

**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: Я. А. Хайруллина
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: 26

$$\begin{cases} -2x_1 - 9x_2 - 3x_3 + 7x_4 = -26 \\ -7x_1 + 8x_2 + 2x_3 + 5x_4 = -25 \\ -6x_1 + 2x_2 = -16 \\ -3x_2 + 8x_3 - 3x_4 = -5 \end{cases}$$

2 Результаты работы

```
Solve of equation Ax=b
[3.00000]
[1.00000]
[-1.00000]
[-2.00000]
det(A) = 4788.0000
A(-1) =
[-0.01921  0.01963  -0.18317  -0.01211]
[-0.05764  0.05890  -0.04950  -0.03634]
[0.00251  0.06266  -0.07393  0.11028]
[0.06433  0.10819  -0.14766  -0.00292]
```

Рис. 1: Вывод программы в консоли

3 Исходный код

Общие файлы для всех подзадач 1 лабораторной работы:

```
1  #ifndef _LAB1_
2  #define _LAB1_
3
4  #include <stdio.h>
5
6  typedef struct Matrix {
7      double** data;
8      unsigned int width;
9      unsigned int height;
10 } Matrix;
11
12 Matrix* create_matrix(void);
13 void remove_matrix(Matrix*);
14 void resize_matrix(Matrix*, const int, const int);
15 void print_matrix(Matrix*, FILE*);
16 void scan_matrix(Matrix*, FILE*);
17 Matrix* multiple_matrix(Matrix*, Matrix*);
18 void exchange_str(Matrix*, int, int);
19 void exchange_col(Matrix*, int, int);
20 Matrix* transpose_matrix(Matrix*);
21
22 #endif

```



```
1  #include "lab1.h"
2  #include <stdlib.h>
3
4  Matrix* create_matrix(void) {
5      Matrix* new_matrix = (Matrix*)malloc(sizeof(Matrix));
6      new_matrix->width = new_matrix->height = 0;
7      new_matrix->data = NULL;
8      return new_matrix;
9  }
10 void remove_matrix(Matrix* matrix) {
11     if (!matrix)
12         return;
13     for (int i = 0; i < matrix->height; i++)
14         free(matrix->data[i]);
15     free(matrix->data);
16     matrix->data = NULL;
17     matrix->height = matrix->width = 0;
18 }
19 void resize_matrix(Matrix* matrix, const int height, const int width) {
20     if (height > 0) {
21         matrix->data = (double**)realloc(matrix->data, sizeof(double*) * height);
22         for (int i = matrix->height; i < height; i++) {
23             matrix->data[i] = (double*) malloc(sizeof(double) * (width <= 0 ? matrix->
```

```

24         width : width));
25         for (int j = 0; j < width; j++)
26             matrix->data[i][j] = 0;
27     }
28     if (width > 0)
29         for (int i = 0; i < matrix->height; i++) {
30             matrix->data[i] = (double*)realloc(matrix->data[i], sizeof(double) * width)
31             ;
32             for (int j = matrix->width; j < width; j++)
33                 matrix->data[i][j] = 0;
34         }
35     if (width > 0)
36         matrix->width = width;
37     if (height > 0)
38         matrix->height = height;
39 }
40 void print_matrix(Matrix* matrix, FILE* stream) {
41     int i, j;
42     if (!matrix->data)
43         return;
44     for (i = 0; i < matrix->height; i++) {
45         fputc('[', stream);
46         for (j = 0; j < matrix->width; j++)
47             fprintf(stream, "%.5f ", matrix->data[i][j]);
48         fprintf(stream, "\b\b]\n");
49     }
50 }
51
52 void scan_matrix(Matrix* matrix, FILE* stream) {
53     int c = 0;
54     float a;
55     for (int i = 0; c != EOF; i++) {
56         resize_matrix(matrix, i + 1, -1);
57         c = 0;
58         for (int j = 0; c != '\n'; j++) {
59             if (!i)
60                 resize_matrix(matrix, -1, j + 1);
61             fscanf(stream, "%f", &a);
62             matrix->data[i][j] = a;
63             c = getc(stream);
64             if (c == EOF) {
65                 resize_matrix(matrix, i, -1);
66                 return;
67             }
68         }
69     }
70 }

