于保护矩阵
```

```
Matrix* A_remain = new Matrix(5, 501);
A->setR(2);
A->setS(2);

for (int j = 0; j < 501; j++){
    a[j] = (1.64-0.024*(j+1))*sin(0.2*(j+1))-0.64*exp(0.1/(j+1));
}
b = 0.16;
c = -0.064;
//init the MATRIX A
for (int j = 0; j < 501; j++){
    if (j > 1) {
        MEle[0][j]  = c;
        AREle[0][j] = c;
    }
    if (j > 0){
        MEle[1][j]  = b;
        AREle[1][j] = b;
    }
    if (j < 500){
        MEle[3][j]  = b;
        AREle[3][j] = b;
    }
    if (j < 499){
        MEle[4][j]  = c;
        AREle[4][j] = c;
    }
    MEle[2][j]  = a[j];
    AREle[2][j] = a[j];
}

//get the one with max module
lambda_max = PowerMethod(A, Epthron);

//translate to get the other
A->Translate(abs(lambda_max));
lambda1 = InversePowerMethod(A, Epthron) + abs(lambda_max);

//compare lambda1 and lambda501
if (lambda1 > lambda_max){
    lambda501 = lambda1;
    lambda1 = lambda_max;
}
else {
    lambda501 = lambda_max;
}

//get lambda_s
Recover(A_remain, A);
lambda_s = InversePowerMethod(A, Epthron);

```

```
    //A stands for L+U, thus the products of the diagonal is detA
    for (int i = 0; i < 501; ++i)
    {
        DET_A*=MEle[s][i];
    }

    cond = abs(lambda_max/lambda_s);

    //translate to get those close to Miu
    for (int i = 0; i < 39; ++i)
    {
        Miu[i] = lambda1 + (i+1)*(lambda501 - lambda1)/40;
        Recover(A_remain, A);
        //translate
        A->Translate(Miu[i]);

        cout << "lambda_miu" << i+1 << " = "<< setprecision(11) << scientific << InversePowerMethod(A, Epthron)+Miu[i] << endl;
    }

    cout << "lambda501 = " << lambda501 << endl;
    cout << "lambda1 = " << lambda1 << endl;
    cout << "lambda_s = " << lambda_s << endl;
    cout << "cond_A = " << cond << endl;
    cout << "det_A = " <<DET_A << endl;

    return 0;
}

int Minimun(int m, int n){
    if (m > n) {
        return n;
    }
    else {
        return m;
    }
}
int Maximum(int m, int n){
    if (m > n) {
        return m;
    }
    else {
        return n;
    }
}
int Maximum(int o, int p, int q){
    if (o > p)
    {
        if (o > q)
        {
            return o;
```

```
            }
            else {
                return q;
            }
        }
        else if (p > q)
        {
            return p;
        }
        else {
            return q;
        }
}
class Matrix{
private:
    int row;
    int column;
    int s, r;
public:
    double **Elements;
    Matrix(int row, int column){
        this->row = row;
        this->column = column;
        Elements = (double**)malloc(sizeof(double*)*row);
        for (int i = 0; i < row; ++i)
        {
            Elements[i] = (double*)malloc(sizeof(double)*column);
        }
    }
    void setValue(){
        this->r = 0;
        this->s = 0;
        double **p = Elements;
        cout << "Input Matrix:\n";
        for (int i = 0; i < this->row; ++i)
        {
            for (int j = 0; j < this->column; ++j)
            {
                cin >> p[i][j];
            }
        }
    }

    //for saving storage
    //s means the number of diagonals that is above the central diagonal
    void setValue(int row, int column, int s, int r){
        double **p = Elements;
        double temp;
        this->s = s;
        this->r = r;
        cout << "Input Matrix:\n";
```

```
        for (int i = 0; i < row; ++i)
        {
            for (int j = 0; j < column; ++j)
            {
                cin >> temp;
                if (temp != 0.0)
                {
                    p[i-j+s][j] = temp;
                }
            }
        }
    }

    void setR(int r){
        this->r = r;
    }
    void setS(int s){
        this->s = s;
    }
    int getR(){
        return this->r;
    }
    int getS(){
        return this->s;
    }
    int getRow(){
        return this->row;
    }
    int getColumn(){
        return this->column;
    }

    void Translate(double distance){
        if (this->getR() == this->getS() == 0){
            for (int i = 0; i < this->getColumn(); ++i)
            {
                Elements[i][i] -= distance;
            }
        }
        else{
            for (int i = 0; i < this->getColumn(); ++i){
                Elements[this->s][i] -= distance;
            }
        }
    }
};

class Vector{
private:
    int size;
public:
```

```
    double* Elements;
    Vector(int size){
        this->size = size;
        Elements = (double*)malloc(sizeof(double)*size);
    }
    void setValue(){
        double *p = Elements;
        cout << "Input Vector:\n";
        for (int i = 0; i < this->size; ++i)
        {
            cin >> p[i];
        }
    }

    int getSize(){
        return this->size;
    }
    double getNorm2(){
        double norm = 0;
        for (int i = 0; i < this->size; i++){
            norm+=Elements[i]*Elements[i];
        }
        return sqrt(norm);
    }
};
class EquationGroup{
private:
    Matrix* A;
    Vector* b;
    Vector* Solution;
public:
    //in order to make storage more economical
    void initEquation(Matrix* A, Vector* b){
        this->A = A;
        this->b = b;
        Solution = new Vector(A->getColumn());
    }

