Белорусский государственный университет

Факультет прикладной математики и информатики

Лабораторная работа №5,6,7,8

Вариант 8

Проблема собственных значений

Выполнил:

Студент 2 курса 7 группы ПМ ФПМИ

Шевцов Евгений

Преподаватель:

Будник Анатолий Михайлович

Минск – 2021

Для всех методов $B = A^T A$

Описание метода Крылова нахождения собственного многочлена

Построим систему векторов $c^{(n+1)}$ следующим образом: возьмём вектор $c^{(0)} = (1, 0, \ldots, 0)^T$, остальные векторы системы найдём по формуле $c^{(k+1)} = Bc^{(k)}$, $k = \overline{0, n-1}$. Получив $n + 1$ векторов с учётом, что матрица имеет размерность n, можем разложить последний вектор по базису из первых n векторов. Таким образом, получим систему:

$$\begin{cases} q_1 {c_1}^{(n-1)} + \cdots + q_n {c_1}^{(0)} = {c_1}^{(n)} \\ \cdots \\ q_1 {c_n}^{(n-1)} + \cdots + q_n {c_n}^{(0)} = {c_n}^{(n)} \end{cases}$$

Решив которую методом Гаусса (что было и сделано в программе) мы получим $q_1, \ldots, q_n$, которые совпадают с коэффициентом характеристического многочлена $\lambda^n - p_1\lambda^{n-1} - \ldots - p_n$, корни которого являются собственными значениями.

Для проверки найденных коэффициентов вычислим корни многочлена с помощью программы Wolfram Alpha. Подставив найденные собственные значения в многочлен, мы получим невязку:

$$\varphi_i = P(\lambda_i)$$

Описание метода Данилевского нахождения системы собственных векторов

В методе Данилевского исходная матрица подобными преобразованиями приводится к нормальной форме Фробениуса. Таким образом, преобразование примет вид $\Phi = S^{-1}BS$.

Для нахождения матрицы S рассмотрим матрицы, с помощью которых B приводится к НФФ. Для этого на каждом шаге k матрица B умножается справа на $M_{n-k} =$

$$\begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ -\frac{{b_{n-k+1,1}}^{k-1}}{{b_{n-k+1,n-k}}^{k-1}} & \cdots & \cdots & \frac{1}{{b_{n-k+1,n-k}}^{k-1}} & \cdots & \cdots & -\frac{{b_{n-k+1,n}}^{k-1}}{{b_{n-k+1,n-k}}^{k-1}} & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & & 1 \end{pmatrix}$$

а слева на на матрицу M_{n-k}^{-1}, равную:

$$\begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ {b_{n-k+1,1}}^{k-1} & \cdots & \cdots & {b_{n-k+1,n-k}}^{k-1} & \cdots & \cdots & {b_{n-k+1,n}}^{k-1} & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & & 1 \end{pmatrix}$$

Таким образом $S = M_{n-1}*\ldots*M_1$ (в программе произведение происходило параллельно преобразованию матрицы B).

В итоге первая строка матрицы B будет состоять из коэффициентов характеристического многочлена матрицы. Используя найденные собственные значения методом Крылова (коэффициенты получатся аналогичными, т.к. оба методы являются точными, значит, можно взять и их) строим вектор $y_i = (\lambda_i^{n-1}, \lambda_i^{n-2}, \ldots, 1)$ и находим собственный вектор для i-го собственного значения по формуле:

$$x_i = \sum_{j=1}^{n} s_{ij} y_j \,, i = \overline{1,n}$$

В матричном виде: x = Sy.

i-й компонент невязки же считаем по формуле:

$$r_i = Bx_i - \lambda_i x_i,\ i = \overline{1,n}$$

Описание итерационного метода вращений (Якоби) нахождение спектра и системы собственных векторов

Для данного метода используется преобразования вращения вида $T_{kl}^T A T_{kl}$, последовательность которых стремится к диагональной матрице, состоящей из собственных значений. Последовательное же произведение матриц T_{kl} даст нам матрицу, столбцы которых образуют систему собственных векторов.

Распишем одну итерацию.

