

**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: М. Д. Жилин  
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: 9**

$$x^3 + x^2 - 2x - 1 = 0$$

### 2 Результаты работы

```
Метод Ньютона
Результат (x_k): 1.2469796037174672
Количество итераций (k): 6
Значение функции при данном корне x_k: 8.881784197001252e-16

Метод простых итераций
Результат (x_k): -0.4450418678900141
Количество итераций (k): 12
Значение функции при данном корне x_k: -5.192102303652746e-11
```

Рис. 1: Вывод программы в консоли

### 3 Исходный код

```
1 from typing import Callable
2
3
4 def func(x: float) -> float:
5     return x ** 3 + x ** 2 - 2 * x - 1
6
7
8 def newton_method(x_0: float, EPS: float, f: Callable) -> tuple[float, int]:
9     """
10     f(x): x^3 + x^2 - 2x - 1
11     : 3x^2 + 2x - 2
12     : 6x + 2
13
14     :
15     f(x_0)*f''(x_0) > 0
16
17     :
18     x_0 = 2 ( )
19     """
20     def d_func(x: float) -> float:
21         return 3 * (x ** 2) + 2 * x - 2
22
23     def eps(val1: float, val2: float) -> float:
24         return abs(val1 - val2)
25
26     if func(x_0) == 0:
27         return x_0, 0
28     x_next, x_curr, k = x_0 - f(x_0)/d_func(x_0), x_0, 1
29     while eps(x_curr, x_next) >= EPS:
30         k += 1
31         x_next, x_curr = x_next - f(x_next)/d_func(x_next), x_next
32     return x_next, k
33
34
35 def simple_iterations_method(x_0: float, a: float, b: float, q: float, EPS: float, f:
36     Callable) -> tuple[float, int]:
37     """
38     f(x): x^3 + x^2 - 2x - 1
39     (x = phi(x)): x = (x^3 + x^2 - 1)/2
40     phi(x) = (x^3 + x^2 - 1)/2
41     phi'(x) = 1.5*x^2 + x
42
43     :
44     1) phi(x)[a, b] x[a, b]
45     2) q: |phi'(x)| <= q < 1 x(a, b)
46
47     :
48     """
```

```

47     q = 0.8
48     [a, b] = [-1/3, (-1 + (1 + 1.5*4*q)**0.5)/3]
49     x_0 = -0.25
50     """
51     def phi(x: float) -> float:
52         return (x ** 3 + x ** 2 - 1)/2
53
54     def eps(val1: float, val2: float) -> float:
55         return q*abs(val1 - val2)/(1-q)
56
57     if func(x_0) == 0:
58         return x_0, 0
59
60     x_next, x_curr, k = x_0 + 2 * EPS, x_0, 0
61     while eps(x_curr, x_next) >= EPS:
62         k += 1
63         x_next, x_curr = phi(x_next), x_next
64     return x_next, k
65
66
67 if __name__ == "__main__":
68     deviation = 1e-9
69     q = 0.8
70
71     result, number_of_iterations = newton_method(x_0=2, EPS=deviation, f=func)
72     print('\n ')
73     print(f' (x_k): {result}\n (k): {number_of_iterations}')
74     print(f'      x_k: {func(result)}')
75     print()
76
77     parameters = {
78         "x_0": -0.25,
79         "a": -1 / 3,
80         "b": (-1 + (1 + 1.5*4*q)**0.5)/3,
81         "q": q,
82         "EPS": deviation,
83         "f": func
84     }
85     result, number_of_iterations = simple_iterations_method(**parameters)
86     print('\n ')
87     print(f' (x_k): {result}\n (k): {number_of_iterations}')
88     print(f'      x_k: {func(result)}')
89     print()

```

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5
6  double func(double x){

```

```

7 |     return pow(x, 3) + pow(x, 2) - 2 * x - 1;
8 | }
9 |
10 |
11 | pair<double, int> newton_method(double x_0, double EPS){
12 |     /*
13 |          $f(x): x^3 + x^2 - 2x - 1$ 
14 |         :  $3x^2 + 2x - 2$ 
15 |         :  $6x + 2$ 
16 |
17 |         :
18 |          $f(x_0)*f''(x_0) > 0$ 
19 |
20 |         :
21 |          $x_0 = 2$  ( )
22 |     */
23 |
24 |     // double dfunc(double x){
25 |     // return 3 * pow(x, 2) + 2 * x - 2;
26 |     // }
27 |
28 |     // double eps(double val1, double val2){
29 |     // return abs(val1 - val2);
30 |     // }
31 |
32 |     auto d_func = [](double x) {
33 |         return 3 * pow(x, 2) + 2 * x - 2;
34 |     };
35 |
36 |     auto eps = [](double val1, double val2) {
37 |         return abs(val1 - val2);
38 |     };
39 |
40 |     if (func(x_0) == 0)
41 |         return make_pair(x_0, 0);
42 |
43 |     int k = 1;
44 |     double x_next = x_0 - func(x_0)/d_func(x_0), x_curr = x_0;
45 |
46 |     while (eps(x_curr, x_next) >= EPS){
47 |         k += 1;
48 |         x_curr = x_next;
49 |         x_next = x_next - func(x_next)/d_func(x_next);
50 |     }
51 |     return make_pair(x_next, k);
52 | }
53 |
54 |
55 |

```