    Vector* getSolution(){
        return Solution;
    }

    void TrianglarDecomposition(){
        int n = A->getRow();
        double u, l;
        for(int k = 0; k < n; k++){

            for(int j = k; j < n; j++){
                u = MEle[k][j];
                for(int t = 0; t <= k-1; t++){
                    u-=MEle[k][t]*MEle[t][j];
```

```
                }

                MEle[k][j] = u;
            }
            for(int i = k+1; i < n; i++){
                l = MEle[i][k];
                for(int t = 0; t <= k-1; t++){
                    l-=MEle[i][t]*MEle[t][k];
                }
                l/=MEle[k][k];

                MEle[i][k] = l;
            }
        }
    }
    void Substitution(){
        int n = A->getRow();
        double y[n];
        y[0] = VEle[0];
        for(int i = 1; i < n; i++){
            y[i] = VEle[i];
            for(int t = 0; t <= i-1; t++){
                y[i]-=MEle[i][t]*y[t];
            }
        }
        SEle[n-1] = y[n-1]/MEle[n-1][n-1];
        for(int i = n-2; i >= 0; i--){
            SEle[i] = y[i];
            for(int t = i+1; t < n; t++){
                SEle[i]-=MEle[i][t]*SEle[t];
            }
            SEle[i]/=MEle[i][i];
        }
    }

    void TrianglarDecompStrip(){
        int s = A->getS();
        int r = A->getR();
        int n = A->getColumn();
        int temp1;
        int temp2;
        for (int k = 0; k < n; k++){
            temp1 = Minimun(k+s, n-1);
            for (int j = k; j <= temp1; j++){
                temp2 = Maximum(0, k-r, j-s);
                for (int t = temp2; t < k; t++){
                    MEle[k-j+s][j] -= MEle[k-t+s][t]*MEle[t-j+s][j];
                }
            }
            temp1 = Minimun(k+r, n-1);
            for (int i = k+1; i <= temp1; i++){
```

```
                temp2 = Maximum(0, i-r, k-s);
                for (int t = temp2; t < k; t++){
                    MEle[i-k+s][k] -= MEle[i-t+s][t]*MEle[t-k+s][k];
                }
                MEle[i-k+s][k] =  MEle[i-k+s][k]/MEle[s][k];
            }
        }
    }

    void SubstitutionStrip(){
        int temp;
        int s = A->getS();
        int r = A->getR();
        int n = b->getSize();

        //保护列向量
        Vector* tp = new Vector(n);
        for (int i = 0; i < n; ++i){
            TEle[i] = VEle[i];
        }
        for (int i = 1; i < n; ++i){
            temp = Maximum(0, i-r);
            for (int t = temp; t < i; t++){
                TEle[i] -= MEle[i-t+s][t]*TEle[t];
            }
        }
        SEle[n-1] = TEle[n-1]/MEle[s][n-1];
        for (int i = n-2; i >= 0; i--){
            SEle[i] = TEle[i];
            temp = Minimun(i+s, n-1);
            for (int t = i+1; t <= temp; t++){
                SEle[i] -= MEle[i-t+s][t]*SEle[t];
            }
            SEle[i] = SEle[i]/MEle[s][i];
        }
        delete tp;
    }

};

double DotProduct(Vector* X, Vector* Y){
    double product = 0;
    for (int i = 0; i < X->getSize(); i++){
        product+=(X->Elements[i]*Y->Elements[i]);
    }
    return product;
}

double PowerMethod(Matrix* A, double Error){
    double Yita;
    double Beta;
```

```
    double FormerBeta;
    double LocalError;
    int column = A->getColumn();
    int row = A->getRow();
    //the s of Matrix_A
    int s = A->getS();
    Vector* y = new Vector(column);
    Vector* u = new Vector(column);

    //generate u0
    for (int i = 0; i < column; i++){
        UEle[i] = 0.1;
    }

    //begin loop, until the error is smaller the given one
    do {
        Yita = u->getNorm2();
        for (int i = 0; i < u->getSize(); i++){
            YEle[i] = UEle[i]/Yita;
        }

        //modify u
        if (A->getR() == A->getS() == 0){
            for (int i = 0; i < row; i++){
                UEle[i] = 0;
                for (int j = 0; j < column; j++){
                    UEle[i] += MEle[i][j]*YEle[j];
                }
            }
        }
        else {
            for (int i = 0; i < column; i++){
                UEle[i] = 0;
                for (int j = 0; j < column; j++){
                    if (i-j+s >= 0 && i-j+s < row)
                    {
                        UEle[i]+=MEle[i-j+s][j]*YEle[j];
                    }
                }
            }

