

**Московский авиационный институт
(национальный исследовательский университет)**

**Институт №8 «Информационные технологии и прикладная
математика»**

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент: И. Б. Белов
Преподаватель: Д. Е. Пивоваров
Группа: М8О-303Б-21
Дата:
Оценка:
Подпись:

Москва, 2024

1.1 LU - разложение матриц

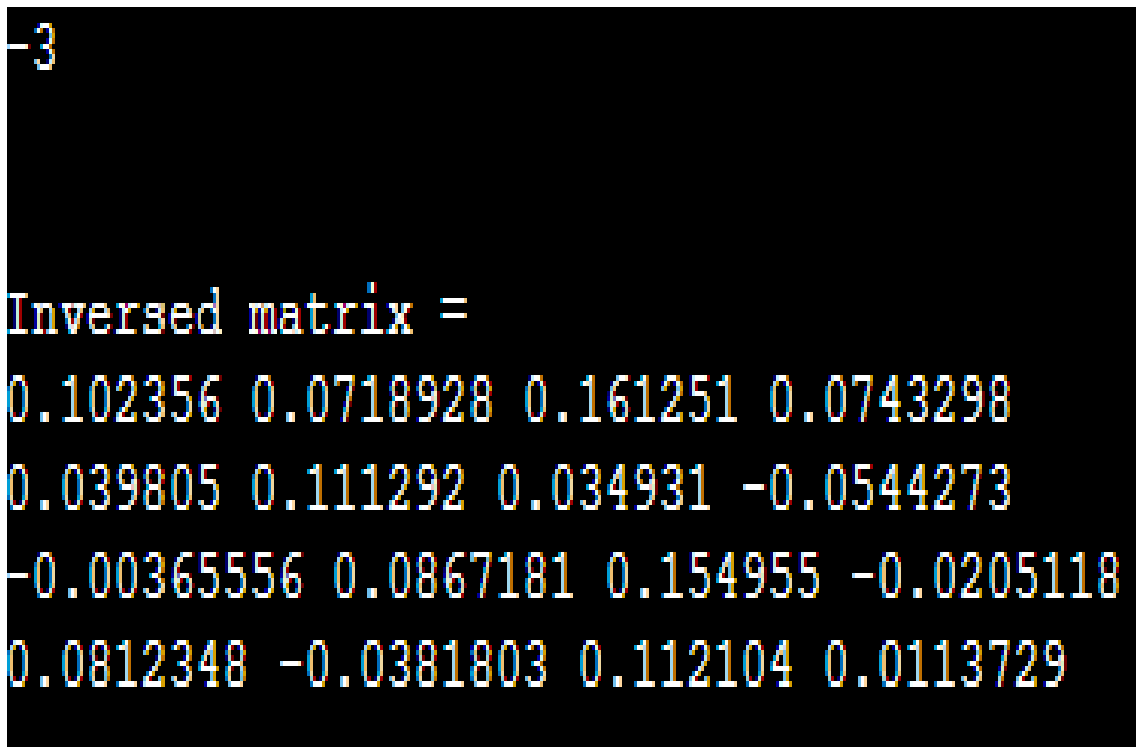
1 Постановка задачи

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: 2

$$\begin{cases} -2x_1 + 7x_2 - 8x_3 + 6x_4 = -39 \\ 4x_1 + 4x_2 - 7x_4 = 41 \\ -x_1 - 3x_2 + 6x_3 + 3x_4 = 4 \\ 9x_1 - 7x_2 - 2x_3 - 8x_4 = 113 \end{cases}$$

2 Результаты работы



```
-3

Inversed matrix =
0.102356 0.0718928 0.161251 0.0743298
0.039805 0.111292 0.034931 -0.0544273
-0.00365556 0.0867181 0.154955 -0.0205118
0.0812348 -0.0381803 0.112104 0.0113729
```