```

```

71
72 Matrix* multiple_matrix(Matrix* A, Matrix* B) {
73     Matrix* result = create_matrix();
74     int i, j, k;
75     if (A->width != B->height)
76         return NULL;
77     resize_matrix(result, A->height, B->width);
78     for (i = 0; i < result->height; i++)
79         for (j = 0; j < result->width; j++)
80             for (k = 0; k < A->width; k++)
81                 result->data[i][j] += A->data[i][k] * B->data[k][j];
82     return result;
83 }
84
85 void exchange_str(Matrix* matrix, int i1, int i2) {
86     double temp;
87     for (int j = 0; j < matrix->width; j++) {
88         temp = matrix->data[i1][j];
89         matrix->data[i1][j] = matrix->data[i2][j];
90         matrix->data[i2][j] = temp;
91     }
92 }
93
94 void exchange_col(Matrix* matrix, int j1, int j2) {
95     double temp;
96     for (int i = 0; i < matrix->width; i++) {
97         temp = matrix->data[i][j1];
98         matrix->data[i][j1] = matrix->data[i][j2];
99         matrix->data[i][j2] = temp;
100     }
101 }
102 Matrix* transpose_matrix(Matrix* matrix) {
103     Matrix* result;
104     int i, j;
105     if (!matrix)
106         return NULL;
107     result = create_matrix();
108     resize_matrix(result, matrix->width, matrix->height);
109     for (i = 0; i < result->height; i++)
110         for (j = 0; j < result->width; j++)
111             result->data[i][j] = matrix->data[j][i];
112     return result;
113 }

```

Коэффициенты перед иксами системы:

```

1 | -2 -9 -3 7
2 | -7 8 2 5
3 | -6 2 0 0
4 | 0 -3 8 -3

```

Вектор b:

```
1 -26
2 -25
3 -16
4 -5
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "lab1.h"
5
6 static inline double absolute(const double a) {
7     return a > 0 ? a : -a;
8 }
9
10 Matrix** LU_decomposition(Matrix* matrix) {
11     int size = matrix->height, max;
12     Matrix** LUP = (Matrix**)malloc(sizeof(Matrix*) * 3);
13
14     for (int i = 0; i < 3; i++) {
15         LUP[i] = create_matrix();
16         resize_matrix(LUP[i], size, size);
17     }
18
19     for (int i = 0; i < size; i++)
20         LUP[2]->data[i][i] = LUP[0]->data[i][i] = 1;
21
22     for (int i = 0; i < size; i++)
23         for (int j = 0; j < size; j++)
24             LUP[1]->data[i][j] = matrix->data[i][j];
25
26     for (int j = 0; j < size; j++) {
27         max = j;
28         for (int i = j + 1; i < size; i++)
29             if (absolute(LUP[1]->data[i][j]) > absolute(LUP[1]->data[max][j]))
30                 max = i;
31         exchange_str(LUP[1], j, max);
32         exchange_str(LUP[2], j, max);
33         exchange_str(LUP[0], j, max);
34         exchange_col(LUP[0], j, max);
35
36         for (int i = j + 1; i < size; i++) {
37             LUP[0]->data[i][j] = LUP[1]->data[i][j] / LUP[1]->data[j][j];
38             for (int k = j; k < size; k++)
39                 LUP[1]->data[i][k] -= LUP[0]->data[i][j] * LUP[1]->data[j][k];
40         }
41     }
42     return LUP;
43 }
```

```

44
45 Matrix* LU_solve(Matrix** LUP, Matrix* vector) {
46     Matrix* result;
47     int i, j;
48     if (!LUP)
49         return NULL;
50     result = multiple_matrix(LUP[2], vector);
51     for (i = 0; i < result->height; i++)
52         for (j = 0; j < i; j++)
53             result->data[i][0] -= result->data[j][0] * LUP[0]->data[i][j];
54     for (i = result->height - 1; i >= 0; i--) {
55         for (j = result->height - 1; j > i; j--)
56             result->data[i][0] -= result->data[j][0] * LUP[1]->data[i][j];
57         result->data[i][0] /= LUP[1]->data[i][i];
58     }
59     return result;
60 }
61
62 Matrix* calculate_decisions(Matrix* matrix, Matrix* vector) {
63     Matrix** LUP, * result;
64     int i;
65     LUP = LU_decomposition(matrix);
66     result = LU_solve(LUP, vector);
67     for (i = 0; i < 3; i++) {
68         remove_matrix(LUP[i]);
69         free(LUP[i]);
70     }
71     free(LUP);
72     return result;
73 }
74
75 double get_determinant(Matrix* matrix) {
76     Matrix** LUP = LU_decomposition(matrix);
77     int i, j = 0;
78     double det;
79     if (!LUP)
80         return 0;
81     for (i = 0; i < LUP[2]->height; i++)
82         if (!LUP[2]->data[i][i])
83             j++;
84     det = (j % 2 == 0) ? 1 : -1;
85     for (i = 0; i < LUP[1]->height; i++)
86         det *= LUP[1]->data[i][i];
87     return det;
88 }
89
90 Matrix* matrix_inversion(Matrix* matrix) {
91     Matrix** LUP = LU_decomposition(matrix), * inverse, * right_part, * temp;
92     int i, j;

