МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационыые технологиии прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Численные методы»

Студент: Д. Д. Наумов
Преподаватель: И. Э. Иванов
Группа: М8О-406Б-17
Дата:
Оценка:
Подпись:

Москва, 2021

Часть 1

Задание

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Вариант: 3

$$\begin{cases} 9x_1 - 5x_2 - 6x_3 + 3x_4 = -8 \\ x_1 - 7x_2 + x_3 = 38 \\ 3x_1 - 4x_2 - 9x_3 = 47 \\ 6x_1 - x_2 + 9x_3 + 8x_4 = -8 \end{cases}$$

Теория

LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е.

$$A = LU$$

где L - нижняя треугольная матрица, U - верхняя треугольная матрица.

LU – разложение может быть построено с использованием метода Гаусса. В результате прямого хода метода Гаусса получим

$$A = A^{(0)} = M_1^{-1}A^{(1)} = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}A^{(n-1)}$$

Где $A^{(n-1)} = U$ верхняя треугольная матрица, а $L = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}$ нижняя треугольная матрица, имеющая вид

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & \dots & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 & 0 \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(2)} & \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}$$

Таким образом, искомое разложение $A = LU$ получено.

В дальнейшем LU – разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида $Ax = b$. Действительно, подставляя LU – разложение в СЛАУ, получим $LUx = b$, или $Ux = L^{-1}y$. Т.е. процесс решения СЛАУ сводится к двум простым этапам.

На первом этапе решается СЛАУ $Lz = b$. Поскольку матрица системы - нижняя треугольная, решение можно записать в явном виде:

$$z_1 = b_1,\ z_i = b_i + \sum_{j=1}^{i} l_{ij}z_j, \qquad i = \overline{2,n}$$

На втором этапе решается СЛАУ $Ux = z$ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_n = \frac{z_n}{u_{nm}},\ x_i = \frac{1}{u_{ii}}\left(z_i - \sum_{j=i+1}^{n} u_{ij}x_j\right), \qquad i = \overline{n-1,1}$$

Отметим, что второй этап эквивалентен обратному ходу методу Гаусса, тогда как первый соответствует преобразованию правой части СЛАУ в процессе прямого хода.

Реализация

```
vvdouble LUDecomposition (vvdouble mat) {
    int n = mat.size();

    vint swapMatrix(n);
    vvdouble l = CreateIdentity(n);
    vvdouble u = mat;
    for (int k = 1; k < n; ++k) {
        for (int i = k; i < n; ++i) {
            l[i][k-1] = u[i][k-1] / u[k-1][k-1];
            for (int j = k - 1; j < n; ++j) {
                u[i][j] = u[i][j] - l[i][k-1]*u[k-1][j];
            }
        }
    }
    for (int i = 0; i < n; ++i) {
        for (int j = i; j < n; ++j) {
            l[i][j] = u[i][j];
        }
    }
    return l;
}

vdouble LUSolve (vvdouble lu, const vdouble& b) {
    int n = lu.size();
    int last = n - 1;
    vdouble z = b;
    vvdouble l = GetL(lu);
    vvdouble u = GetU(lu);
    for (int i = 0; i < n; ++i) {
        for (int j = i+1; j < n; ++j) {
            z[j] -= l[j][i] * z[i];
        }
    }

    vdouble x = z;
    for (int i = n - 1; i >= 0; --i) {
        x[i] /= u[i][i];
        for (int j = i - 1; j >= 0; --j) {
            x[j] -= x[i] * u[j][i];
        }
    }

    return x;
}
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part1.t
4
9 -5 -6 3
1 -7 1 0
3 -4 -9 0
6 -1 9 8

-8 38 47 -8
(base) MacBook-Air-Dima:Program dandachok$ ./part1 <tests/input/part1.t
L: {[1, 0, 0, 0]
[0.1111, 1, 0, 0]
[0.3333, 0.3621, 1, 0]
[0.6667, -0.3621, -1.789, 1]
}
U:{[9, -5, -6, 3]
[0, -6.444, 1.667, -0.3333]
[0, 0, -7.603, -0.8793]
[0, 0, 0, 4.306]
}
LU solve:[-15.98, -8.668, -6.697, 17.44]
Gausse solve:[-15.98, -8.668, -6.697, 17.44]
```