Рис. 1: Вывод программы в консоли

3 Исходный код

```
1 | #include <bits/stdc++.h>
2 |
3 | using namespace std;
4 | using matrix = vector<vector<double> >;
5 |
6 |
7 |
8 | pair<matrix, matrix> lu_decomposition(matrix& coefficients, matrix& results) {
9 |     int n1=coefficients.size(), m1=coefficients[0].size(), m2 = results[0].size();
10 |     matrix L(n1), U = coefficients;
11 |     for (int i=0; i<n1; i++)
12 |         for (int j=0; j<m1; j++)
13 |             L[i].push_back(0);
14 |
15 |     for (int k=0; k<n1; k++) {
16 |         if (U[k][k] == 0) {
17 |             for (int i=k+1; i<n1; i++) {
18 |                 if (U[i][k] != 0) {
19 |                     swap(U[k], U[i]);
20 |                     swap(L[k], L[i]);
21 |                     swap(coefficients[k], coefficients[i]);
22 |                     swap(results[k], results[i]);
23 |                     break;
24 |                 }
25 |             }
26 |         }
27 |         L[k][k] = 1;
28 |         for (int i=k+1; i<n1; i++) {
29 |             L[i][k] = U[i][k]/U[k][k];
30 |             if (U[i][k] == 0)
31 |                 continue;
32 |             for(int j=k; j<m1; j++)
33 |                 U[i][j] -= L[i][k]*U[k][j];
34 |
35 |         }
36 |     }
37 |
38 |     return make_pair(L, U);
39 | }
40 |
41 |
42 | double get_determinant(matrix& coefficients, matrix& results) {
43 |     auto [_, U] = lu_decomposition(coefficients, results);
44 |     double det = 1;
45 |     for (int i=0; i<coefficients.size(); i++)
46 |         det *= U[i][i];
47 |     return det;
```

```

48 }
49
50
51 matrix calculate_decisions(matrix& coefficients, matrix& results) {
52     auto [L, U] = lu_decomposition(coefficients, results);
53     matrix res = results;
54
55     for (int k=0; k<res[0].size(); k++)
56         for (int i=0; i<res.size(); i++)
57             for (int j=0; j<i; j++)
58                 res[i][k] -= res[j][k]*L[i][j];
59     for (int k=0; k<res[0].size(); k++) {
60         for (int i=coefficients.size()-1; i>-1; i--) {
61             for (int j=i+1; j<results.size(); j++) {
62                 res[i][k] -= res[j][k]*U[i][j];
63             }
64             res[i][k] /= U[i][i];
65         }
66     }
67
68     return res;
69 }
70
71
72 matrix get_inverse_matrix(matrix& matrix1) {
73     matrix E(matrix1.size());
74     for (int i=0; i<matrix1.size(); i++)
75         for (int j=0; j<matrix1.size(); j++)
76             E[i].push_back((i == j) ? 1 : 0);
77     return calculate_decisions(matrix1, E);
78 }
79
80 void print_matrix(const matrix& matrix1) {
81     for(const auto& vect: matrix1) {
82         for (auto x: vect)
83             cout << x << " ";
84         cout << endl;
85     }
86 }
87
88 int main() {
89     matrix coefficient_matrix{
90         {2, 7, -8, 6},
91         {4, 4, 0, -7},
92         {-1, -3, 6, 3},
93         {9, -7, -2, -8}
94     };
95
96     matrix equation_roots = {

```

```

97         {-39},
98         {41},
99         {4},
100        {113}
101    };
102
103    auto [l, u] = lu_decomposition(coefficient_matrix, equation_roots);
104
105    cout << endl << "L =" << endl;
106    print_matrix(l);
107
108    cout << endl << endl;
109    cout << "U =" << endl;
110    print_matrix(u);
111
112    cout << endl << endl;
113    cout << "det = " << get_determinant(coefficient_matrix, equation_roots) << endl;
114
115    cout << endl << endl << "Decisions =" << endl;
116    matrix decisions = calculate_decisions(coefficient_matrix, equation_roots);
117    print_matrix(decisions);
118
119    cout << endl << endl << "Inversed matrix =" << endl;
120    matrix inversed = get_inverse_matrix(coefficient_matrix);
121    print_matrix(inversed);
122    return 0;
123 }

```

1.2 Метод прогонки

4 Постановка задачи

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: 2

$$\begin{cases} 10x_1 + 5x_2 = -120 \\ 3x_1 + 10x_2 - 2x_3 = -91 \\ 2x_2 - 9x_3 - 5x_4 = 5 \\ 5x_3 - 16x_4 - 4x_5 = -74 \\ -8x_4 + 16x_5 = -56 \end{cases}$$

5 Результаты работы

```
The solution for matrix:
10      5      0      0      0
3       10     -2      0      0
0        2     -9     -5      0
0        0      5     16     -4
0        0      0     -8     16
Is this vector:
-9 -6 2 -7 -7
```

