

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №1
по курсу «Численные методы»

Студент: Д. Д. Наумов
Преподаватель: И. А. 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

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & \dots & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 & 0 \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(3)} & \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}$$

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 $Ux = L^{-1}b$. 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:

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

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

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

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

Реализация

```
1 | vdouble LUDecomposition (vdouble mat) {
2 |     int n = mat.size();
3 |
4 |     vint swapMatrix(n);
5 |     vdouble l = CreateIdentity(n);
6 |     vdouble u = mat;
7 |     for (int k = 1; k < n; ++k) {
8 |         for (int i = k; i < n; ++i) {
9 |             l[i][k-1] = u[i][k-1] / u[k-1][k-1];
10 |             for (int j = k - 1; j < n; ++j) {
11 |                 u[i][j] = u[i][j] - l[i][k-1]*u[k-1][j];
12 |             }
13 |         }
14 |     }
15 |     for (int i = 0; i < n; ++i) {
16 |         for (int j = i; j < n; ++j) {
17 |             l[i][j] = u[i][j];
18 |         }
19 |     }
20 |     return l;
21 | }
22 |
23 | vdouble LUSolve (vdouble lu, const vdouble& b) {
24 |     int n = lu.size();
25 |     int last = n - 1;
26 |     vdouble z = b;
27 |     vdouble l = GetL(lu);
28 |     vdouble u = GetU(lu);
29 |     for (int i = 0; i < n; ++i) {
30 |         for (int j = i+1; j < n; ++j) {
31 |             z[j] -= l[j][i] * z[i];
32 |         }
33 |     }
34 |
35 |     vdouble x = z;
36 |     for (int i = n - 1; i >= 0; --i) {
37 |         x[i] /= u[i][i];
38 |         for (int j = i - 1; j >= 0; --j) {
39 |             x[j] -= x[i] * u[j][i];
40 |         }
41 |     }
42 |
43 |     return x;
44 | }
```

Результаты

Пример работы программы

```
1 (base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part1.t
2 4
3 9 -5 -6 3
4 1 -7 1 0
5 3 -4 -9 0
6 6 -1 9 8
7
8 -8 38 47 -8
9 (base) MacBook-Air-Dima:Program dandachok$ ./part1 <tests/input/part1.t
10 L: {[1, 0, 0, 0]
11 [0.1111, 1, 0, 0]
12 [0.3333, 0.3621, 1, 0]
13 [0.6667, -0.3621, -1.789, 1]
14 }
15 U: {[9, -5, -6, 3]
16 [0, -6.444, 1.667, -0.3333]
17 [0, 0, -7.603, -0.8793]
18 [0, 0, 0, 4.306]
19 }
20 LU solve: [-15.98, -8.668, -6.697, 17.44]
21 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:

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

решение которой будем искать в виде

$$x_i = P_i x_{i+1} + Q_i, \quad i = \overline{1, n}$$

где P, Q – прогоночные коэффициенты, подлежащие определению.

Прогоночные коэффициенты вычисляются по следующим формулам

$$\begin{aligned} P_1 &= -\frac{c_1}{b_1}, & Q_1 &= \frac{d_1}{b_1}, & i &= 1 \\ P_i &= -\frac{c_i}{b_i + a_i P_{i-1}}, & Q_i &= \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}, & i &= \overline{2, n-1} \\ P_n &= 0, & Q_n &= \frac{d_n - a_n Q_{n-1}}{b_n + a_n P_{n-1}}, & i &= n \end{aligned}$$

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

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

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

