

**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**

## 2.1 Методы простой итерации и Ньютона

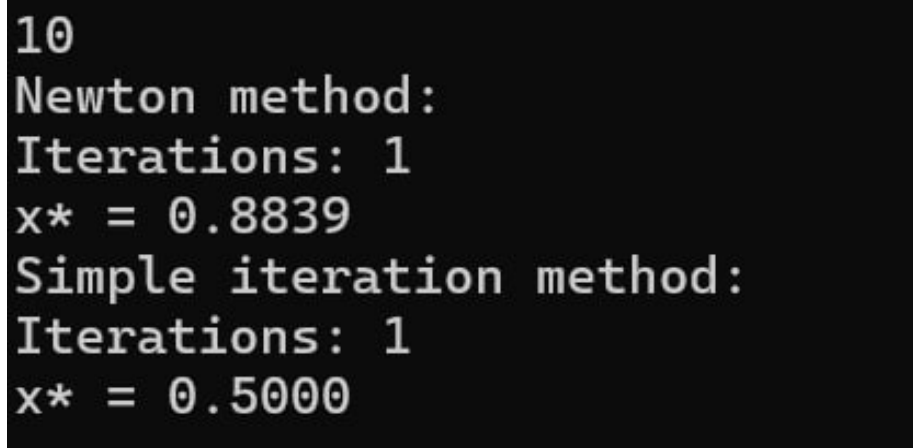
### 1 Постановка задачи

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

**Вариант: 26**

$$\lg(x + 1) - x + 0.5 = 0$$

### 2 Результаты работы



```
10
Newton method:
Iterations: 1
x* = 0.8839
Simple iteration method:
Iterations: 1
x* = 0.5000
```

Рис. 1: Вывод программы в консоли

### 3 Исходный код

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 | #include <math.h>
5 |
6 | const double a = 0;
7 |
8 | double Function(double x) {
9 |     return log10(x + 1) - x + 0.5;
10 | }
11 |
12 | double Derivative(double x) {
13 |     return (1 / (log(10) * (x + 1))) - 1;
14 | }
15 |
16 | double simple_iteration_function(double x) {
17 |     return log10(x + 1) + 0.5;
18 | }
19 |
20 | double absolute(double a) {
21 |     return a > 0 ? a : -a;
22 | }
23 |
24 | double simple_iteration_method(double (*Function)(double), double eps) {
25 |     const double constanta = 0.5;
26 |     double result, prev = a;
27 |     int iter;
28 |     for (iter = 1; eps < constanta / (1 - constanta) * absolute((result = Function(prev
29 |         )) - prev); iter++) {
30 |         prev = result;
31 |     }
32 |     printf("Iterations: %d\n", iter);
33 |     return result;
34 | }
35 | double newton_method(double (*Function)(double), double (*Derivative)(double), double
36 |     eps) {
37 |     double result, prev = a;
38 |     int iter;
39 |     for (iter = 1; eps < absolute((result = prev - Function(prev) / Derivative(prev)) -
40 |         prev); iter++) {
41 |         prev = result;
42 |     }
43 |     printf("Iterations: %d\n", iter);
44 |     return result;
45 | }
```