Рис. 2: Вывод программы в консоли

6 Исходный код

```
1 | #include <iostream>
2 | #include <fstream>
3 | #include <vector>
4 | #include <sstream>
5 | #include <cmath>
6 |
7 | class NotANeededMatrix : public std::exception {
8 | public:
9 |     const char* what() const noexcept override {
10 |         return "Sorry this matrix doesn't fit the requirements!\n";
11 |     }
12 | };
13 |
14 | double find_det(const std::vector<std::vector<double>>& matrix) {
15 |     double det = 1;
16 |     for (size_t i = 0; i < matrix.size(); ++i) {
17 |         det *= matrix[i][i];
18 |     }
19 |     return det;
20 | }
21 |
22 | void check_coefficients(double a = 0, double b = 0, double c = 0) {
23 |     return;
24 | }
25 |
26 | std::vector<double> tridiagonal_solution(const std::vector<std::vector<double>>&
27 |     matrix, const std::vector<double>& vector_b) {
28 |     check_coefficients(matrix[0][0], matrix[0][1]);
29 |     std::vector<double> vector_alphas = {-matrix[0][1] / matrix[0][0]};
30 |     std::vector<double> vector_betas = {vector_b[0] / matrix[0][0]};
31 |
32 |     for (size_t i = 1; i < matrix.size() - 1; ++i) {
33 |         check_coefficients(matrix[i][i - 1], matrix[i][i], matrix[i][i + 1]);
34 |         double y_i = matrix[i][i] + matrix[i][i - 1] * vector_alphas[i - 1];
35 |         double a_i = -matrix[i][i + 1] / y_i;
36 |         double b_i = (vector_b[i] - matrix[i][i - 1] * vector_betas[i - 1]) / y_i;
37 |         vector_alphas.push_back(a_i);
38 |         vector_betas.push_back(b_i);
39 |     }
40 |
41 |     check_coefficients(matrix.back()[matrix.back().size() - 2], matrix.back().back());
42 |     double y_n = (matrix.back().back() + matrix.back()[matrix.back().size() - 2] *
43 |         vector_alphas.back());
44 |     std::vector<double> vector_x = {round((vector_b.back() - matrix.back()[matrix.back
45 |         ().size() - 2] * vector_betas.back()) / y_n)};
46 |
47 |     for (int i = static_cast<int>(matrix.size()) - 2; i >= 0; --i) {
```

```

45         vector_x.insert(vector_x.begin(), round(vector_alphas[i] * vector_x[0] +
46             vector_betas[i]));
47     }
48     return vector_x;
49 }
50
51 int main() {
52
53     std::vector<std::vector<double>> matrix = {
54         {10.0, 5.0, 0, 0, 0},
55         {3.0, 10.0, -2.0, 0.0, 0},
56         {0, 2, -9, -5, 0},
57         {0, 0, 5, 16, -4},
58         {0, 0, 0, -8, 16}
59     };
60
61     std::vector<double> vector = {-120, -91, 5, -74, -56};
62     find_det(matrix);
63     auto solution = tridiagonal_solution(matrix, vector);
64     std::cout << "The solution for matrix:\n";
65     for (const auto& row : matrix) {
66         for (double val : row) {
67             std::cout << val << "\t";
68         }
69         std::cout << std::endl;
70     }
71     std::cout << "Is this vector:\n";
72     for (double val : solution) {
73         std::cout << val << " ";
74     }
75     std::cout << std::endl;
76 }