```
1 | vdouble SweepMethod (vdouble mat) {
2 |     int n = mat.size();
3 |     vdouble p(n, 0);
4 |     vdouble q(n, 0);
5 |     p[0] = -mat[0][2] / mat[0][1];
6 |     q[0] = mat[0][3] / mat[0][1];
7 |     for (int i = 1; i < n; ++i) {
8 |         double a = mat[i][0];
9 |         double b = mat[i][1];
10 |        double c = mat[i][2];
11 |        double d = mat[i][3];
12 |        p[i] = - c / (b + a*p[i - 1]);
13 |        q[i] = (d - a*q[i-1]) / (b + a*p[i-1]);
14 |    }
15 |    p[n - 1] = 0;
16 |    vdouble x(n);
17 |    x[n - 1] = q[n - 1];
18 |    for (int i = n - 2; i >= 0; --i) {
19 |        x[i] = p[i]*x[i + 1] + q[i];
20 |    }
21 |    return x;
22 | }
```


Результаты

```
1 | (base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part2.t
2 | 5
3 | 0 13 -5 -66
4 | -4 9 -5 -47
5 | -1 -12 -6 -43
6 | 6 20 -5 -74
7 | 4 5 0 14
8 | (base) MacBook-Air-Dima:Program dandachok$ ./part2 <tests/input/part2.t
9 | [-7, -5, 6, -4, 6]
10 | (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

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

Приведем СЛАУ к эквивалентному виду

[illegible]

или в векторно-матричной форме

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

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

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

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

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

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

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:

[illegible]

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

```
1 | vdouble SimpleIter (const vdouble& m, vdouble b, double eps) {
2 |     int n = m.size();
3 |     vdouble a = m;
4 |     for (int i = 0; i < n; ++i) {
5 |         b[i] /= m[i][i];
6 |         for (int j = 0; j < n; ++j) {
7 |             a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
8 |         }
9 |     }
10 |     vdouble x = b;
11 |     vdouble prev_x;
12 |     int count_iter = 0;
13 |     do {
14 |         count_iter++;
15 |         prev_x = x;
16 |         x = b + a * prev_x;
17 |     } while (Norma(x - prev_x) > eps);
18 |     std::cout << "Simple method iters count: " << count_iter << '\n';
19 |     return x;
20 | }
21 |
22 | vdouble ZeidelMethod (const vdouble& m, vdouble b, double eps) {
23 |     int n = m.size();
24 |     vdouble a = m;
25 |     for (int i = 0; i < n; ++i) {
26 |         b[i] /= m[i][i];
27 |         for (int j = 0; j < n; ++j) {
28 |             a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
29 |         }
30 |     }
31 |     vdouble x = b;
32 |     vdouble prev_x;
33 |     double count_iter = 0;
34 |     do {
35 |         count_iter++;
36 |         prev_x = x;
37 |         for (int i = 0; i < n; ++i) {
38 |             x[i] = b[i];
39 |             for (int j = 0; j < n; ++j) {
40 |                 x[i] += x[j] * a[i][j];
41 |             }
42 |         }
43 |     } while (Norma(x - prev_x) > eps);
44 |     std::cout << "Zeidel method iter count: " << count_iter << '\n';
45 |     return x;
46 | }
```

Результаты

```
1 | (base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part3.t
2 | 4 4
3 | -23 -7 5 2
4 | -7 -21 4 9
5 | 9 5 -31 -8
6 | 0 1 -2 10
7 |
8 | -26 -55 -58 -24
9 |
10 | 0.01
11 | (base) MacBook-Air-Dima:Program dandachok$ ./part3 <tests/input/part3.t
12 | Simple method iters count: 7
13 | Zeidel method iter count: 5
14 | Simple ans:
15 | [0.9998, 2, 3, -2]
16 | Zeidel ans:
17 | [0.9999, 2, 3, -2]
18 | (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 Зейделя 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 $A_{n \times n}$ ($A = A^T$) 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 $\Lambda = U^T A U$, 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 $A^{(k+1)}$

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

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

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

Если $t(A^{(k+1)}) > \varepsilon$, 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 $t(A^{(k+1)}) < \varepsilon$, то итерационный процесс останавливается, и в качестве искоемых 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 $U^{(1)} = U^{(0)} U^{(1)} \dots U^{(k)}$, 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.

$$(x_l, x_m) \approx 0, \quad l \neq m.$$

Реализация