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.1111 & 1 & 0 & 0 \\ 0.3333 & 0.3621 & 1 & 0 \\ 0.6667 & -0.3621 & -1.789 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 9 & -5 & -6 & 3 \\ 0 & -6.444 & 1.667 & -0.3333 \\ 0 & 0 & -7.603 & -0.8793 \\ 0 & 0 & 0 & 4.306 \end{pmatrix}$$

$$x = \begin{pmatrix} -8 \\ 38 \\ 47 \\ -8 \end{pmatrix}$$

Часть 2

Задание

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Вариант: 3

$$\begin{cases} 13x_1 - 5x_2 = -66 \\ -4x_1 + 9x_2 - 5x_3 = -47 \\ -x_2 - 12x_3 - 6x_4 = -43 \\ 6x_3 + 20x_4 - 5x_5 = -74 \\ 4x_4 + 5x_5 = 14 \end{cases}$$

Теория

Метод прогонки является одним из эффективных методов решения СЛАУ с трехдиагональными матрицами, возникающих при конечно-разностной аппроксимации задач для обыкновенных дифференциальных уравнений (ОДУ) и уравнений в частных производных второго порядка и является частным случаем метода Гаусса. Рассмотрим следующую СЛАУ:

$$\begin{cases} b_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ \qquad a_3x_2 + b_3x_3 + c_3x_4 = d_3 \\ \\ \qquad\qquad a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ \qquad\qquad a_nx_{n-1} + b_nx_n = d_n \end{cases}$$

решение которой будем искать в виде

$$x_i = P_ix_{i+1} + Q_i, \qquad i = \overline{1,n}$$

где P, Q – прогоночные коэффициенты, подлежащие определению.

Прогоночные коэффициенты вычисляются по следующим формулам

$$\begin{aligned} P_1 &= -\frac{c_1}{b_1}, & Q_1 &= \frac{d_1}{b_1}, & i &= 1 \\ P_i &= -\frac{c_i}{b_i + a_iP_{i-1}}, & Q_i &= \frac{d_i - a_iQ_{i-1}}{b_i + a_iP_{i-1}}, & i &= \overline{2,n-1} \\ P_n &= 0, & Q_n &= \frac{d_i - a_iQ_{i-1}}{b_i + a_iP_{i-1}}, & i &= n \end{aligned}$$

Обратный ход метода прогонки осуществляется в соответствии с выражением

Общее число операций в методе прогонки равно $8n + 1$, т.е. пропорционально числу уравнений. Такие методы решения СЛАУ называют экономичными. Для сравнения число операций в методе Гаусса пропорционально n^3. Для устойчивости метода прогонки достаточно выполнение следующих условий

$$\begin{aligned} a_i \neq 0,\ c_i \neq 0, \qquad & i = \overline{2,n-1} \\ |b_i| \geqslant |a_i| + |c_i| \qquad & i = \overline{1,b} \end{aligned}$$

Причем строгое неравенство имеет место хотя бы при одном. Здесь устойчивость понимается в смысле не накопления погрешности решения в ходе вычислительного процесса при малых погрешностях входных данных (правых частей и элементов матрицы СЛАУ).

