

**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: Голошумов М.С.
Преподаватель: Пивоваров Д.Е.
Группа: М8О-303Б-21
Дата:
Оценка:
Подпись:

Москва, 2024

2.1 Методы простой итерации и Ньютона

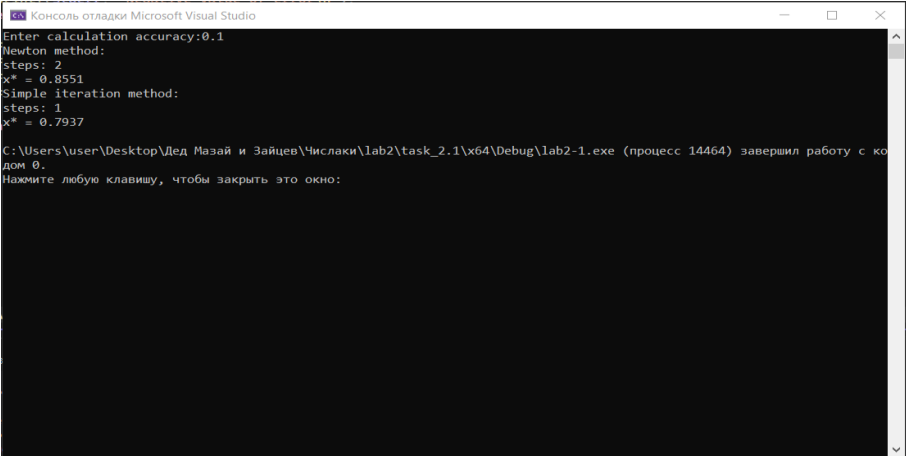
1 Постановка задачи

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: 4

$$x^3 + x^2 - x - 0.5 = 0$$

2 Результаты работы



```
Консоль отладки Microsoft Visual Studio
Enter calculation accuracy:0.1
Newton method:
steps: 2
x* = 0.8551
Simple iteration method:
steps: 1
x* = 0.7937
C:\Users\user\Desktop\Дед Мазай и Зайцев\Числаки\lab2\task_2.1\x64\Debug\lab2-1.exe (процесс 14464) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

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 BEGIN_VALUE = 1.0;
7 |
8 | double Function(double x) {
9 |     return (x * x * x) + x * x - x - 0.5;
10 | }
11 |
12 | double Derivative(double x) {
13 |     return 3 * x * x + 2 * x - 1;
14 | }
15 |
16 | double simple_iteration_function(double x) {
17 |     return pow(0.5 + x - x * x, 1.0 / 3.0);
18 | }
19 |
20 | double absolute(double a) {
21 |     return a > 0 ? a : -a;
22 | }
23 |
24 | double simple_iteration_method(double (*Function)(double), double epsilon) {
25 |     const double SIMPLE_ITERATION_CONSTANT = 0.01;
26 |     double result, previous = BEGIN_VALUE;
27 |     int step;
28 |     for (step = 1; epsilon < SIMPLE_ITERATION_CONSTANT / (1 - SIMPLE_ITERATION_CONSTANT
29 |         ) * absolute((result = Function(previous)) - previous); step++) {
30 |         previous = result;
31 |     }
32 |     printf("steps: %d\n", step);
33 |     return result;
34 | }
35 | double newton_method(double (*Function)(double), double (*Derivative)(double), double
36 |     epsilon) {
37 |     double result, previous = BEGIN_VALUE;
38 |     int step;
39 |     for (step = 1; epsilon < absolute((result = previous - Function(previous) /
40 |         Derivative(previous)) - previous); step++) {
41 |         previous = result;
42 |     }
43 |     printf("steps: %d\n", step);
44 |     return result;
45 | }
```

```

45 | int main(void) {
46 |     float epsilon;
47 |
48 |     printf("Enter calculation accuracy:");
49 |     scanf("%f", &epsilon);
50 |
51 |     if (epsilon <= 0) {
52 |         fprintf(stderr, "Negative value of error\n");
53 |         return 0;
54 |     }
55 |     printf("Newton method:\n");
56 |     printf("x* = %.4f\n", newton_method(Function, Derivative, epsilon));
57 |     printf("Simple iteration method:\n");
58 |     printf("x* = %.4f\n", simple_iteration_method(simple_iteration_function, epsilon));
59 |
60 |     return 0;
61 | }