```

56 pair<double, int> simple_iterations_method(double x_0, double a, double b, double q,
    double EPS){
57
58     /*
59      $f(x): x^3 + x^2 - 2x - 1$ 
60      $(x = \phi(x)): x = (x^3 + x^2 - 1)/2$ 
61      $\phi(x) = (x^3 + x^2 - 1)/2$ 
62      $\phi'(x) = 1.5*x^2 + x$ 
63
64     :
65     1)  $\phi(x)[a, b]$   $x[a, b]$ 
66     2)  $q: |\phi'(x)| \leq q < 1$   $x(a, b)$ 
67
68     :
69      $q = 0.8$ 
70      $[a, b] = [-1/3, (-1 + (1 + 1.5*4*q)**0.5)/3]$ 
71      $x_0 = -0.25$ 
72     */
73
74     auto phi = [](double x) {
75         return (pow(x, 3) + pow(x, 2) - 1)/2;
76     };
77
78     auto eps = [](double val1, double val2, double q) {
79         return q*abs(val1 - val2)/(1-q);
80     };
81
82     if (func(x_0) == 0)
83         return make_pair(x_0, 0);
84
85     int k = 0;
86     double x_next = x_0 + 2 * EPS, x_curr = x_0;
87     while (eps(x_curr, x_next, q) >= EPS){
88         k += 1;
89         x_curr = x_next;
90         x_next = phi(x_next);
91     }
92     return make_pair(x_next, k);
93 }
94
95
96 int main(){
97     double deviation = 1e-9, q = 0.8, result;
98     int number_of_iterations;
99
100     tie(result, number_of_iterations) = newton_method(2, deviation);
101     cout << "\nNewton's method\n";
102     cout << "Result (x_k): " << result << endl << "Number of iterations (k): " <<
        number_of_iterations << endl;

```

```

103 | cout << "The value of the function at the given root x_k: " << func(result) << endl
    | << endl;
104 |
105 | tie(result, number_of_iterations) = simple_iterations_method(-0.25, -1/3, (-1 +
    | sqrt(1 + 1.5*4*q))/3, q, deviation);
106 | cout << "\nThe method of simple iterations\n";
107 | cout << "Result (x_k): " << result << endl << "Number of iterations (k): " <<
    | number_of_iterations << endl;
108 | cout << "The value of the function at the given root x_k: " << func(result) << endl
    | << endl;
109 |
110 | return 1;
111 | }

```

## 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: 9**

$$\begin{cases} x_1^2 + x_2^2 - 16 = 0 \\ x_1 - e^{x_2} + 4 = 0 \end{cases}$$

### 5 Результаты работы

```
Метод Ньютона
Результат (x_k): [-3.7508423759566107, -1.389669554508614]
Количество итераций (k): 5
Значение функций при данном корне x_k: (0.0, 0.0)

Метод простых итераций
Результат (x_k): [-3.750842375910453, -1.3896695543408484]
Количество итераций (k): 111
Значение функций при данном корне x_k: (-8.125358164079444e-10, 4.357403327048814e-12)
```

Рис. 2: Вывод программы в консоли



## 6 Исходный код

```
1 from typing import Callable
2 import math
3
4 f1 = lambda x: x[0]**2 + x[1]**2 - 16
5 f2 = lambda x: x[0] - math.e**x[1] + 4
6
7
8 def newton_method(x_0: list[float], EPS: float) -> tuple[list[float], int]:
9     """
10     f1(x1, x2): x1^2 + x2^2 - 16
11     f2(x1, x2): x1 - e^(x2) + 4
12
13     f1 x1: 2*x1
14     f1 x2: 2*x2
15
16     f2 x1: 1
17     f2 x2: -e^(x2)
18
19     ( ):
20     x1 = -3.5
21     x2 = -2
22     """
23
24     def eps(vect1: list[float], vect2: list[float]) -> float:
25         return max((abs(vect1[i]-vect2[i]) for i in range(len(vect1))))
26
27     def det(x: list[float], matrix: list[list[Callable]]) -> float:
28         return matrix[0][0](x) * matrix[1][1](x) - matrix[0][1](x) * matrix[1][0](x)
29
30     df1_x1 = lambda x: 2*x[0]
31     df1_x2 = lambda x: 2*x[1]
32     df2_x1 = lambda x: 1
33     df2_x2 = lambda x: -math.e**x[1]
34
35     A1 = [
36         [f1, df1_x2],
37         [f2, df2_x2]
38     ]
39
40     A2 = [
41         [df1_x1, f1],
42         [df2_x1, f2]
43     ]
44
45     J = [
46         [df1_x1, df1_x2],
47         [df2_x1, df2_x2]
```