```

45 | int main(void) {
46 |     float eps;
47 |
48 |     scanf("%f", &eps);
49 |
50 |     if (eps <= 0) {
51 |         fprintf(stderr, "Error\n");
52 |         return 0;
53 |     }
54 |     printf("Newton method:\n");
55 |     printf("x* = %.4f\n", newton_method(Function, Derivative, eps));
56 |     printf("Simple iteration method:\n");
57 |     printf("x* = %.4f\n", simple_iteration_method(simple_iteration_function, eps));
58 |
59 |     return 0;
60 | }

```

## 2.2 Методы простой итерации и Ньютона

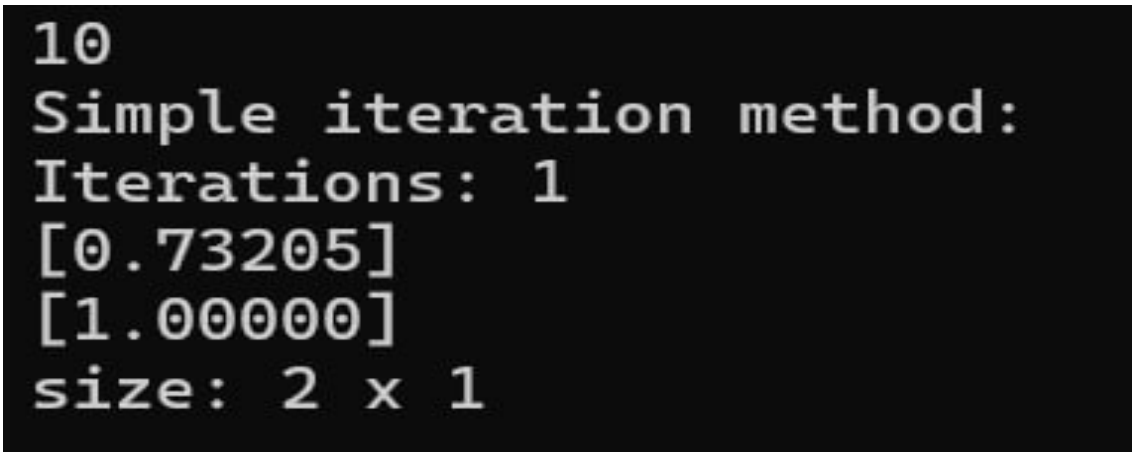
### 4 Постановка задачи

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

**Вариант: 26**

$$\begin{cases} 2x_1^2 - x_2 + x_2^2 - 2 = 0 \\ x_1 - \sqrt{x_2 + 2} + 1 = 0 \end{cases}$$

### 5 Результаты работы



```
10
Simple iteration method:
Iterations: 1
[0.73205]
[1.00000]
size: 2 x 1
```

Рис. 2: Вывод программы в консоли

## 6 Исходный код

```
1 | #ifndef _LAB2_
2 | #define _LAB2_
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 | Matrix* multiple_matrix(Matrix*, Matrix*);
17 | Matrix* subtraction(Matrix*, Matrix*);
18 | double matrix_norm(Matrix*);
19 | Matrix* gauss_method(Matrix*, Matrix*);
20 |
21 | #endif
22 |
23 | #include "lab2.h"
24 | #include <stdlib.h>
25 |
26 |
27 | static inline double absolute(const double a) {
28 |     return a > 0 ? a : -a;
29 | }
30 |
31 | Matrix* create_matrix(void) {
32 |     Matrix* new_matrix = (Matrix*)malloc(sizeof(Matrix));
33 |     new_matrix->width = new_matrix->height = 0;
34 |     new_matrix->data = NULL;
35 |     return new_matrix;
36 | }
37 |
38 | void remove_matrix(Matrix* matrix) {
39 |     int i;
40 |     if (!matrix)
41 |         return;
42 |     for (i = 0; i < matrix->height; i++)
43 |         free(matrix->data[i]);
44 |     free(matrix->data);
45 |     matrix->data = NULL;
46 |     matrix->height = matrix->width = 0;
47 | }
48 |
49 | void resize_matrix(Matrix* matrix, const int height, const int width) {
50 |     int i, j;
51 |     if (height > 0) {
```