```

2.2 Методы простой итерации и Ньютона

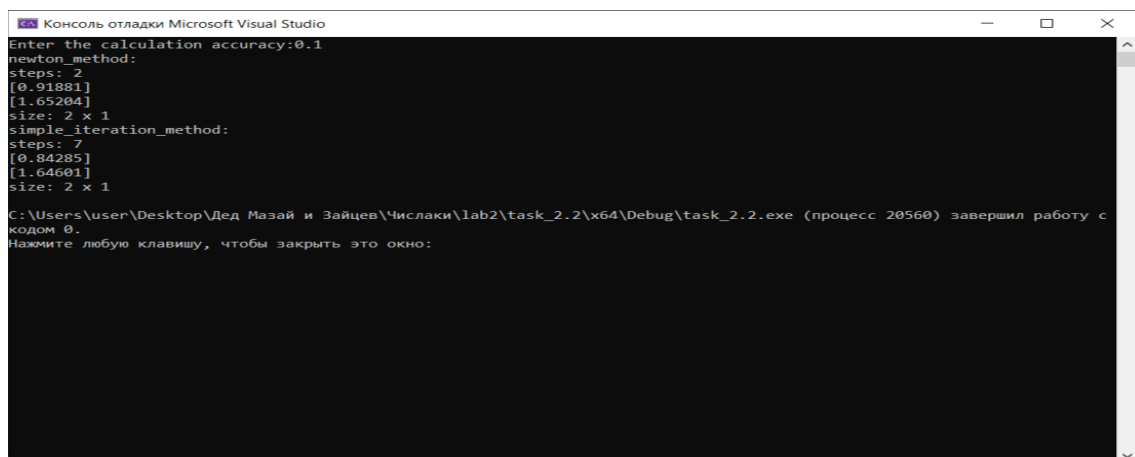
4 Постановка задачи

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: 4

$$\begin{cases} x_1 - \cos(x_2) = 1 \\ x_2 - \lg(x_1 + 1) = 1 \end{cases}$$

5 Результаты работы



```
Консоль отладки Microsoft Visual Studio
Enter the calculation accuracy:0.1
newton_method:
steps: 2
[0.91881]
[1.65284]
size: 2 x 1
simple_iteration_method:
steps: 7
[0.84285]
[1.64601]
size: 2 x 1
C:\Users\user\Desktop\Дед Маай и Зайцев\Ислаки\lab2\task_2.2\x64\Debug\task_2.2.exe (процесс 20560) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

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* multiplication(Matrix*, Matrix*);
17 Matrix* subtraction(Matrix*, Matrix*);
18 double matrix_norm(Matrix*);
19 Matrix* gauss_method(Matrix*, Matrix*);
20
21 #endif
22 #pragma once
23
24 #include "read.h"
25 #include <stdlib.h>
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;
```

```

26     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     }
35     if (width > 0)
36         for (i = 0; i < matrix->height; i++) {
37             matrix->data[i] = (double*)realloc(matrix->data[i], sizeof(double) * width)
38                 ;
39             for (j = matrix->width; j < width; j++)
40                 matrix->data[i][j] = 0;
41         }
42     if (width > 0)
43         matrix->width = width;
44     if (height > 0)
45         matrix->height = height;
46 }
47 void print_matrix(Matrix* matrix, FILE* stream) {
48     int i, j;
49     if (!matrix->data)
50         return;
51     for (i = 0; i < matrix->height; i++) {
52         fputc('[', stream);
53         for (j = 0; j < matrix->width; j++)
54             fprintf(stream, "%.5Lf ", matrix->data[i][j]);
55         fprintf(stream, "\b\b\n");
56     }
57     fprintf(stream, "size: %d x %d\n", matrix->height, matrix->width);
58 }
59 Matrix* multiplication(Matrix* A, Matrix* B) {
60     Matrix* result;
61     int i, j, k;
62     if (A->width != B->height)
63         return NULL;
64     result = create_matrix();
65     resize_matrix(result, A->height, B->width);
66     for (i = 0; i < result->height; i++)
67         for (j = 0; j < result->width; j++)
68             for (k = 0; k < A->width; k++)
69                 result->data[i][j] += A->data[i][k] * B->data[k][j];
70     return result;
71 }
72 Matrix* subtraction(Matrix* A, Matrix* B) {
73     Matrix* result;
74     int i, j;