Реализация

```
vdouble SweepMethod (vvdouble mat) {
    int n = mat.size();
    vdouble p(n, 0);
    vdouble q(n, 0);
    p[0] = -mat[0][2] / mat[0][1];
    q[0] = mat[0][3] / mat[0][1];
    for (int i = 1; i < n; ++i) {
        double a = mat[i][0];
        double b = mat[i][1];
        double c = mat[i][2];
        double d = mat[i][3];
        p[i] = - c / (b + a*p[i - 1]);
        q[i] = (d - a*q[i-1]) / (b + a*p[i-1]);
    }
    p[n - 1] = 0;
    vdouble x(n);
    x[n - 1] = q[n - 1];
    for (int i = n - 2; i >= 0; --i) {
        x[i] = p[i]*x[i + 1] + q[i];
    }
    return x;
}
```

Результаты

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part2.t
5
0 13 -5 -66
-4 9 -5 -47
-1 -12 -6 -43
6 20 -5 -74
4 5 0 14
(base) MacBook-Air-Dima:Program dandachok$ ./part2 <tests/input/part2.t
[-7, -5, 6, -4, 6]
(base) MacBook-Air-Dima:Program dandachok$
```

$$x = \begin{pmatrix} -7 \\ -5 \\ 6 \\ -4 \\ 6 \end{pmatrix}$$

Часть 3

Задание

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант: 3

$$\begin{cases} -23x_1 - 7x_2 + 5x_3 + 2x_4 = -26 \\ -7x_1 - 21x_2 + 4x_3 + 9x_4 = -55 \\ 9x_1 + 5x_2 - 31x_3 - 8x_4 = -58 \\ x_2 - 2x_3 + 10x_4 = -24 \end{cases}$$

Теория

Метод простых итераций

При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы. Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются итерационными Рассмотрим СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \quad\quad \vdots \quad\quad \ddots \quad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

Приведем СЛАУ к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \cdots + \alpha_{nn}x_n \end{cases}$$

или в векторно-матричной форме

$$x = \beta + \alpha x$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \quad \alpha \neq \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & \ddots & \dots \\ \alpha_{n1} & \dots & \alpha_{nn} \end{pmatrix}$$

Разрешим систему относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0$, $i = \overline{1,n}$ (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением).

Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы

$$\begin{aligned} \beta_i &= \frac{b_i}{a_{ii}}, \qquad && i = \overline{1,n}; \\ \alpha_{ij} &= -\frac{a_{ij}}{a_{ii}}, \qquad && i = \overline{1,n}, i \neq j; \\ \alpha_{ij} &= 0, \qquad && i = \overline{1,n}, i = j; \end{aligned}$$

При таком способе приведения исходной СЛАУ к эквивалентному виду метод простых итераций носит название метода Якоби. Тогда метод простых итераций примет вид

$$\begin{cases} x^{(0)} = \beta \\ x^{(1)} = \beta + \alpha x^{(0)} \\ x^{(2)} = \beta + \alpha x^{(1)} \\ \cdots\cdots\cdots\cdots \\ x^{(k)} = \beta + \alpha x^{(k-1)} \end{cases}$$

Видно преимущество итерационных методов по сравнению, например, с рассмотренным выше методом Гаусса. В вычислительном процессе участвуют только произведения матрицы на вектор, что позволяет работать только с ненулевыми элементами матрицы, значительно упрощая процесс хранения и обработки матриц. Имеет место следующее достаточное условие сходимости метода простых итераций. Метод простых итераций сходится к единственному решению СЛАУ (а, следовательно, и к решению исходной СЛАУ) при любом начальном приближении, если какая-либо норма матрицы эквивалентной системы меньше единицы. Приведем также необходимое и достаточное условие сходимости метода простых итераций. Для сходимости итерационного процесса необходимо и достаточно, чтобы спектр матрицы эквивалентной системы лежал внутри круга с радиусом, равным единице.

Метод Зейделя

Метод Зейделя для известного вектора итерации имеет вид:

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \ldots + \alpha_n x_n^k \\ x_2^{k+1} = \beta_2 + \alpha_{21}x_1^{k+1} + \alpha_{22}x_2^k + \ldots + \alpha_n x_n^k \\ x_3^{k+1} = \beta_3 + \alpha_{31}x_1^{k+1} + \alpha_{32}x_2^{k+1} + \alpha_{33}x_3^k + \ldots + \alpha_{3n}x_n^k \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{k+1} = \beta_n + \alpha_{n1}x_1^{k+1} + \alpha_{n2}x_2^{k+1} + \ldots + \alpha_{nn-1}x_{n-1}^{k+1} + \alpha_{nn}x_n^k \end{cases}$$

Из этой системы видно $x^{k+1} = \beta + Bx^{k+1} + Cx^k$, что где B - нижняя треугольная матрица с диагональными элементами, равными нулю, а C - верхняя треугольная матрица с диагональными элементами, отличными от нуля $\alpha = B+C$. Следовательно

Реализация