```

27     matrix->data = (double**)realloc(matrix->data, sizeof(double*) * height);
28     for (i = matrix->height; i < height; i++) {
29         matrix->data[i] = (double*)malloc(sizeof(double) * (width <= 0 ? matrix->
30             width : width));
31         for (j = 0; j < width; j++)
32             matrix->data[i][j] = 0;
33     }
34     if (width > 0)
35         for (i = 0; i < matrix->height; i++) {
36             matrix->data[i] = (double*)realloc(matrix->data[i], sizeof(double) * width)
37                 ;
38             for (j = matrix->width; j < width; j++)
39                 matrix->data[i][j] = 0;
40         }
41     if (width > 0)
42         matrix->width = width;
43     if (height > 0)
44         matrix->height = height;
45 }
46 void print_matrix(Matrix* matrix, FILE* stream) {
47     int i, j;
48     if (!matrix->data)
49         return;
50     for (i = 0; i < matrix->height; i++) {
51         fputc('[', stream);
52         for (j = 0; j < matrix->width; j++)
53             fprintf(stream, "%.5Lf ", matrix->data[i][j]);
54         fprintf(stream, "\b\b\n");
55     }
56     fprintf(stream, "size: %d x %d\n", matrix->height, matrix->width);
57 }
58 Matrix* multiple_matrix(Matrix* A, Matrix* B) {
59     Matrix* result;
60     int i, j, k;
61     if (A->width != B->height)
62         return NULL;
63     result = create_matrix();
64     resize_matrix(result, A->height, B->width);
65     for (i = 0; i < result->height; i++)
66         for (j = 0; j < result->width; j++)
67             for (k = 0; k < A->width; k++)
68                 result->data[i][j] += A->data[i][k] * B->data[k][j];
69     return result;
70 }
71 Matrix* subtraction(Matrix* A, Matrix* B) {
72     Matrix* result;
73     int i, j;
74     if (A->height != B->height || A->width != B->width)

```

```

74     return NULL;
75     result = create_matrix();
76     resize_matrix(result, A->height, A->width);
77     for (i = 0; i < result->height; i++)
78         for (j = 0; j < result->width; j++)
79             result->data[i][j] = A->data[i][j] - B->data[i][j];
80     return result;
81 }
82 void exchange_str(Matrix* matrix, int i1, int i2) {
83     double temp;
84     int j;
85     for (j = 0; j < matrix->width; j++) {
86         temp = matrix->data[i1][j];
87         matrix->data[i1][j] = matrix->data[i2][j];
88         matrix->data[i2][j] = temp;
89     }
90 }
91 void exchange_col(Matrix* matrix, int j1, int j2) {
92     double temp;
93     int i;
94     for (i = 0; i < matrix->height; i++) {
95         temp = matrix->data[i][j1];
96         matrix->data[i][j1] = matrix->data[i][j2];
97         matrix->data[i][j2] = temp;
98     }
99 }
100 double matrix_norm(Matrix* matrix) {
101     double sum = 0, norm = 0;
102     int i, j;
103     if (!matrix)
104         return 0;
105     for (i = 0; i < matrix->height; i++) {
106         for (j = 0; j < matrix->width; j++)
107             sum += absolute(matrix->data[i][j]);
108         norm = sum > norm ? sum : norm;
109         sum = 0;
110     }
111     return norm;
112 }
113 Matrix** LU_decomposition(Matrix* matrix) {
114     int i, j, k, size = matrix->height, max;
115     Matrix** LUP = (Matrix**)malloc(sizeof(Matrix*) * 3);
116     if (matrix->height != matrix->width)
117         return NULL;
118     for (i = 0; i < 3; i++) {
119         LUP[i] = create_matrix();
120         resize_matrix(LUP[i], size, size);
121     }
122     for (i = 0; i < size; i++)

```