```

```

93     if (!LUP)
94         return NULL;
95     inverse = create_matrix();
96     resize_matrix(inverse, matrix->height, matrix->width);
97     right_part = create_matrix();
98     resize_matrix(right_part, matrix->height, 1);
99     right_part->data[0][0] = 1;
100    for (j = 0; j < inverse->width; j++) {
101        temp = LU_solve(LUP, right_part);
102        for (i = 0; i < inverse->height; i++)
103            inverse->data[i][j] = temp->data[i][0];
104        if (j != inverse->width - 1)
105            exchange_str(right_part, j + 1, j);
106        remove_matrix(temp);
107        free(temp);
108    }
109    {
110        for (i = 0; i < 3; i++) {
111            remove_matrix(LUP[i]);
112            free(LUP[i]);
113        }
114        free(LUP);
115        remove_matrix(right_part);
116        free(right_part);
117    }
118    return inverse;
119 }
120
121 int main(void) {
122     int i;
123     Matrix* matrix = create_matrix(), * vector = create_matrix(), * result;
124     FILE* fmatrix, * fvector;
125
126     fmatrix = fopen("lab1-1m.txt", "r");
127     fvector = fopen("lab1-1v.txt", "r");
128
129     if (fmatrix == NULL || fvector == NULL) {
130         fprintf(stderr, "Invalid name of file\n");
131         return 0;
132     }
133
134     scan_matrix(matrix, fmatrix);
135     fclose(fmatrix);
136     scan_matrix(vector, fvector);
137     fclose(fvector);
138
139     if (result = calculate_decisions(matrix, vector)) {
140         printf("Solve of equation Ax=b\n");
141         print_matrix(result, stdout);

```



```

142     remove_matrix(result);
143     free(result);
144 }
145 if (matrix->height == matrix->width) {
146     printf("det(A) = %.4f\n", get_determinant(matrix));
147 }
148 if (result = matrix_inversion(matrix)) {
149     printf("A(-1) = \n");
150     print_matrix(result, stdout);
151     remove_matrix(result);
152     free(result);
153 }
154 remove_matrix(vector);
155 remove_matrix(matrix);
156 free(vector);
157 free(matrix);
158
159 return 0;
160 }

```

1.2 Метод прогонки

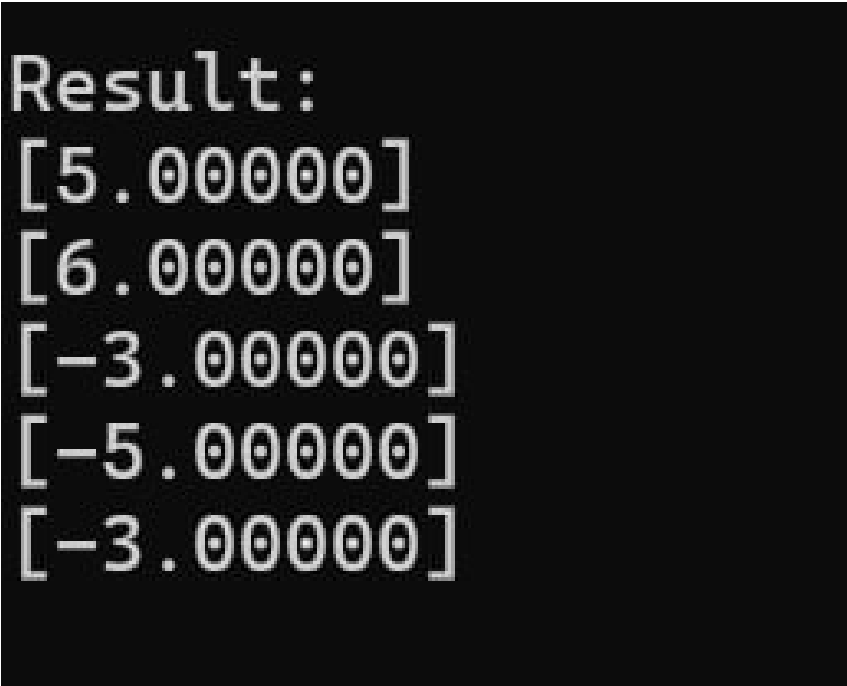
4 Постановка задачи

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: 26

$$\begin{cases} -12x_1 - 7x_2 = -102 \\ -7x_1 - 11x_2 - 3x_3 = -92 \\ -7x_2 + 21x_3 - 8x_4 = -65 \\ 4x_3 - 13x_4 + 5x_5 = 38 \\ -6x_4 + 14x_5 = -12 \end{cases}$$

5 Результаты работы



```
Result:
[5.00000]
[6.00000]
[-3.00000]
[-5.00000]
[-3.00000]
```

Рис. 2: Вывод программы в консоли

6 Исходный код

Коэффициенты перед искоми системы:

```
1 | 0 -12 -7
2 | -7 -11 -3
3 | -7 21 -8
4 | 4 -13 5
5 | -6 14 0
```