Вначале находится максимальный по модулю <u>недиагональный</u> элемент b_{kl}. В программе это делается в самой итерации. Далее идёт подсчёт $tg2\varphi = \frac{2b_{kl}}{b_{kk}-b_{ll}}$. Затем находятся элементы матрицы вращения:

$$cos\varphi = \sqrt{\frac{1}{2}(1 + \frac{1}{\sqrt{1+tg^2\varphi}})}, \qquad sin\varphi = \sqrt{\frac{1}{2}(1 - \frac{1}{\sqrt{1+tg^2\varphi}})}$$

Матрица вращения при этом принимает вид:

$$T_{kl} = \begin{pmatrix} 1 & & & & & \\ & \ldots & & & & \\ & & cos\varphi & & -sin\varphi & \\ & & & \ldots & & \\ & & sin\varphi & & cos & \\ & & & & & \ldots \\ & & & & & 1 \end{pmatrix}$$

Далее происходит ранее упомянутое преобразование подобия и последовательное произведение T_{kl} до тех пор, пока не выполнится условие:

$$\sqrt{\sum_{i,j=1, i\neq j}^{n} {b_{ij}}^2} < \varepsilon$$

По условию $\varepsilon = 10^{-5}$.

Описание степенного метода нахождения минимального собственного значения

Для нахождения минимального собственного значения будем использовать вариацию степенного метода – метод обратных итераций. Для него нам требуется найти обратную матрицу для матрицы B. В программе это сделано методом Гаусса, реализованного в первой лабораторной.

Далее берётся произвольный вектор $y^{(0)}$ (в программе взят вектор $(1, \ldots, 1)^T$) в качестве начального. На k-й итерации формула выглядит следующим образом:

$$y^{(k+1)} = B^{-1}y^{(k)}$$

Приближение собственных значений вычисляется как среднее арифметическое отношения координат $y^{(k+1)}$ и $y^{(k)}$, т.е. по формуле:

$$\lambda_k = \sum_{i=1}^{n} \frac{{y_i}^{k+1}}{{y_i}^{k}}$$

Данный процесс выполняем, пока $|\lambda_k - \lambda_{k+1}| > \varepsilon$. В конце получим максимальное собственное значение B^{-1}. Возведя его в -1 степень, получим минимальное собственное значение B.

Собственным вектором же будет являться $y^{(k+1)}$.

Листинг

Общие функции:

```
#include <iostream>
#include <vector>
#include <iomanip>

std::vector<std::vector<double>> transpositionMatrix(std::vector<std::vector<double>> M)
{
    for (int i = 0; i < 4; ++i) {
        for (int j = i + 1; j < 5; ++j) {
            std::swap(M[i][j], M[j][i]);
        }
    }

    return M;
}

std::vector<std::vector<double>> multMatrix(const std::vector<std::vector<double>> M1,
const std::vector<std::vector<double>> M2) {
    std::vector<std::vector<double>> M3 = {
      {0.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 0.0} };

    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 5; ++j) {
            for (int k = 0; k < 5; ++k) {
                M3[i][j] += M1[i][k] * M2[k][j];
            }
        }
    }
    return M3;
}

std::vector<double> multVec(const std::vector<std::vector<double>> M, const
std::vector<double> f) {
    std::vector<double> b = { 0., 0., 0., 0., 0. };

    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 5; ++j) {
            b[i] += M[i][j] * f[j];
        }
    }

    return b;
}

double getLambda(const std::vector<double> f1, const std::vector<double> f2) {
    double res = 0.;

    for (int i = 0; i < 5; ++i) {
        res += f1[i]/ f2[i];
    }

    return res / 5.;
}

std::vector<double> normV(std::vector<double> f) {
    double max = 0;
    for (int i = 0; i < 5; ++i) {
```

```
        if (abs(f[i]) > abs(max)) {
            max = f[i];
        }
    }
    for (int i = 0; i < 5; ++i) {
        f[i] /= max;
    }
    return f;
}

//Для Якоби
double getEps(std::vector<std::vector<double>> M) {
    double eps = 0.;
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 5; ++j) {
            if (i == j) {
                continue;
            }
            eps += pow(M[i][j], 2);
        }
    }

    return sqrt(eps);
}

//Для степенного
double getEps(double lk, double lk1) {
    return abs(lk - lk1);
}
```