```


1.3 Метод простых итераций. Метод Зейделя

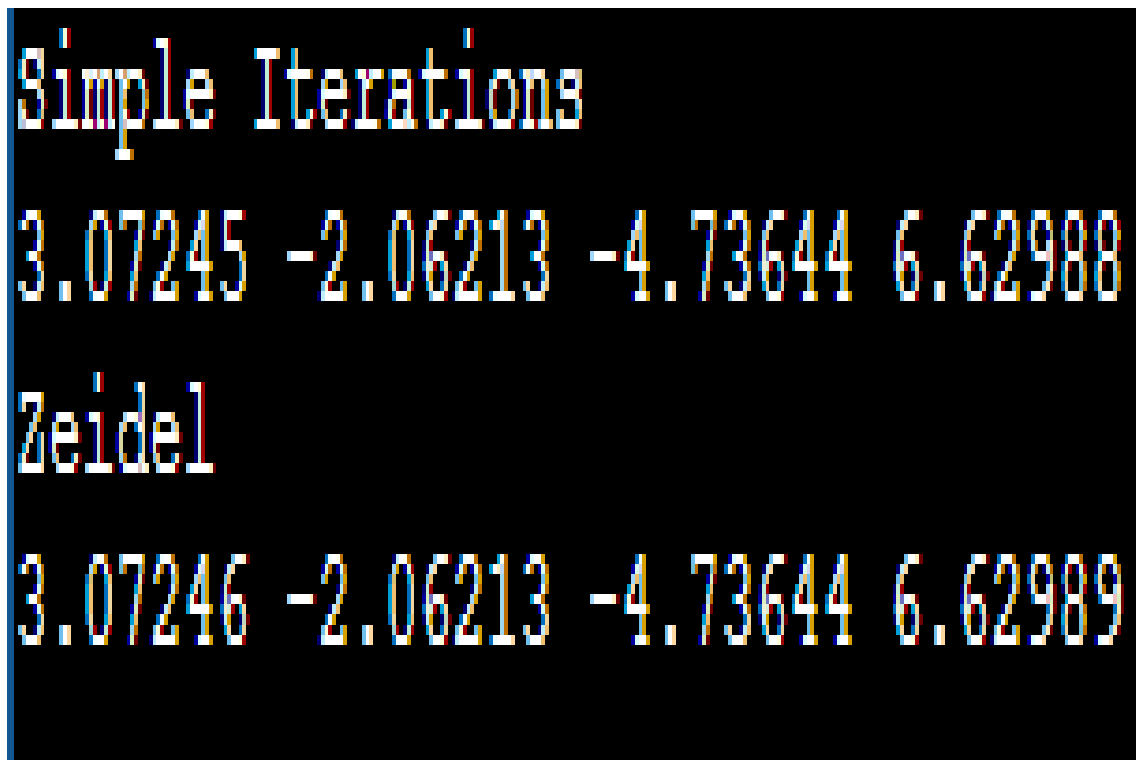
7 Постановка задачи

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: 2

$$\begin{cases} 24x_1 + 2x_2 + 4x_3 - 9x_4 = -9 \\ -6x_1 - 27x_2 - 8x_3 - 6x_4 = -76 \\ -4x_1 + 8x_2 + 19x_3 + 6x_4 = -79 \\ 4x_1 + 5x_2 - 3x_3 - 13x_4 = -70 \end{cases}$$

8 Результаты работы



```
Simple Iterations
3.07245 -2.06213 -4.73644 6.62988

Zeidel
3.07246 -2.06213 -4.73644 6.62989
```

Рис. 3: Вывод программы в консоли

9 Исходный код

```
1  #include <fstream>
2  #include <sstream>
3  #include <iostream>
4  #include <vector>
5  #include <cmath>
6
7  std::vector<std::vector<double>> deepcopy(const std::vector<std::vector<double>>&
8      matrix) {
9      std::vector<std::vector<double>> result(matrix.size(), std::vector<double>(matrix
10         [0].size()));
11      for (size_t i = 0; i < matrix.size(); ++i) {
12          for (size_t j = 0; j < matrix[i].size(); ++j) {
13              result[i][j] = matrix[i][j];
14          }
15      }
16      return result;
17  }
18
19  std::pair<std::vector<std::vector<double>>, std::vector<double>> read_matrix_from_file
20      (const std::string& filename) {
21      std::ifstream file(filename);
22      std::vector<std::vector<double>> matrix;
23      std::vector<double> vector;
24      std::string line;
25      while (std::getline(file, line)) {
26          if (!line.empty()) {
27              std::istringstream iss(line);
28              std::vector<double> row;
29              double value;
30              while (iss >> value) {
31                  row.push_back(value);
32              }
33              if (vector.empty()) {
34                  vector = row;
35              } else {
36                  matrix.push_back(row);
37              }
38          }
39      }
40      return std::make_pair(matrix, vector);
41  }
42
43  double matrix_norm1(const std::vector<std::vector<double>>& matrix) {
44      double max_sum = 0.0;
45      for (const auto& row : matrix) {
```

```

45     double row_sum = 0.0;
46     for (double val : row) {
47         row_sum += std::abs(val);
48     }
49     if (row_sum > max_sum) {
50         max_sum = row_sum;
51     }
52 }
53 return max_sum;
54 }
55
56
57 double dot_product(const std::vector<double>& a, const std::vector<double>& b) {
58     double result = 0.0;
59     for (size_t i = 0; i < a.size(); ++i) {
60         result += a[i] * b[i];
61     }
62     return result;
63 }
64
65 std::vector<double> subtract_vectors(const std::vector<double>& a, const std::vector<
66     double>& b) {
67     std::vector<double> result(a.size());
68     for (size_t i = 0; i < a.size(); ++i) {
69         result[i] = a[i] - b[i];
70     }
71     return result;
72 }
73
74 double norm(const std::vector<double>& vector) {
75     double max_val = vector[0];
76     for (size_t i = 1; i < vector.size(); ++i) {
77         if (vector[i] > max_val) {
78             max_val = vector[i];
79         }
80     }
81     return max_val;
82 }
83
84 std::pair<std::vector<std::vector<double>>, std::vector<double>> normal_view(const std
85     ::vector<std::vector<double>>& matrix, const std::vector<double>& vector) {
86     std::vector<std::vector<double>> res = matrix;
87     std::vector<double> res_v = vector;
88     for (size_t i = 0; i < res.size(); ++i) {
89         double delim = res[i][i];
90         for (size_t j = 0; j < res.size(); ++j) {
91             res[i][j] /= -delim;
92         }
93         res_v[i] /= delim;