```

123     LUP[2]->data[i][i] = LUP[0]->data[i][i] = 1;
124     for (i = 0; i < size; i++)
125         for (j = 0; j < size; j++)
126             LUP[1]->data[i][j] = matrix->data[i][j];
127     for (j = 0; j < size; j++) {
128         max = j;
129         for (i = j + 1; i < size; i++)
130             if (absolute(LUP[1]->data[i][j]) > absolute(LUP[1]->data[max][j]))
131                 max = i;
132         if (!LUP[1]->data[max][j])
133             return NULL;
134         exchange_str(LUP[1], j, max);
135         exchange_str(LUP[2], j, max);
136         exchange_str(LUP[0], j, max);
137         exchange_col(LUP[0], j, max);
138         for (i = j + 1; i < size; i++) {
139             LUP[0]->data[i][j] = LUP[1]->data[i][j] / LUP[1]->data[j][j];
140             for (k = j; k < size; k++)
141                 LUP[1]->data[i][k] -= LUP[0]->data[i][j] * LUP[1]->data[j][k];
142         }
143     }
144     return LUP;
145 }
146 Matrix* LU_solve(Matrix** LUP, Matrix* vector) {
147     Matrix* result;
148     int i, j;
149     if (!LUP)
150         return NULL;
151     result = multiple_matrix(LUP[2], vector);
152     for (i = 0; i < result->height; i++)
153         for (j = 0; j < i; j++)
154             result->data[i][0] -= result->data[j][0] * LUP[0]->data[i][j];
155     for (i = result->height - 1; i >= 0; i--) {
156         for (j = result->height - 1; j > i; j--)
157             result->data[i][0] -= result->data[j][0] * LUP[1]->data[i][j];
158         result->data[i][0] /= LUP[1]->data[i][i];
159     }
160     return result;
161 }
162 Matrix* gauss_method(Matrix* matrix, Matrix* vector) {
163     Matrix** LUP, * result;
164     int i;
165     if (matrix->height != vector->height || vector->width != 1) {
166         return NULL;
167     }
168     LUP = LU_decomposition(matrix);
169     result = LU_solve(LUP, vector);
170     for (i = 0; i < 3; i++) {
171         remove_matrix(LUP[i]);

```

```

172     free(LUP[i]);
173 }
174 free(LUP);
175 return result;
176 }

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <string.h>
5 #include "lab2.h"
6
7 int n = 0;
8 Matrix* values;
9 double constanta = 0.5;
10
11 void values_init(void) {
12     values = create_matrix();
13     resize_matrix(values, 2, 1);
14     values->data[0][0] = values->data[1][0] = 1;
15 }
16
17 Matrix* Function(Matrix* vector) {
18     Matrix* result;
19     if (vector->height != 2 || vector->width != 1) {
20         fprintf(stderr, "Invalid size of vector\n");
21         return NULL;
22     }
23     result = create_matrix();
24     resize_matrix(result, 2, 1);
25     result->data[0][0] = 2 * pow(vector->data[0][0], 2) - vector->data[1][0] + pow(
        vector->data[1][0], 2) - 2;
26     result->data[1][0] = vector->data[0][0] - pow(vector->data[1][0] + 2, 0.5) + 1;
27     return result;
28 }
29
30 Matrix* Jacobi_matrix(Matrix* vector) {
31     Matrix* result;
32     if (vector->height != 2 || vector->width != 1) {
33         fprintf(stderr, "Error\n");
34         return NULL;
35     }
36     result = create_matrix();
37     resize_matrix(result, 2, 2);
38     result->data[0][0] = 4 * vector->data[0][0];
39     result->data[0][1] = 2 * vector->data[1][0] - 1;
40     result->data[1][0] = 1;
41     result->data[1][1] = -0.5 / sqrt(vector->data[1][0] + 2);
42     return result;
43 }

```