```

```

73     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 matrix_exchange_strings(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 matrix_exchange_columns(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
115     int i, j, k, size = matrix->height, max;
116     Matrix** LUP = (Matrix**)malloc(sizeof(Matrix*) * 3);
117     if (matrix->height != matrix->width)
118         return NULL;
119
120     for (i = 0; i < 3; i++) {
121         LUP[i] = create_matrix();

```



```

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

```

```

171     if (matrix->height != vector->height || vector->width != 1) {
172         return NULL;
173     }
174     LUP = LU_decomposition(matrix);
175     result = LU_solve(LUP, vector);
176     for (i = 0; i < 3; i++) {
177         remove_matrix(LUP[i]);
178         free(LUP[i]);
179     }
180     free(LUP);
181     return result;
182 }

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <string.h>
5  #include "read.h"
6
7  int NEWTON = 1;
8
9  Matrix* BEGIN_VALUES;
10 double SIMPLE_ITERATION_CONSTANT = 0.5;
11
12 void begin_values_initialization(void) {
13     BEGIN_VALUES = create_matrix();
14     resize_matrix(BEGIN_VALUES, 2, 1);
15     BEGIN_VALUES->data[0][0] = BEGIN_VALUES->data[1][0] = 1;
16 }
17
18 void begin_values_removing(void) {
19     remove_matrix(BEGIN_VALUES);
20     free(BEGIN_VALUES);
21 }
22
23 Matrix* Function(Matrix* vector) {
24     Matrix* result;
25     if (vector->height != 2 || vector->width != 1) {
26         fprintf(stderr, "Invalid size of vector\n");
27         return NULL;
28     }
29     result = create_matrix();
30     resize_matrix(result, 2, 1);
31     result->data[0][0] = vector->data[0][0] - cos(vector->data[1][0]) - 1;
32     result->data[1][0] = vector->data[1][0] - log1(vector->data[0][0] + 1) - 1;
33     return result;
34 }
35
36 Matrix* Jacobi_matrix(Matrix* vector) {
37     Matrix* result;

```

```

38     if (vector->height != 2 || vector->width != 1) {
39         fprintf(stderr, "Invalid size of vector\n");
40         return NULL;
41     }
42     result = create_matrix();
43     resize_matrix(result, 2, 2);
44     result->data[0][0] = 1; result->data[0][1] = sin(vector->data[1][0]);
45     result->data[1][0] = -1 / (vector->data[0][0] + 1); result->data[1][1] = 1;
46     return result;
47 }
48
49 Matrix* simple_iteration_function(Matrix* vector) {
50
51     Matrix* result;
52     if (vector->height != 2 || vector->width != 1) {
53         fprintf(stderr, "Invalid size of vector\n");
54         return NULL;
55     }
56     result = create_matrix();
57     resize_matrix(result, 2, 1);
58     result->data[0][0] = cos(vector->data[1][0]) + 1;
59     result->data[1][0] = log1(vector->data[0][0] + 1.0) + 1;
60     return result;
61 }
62
63 Matrix* simple_iteration_method(Matrix* (*Function)(Matrix*), double epsilon) {
64     Matrix* current, * previous, * error_vector;
65     int step;
66     double err;
67     if (!BEGIN_VALUES || BEGIN_VALUES->width != 1)
68         return NULL;
69     previous = create_matrix();
70     resize_matrix(previous, BEGIN_VALUES->height, 1);
71     for (step = 0; step < BEGIN_VALUES->height; step++)
72         previous->data[step][0] = BEGIN_VALUES->data[step][0];
73     current = Function(previous);
74     if (!current)
75         return NULL;
76     error_vector = subtraction(current, previous);
77     for (step = 1; SIMPLE_ITERATION_CONSTANT / (1 - SIMPLE_ITERATION_CONSTANT) *
78         (err = matrix_norm(error_vector)) > epsilon; step++) {
79
80         remove_matrix(previous);
81         free(previous);
82         previous = current;
83         current = Function(previous);
84         remove_matrix(error_vector);
85         free(error_vector);
86         error_vector = subtraction(current, previous);