```
1 vdouble JakobiMethod (const vdouble& mat, double eps) {
2     int n = mat.size();
3     int im;
4     int jm;
5     vdouble a = mat;
6     vdouble r = CreateIdentity(n);
7     for (int i = 0; SqrtSumNDElem(a) > eps; ++i) {
8
9         FindMaxNDElem(a, im, jm); //find max not diagonal elem
10        double q;
11        if (a[im][im] == a[jm][jm]) {
12            q = pi / 4;
13        } else {
14            q = 0.5 * atan (2*a[im][jm] / (a[im][im] - a[jm][jm]));
15        }
16
17        vdouble u = CreateIdentity(n);
18        u[im][im] = cos(q);
19        u[im][jm] = -sin(q);
20        u[jm][im] = sin(q);
21        u[jm][jm] = cos(q);
22        vdouble ut = Trans(u);
23
24        r = r*u;
25        a = (ut * a) * u;
26
27    }
28
29    for (int i = 0; i < n; ++i) {
30        std::cout << "x" << i << ": " << r[i];
31    }
32    std::cout << '\n';
33    vdouble res;
34    for (int i = 0; i < n; ++i) {
35        res.push_back(a[i][i]);
36    }
37    return res;
38 }
```

Результаты

Пример работы программы

```
1 (base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part4.t
2 3
3
4 5 5 3
5 5 -4 1
6 3 1 2
7
8 0.3
9 (base) MacBook-Air-Dima:Program dandachok$ ./part4 <tests/input/part4.t
10 x0: [0.8306, -0.4152, -0.3712]
11 x1: [0.3604, 0.9088, -0.2103]
12 x2: [0.4246, 0.04089, 0.9045]
13
14 [8.705, -6.239, 0.5339]
15 (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 матрица того же размера

Матрица Хаусхолдера H_1 вычисляется

$$\nu_1^1 = a_0 + \text{sign}(a_{11}^0) \sqrt{\left(\sum_{j=1}^n (a_{j1}^0)^2 \right)}$$

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

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

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

$$A^{(0)} = A,$$

$$A^{(0)} = Q^{(0)} R^{(0)} - \text{производится QR-разложение}$$

$$A^{(1)} = R^{(0)} Q^{(0)} - \text{перемножение матриц}$$

.....

$$A^{(k)} = Q^{(k)} R^{(k)} - \text{производится QR-разложение}$$

$$A^{(k+1)} = R^{(k)} Q^{(k)} - \text{перемножение матриц}$$

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

```
1 | vdouble HausseholderMatrix (const vdouble& v) {
2 |     return CreateIdentity(v.size()) - 2 / (Trans(v) * v)[0][0] * (v * Trans(v));
3 | }
4 |
5 | void QRDecomposition (vdouble a, vdouble& q, vdouble& r) {
6 |     int n = a.size();
7 |     q = CreateIdentity(n);
8 |     for (int i = 0; i < n - 1; ++i) {
9 |         vdouble b = GetColumn(a, i);
10 |         vdouble h = HausseholderMatrix(b);
11 |         a = h * a;
12 |         q = q * h;
13 |     }
14 |     r = a;
15 | }
16 |
17 | vdouble QRMethod (vdouble a, double eps) {
18 |     int n = a.size();
19 |     vdouble q, r, an = a;
20 |     vcomplex l(n);
21 |     int i = 0;
22 |     do {
23 |         std::swap(a, an);
24 |         QRDecomposition(a, q, r);
25 |         an = r * q;
26 |     } while (!FinishIterProc(an, a, eps));
27 |     return an;
28 | }
```


Результаты

Пример работы программы

```
1 (base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part5.t
2 3
3
4 5 -5 -6
5 -1 -8 -5
6 2 7 -3
7
8 0.1
9 (base) MacBook-Air-Dima:Program dandachok$ ./part5 <tests/input/part5.t
10 QR Method iter count: 41
11 A:
12 {[-7.358, 5.286, -8.236]
13 [-7.147, -3.473, 1.266]
14 [-1.089e-07, 1.565e-08, 4.832]}
15 (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 собственные значения:

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