```
vdouble SimpleIter (const vvdouble& m, vdouble b, double eps) {
    int n = m.size();
    vvdouble a = m;
    for (int i = 0; i < n; ++i) {
        b[i] /= m[i][i];
        for (int j = 0; j < n; ++j) {
            a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
        }
    }
    vdouble x = b;
    vdouble prev_x;
    int count_iter = 0;
    do {
        count_iter++;
        prev_x = x;
        x = b + a * prev_x;
    } while (Norma(x - prev_x) > eps);
    std::cout << "Simple method iters count: " << count_iter << '\n';
    return x;
}

vdouble ZeidelMethod (const vvdouble& m, vdouble b, double eps) {
    int n = m.size();
    vvdouble a = m;
    for (int i = 0; i < n; ++i) {
        b[i] /= m[i][i];
        for (int j = 0; j < n; ++j) {
            a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
        }
    }
    vdouble x = b;
    vdouble prev_x;
    double count_iter = 0;
    do {
        count_iter++;
        prev_x = x;
        for (int i = 0; i < n; ++i) {
            x[i] = b[i];
            for (int j = 0; j < n; ++j) {
                x[i] += x[j] * a[i][j];
            }
        }
    } while (Norma(x - prev_x) > eps);
    std::cout << "Zeidel method iter count: " << count_iter << '\n';
    return x;
}
```

Результаты

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part3.t
4 4
-23 -7 5 2
-7 -21 4 9
9 5 -31 -8
0 1 -2 10

-26 -55 -58 -24

0.01
(base) MacBook-Air-Dima:Program dandachok$ ./part3 <tests/input/part3.t
Simple method iters count: 7
Zeidel method iter count: 5
Simple ans:
[0.9998, 2, 3, -2]
Zeidel ans:
[0.9999, 2, 3, -2]
(base) MacBook-Air-Dima:Program dandachok$
```

С помощью метода простых итераций ответ был получен за 7 шагов($\varepsilon = 0.01$):

$$x = \begin{pmatrix} 0.9998 \\ 2 \\ 3 \\ -2 \end{pmatrix}$$

С помощью улучшения Зейделя ответ был получен за 5 шагов($\varepsilon = 0.01$):

$$x = \begin{pmatrix} 0.9999 \\ 2 \\ 3 \\ -2 \end{pmatrix}$$

Как и ожидалось метод Зейдаля за меньшее число итераций доходит до заданной точности.

Часть 4

Задание

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант: 3

$$\begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

Теория

Метод вращений Якоби применим только для симметрических матриц A_{nxn} ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где Λ - диагональная матрица с собственными значениями на главной диагонали

Пусть дана симметрическая матрица A. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица $A^{(k)}$ на k–й итерации, при этом для $k = 0$ $A^{(0)} = A$.

1. Выбирается максимальный по модулю недиагональный элемент $a(k)$ матрицы

$$A^{(k)} \left(|a_{ij}^{(k)}| = \max_{l<m} |a_{lm}^{(k)}| \right)$$

2. Ставится задача найти такую ортогональную матрицу $U^{(k)}$, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$.