Метод Крылова

```
std::vector<std::vector<double>> A_Result = multMatrix(A, transpositionMatrix(A));

    std::vector<std::vector<double>> C;
    C.push_back({1., 0., 0., 0., 0.});

    for (int i = 0; i < 5; ++i) {
        std::vector<double> Ck = multVec(A_Result, C[i]);
        C.push_back(Ck);
    }

    std::vector<std::vector<double>> C_result = {
       {C[4][0], C[3][0], C[2][0], C[1][0], C[0][0], C[5][0]},
      {C[4][1], C[3][1], C[2][1], C[1][1], C[0][1], C[5][1]},
      {C[4][2], C[3][2], C[2][2], C[1][2], C[0][2], C[5][2]},
      {C[4][3], C[3][3], C[2][3], C[1][3], C[0][3], C[5][3]},
      {C[4][4], C[3][4], C[2][4], C[1][4], C[0][4], C[5][4]} };

    for (int i = 0; i < 4; ++i) {
        for (int j = i + 1; j < 5; ++j) {
            for (int k = i + 1; k < 6; ++k) {
                C_result[j][k] -= C_result[j][i] / C_result[i][i] * C_result[i][k];
            }
        }
        for (int j = i + 1; j < 5; ++j) {
            C_result[j][i] = 0.;
        }
    }

    std::vector<double> answer = { 0., 0., 0., 0., 0. };

    for (int k = 4; k >= 0; --k) {
```

```
        double sum = 0;
        for (int j = k + 1; j < 5; ++j) {
            sum += C_result[k][j] * answer[j];
        }
        answer[k] = (C_result[k][5] - sum) / C_result[k][k];
    }
```

Метод Данилевского

```
std::vector<std::vector<double>> F = multMatrix(A, transpositionMatrix(A));
    std::vector<std::vector<double>> S = {
      {1.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 1.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 1.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 1.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 1.0} };

    for (int i = 3; i >= 0; --i) {
        std::vector<std::vector<double>> M = {
      {1.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 1.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 1.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 1.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 1.0} };
        std::vector<std::vector<double>> M1 = {
      {1.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 1.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 1.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 1.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 1.0} };
        for (int j = 0; j < 5; ++j) {
            M1[i][j] = F[i + 1][j];
            if (i != j) {
                M[i][j] = -F[i + 1][j] / F[i + 1][i];
            }
        }
        M[i][i] = 1 / F[i + 1][i];

        F = multMatrix(M1, multMatrix(F, M));
        S = multMatrix(S, M);
    }

    std::vector<double> lambda = { 0.23075, 0.31026, 0.691624, 0.936767, 1.17864 };
```

Метод Якоби (Итерационный метод вращений)

```
std::vector<std::vector<double>> M = multMatrix(A, transpositionMatrix(A));

    const double EPS = 0.00001;

    int counter = 0;

    std::vector<std::vector<double>> resultT = {
      {1.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 1.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 1.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 1.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 1.0} };

    while (getEps(M) > EPS) {
        counter++;
        double max = 0;
        int maxI = 0;
        int maxJ = 0;
```

```
        for (int i = 0; i < 5; ++i) {
            for (int j = 0; j < 5; ++j) {
                if (i == j) {
                    continue;
                }
                else {
                    if (abs(M[i][j]) > abs(max)) {
                        max = M[i][j];
                        maxI = i;
                        maxJ = j;
                    }
                }
            }
        }

        double tg = 2 * max / (M[maxI][maxI] - M[maxJ][maxJ]);
        double cos = sqrt((1 + 1 / sqrt(1 + pow(tg, 2))) / 2);
        double sin = sqrt(1 - pow(cos, 2));

        std::vector<std::vector<double>> T = {
      {1.0, 0.0, 0.0, 0.0, 0.0},
      {0.0, 1.0, 0.0, 0.0, 0.0},
      {0.0, 0.0, 1.0, 0.0, 0.0},
      {0.0, 0.0, 0.0, 1.0, 0.0},
      {0.0, 0.0, 0.0, 0.0, 1.0} };
        T[maxI][maxI] = cos;
        T[maxJ][maxJ] = cos;
        T[maxI][maxJ] = -sin;
        T[maxJ][maxI] = sin;

        M = multMatrix(transpositionMatrix(T), multMatrix(M, T));
        resultT = multMatrix(resultT, T);
    }
```