```

44
45 Matrix* simple_iteration_function(Matrix* vector) {
46     Matrix* result;
47     if (vector->height != 2 || vector->width != 1) {
48         fprintf(stderr, "Error\n");
49         return NULL;
50     }
51     result = create_matrix();
52     resize_matrix(result, 2, 1);
53     result->data[0][0] = pow(vector->data[1][0] + 2, 0.5) - 1;
54     result->data[1][0] = pow(vector->data[1][0] + 2 - 2 * pow(vector->data[0][0], 2),
55         0.5);
56     return result;
57 }
58 Matrix* simple_iteration_method(Matrix* (*Function)(Matrix*), double eps) {
59     Matrix* current, * prev, * error_vector;
60     int iter;
61     double err;
62     if (!values || values->width != 1)
63         return NULL;
64     prev = create_matrix();
65     resize_matrix(prev, values->height, 1);
66     for (iter = 0; iter < values->height; iter++)
67         prev->data[iter][0] = values->data[iter][0];
68     current = Function(prev);
69     if (!current)
70         return NULL;
71     error_vector = subtraction(current, prev);
72     for (iter = 1; constanta / (1 - constanta) * (err = matrix_norm(error_vector)) >
73         eps; iter++) {
74         remove_matrix(prev);
75         free(prev);
76         prev = current;
77         current = Function(prev);
78         remove_matrix(error_vector);
79         free(error_vector);
80         error_vector = subtraction(current, prev);
81     }
82     printf("Iterations: %d\n", iter);
83     remove_matrix(error_vector);
84     free(error_vector);
85     remove_matrix(prev);
86     free(prev);
87     return current;
88 }
89 Matrix* newton_method(Matrix* (*Function)(Matrix*), Matrix* (*Jacobi_matrix)(Matrix*),
90     double eps) {

```

```

90 Matrix* current, * prev, * jacobi_matrix, * error_vector, * temp;
91 int iter;
92 double err;
93 if (!values || values->width != 1)
94     return NULL;
95 prev = create_matrix();
96 resize_matrix(prev, values->height, 1);
97
98 for (iter = 0; iter < values->height; iter++)
99     prev->data[iter][0] = values->data[iter][0];
100
101 jacobi_matrix = Jacobi_matrix(prev);
102 current = multiple_matrix(jacobi_matrix, prev);
103 temp = Function(prev);
104 error_vector = subtraction(current, temp);
105
106 remove_matrix(current);
107 remove_matrix(temp);
108 free(current);
109 free(temp);
110
111 current = gauss_method(jacobi_matrix, error_vector);
112
113 remove_matrix(error_vector);
114 free(error_vector);
115 remove_matrix(jacobi_matrix);
116 free(jacobi_matrix);
117
118 if (!current)
119     return NULL;
120 error_vector = subtraction(current, prev);
121
122 for (iter = 1; (err = matrix_norm(error_vector)) > eps; iter++) {
123     remove_matrix(prev);
124     free(prev);
125     prev = current;
126     remove_matrix(error_vector);
127     free(error_vector);
128     jacobi_matrix = Jacobi_matrix(prev);
129     current = multiple_matrix(jacobi_matrix, prev);
130     temp = Function(prev);
131     error_vector = subtraction(current, temp);
132     remove_matrix(current);
133     remove_matrix(temp);
134     free(current);
135     free(temp);
136     current = gauss_method(jacobi_matrix, error_vector);
137     remove_matrix(error_vector);
138     free(error_vector);

```

```

139     error_vector = subtraction(current, prev);
140     remove_matrix(jacobi_matrix);
141     free(jacobi_matrix);
142 }
143 printf("Iterations: %d\n", iter);
144
145     remove_matrix(error_vector);
146     free(error_vector);
147     remove_matrix(prev);
148     free(prev);
149     return current;
150 }
151
152 int main(void) {
153     int i;
154     float eps;
155     scanf("%f", &eps);
156     Matrix* result;
157     values_init();
158     if (eps <= 0) {
159         fprintf(stderr, "Error\n");
160         return 0;
161     }
162     if (n) {
163         printf("Newton method:\n");
164         result = newton_method(Function, Jacobi_matrix, eps);
165         print_matrix(result, stdout);
166         remove_matrix(result);
167         free(result);
168     }
169     else {
170         printf("Simple iteration method:\n");
171         result = simple_iteration_method(simple_iteration_function, eps);
172         print_matrix(result, stdout);
173         remove_matrix(result);
174         free(result);
175     }
176
177     return 0;
178 }

```