```

```

87     }
88     printf("steps: %d\n", step);
89
90     remove_matrix(error_vector);
91     free(error_vector);
92     remove_matrix(previous);
93     free(previous);
94     return current;
95 }
96
97 Matrix* newton_method(Matrix* (*Function)(Matrix*), Matrix* (*Jacobi_matrix)(Matrix*),
98     double epsilon) {
99     Matrix* current, * previous, * jacobi_matrix, * error_vector, * temp;
100     int step;
101     double err;
102     if (!BEGIN_VALUES || BEGIN_VALUES->width != 1)
103         return NULL;
104     previous = create_matrix();
105     resize_matrix(previous, BEGIN_VALUES->height, 1);
106
107     for (step = 0; step < BEGIN_VALUES->height; step++)
108         previous->data[step][0] = BEGIN_VALUES->data[step][0];
109
110     jacobi_matrix = Jacobi_matrix(previous);
111     current = multiplication(jacobi_matrix, previous);
112     temp = Function(previous);
113     error_vector = subtraction(current, temp);
114
115     remove_matrix(current);
116     remove_matrix(temp);
117     free(current);
118     free(temp);
119
120     current = gauss_method(jacobi_matrix, error_vector);
121
122     remove_matrix(error_vector);
123     free(error_vector);
124     remove_matrix(jacobi_matrix);
125     free(jacobi_matrix);
126
127     if (!current)
128         return NULL;
129     error_vector = subtraction(current, previous);
130
131     for (step = 1; (err = matrix_norm(error_vector)) > epsilon; step++) {
132         remove_matrix(previous);
133         free(previous);
134         previous = current;

```

```

135     remove_matrix(error_vector);
136     free(error_vector);
137     jacobi_matrix = Jacobi_matrix(previous);
138     current = multiplication(jacobi_matrix, previous);
139     temp = Function(previous);
140     error_vector = subtraction(current, temp);
141     remove_matrix(current);
142     remove_matrix(temp);
143     free(current);
144     free(temp);
145     current = gauss_method(jacobi_matrix, error_vector);
146     remove_matrix(error_vector);
147     free(error_vector);
148     error_vector = subtraction(current, previous);
149     remove_matrix(jacobi_matrix);
150     free(jacobi_matrix);
151 }
152 printf("steps: %d\n", step);
153
154     remove_matrix(error_vector);
155     free(error_vector);
156     remove_matrix(previous);
157     free(previous);
158     return current;
159 }
160
161 int main(void) {
162     int i;
163     float epsilon;
164     printf("Enter the calculation accuracy:");
165     scanf_s("%f", &epsilon);
166
167     Matrix* result;
168     begin_values_initialization();
169
170     if (epsilon <= 0) {
171         fprintf(stderr, "Negative value of error\n");
172         return 0;
173     }
174     printf("newton_method:\n");
175     result = newton_method(Function, Jacobi_matrix, epsilon);
176     print_matrix(result, stdout);
177     remove_matrix(result);
178     free(result);
179
180
181     printf("simple_iteration_method:\n");
182     result = simple_iteration_method(simple_iteration_function, epsilon);
183     print_matrix(result, stdout);

```

```
184 || remove_matrix(result);  
185 || free(result);  
186 ||  
187 || begin_values_removing();  
188 ||  
189 || return 0;  
190 || }
```