Вектор b:

```
1 | -102
2 | -92
3 | -65
4 | 38
5 | -12
```

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include "lab1.h"
4 |
5 | static inline double absolute(double a) {
6 |     return a > 0 ? a : -a;
7 | }
8 | Matrix* tridiagonal_algorithm(Matrix* matrix, Matrix* vector) {
9 |     Matrix* PQ, * result;
10 |     int i;
11 |     PQ = create_matrix();
12 |     resize_matrix(PQ, matrix->height - 1, 2);
13 |     result = create_matrix();
14 |     resize_matrix(result, matrix->height, 1);
15 |
16 |     PQ->data[0][0] = -matrix->data[0][2] / matrix->data[0][1];
17 |     PQ->data[0][1] = vector->data[0][0] / matrix->data[0][1];
18 |     if (absolute(matrix->data[0][1]) < absolute(matrix->data[0][2]) ||
19 |         absolute(matrix->data[matrix->height - 1][1]) < absolute(matrix->data[matrix->
20 |             height - 1][2])) {
21 |         fprintf(stderr, "Singular matrix\n");
22 |         return NULL;
23 |     }
24 |     for (i = 1; i < PQ->height; i++) {
25 |         if (absolute(matrix->data[i][1]) < absolute(matrix->data[i][0]) + absolute(
26 |             matrix->data[i][2])) {
27 |             fprintf(stderr, "Singular matrix\n");
28 |             return NULL;
29 |         }
30 |         double temp = matrix->data[i][0] * PQ->data[i - 1][0] + matrix->data[i][1];
31 |         PQ->data[i][0] = -matrix->data[i][2] / temp;
32 |         PQ->data[i][1] = (vector->data[i][0] - matrix->data[i][0] * PQ->data[i - 1][1])
33 |             / temp;
```

```

31     }
32     i = result->height - 1;
33     result->data[i][0] = (vector->data[i][0] - matrix->data[i][0] * PQ->data[i - 1][1])
34         /
35         (matrix->data[i][0] * PQ->data[i - 1][0] + matrix->data[i][1]);
36     for (i = result->height - 2; i >= 0; i--)
37         result->data[i][0] = PQ->data[i][0] * result->data[i + 1][0] + PQ->data[i][1];
38     remove_matrix(PQ);
39     free(PQ);
40     return result;
41 }
42 Matrix* (* const TDMA)(Matrix*, Matrix*) = tridiagonal_algorithm;
43
44 int main(void) {
45     int i;
46     Matrix* matrix = create_matrix(), * vector = create_matrix(), * result;
47     FILE* fmatrix, * fvector;
48
49     fmatrix = fopen("lab1-2m.txt", "r");
50     fvector = fopen("lab1-2v.txt", "r");
51
52     if (!fmatrix || !fvector) {
53         fprintf(stderr, "Invalid name of file\n");
54         return 0;
55     }
56
57     scan_matrix(matrix, fmatrix);
58     fclose(fmatrix);
59     scan_matrix(vector, fvector);
60     fclose(fvector);
61     if (result = TDMA(matrix, vector)) {
62         printf("Result: \n");
63         print_matrix(result, stdout);
64         remove_matrix(result);
65         free(result);
66     }
67     remove_matrix(vector);
68     remove_matrix(matrix);
69     free(vector);
70     free(matrix);
71
72     return 0;
73 }

```

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: 26

$$\begin{cases} 18x_1 - 2x_3 + 7x_4 = 50 \\ -x_1 + 14x_2 - 3x_3 + 2x_4 = 2 \\ 5x_1 + 5x_2 + 26x_3 + 7x_4 = 273 \\ -2x_1 - 6x_2 + 9x_3 + 24x_4 = 111 \end{cases}$$

8 Результаты работы

```
Estimate of number of iteration: k + 1 less then 3
Step: 2
Solve:
[2.14583]
[1.88542]
[8.47957]
[2.09534]
```

Рис. 3: Вывод программы в консоли

9 Исходный код

Эпсилон:

```
1 || 10
```

Коэффициенты перед иксами системы:

```
1 || 18 0 -2 7
2 || -1 14 -3 2
3 || 5 5 26 7
4 || -2 -6 9 24
```

Вектор b:

```
1 || 50
2 || 2
3 || 273
4 || 111
```

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <math.h>
4 | #include "lab1.h"
5 |
6 | static inline double absolute(const double a) {
7 |     return a > 0 ? a : -a;
8 | }
9 | double matrix_norm(Matrix* matrix) {
10 |     double sum = 0, norm = 0;
11 |     int i, j;
12 |     if (!matrix)
13 |         return 0;
14 |     for (i = 0; i < matrix->height; i++) {
15 |         for (j = 0; j < matrix->width; j++)
16 |             sum += absolute(matrix->data[i][j]);
17 |         norm = sum > norm ? sum : norm;
18 |         sum = 0;
19 |     }
20 |     return norm;
21 | }
22 | int estimation(double eps, double a, double b) {
23 |     return (int)(log(eps * (1 - a) / b) / log(a));
24 | }
25 | double error(Matrix* vector_prev, Matrix* vector_cur, double alpha_norm) {
26 |     int i;
27 |     double epsilon_k;
28 |     Matrix* temp = create_matrix();
29 |     resize_matrix(temp, vector_prev->height, 1);
30 |     for (i = 0; i < vector_prev->height; i++)
31 |         temp->data[i][0] = vector_prev->data[i][0] - vector_cur->data[i][0];
```

```

32     if (alpha_norm < 1)
33         epsilon_k = matrix_norm(temp) * alpha_norm / (1 - alpha_norm);
34     else
35         epsilon_k = matrix_norm(temp);
36     remove_matrix(temp);
37     free(temp);
38     return epsilon_k;
39 }
40 Matrix* seidel_method(Matrix* alpha, Matrix* betta, double epsilon) {
41     Matrix* current, * previous;
42     double alpha_norm = matrix_norm(alpha), err = epsilon + 1;
43     int i, j, step;
44     if (alpha_norm > 1) {
45         fprintf(stderr, "Norm of alpha is %.4f\n", alpha_norm);
46         return NULL;
47     }
48     if (alpha_norm != 1) {
49         printf("Estimate of number of iteration: k + 1 less then %d\n",
50             estimation(epsilon, alpha_norm, matrix_norm(betta)));
51     }
52     current = create_matrix();
53     resize_matrix(current, betta->height, 1);
54     for (i = 0; i < betta->height; i++)
55         current->data[i][0] = betta->data[i][0];
56     previous = create_matrix();
57     resize_matrix(previous, betta->height, 1);
58     for (step = 1; err >= epsilon; step++) {
59         for (i = 0; i < betta->height; i++)
60             previous->data[i][0] = current->data[i][0];
61         for (i = 0; i < betta->height; i++) {
62             current->data[i][0] = betta->data[i][0];
63             for (j = 0; j < alpha->width; j++)
64                 current->data[i][0] += alpha->data[i][j] * (j < i ? current->data[j][0]
65                     : previous->data[j][0]);
66         }
67         err = error(previous, current, alpha_norm);
68     }
69     printf("Step: %d\n", step);
70     remove_matrix(previous);
71     free(previous);
72     return current;
73 }
74 Matrix* simple_iteration_method(Matrix* matrix, Matrix* vector, double epsilon) {
75     Matrix* alpha, * betta, * result;
76     double norm;
77     int i, j;
78     if (!matrix || !vector || matrix->width != matrix->height || matrix->height !=
79         vector->height || vector->width != 1) {

```

```

79     fprintf(stderr, "Matrix or vector have has the incorrect size\n");
80     return NULL;
81 }
82 for (i = 0; i < matrix->height; i++)
83     if (!matrix->data[i][i]) {
84         fprintf(stderr, "Singular matrix\n");
85         return NULL;
86     }
87 alpha = create_matrix();
88 resize_matrix(alpha, matrix->height, matrix->width);
89 betta = create_matrix();
90 resize_matrix(betta, vector->height, 1);
91 for (i = 0; i < matrix->height; i++) {
92     for (j = 0; j < matrix->width; j++)
93         if (i == j)
94             alpha->data[i][j] = 0;
95         else
96             alpha->data[i][j] = -matrix->data[i][j] / matrix->data[i][i];
97     betta->data[i][0] = vector->data[i][0] / matrix->data[i][i];
98 }
99
100 result = seidel_method(alpha, betta, epsilon);
101 remove_matrix(alpha);
102 remove_matrix(betta);
103 free(alpha);
104 free(betta);
105
106 return result;
107 }
108
109 int main(void) {
110     int i;
111     double epsilon;
112     Matrix* matrix = create_matrix(), * vector = create_matrix(), * result;
113     FILE* fmatrix, * fvector, * fepsilon;
114
115     fmatrix = fopen("lab1-3m.txt", "r");
116     fvector = fopen("lab1-3v.txt", "r");
117     fepsilon = fopen("lab1-3e.txt", "r");
118
119     fscanf(fepsilon, "%lf", &epsilon);
120
121     if (epsilon <= 0) {
122         fprintf(stderr, "Negative value of error\n");
123         return 0;
124     }
125     if (!fmatrix || !fvector) {
126         fprintf(stderr, "Invalid name of file\n");
127         return 0;

```



```

128     }
129
130     scan_matrix(matrix, fmatrix);
131     fclose(fmatrix);
132     scan_matrix(vector, fvector);
133     fclose(fvector);
134     fclose(fepsilon);
135     if (result = simple_iteration_method(matrix, vector, epsilon)) {
136         printf("Solve: \n");
137         print_matrix(result, stdout);
138         remove_matrix(result);
139         free(result);
140     }
141     remove_matrix(vector);
142     remove_matrix(matrix);
143     free(vector);
144     free(matrix);
145     return 0;
146 }

```

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: 26

$$\begin{pmatrix} -4 & 1 & -7 \\ 1 & 9 & 1 \\ -7 & 1 & 7 \end{pmatrix}$$

11 Результаты работы

```
Vectors:
[0.89939  0.00000 -0.43714]
[0.00000  1.00000  0.00000]
[0.43714  0.00000  0.89939]
Values:
[-7.40225]
[9.00000]
[10.40225]
```

Рис. 4: Вывод программы в консоли

12 Исходный код

Эпсилон:

1 || 5

Матрица:

1 || -4 1 -7
2 || 1 9 1
3 || -7 1 7

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <math.h>
4 | #include "lab1.h"
5 |
6 | static inline double absolute(const double a) {
7 |     return a > 0 ? a : -a;
8 | }
9 | Matrix* create_rotation_matrix(unsigned int size, unsigned int l, unsigned int m,
10 |     double phi) {
11 |     Matrix* rotate;
12 |     int i;
13 |     if (size <= 1 || size <= m || l == m)
14 |         return NULL;
15 |     rotate = create_matrix();
16 |     resize_matrix(rotate, size, size);
17 |     for (i = 0; i < size; i++)
18 |         rotate->data[i][i] = 1;
19 |     rotate->data[l][l] = rotate->data[m][m] = cos(phi);
20 |     rotate->data[l][m] = -(rotate->data[m][l] = sin(phi));
21 |     return rotate;
22 | }
23 | double error(Matrix* matrix) {
24 |     int i, j;
25 |     double err = 0;
26 |     if (!matrix)
27 |         return 0;
28 |     for (i = 0; i < matrix->height; i++)
29 |         for (j = i + 1; j < matrix->width; j++)
30 |             err += matrix->data[i][j] * matrix->data[i][j];
31 |     return sqrt(err);
32 | }
33 | Matrix* rotation_method(Matrix* matrix, Matrix** lambda, double epsilon) {
34 |     int i, j, i_max, j_max, step;
35 |     double max = 0, err = epsilon + 1;
36 |     Matrix* result, * rotate, * temp, * temp2;
37 |     if (!matrix || matrix->height != matrix->width) {
38 |         fprintf(stderr, "Not square matrix\n");
39 |         return NULL;
```

```

39     }
40     *lambda = create_matrix();
41     resize_matrix(*lambda, matrix->height, matrix->width);
42     result = create_rotation_matrix((*lambda)->height, 1, 0, 0);
43     for (i = 0; i < matrix->height; i++)
44         for (j = 0; j < matrix->width; j++)
45             (*lambda)->data[i][j] = matrix->data[i][j];
46
47     for (step = 1; err >= epsilon; step++) {
48         for (i = 0; i < (*lambda)->height; i++)
49             for (j = i + 1; j < (*lambda)->width; j++)
50                 if (max < absolute((*lambda)->data[i][j])) {
51                     max = absolute((*lambda)->data[i][j]);
52                     i_max = i;
53                     j_max = j;
54                 }
55         rotate = create_rotation_matrix((*lambda)->height, i_max, j_max,
56             0.5 * atan(2 * (*lambda)->data[i_max][j_max] /
57                 ((*lambda)->data[i_max][i_max] - (*lambda)->data[j_max][j_max]]));
58         max = 0;
59         temp = multiple_matrix(result, rotate);
60         for (i = 0; i < result->height; i++)
61             for (j = 0; j < result->width; j++)
62                 result->data[i][j] = temp->data[i][j];
63         remove_matrix(temp);
64         free(temp);
65         temp = transpose_matrix(rotate);
66         temp2 = multiple_matrix(temp, *lambda);
67         remove_matrix(temp);
68         free(temp);
69         remove_matrix(*lambda);
70         free(*lambda);
71         *lambda = multiple_matrix(temp2, rotate);
72         remove_matrix(temp2);
73         free(temp2);
74         remove_matrix(rotate);
75         free(rotate);
76         err = error(*lambda);
77     }
78     for (i = 0; i < (*lambda)->height; i++)
79         (*lambda)->data[i][0] = (*lambda)->data[i][i];
80     resize_matrix(*lambda, (*lambda)->height, 1);
81     return result;
82 }
83
84 int main(void) {
85     int i;
86     double epsilon;
87     Matrix* matrix = create_matrix(), * vector, * result;

```