```

```

92     res[i][i] = 0;
93 }
94 return std::make_pair(res, res_v);
95 }
96
97 std::vector<double> sum_vectors(const std::vector<double>& vect1, const std::vector<
98     double>& vect2) {
99     std::vector<double> result(vect1.size());
100     for (size_t i = 0; i < vect1.size(); ++i) {
101         result[i] = vect1[i] + vect2[i];
102     }
103     return result;
104 }
105
106 std::vector<double> prod_matrix(const std::vector<std::vector<double>>& a, const std:::
107     vector<double>& vector) {
108     std::vector<double> result(a.size());
109     for (size_t i = 0; i < a.size(); ++i) {
110         double sum = 0.0;
111         for (size_t j = 0; j < a[i].size(); ++j) {
112             sum += a[i][j] * vector[j];
113         }
114         result[i] = sum;
115     }
116     return result;
117 }
118
119 std::vector<double> gauss_seidel(const std::vector<std::vector<double>>& a, const std
120     ::vector<double>& b, double epsilon) {
121     auto [a_norm, b_norm] = normal_view(a, b);
122     double alpha_norm = matrix_norm1(a_norm);
123     std::vector<double> x_start(a_norm.size(), 0);
124     std::vector<double> x_new = b_norm;
125
126     while (true) {
127         if (alpha_norm / (1 - alpha_norm) * norm(subtract_vectors(x_new, x_start)) <=
128             epsilon) {
129             break;
130         }
131         x_start = x_new;
132         for (size_t j = 0; j < a_norm.size(); ++j) {
133             double x_res = 0;
134             for (size_t l = 0; l < a_norm.size(); ++l) {
135                 x_res += x_new[l] * a_norm[j][l];
136             }
137             x_res += b_norm[j];
138             x_new[j] = x_res;
139         }
140     }
141 }

```

```

137     return x_new;
138 }
139
140 std::vector<double> simple_iteration(const std::vector<std::vector<double>>& a, const
    std::vector<double>& b, double epsilon) {
141     auto [a_norm, b_norm] = normal_view(a, b);
142     double alpha_norm = matrix_norm1(a_norm);
143     std::vector<double> x_start(a_norm.size(), 0);
144     size_t max_iters = 100000;
145     std::vector<double> x_new = b_norm;
146
147     for (size_t j = 0; j < max_iters; ++j) {
148         if (alpha_norm / (1 - alpha_norm) * norm(subtract_vectors(x_new, x_start)) >
            epsilon) {
149             x_start = x_new;
150             x_new = sum_vectors(prod_matrix(a_norm, x_new), b_norm);
151         } else {
152             break;
153         }
154     }
155     return x_new;
156 }
157 int main() {
158     std::vector<std::vector<double>> matrix = {
159         {24.0, 2.0, 4.0, -9.0},
160         {-6.0, 27.0, -8.0, -6.0},
161         {-4.0, 8.0, 19.0, 6.0},
162         {4.0, 5.0, -3.0, -13.0}
163     };
164
165     // 1x4
166     std::vector<double> vector = {-9.0, -76.0, -79.0, -70.0};
167     double epsilon = 0.0001;
168     auto result = simple_iteration(matrix, vector, epsilon);
169     auto result2 = gauss_seidel(matrix, vector, epsilon);
170     std::cout << "Simple Iterations\n";
171     for (double element : result) {
172         std::cout << element << " ";
173     }
174     std::cout << std::endl;
175     std::cout << "Zeidel\n";
176     for (double element : result2) {
177         std::cout << element << " ";
178     }
179
180     std::cout << std::endl;
181     return 0;
182 }