3. Строится матрица $A^{(k+1)}$

   $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$, где $a_{ij}^{(k+1)} \approx 0$.

   В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

$$t\left(A^{(k+1)}\right) = \sqrt{\left(\sum_{l,m;l<m} \left(a_{lm}^{(k+1)}\right)^2 \right)}$$

   .

Если $t\left(A^{(k+1)}\right) > \varepsilon$, то итерационный процесс

$$A^{(k)} = U^{(k)T} A^{(k)} U^{(k)} = U^{(k)T} U^{(k-1)T} \dots U^{(0)} A^{(0)} U^{(0)} U^{(1)} \dots U^{(k)}$$

продолжается. Если $t\left(A^{(k+1)}\right) < \varepsilon$, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются $\lambda_1 \approx a_{11}, \lambda_2 \approx a_{22}, \dots, \lambda_n \approx a_{nn}$. Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U^{(1)} = U^{(0)}U^{(1)}...U^{(k)}$, т.е.

$$x^1 = \begin{pmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{n1} \end{pmatrix}, \quad x^2 = \begin{pmatrix} u_{12} \\ u_{22} \\ \vdots \\ u_{n2} \end{pmatrix}, \quad \dots, \quad x^n = \begin{pmatrix} u_{1n} \\ u_{2n} \\ \vdots \\ u_{nn} \end{pmatrix}$$

причем эти собственные векторы будут ортогональны между собой, т.е.
$(x_l, x_m) \approx 0$, $l \neq m$.

Реализация

```
vdouble JakobiMethod (const vvdouble& mat, double eps) {
    int n = mat.size();
    int im;
    int jm;
    vvdouble a = mat;
    vvdouble r = CreateIdentity(n);
    for (int i = 0; SqrtSumNDElem(a) > eps; ++i) {

        FindMaxNDElem(a, im, jm); //find max not diagonal elem
        double q;
        if (a[im][im] == a[jm][jm]) {
            q = pi / 4;
        } else {
            q = 0.5 * atan (2*a[im][jm] / (a[im][im] - a[jm][jm]));
        }

        vvdouble u = CreateIdentity(n);
        u[im][im] = cos(q);
        u[im][jm] = -sin(q);
        u[jm][im] = sin(q);
        u[jm][jm] = cos(q);
        vvdouble ut = Trans(u);

        r = r*u;
        a = (ut * a) * u;

    }

    for (int i = 0; i < n; ++i) {
        std::cout << "x" << i << ": " << r[i];
    }
    std::cout << '\n';
    vdouble res;
    for (int i = 0; i < n; ++i) {
        res.push_back(a[i][i]);
    }
    return res;
}
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part4.t
3

5 5 3
5 -4 1
3 1 2

0.3
(base) MacBook-Air-Dima:Program dandachok$ ./part4 <tests/input/part4.t
x0: [0.8306, -0.4152, -0.3712]
x1: [0.3604, 0.9088, -0.2103]
x2: [0.4246, 0.04089, 0.9045]

[8.705, -6.239, 0.5339]
(base) MacBook-Air-Dima:Program dandachok$
```

Для матрицы $A = \begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$ найдены собственные значения

$$\lambda_1 = 8.705, \quad \lambda_2 = -6.239, \quad \lambda_3 = 0.5339$$

и следующие собственные векторы

$$x_1 = \begin{pmatrix} 0.8306 \\ -0.4152 \\ -0.3712 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 0.3604 \\ 0.9088 \\ -0.2103 \end{pmatrix}, \quad x_3 = \begin{pmatrix} 0.4246 \\ 0.04089 \\ 0.9045 \end{pmatrix},$$