        }

        FormerBeta = Beta;
        Beta = DotProduct(y, u);
        LocalError = abs(Beta - FormerBeta)/abs(Beta);
    } while (LocalError > Error);

    return Beta;
}
double InversePowerMethod(Matrix* A, double Error){
```

```
    double Beta;
    double FormerBeta;
    double LocalError;
    double Yita;
    int column = A->getColumn();
    int row = A->getRow();

    Vector* y = new Vector(column);
    Vector* u = new Vector(column);

    EquationGroup* E = new EquationGroup();
    E->initEquation(A, y);

    //do the decomposition
    if (A->getR() == A->getS() == 0){
        E->TrianglarDecomposition();
    }
    else {
        E->TrianglarDecompStrip();
    }

    //generate u0
    for (int i = 0; i < column; i++){
        UEle[i] = 0.1;
    }
    //begin loop, until the error is smaller the given one
    do {
        Yita = u->getNorm2();
        for (int i = 0; i < column; i++){
            YEle[i] = UEle[i]/Yita;
        }

        //solve equation to get u
        E->initEquation(A, y);

        if (A->getR() == A->getS() == 0){
            E->Substitution();
        }
        else {
            E->SubstitutionStrip();
        }

        for (int i = 0; i < column; i++){
            UEle[i] = E->getSolution()->Elements[i];
        }

        FormerBeta = Beta;
        Beta = DotProduct(y, u);
        LocalError = abs(1/Beta - 1/FormerBeta)/abs(1/Beta);
    } while (LocalError > Error);

```

```
    return 1/Beta;
}

void Recover(Matrix* A_for_copying, Matrix* A_for_targeting){
    int row = A_for_targeting->getRow();
    int column = A_for_targeting->getColumn();
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < column; ++j)
        {
            A_for_targeting->Elements[i][j] = A_for_copying->Elements[i][j];
        }
    }
}
```

3 打印结果

```
lambda_miu1 = -1.01829340331e+01
lambda_miu2 = -9.58570742506e+00
lambda_miu3 = -9.17267242393e+00
lambda_miu4 = -8.65228400790e+00
lambda_miu5 = -8.09348380867e+00
lambda_miu6 = -7.65940540769e+00
lambda_miu7 = -7.11968464869e+00
lambda_miu8 = -6.61176433940e+00
lambda_miu9 = -6.06610322659e+00
lambda_miu10 = -5.58510105263e+00
lambda_miu11 = -5.11408352981e+00
lambda_miu12 = -4.57887217687e+00
lambda_miu13 = -4.09647092626e+00
lambda_miu14 = -3.55421121575e+00
lambda_miu15 = -3.04109001813e+00
lambda_miu16 = -2.53397031113e+00
lambda_miu17 = -2.00323076956e+00
lambda_miu18 = -1.50355761123e+00
lambda_miu19 = -9.93558606008e-01
lambda_miu20 = -4.87042673885e-01
lambda_miu21 = 2.23173624957e-02
lambda_miu22 = 5.32417474207e-01
lambda_miu23 = 1.05289896269e+00
lambda_miu24 = 1.58944588188e+00
lambda_miu25 = 2.06033046027e+00
lambda_miu26 = 2.55807559707e+00
lambda_miu27 = 3.08024050931e+00
lambda_miu28 = 3.61362086769e+00
lambda_miu29 = 4.09137851045e+00
lambda_miu30 = 4.60303537828e+00
lambda_miu31 = 5.13292428390e+00
lambda_miu32 = 5.59490634808e+00
```

```
lambda_miu33 = 6.08093385703e+00
lambda_miu34 = 6.68035409211e+00
lambda_miu35 = 7.29387744813e+00
lambda_miu36 = 7.71711171424e+00
lambda_miu37 = 8.22522001405e+00
lambda_miu38 = 8.64866606519e+00
lambda_miu39 = 9.25420034458e+00
lambda501 = 9.72463409966e+00
lambda1 = −1.07001136138e+01
lambda_s = −5.55791079436e−03
cond_A = 1.92520427364e+03
det_A = 2.77278614175e+118
```

4 讨论

对于使用反幂法之后矩阵 A 改变的考虑

由于使用反幂法时，需要对矩阵 A 和一个列向量求解线性方程组，而在使用 Doolittle 分解求解的时候，为了节省存储空间，会把分解后的 L 和 U 矩阵存在原本的矩阵 A 的位置，导致在每次使用反幂法之后，矩阵 A 都会被改变。本题中使用一个矩阵来保留原矩阵 A 的值，在每次进行反幂法之后都使用一个函数（源程序 492 行），把被修改过的 A 进行恢复。这样既需要多出一个矩阵的存储空间，还需要多次恢复该矩阵，增加了操作次数（尽管和内部求解方程组的时间复杂度 $O(n^3)$ 相比，并不会改变时间复杂度的量级）。而如果在 Doolittle 分解的时候不把 L 和 U 存在 A 的位置，而是开辟另一块内存空间，多次回带结束之后释放该空间，这样重复使用反幂法的时候也最多只有一个矩阵的空间占用，并且在时间上不需要像上面一样进行一个二重循环，也可以节省算法的操作时间。所以**对于本题来说**，或许可以修改 Doolittle 算法来避免这个恢复矩阵 A 的过程。