```

1.4 Метод вращений

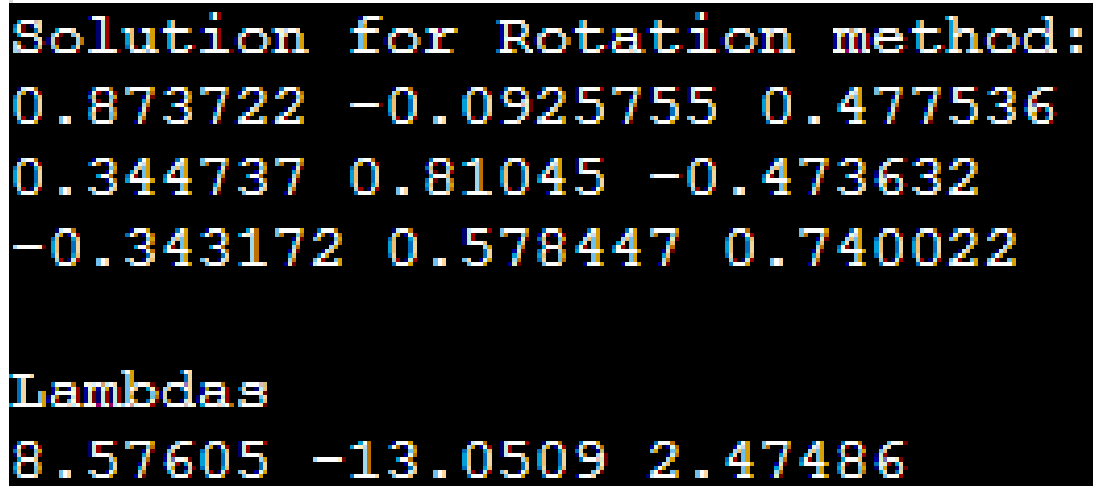
10 Постановка задачи

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: 2

$$\begin{pmatrix} -9 & 7 & 5 \\ 7 & 8 & 9 \\ 5 & 9 & 8 \end{pmatrix}$$

11 Результаты работы



```
Solution for Rotation method:
0.873722 -0.0925755 0.477536
0.344737 0.81045 -0.473632
-0.343172 0.578447 0.740022

Lambdas
8.57605 -13.0509 2.47486
```

Рис. 4: Вывод программы в консоли

12 Исходный код

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <cmath>
5
6  std::vector<std::vector<double>> find_transport_matrix(const std::vector<std::vector<
    double>>& matrix) {
7      std::vector<std::vector<double>> transport_matrix = matrix;
8      for (size_t i = 0; i < matrix.size(); ++i) {
9          for (size_t j = i + 1; j < matrix.size(); ++j) {
10             std::swap(transport_matrix[i][j], transport_matrix[j][i]);
11         }
12     }
13     return transport_matrix;
14 }
15
16 std::vector<std::vector<double>> make_e_matrix(int size) {
17     std::vector<std::vector<double>> e_matrix(size, std::vector<double>(size, 0));
18     for (int i = 0; i < size; ++i) {
19         e_matrix[i][i] = 1.0;
20     }
21     return e_matrix;
22 }
23
24 double scholar_multiply(const std::vector<double>& vector_1, const std::vector<double
    >& vector_2) {
25     double result = 0;
26     for (size_t i = 0; i < vector_1.size(); ++i) {
27         result += vector_1[i] * vector_2[i];
28     }
29     return result;
30 }
31
32
33 std::vector<std::vector<double>> multiply_matrix(const std::vector<std::vector<double
    >>& matrix_1, const std::vector<std::vector<double>>& matrix_2) {
34     size_t n = matrix_1.size();
35     size_t m = matrix_1[0].size();
36     size_t p = matrix_2[0].size();
37     if (m != matrix_2.size()) {
38         throw std::invalid_argument("Matrix dimensions do not match.");
39     }
40
41     std::vector<std::vector<double>> result(n, std::vector<double>(p, 0));
42     for (size_t i = 0; i < n; ++i) {
43         for (size_t j = 0; j < p; ++j) {
44             for (size_t k = 0; k < m; ++k) {
```

```

45         result[i][j] += matrix_1[i][k] * matrix_2[k][j];
46     }
47 }
48 }
49 return result;
50 }
51
52
53 std::vector<int> find_max_non_diagonal(const std::vector<std::vector<double>>& matrix)
54 {
55     double max_val = matrix[0][1];
56     std::vector<int> max_indices = {0, 1};
57     for (size_t i = 0; i < matrix.size(); ++i) {
58         for (size_t j = i + 1; j < matrix[i].size(); ++j) {
59             if (std::abs(matrix[i][j]) > std::abs(max_val)) {
60                 max_val = matrix[i][j];
61                 max_indices = {static_cast<int>(i), static_cast<int>(j)};
62             }
63         }
64     }
65     return max_indices;
66 }
67
68 double find_phi(const std::vector<std::vector<double>>& matrix, int i, int j) {
69     if (matrix[i][i] == matrix[j][j]) {
70         return M_PI / 4;
71     }
72     return 0.5 * std::atan((2 * matrix[i][j]) / (matrix[i][i] - matrix[j][j]));
73 }
74
75 std::vector<std::vector<double>> make_rotation_matrix(double angle, size_t size, int
76 i_, int j_) {
77     std::vector<std::vector<double>> res = make_e_matrix(size);
78     res[i_][j_] = -std::sin(angle);
79     res[j_][i_] = std::sin(angle);
80     res[i_][i_] = res[j_][j_] = std::cos(angle);
81     return res;
82 }
83
84 double find_lim(const std::vector<std::vector<double>>& matrix) {
85     double res = 0;
86     for (size_t i = 0; i < matrix.size() - 1; ++i) {
87         for (size_t j = i + 1; j < matrix[i].size(); ++j) {
88             res += matrix[i][j] * matrix[i][j];
89         }
90     }
91     return std::sqrt(res);
92 }

```