Степенной метод

```
const double EPS = 0.00001;

    std::vector<std::vector<double>> A1 = {
        {1.67953, 0.291252, 0.513653,  0.32638, -0.737916},
        {0.291252, 3.87842, 0.500924, 0.384012, -0.280008 },
        {0.513653, 0.500924, 1.5147, 0.051476, -0.903941},
        {-0.32638, 0.384012, 0.051476, 1.24609, 0.169331},
        {-0.737916, -0.280008, -0.903941, 0.169331, 2.59987} };

    std::vector<double> y = {1, 1, 1, 1, 1};

    double l = 1;
    double l1 = 0;

    while (getEps(l, l1) > EPS) {
        std::vector<double> y1 = multVec(A1, y);
        l = l1;
        l1 = getLambda(y1, y);
        y = normV(y1);

    }

    l1 = 1 / l1;
```

Полученные значения

Метод Крылова

```
==================================================qk==================================================

                  3.34804      -4.1574      2.35346    -0.596922    0.0546702

==========================================Lambda (Wolfram)==========================================

                  0.23075      0.31026     0.691624     0.936767      1.17864

============================================Neural============================================

              -8.64646e-08 -1.36861e-07 -5.50732e-07 -9.25458e-07 -1.41092e-06
```

Метод Данилевского

```
==============================Eigenvector 1 ==============================
       0.315005              1       0.394989      0.0705505      -0.494602

==============================Neural vector 1 ==============================
    2.78663e-06   -8.95657e-06   -4.04476e-06   -1.49191e-05   -5.55112e-16


==============================Eigenvector 2 ==============================
      -0.539085       0.816629      -0.442089       0.321697              1

==============================Neural vector 2 ==============================
    4.41085e-06    -1.4177e-05   -6.40228e-06   -2.36149e-05     -4.996e-16


==============================Eigenvector 3 ==============================
              1       0.154419      -0.298972      -0.924336       0.578349

==============================Neural vector 3 ==============================
    1.77493e-05   -5.70485e-05   -2.57629e-05   -9.50267e-05   -3.66374e-15


==============================Eigenvector 4 ==============================
       0.685852       -0.28891              1       0.854696       0.772926

==============================Neural vector 4 ==============================
    2.98262e-05    -9.5865e-05   -4.32924e-05   -0.000159684   -5.77316e-15


==============================Eigenvector 5 ==============================
      -0.699351      0.0420404              1      -0.877062       0.312977

==============================Neural vector 5 ==============================
    4.54721e-05   -0.000146153   -6.60021e-05   -0.000243449    1.11022e-15

        3.34804        -4.1574        2.35346      -0.596922      0.0546702
              1              0              0              0              0
              0              1              0              0              0
    2.23631e-16   -7.32909e-16              1   -2.44792e-16    2.13371e-17
              0              0              0              1              0
```