```

48     ]
49
50     A = [A1, A2]
51
52     x_next, x_curr, k = [x_0[i] - det(x_0, A[i])/det(x=x_0, matrix=J) for i in range(
53         len(x_0))], x_0, 1
54     while eps(x_curr, x_next) >= EPS:
55         k += 1
56         x_next, x_curr = [x_next[i] - det(x_next, A[i])/det(x=x_next, matrix=J) for i
57             in range(len(x_next))], x_next
58     return x_next, k
59
60 def simple_iterations_method(x_0: list[float], q: float, EPS: float) -> tuple[list[
61     float], int]:
62     """
63     f1(x1, x2): x1^2 + x2^2 - 16
64     f2(x1, x2): x1 - e^(x2) + 4
65
66     (x1 = phi1(x1, x2)): x1 = e^(x2) - 4
67     (x2 = phi2(x1, x2)): x2 = sqrt(16 - x1^2)
68
69     phi1(x1, x2) = e^(x2) - 4
70     phi1_dx1(x1, x2) = 0
71     phi1_dx2(x1, x2) = e^(x2)
72
73     phi2(x1, x2) = sqrt(16 - x1^2)
74     phi1_dx1(x1, x2) = -x1/sqrt(16 - x1^2)
75     phi1_dx2(x1, x2) = 0
76     """
77
78     def eps(vect1: list[float], vect2: list[float]) -> float:
79         return q*max((abs(vect1[i] - vect2[i]) for i in range(len(vect1))))/(1-q)
80
81     phi1 = lambda x: math.e ** x[1] - 4
82     phi2 = lambda x: -math.sqrt(16-x[0]**2)
83
84     x_next, x_curr, k = x_0, [3*i for i in x_0], 0
85     while eps(x_curr, x_next) >= EPS:
86         k += 1
87         x_next, x_curr = [phi1(x_next), phi2(x_next)], x_next
88     return x_next, k
89
90 if __name__ == "__main__":
91     deviation = 1e-9
92
93     result, number_of_iterations = newton_method(x_0=[-3.5, -2], EPS=deviation)
94     print('\n ')

```

```

94     print(f' (x_k): {result}\n (k): {number_of_iterations}')
95     print(f'      x_k: {f1(result), f2(result)}')
96     print()
97
98     result, number_of_iterations = simple_iterations_method(x_0=[-3.5, -2], q=0.9, EPS=
99         deviation)
100     print('\n ')
101     print(f' (x_k): {result}\n (k): {number_of_iterations}')
102     print(f'      x_k: {f1(result), f2(result)}')
103     print()

```

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5
6  auto f1 = [](vector<double> x) {
7      return pow(x[0], 2) + pow(x[1], 2) - 16;
8  };
9
10 auto f2 = [](vector<double> x) {
11     return x[0] - exp(x[1]) + 4;
12 };
13
14 pair<vector<double>, int> newton_method(vector<double> x_0, double EPS) {
15     /*
16         f1(x1, x2): x1^2 + x2^2 - 16
17         f2(x1, x2): x1 - e^(x2) + 4
18
19         f1  x1: 2*x1
20         f1  x2: 2*x2
21
22         f2  x1: 1
23         f2  x2: -e^(x2)
24
25         ( ):
26         x1 = -3.5
27         x2 = -2
28     */
29     auto eps = [](const vector<double>& vect1, const vector<double>& vect2) {
30         double max_diff = 0.0;
31         for (size_t i = 0; i < vect1.size(); ++i) {
32             max_diff = max(max_diff, abs(vect1[i] - vect2[i]));
33         }
34         return max_diff;
35     };
36
37     auto det = [](const vector<double>& x, const vector<vector<function<double(vector<
38         double>>>>& matrix) {
39         return matrix[0][0](x) * matrix[1][1](x) - matrix[0][1](x) * matrix[1][0](x);