```

88 FILE* fmatrix, * fepsilon;
89
90 fmatrix = fopen("lab1-4m.txt", "r");
91 fepsilon = fopen("lab1-4e.txt", "r");
92 fscanf(fepsilon, "%lf", &epsilon);
93
94 if (epsilon <= 0) {
95     fprintf(stderr, "Negative value of error\n");
96     return 0;
97 }
98 if (!fmatrix || !fepsilon) {
99     fprintf(stderr, "Invalid name of file\n");
100    return 0;
101 }
102
103 scan_matrix(matrix, fmatrix);
104 fclose(fmatrix);
105 fclose(fepsilon);
106
107 if (result = rotation_method(matrix, &vector, epsilon)) {
108     printf("Vectors: \n");
109     print_matrix(result, stdout);
110     printf("Values: \n");
111     print_matrix(vector, stdout);
112     remove_matrix(result);
113     free(result);
114     remove_matrix(vector);
115     free(vector);
116 }
117 remove_matrix(matrix);
118 free(matrix);
119 return 0;
120 }

```

1.5 QR – разложение матриц

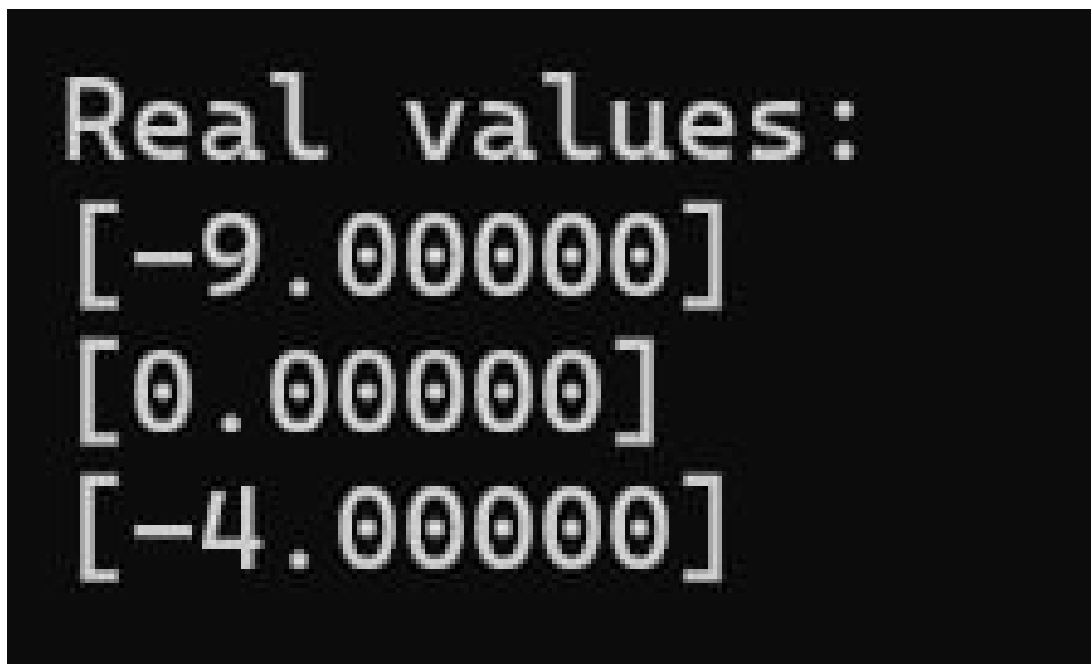
13 Постановка задачи

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: 26

$$\begin{pmatrix} -9 & -9 & -3 \\ -9 & 0 & -2 \\ -5 & -1 & -4 \end{pmatrix}$$

14 Результаты работы



```
Real values:
[-9.000000]
[0.000000]
[-4.000000]
```

Рис. 5: Вывод программы в консоли

15 Исходный код

Эпсилон:

1 || 15

Матрица:

1 | -9 -9 -3
2 | -9 0 -2
3 | -5 -1 -4

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <math.h>
4 | #include "lab1.h"
5 |
6 | static inline double absolute(const double a) {
7 |     return a > 0 ? a : -a;
8 | }
9 | static inline sign(const double a) {
10 |     return a > 0 ? 1 : (a < 0 ? -1 : 0);
11 | }
12 | Matrix** QR_decomposition(Matrix* matrix) {
13 |     Matrix** QR, * vector, * mat1, * mat2;
14 |     int i, j, k;
15 |     if (!matrix || matrix->height != matrix->width) {
16 |         return NULL;
17 |     }
18 |     QR = (Matrix**)malloc(sizeof(Matrix*) * 2);
19 |     QR[0] = create_matrix();
20 |     resize_matrix(QR[0], matrix->height, matrix->width);
21 |     QR[1] = create_matrix();
22 |     resize_matrix(QR[1], matrix->height, matrix->width);
23 |
24 |     for (i = 0; i < matrix->height; i++) {
25 |         for (j = 0; j < matrix->width; j++)
26 |             QR[1]->data[i][j] = matrix->data[i][j];
27 |         QR[0]->data[i][i] = 1;
28 |     }
29 |
30 |     for (j = 0; j < matrix->width - 1; j++) {
31 |         double s = 0;
32 |         vector = create_matrix();
33 |         resize_matrix(vector, matrix->height, 1);
34 |         for (i = 0; i < matrix->height; i++)
35 |             vector->data[i][0] = i < j ? 0 : QR[1]->data[i][j];
36 |         for (i = j; i < vector->height; i++)
37 |             s += vector->data[i][0] * vector->data[i][0];
38 |         vector->data[j][0] += sign(vector->data[j][0]) * sqrt(s);
39 |     }
```

```

40     s = 0;
41     mat1 = transpose_matrix(vector);
42     mat2 = multiple_matrix(vector, mat1);
43     remove_matrix(mat1);
44     free(mat1);
45
46     for (i = j; i < vector->height; i++)
47         s += mat2->data[i][i];
48     for (i = 0; i < vector->height; i++)
49         for (k = 0; k < vector->height; k++)
50             mat2->data[i][k] = (i == k ? 1 : 0) - 2 * mat2->data[i][k] / s;
51     mat1 = multiple_matrix(QR[0], mat2);
52
53     for (i = 0; i < QR[0]->height; i++)
54         for (k = 0; k < QR[0]->width; k++)
55             QR[0]->data[i][k] = mat1->data[i][k];
56
57     remove_matrix(mat1);
58     free(mat1);
59
60     mat1 = multiple_matrix(mat2, QR[1]);
61     for (i = 0; i < QR[1]->height; i++)
62         for (k = 0; k < QR[1]->width; k++)
63             QR[1]->data[i][k] = mat1->data[i][k];
64
65     remove_matrix(mat1);
66     free(mat1);
67     remove_matrix(mat2);
68     free(mat2);
69     remove_matrix(vector);
70     free(vector);
71 }
72 return QR;
73 }
74 int stop(Matrix* matrix, double epsilon) {
75     int mark = 0, i, j;
76
77     for (j = 0; j < matrix->width - 1; j++) {
78         double error = 0;
79         for (i = j + 1; i < matrix->height; i++)
80             error += matrix->data[i][j] * matrix->data[i][j];
81         if (sqrt(error) > epsilon)
82             if (sqrt(error - matrix->data[j + 1][j] * matrix->data[j + 1][j]) > epsilon
83                 || mark)
84                 return 0;
85             else
86                 mark = 1;
87         else
88             mark = 0;