Метод Якоби (Итерационный метод вращений)

```
================================Result Matrix (diag - lambda)==========================
    0.230754  2.45423e-15  7.47038e-10  1.14321e-07 -1.70207e-12
  2.5479e-15      0.69165 -1.83295e-07 -5.08064e-12 -1.57556e-10
 7.47038e-10 -1.83295e-07      1.17865  1.27333e-10  1.11101e-14
 1.14321e-07 -5.08078e-12  1.27333e-10     0.310252  8.34482e-07
-1.70207e-12 -1.57556e-10  1.11101e-14  8.34482e-07     0.936732


============================================Neural=====================================

              7.46503e-08  5.48773e-07  1.58873e-06  1.35807e-07  9.33777e-07

============================Eigenvector 1 ============================
       0.315036              1       0.395011      0.0705305      -0.494646

============================Neural vector 1 ============================
   -4.13718e-08     6.2159e-08   -3.31613e-08    2.40594e-08    7.62631e-08

============================Eigenvector 2 ============================
              1       0.154291      -0.298909      -0.924219       0.578329

============================Neural vector 2 ============================
    8.35411e-08   -5.04462e-09   -1.19257e-07    1.04787e-07   -3.72854e-08

============================Eigenvector 3 ============================
      -0.699413      0.0420193              1      -0.877321       0.312975

============================Neural vector 3 ============================
   -1.21052e-07   -1.91802e-08    3.58344e-08    1.11643e-07   -6.94844e-08

============================Eigenvector 4 ============================
      -0.539118       0.816434      -0.442105       0.321708              1

============================Neural vector 4 ============================
   -3.66721e-07    4.88846e-08   -5.28213e-07   -4.26705e-07   -3.33868e-07

============================Eigenvector 5 ============================
       0.686289      -0.289065              1       0.854679       0.773142

============================Neural vector 5 ============================
   -2.99616e-07    4.53564e-07   -2.45585e-07    1.78824e-07      5.555e-07

==================================Number of iterations================================
                              103
```

Степенной метод

```
====================================Min lambda=====================================
                                    0.230265


================================Eigenvector=========================================
       0.315612              1       0.395623       0.070598       -0.494783

================================Number of iterations================================
                                    27
```

Вывод

В точных методах Крылова и Данилевского были найдены коэффициенты характеристического многочлена. Однако ни тот, ни другой метод не находит его корни, следовательно, они искались «за пределами» данных методов, а именно: с помощью WolframAlpha, что привело к погрешностям вычислений, т.к. данная программа выдаёт значения с определённой точностью. Таким образом, несмотря на то, что эти методы являются точными, мы имеем погрешность корней порядка 10^{-6} (и там, и там будет погрешность одна и та же, т.к. коэффициенты по итогу многочленов получились одинаковыми). Аналогичная причина погрешности получена в нахождении собственных векторов (вплоть до 10^{-4}). Также следует заметить, что нормальная форма Фробениуса была найдена с небольшой (порядка 10^{-16}) погрешностью, связанной с округлениями в вычислениях.

Если бы точные значения вычислялись без погрешности программы, то их погрешность была бы связана на прямую с погрешностью найденных коэффициентов. В случае метода Крылова (напомню: там решается система методом Гаусса) систему можно было бы решить с помощью выбора главного элемента по матрице, а т.ж. ортогональными методами, не влияющие на обусловленность системы, с целью нахождения более точных коэффициентов. В случае метода Данилевского преобразования подобия сильно ухудшают число обусловленности матрицы, что также может сказаться на дальнейшем поиске собственных значений и векторов этим методом.

В методе Якоби, который является итерационным, собственные значения и вектора найдены с погрешностью порядка 10^{-6}. В случае с собственными векторами у нас получились более точные значения в связи с использованием не точных решений из WolframAlpha и возможных погрешностях при вычислениях. При этом в методе Якоби не накапливается погрешность, т.к. каждый новый процесс нивелирует погрешность старого.

В степенном методе изначально находилась обратная матрица B, что сопровождалось накоплением ошибок, т.к. находилась она методом Гаусса. Метод так же является итерационным, но используется для нахождения максимального (с помощью B^{-1} – минимального) собственного значения и соответствующего ему вектора. Собственное значение совпадает с точностью до 10^{-4}. Аналогичная ситуация и с собственным вектором, соответствующего этому собственному значению.

Погрешность метода Якоби и степенного связана с различным условием выхода из итерационного процесса (в случае Якоби – сумма диагональных элементов $< \varepsilon$, в случае степенного – модуль разность искомого собственного значения на соседних шагах $< \varepsilon$). Несмотря на это, из-за округлений в вычислениях собственных значений у точных методов погрешность получилась, вообще говоря, меньше, чем в точных.

Для сравнения собственных векторов они нормировались по кубической норме. Значения их координат во всех методах совпадали с точность 10^{-4}.