```

92
93
94 std::pair<std::vector<std::vector<double>>, std::vector<double>> rotation_solution(
    const std::vector<std::vector<double>>& matrix, double epsilon) {
95     std::vector<std::vector<double>> matrix_a = matrix;
96     std::vector<std::vector<double>> matrix_res_u = make_e_matrix(matrix.size());
97     while (std::abs(find_lim(matrix_a)) > epsilon) {
98         std::vector<int> res_ij = find_max_non_diagonal(matrix_a);
99         double phi = find_phi(matrix_a, res_ij[0], res_ij[1]);
100        std::vector<std::vector<double>> matrix_u = make_rotation_matrix(phi, matrix_a.
            size(), res_ij[0], res_ij[1]);
101        matrix_res_u = multiply_matrix(matrix_res_u, matrix_u);
102        matrix_a = multiply_matrix(multiply_matrix(find_transport_matrix(matrix_u),
            matrix_a), matrix_u);
103    }
104    std::vector<std::vector<double>> result_vector;
105    std::vector<double> lambdas(matrix_a.size());
106    for (size_t i = 0; i < matrix_a.size(); ++i) {
107        result_vector.push_back({matrix_res_u[i]});
108        lambdas[i] = matrix_a[i][i];
109    }
110
111    return std::make_pair(matrix_res_u, lambdas);
112 }
113
114 int main() {
115     std::vector<std::vector<double>> matrix = {
116         {7.0, 3.0, -1.0},
117         {3.0, -7.0, -8.0},
118         {-1.0, -8.0, -2.0}
119     };
120
121
122     auto rotation_solution_result = rotation_solution(matrix, 0.01);
123     auto matrix_res_u = rotation_solution_result.first;
124     auto lambdas = rotation_solution_result.second;
125
126     std::cout << "Solution for Rotation method:\n";
127     for (const auto& row : matrix_res_u) {
128         for (double val : row) {
129             std::cout << val << " ";
130         }
131         std::cout << std::endl;
132     }
133     std::cout << std::endl;
134     std::cout << "Lambdas\n";
135     for (auto const& var: lambdas) {
136         std::cout << var << " ";
137     }

```

```
138 ||  
139 ||   return 0;  
140 || }
```

1.5 QR – разложение матриц

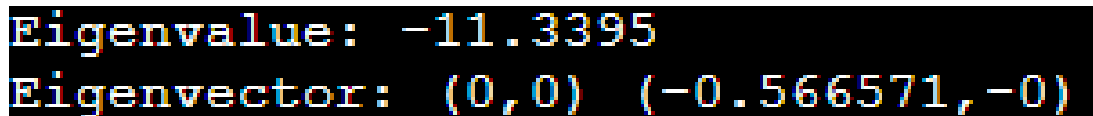
13 Постановка задачи

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: 2

$$\begin{pmatrix} -6 & -4 & 0 \\ -7 & 6 & -7 \\ -2 & -6 & -7 \end{pmatrix}$$

14 Результаты работы



```
Eigenvalue: -11.3395  
Eigenvector: (0,0) (-0.566571,-0)
```

Рис. 5: Вывод программы в консоли

15 Исходный код

```
1  #include <iostream>
2  #include <utility>
3  #include <vector>
4  #include <cmath>
5  #include <complex>
6
7  double row_col_mul(const std::vector<double>& row, const std::vector<double>& col) {
8      double result = 0;
9      for (size_t i = 0; i < row.size(); ++i) {
10         result += row[i] * col[i];
11     }
12     return result;
13 }
14
15 std::pair<std::complex<double>, std::complex<double>> solve_equation(double a, double
    b, double c) {
16     std::complex<double> delta = std::sqrt(std::complex<double>(b * b - 4 * a * c, 0));
17     std::complex<double> root1 = (b + delta) / (2 * a);
18     std::complex<double> root2 = (b - delta) / (2 * a);
19     return std::make_pair(root1, root2);
20 }
21
22 std::vector<std::vector<double>> col_row_mul(const std::vector<double>& col, const std
    ::vector<double>& row) {
23     std::vector<std::vector<double>> result(col.size(), std::vector<double>(row.size(),
        0));
24     for (size_t i = 0; i < col.size(); ++i) {
25         for (size_t j = 0; j < row.size(); ++j) {
26             result[i][j] = col[i] * row[j];
27         }
28     }
29     return result;
30 }
31
32 std::vector<std::vector<double>> matrix_matrix_mul(const std::vector<std::vector<
    double>>& matrix1, const std::vector<std::vector<double>>& matrix2) {
33     std::vector<std::vector<double>> result(matrix1.size(), std::vector<double>(matrix2
        [0].size(), 0));
34     for (size_t i = 0; i < matrix1.size(); ++i) {
35         for (size_t j = 0; j < matrix2[0].size(); ++j) {
36             for (size_t k = 0; k < matrix1[0].size(); ++k) {
37                 result[i][j] += matrix1[i][k] * matrix2[k][j];
38             }
39         }
40     }
41     return result;
42 }
```