```



```

88     }
89     return 1;
90 }
91 Matrix** QR_algorithm(Matrix* matrix, double epsilon) {
92     Matrix** QR, * alpha;
93     int step, i, j;
94     alpha = create_matrix();
95     resize_matrix(alpha, matrix->height, matrix->width);
96     for (i = 0; i < matrix->height; i++)
97         for (j = 0; j < matrix->width; j++)
98             alpha->data[i][j] = matrix->data[i][j];
99     for (step = 1; !stop(alpha, epsilon); step++) {
100         QR = QR_decomposition(alpha);
101         remove_matrix(alpha);
102         free(alpha);
103         alpha = multiple_matrix(QR[1], QR[0]);
104
105         remove_matrix(QR[0]);
106         free(QR[0]);
107         remove_matrix(QR[1]);
108         free(QR[1]);
109         free(QR);
110     }
111     QR = (Matrix**)malloc(sizeof(Matrix*) * 2);
112     QR[0] = create_matrix();
113     QR[1] = create_matrix();
114     for (j = 0; j < alpha->width - 1; j++) {
115         double s = 0;
116         for (i = j + 1; i < alpha->height; i++)
117             s += alpha->data[i][j] * alpha->data[i][j];
118         if (epsilon > sqrt(s)) {
119             resize_matrix(QR[0], QR[0]->height + 1, 1);
120             QR[0]->data[QR[0]->height - 1][0] = alpha->data[j][j];
121         } else {
122             double b = alpha->data[j][j] + alpha->data[j + 1][j + 1];
123             double c = alpha->data[j][j] * alpha->data[j + 1][j + 1] - alpha->data[j][j + 1] * alpha->data[j + 1][j];
124             resize_matrix(QR[1], QR[1]->height + 2, 2);
125             QR[1]->data[QR[1]->height - 1][0] = QR[1]->data[QR[1]->height - 2][0] = 0.5 * b;
126             QR[1]->data[QR[1]->height - 1][1] = -(QR[1]->data[QR[1]->height - 2][1] = 0.5 * sqrt(4 * c - b * b));
127             j++;
128         }
129     }
130     if (QR[0]->height + QR[1]->height != alpha->height) {
131         resize_matrix(QR[0], QR[0]->height + 1, 1);
132         QR[0]->data[QR[0]->height - 1][0] = alpha->data[alpha->height - 1][alpha->
            height - 1];

```

```

133     }
134     remove_matrix(alpha);
135     free(alpha);
136     return QR;
137 }
138
139 int main(void) {
140     int i;
141     double epsilon;
142     Matrix* matrix = create_matrix(), ** result;
143     FILE* fmatrix, * fepsilon;
144
145     fmatrix = fopen("lab1-5m.txt", "r");
146     fepsilon = fopen("lab1-5e.txt", "r");
147     fscanf(fepsilon, "%lf", &epsilon);
148     if (epsilon <= 0) {
149         fprintf(stderr, "Negative value of error\n");
150         return 0;
151     }
152     if (!fmatrix || !fepsilon) {
153         return 0;
154     }
155
156     scan_matrix(matrix, fmatrix);
157     fclose(fmatrix);
158     fclose(fepsilon);
159     if (result = QR_algorithm(matrix, epsilon)) {
160         if (result[0]->height) {
161             printf("Real values:\n");
162             print_matrix(result[0], stdout);
163         }
164         if (result[1]->height) {
165             printf("Complex values:\n");
166             print_matrix(result[1], stdout);
167         }
168         remove_matrix(result[0]);
169         free(result[0]);
170         remove_matrix(result[1]);
171         free(result[1]);
172         free(result);
173     }
174     remove_matrix(matrix);
175     free(matrix);
176     return 0;
177 }

```