```

```

39     };
40
41     auto df1_x1 = [](const vector<double>& x) { return 2 * x[0]; };
42     auto df1_x2 = [](const vector<double>& x) { return 2 * x[1]; };
43     auto df2_x1 = [](const vector<double>& x) { return 1; };
44     auto df2_x2 = [](const vector<double>& x) { return -exp(x[1]); };
45
46     vector<vector<function<double(vector<double>)>>>> A1 = {
47         {f1, df1_x2},
48         {f2, df2_x2}
49     };
50
51     vector<vector<function<double(vector<double>)>>>> A2 = {
52         {df1_x1, f1},
53         {df2_x1, f2}
54     };
55
56     vector<vector<function<double(vector<double>)>>>> J = {
57         {df1_x1, df1_x2},
58         {df2_x1, df2_x2}
59     };
60
61     vector<vector<vector<function<double(vector<double>)>>>> A = {A1, A2};
62
63     vector<double> x_next, x_curr = x_0;
64     x_next = {
65         x_0[0] - det(x_0, A[0])/det(x_0, J),
66         x_0[1] - det(x_0, A[1])/det(x_0, J)
67     };
68     int k = 1;
69
70     while (eps(x_curr, x_next) >= EPS){
71         k += 1;
72         x_curr = x_next;
73         x_next = {
74             x_next[0] - det(x_next, A[0])/det(x_next, J),
75             x_next[1] - det(x_next, A[1])/det(x_next, J)
76         };
77     }
78     return make_pair(x_next, k);
79 }
80
81
82
83
84
85 pair<vector<double>, int> simple_iterations_method(vector<double> x_0, double q,
86     double EPS) {
87     /*

```

```

87     f1(x1, x2): x1^2 + x2^2 - 16
88     f2(x1, x2): x1 - e^(x2) + 4
89
90     (x1 = phi1(x1, x2)): x1 = e^(x2) - 4
91     (x2 = phi2(x1, x2)): x2 = sqrt(16 - x1^2)
92
93     phi1(x1, x2) = e^(x2) - 4
94     phi1_dx1(x1, x2) = 0
95     phi1_dx2(x1, x2) = e^(x2)
96
97     phi2(x1, x2) = sqrt(16 - x1^2)
98     phi1_dx1(x1, x2) = -x1/sqrt(16 - x1^2)
99     phi1_dx2(x1, x2) = 0
100 */
101
102 auto eps = [](const vector<double>& vect1, const vector<double>& vect2, double q) {
103     double max_diff = 0.0;
104     for (size_t i = 0; i < vect1.size(); ++i) {
105         max_diff = max(max_diff, abs(vect1[i] - vect2[i]));
106     }
107     return q*max_diff/(1-q);
108 };
109
110 auto phi1 = [](const vector<double>& x) {
111     return exp(x[1]) - 4;
112 };
113
114 auto phi2 = [](const vector<double>& x) {
115     return -sqrt(16 - pow(x[0], 2));
116 };
117
118 vector<double> x_next = x_0, x_curr = {x_0[0]*3, x_0[1]*3};
119 int k = 1;
120
121 while (eps(x_curr, x_next, q) >= EPS){
122     k += 1;
123     x_curr = x_next;
124     x_next = {
125         phi1(x_next),
126         phi2(x_next)
127     };
128 }
129 return make_pair(x_next, k);
130 }
131
132
133 int main(){
134     double deviation = 1e-9;
135     vector<double> result, x_0 = {-3.5, -2};

```

```

136     int number_of_iterations;
137
138     tie(result, number_of_iterations) = newton_method(x_0, deviation);
139     cout << "\nNewton's method\n";
140     cout << "Result (x_k): [" << result[0] << ", " << result[1] << "]" << endl << "
        Number of iterations (k): " << number_of_iterations << endl;
141     cout << "The value of the function at the given root x_k: " << f1(result) << " " <<
        f2(result) << endl << endl;
142
143     tie(result, number_of_iterations) = simple_iterations_method(x_0, 0.9, deviation);
144     cout << "\nhe method of simple iterations\n";
145     cout << "Result (x_k): [" << result[0] << ", " << result[1] << "]" << endl << "
        Number of iterations (k): " << number_of_iterations << endl;
146     cout << "The value of the function at the given root x_k: " << f1(result) << " " <<
        f2(result) << endl << endl;
147
148     return 1;
149 }

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 figure, axes = plt.subplots(1)
6
7 angle = np.linspace(0, 2 * np.pi, 200)
8 x1 = 4 * np.cos(angle)
9 x2 = 4 * np.sin(angle)
10 axes.plot(x1, x2)
11
12 x2 = np.linspace(-5, 2.5, 500)
13 x1 = list(map(lambda i: np.e**i - 4, x2))
14 axes.plot(x1, x2)
15
16
17 plt.xticks(np.arange(min(*x1, *x2)-1, max(*x1, *x2)+1, 1.0))
18 plt.yticks(np.arange(min(*x1, *x2)-1, max(*x1, *x2)+1, 1.0))
19 axes.set_aspect(1)
20 plt.grid()
21 plt.show()

```