```

43
44 std::vector<std::vector<double>> get_e_matrix(size_t size) {
45     std::vector<std::vector<double>> result(size, std::vector<double>(size, 0));
46     for (size_t i = 0; i < size; ++i) {
47         result[i][i] = 1;
48     }
49     return result;
50 }
51
52 int sign(double element) {
53     if (element > 0) {
54         return 1;
55     } else if (element < 0) {
56         return -1;
57     }
58     return 0;
59 }
60
61 double vector_second_norm_2(const std::vector<double>& vector) {
62     double result = 0;
63     for (double val : vector) {
64         result += val * val;
65     }
66     return std::sqrt(result);
67 }
68
69 std::vector<std::vector<double>> get_h(const std::vector<double>& vector) {
70     double lower = row_col_mul(vector, vector);
71     auto upper = col_row_mul(vector, vector);
72     auto E = get_e_matrix(upper.size());
73     std::vector<std::vector<double>> second(upper.size(), std::vector<double>(upper.
74         size(), 0));
75     for (size_t i = 0; i < upper.size(); ++i) {
76         for (size_t j = 0; j < upper.size(); ++j) {
77             second[i][j] = -2 / lower * upper[i][j];
78         }
79     }
80     std::vector<std::vector<double>> result(upper.size(), std::vector<double>(upper.
81         size(), 0));
82     for (size_t i = 0; i < upper.size(); ++i) {
83         for (size_t j = 0; j < upper.size(); ++j) {
84             result[i][j] = E[i][j] + second[i][j];
85         }
86     }
87     return result;
88 }
89
90 std::pair<std::vector<std::vector<double>>, std::vector<std::vector<double>>> gen_qr(
91     std::vector<std::vector<double>> matrix) {

```

```

89     auto A = matrix;
90     std::vector<std::vector<double>>> Q(matrix.size(), std::vector<double>(matrix.size()
    , 0));
91     std::vector<std::vector<std::vector<double>>>> H_all;
92     std::vector<double> vector(matrix.size());
93     for (size_t i = 0; i < matrix.size() - 1; ++i) {
94         vector.clear();
95         std::vector<double> row;
96         for (const auto& col : A) {
97             row.push_back(col[i]);
98         }
99         vector.insert(vector.end(), i, 0);
100        vector.push_back(A[i][i] + sign(A[i][i]) * vector_second_norm_2(row));
101        vector.insert(vector.end(), row.begin() + i + 1, row.end());
102        auto H = get_h(vector);
103        A = matrix_matrix_mul(H, A);
104        H_all.push_back(H);
105    }
106    Q = H_all[0];
107    for (size_t i = 1; i < H_all.size(); ++i) {
108        Q = matrix_matrix_mul(Q, H_all[i]);
109    }
110    return std::make_pair(Q, A);
111 }
112
113 std::pair<double, std::pair<std::complex<double>, std::complex<double>>>> qr_solve(std
    ::vector<std::vector<double>>> matrix, double eps) {
114     auto A = std::move(matrix);
115     while (sqrt(A[1][0] * A[1][0] + A[2][0] * A[2][0]) > eps) {
116         auto qr = gen_qr(A);
117         A = matrix_matrix_mul(qr.second, qr.first);
118     }
119     auto roots = solve_equation(A[1][1], A[2][2], A[1][2] * A[2][1]);
120     return std::make_pair(A[0][0], roots);
121 }
122
123 int main() {
124     // Example usage
125     std::vector<std::vector<double>>> matrix = {{-6, -4, 0},
126                                                {-7, 6, -7},
127                                                {-2, -6, -7}};
128     double epsilon = 0.0001;
129     auto result = qr_solve(matrix, epsilon);
130     std::cout << "Eigenvalue: " << result.first << std::endl;
131     std::cout << "Eigenvector: ";
132     std::cout << result.second.first << " " << result.second.second;
133     std::cout << std::endl;
134     return 0;
135 }

```