Часть 5

Задание

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант: 3

$$\begin{pmatrix} 5 & -5 & -6 \\ -1 & -8 & -5 \\ 2 & 7 & -3 \end{pmatrix}$$

Теория

В основе QR-алгоритма лежит представление матрицы в виде, где $A = QR$ - ортогональная матрица, R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы. Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

$$H = E - \frac{2}{\nu^T \nu} \nu \nu^T$$

где ν - произвольный ненулевой вектор-столбец, E-единичная матрица, $\nu\nu^T$-квадратная матрицаого же размера

Матрица Хаусхолдера H_1 вычисляется

$$\nu_1^1 = a_0 + sign(a_{11}^0)\sqrt{\left(\sum_{j=1}^{n}(a_{j1}^0)\right)}$$

$$\nu_i^1 = a_{i1}^0, \qquad i = \overline{1,n}$$

$$H_1 = E - 2\frac{\nu^1 \nu^{1T}}{\nu^{1T} \nu^1}$$

Процедура QR - разложения многократно используется в QR - алгоритме вычисления собственных значений. Строится следующий итерационный процесс:

$$\begin{aligned}
&A^{(0)} = A,\\
&A^{(0)} = Q^{(0)} R^{(0)} - \text{производится QR-разложение}\\
&A^{(1)} = R^{(0)} Q^{(0)} - \text{перемножение матриц}\\
&\ldots\ldots\ldots\ldots\\
&A^{(k)} = Q^{(k)} R^{(k)} - \text{производится QR-разложение}\\
&A^{(k+1)} = R^{(k)} Q^{(k)} - \text{перемножение матриц}
\end{aligned}$$

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений мож-

но использовать следующее неравенство:

$$\left(\sum_{l=m+1}^{n} (a_{lm}^{k})^2 \right)^{1/2}$$

При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

Реализация

```
vvdouble HausseholderMatrix (const vvdouble& v) {
    return CreateIdentity(v.size()) - 2 / (Trans(v) * v)[0][0] * (v * Trans(v));
}

void QRDecomposition (vvdouble a, vvdouble& q, vvdouble& r) {
    int n = a.size();
    q = CreateIdentity(n);
    for (int i = 0; i < n - 1; ++i) {
        vvdouble b = GetColumn(a, i);
        vvdouble h = HausseholderMatrix(b);
        a = h * a;
        q = q * h;
    }
    r = a;
}

vvdouble QRMethod (vvdouble a, double eps) {
    int n = a.size();
    vvdouble q, r, an = a;
    vcomplex l(n);
    int i = 0;
    do {
        std::swap(a, an);
        QRDecomposition(a, q, r);
        an = r * q;
    } while (!FinishIterProc(an, a, eps));
    return an;
}
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part5.t
3

5 -5 -6
-1 -8 -5
2 7 -3

0.1
(base) MacBook-Air-Dima:Program dandachok$ ./part5 <tests/input/part5.t
QR Method iter count: 41
A:
{[-7.358, 5.286, -8.236]
[-7.147, -3.473, 1.266]
[-1.089e-07, 1.565e-08, 4.832]}
(base) MacBook-Air-Dima:Program dandachok$
```

В результате получилась матрица:

$$A^{(41)} = \begin{pmatrix} -7.358 & 5.286 & -8.236 \\ -7.147 & -3.473 & 1.266 \\ -1.08e-7 & 1.6e-8 & 4.832 \end{pmatrix}$$

Видно, что поддиагональные элементы достаточно малые, в то же время отчетливо прослеживается комплексно-сопряженная пара собственных значений, соответствующая блоку, образуемому элементами первого и второго столбцов. Несмотря на значительное изменение в ходе итераций самих этих элементов, собственные значения, соответствующие данному блоку и определяемые из решения квадратного уравнения $(a_{11} - \lambda)(a_{22} - \lambda) = a_{12}a_{21}$

После решения уравнения получаются следующие собстенные значения:

$$\lambda_1 \approx -5.4155 + 5.83i, \quad \lambda_2 \approx -5.4155 - 5.83i, \quad \lambda_3 \approx 4.832$$