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

```
from typing import Callable


def func(x: float) -> float:
    return x ** 3 + x ** 2 - 2 * x - 1


def newton_method(x_0: float, EPS: float, f: Callable) -> tuple[float, int]:
    """
     f(x): x^3 + x^2 - 2x - 1
     : 3x^2 + 2x - 2
     : 6x + 2

     :
    f(x_0)*f''(x_0) > 0

          :
    x_0 = 2 ( )
    """
    def d_func(x: float) -> float:
        return 3 * (x ** 2) + 2 * x - 2

    def eps(val1: float, val2: float) -> float:
        return abs(val1 - val2)

    if func(x_0) == 0:
        return x_0, 0
    x_next, x_curr, k = x_0 - f(x_0)/d_func(x_0), x_0, 1
    while eps(x_curr, x_next) >= EPS:
        k += 1
        x_next, x_curr = x_next - f(x_next)/d_func(x_next), x_next
    return x_next, k


def simple_iterations_method(x_0: float, a: float, b: float, q: float, EPS: float, f: Callable) -> tuple[float, int]:
    """
     f(x): x^3 + x^2 - 2x - 1
        (x = phi(x)): x = (x^3 + x^2 - 1)/2
    phi(x) = (x^3 + x^2 - 1)/2
    phi'(x) = 1.5*x^2 + x

     :
    1) phi(x)[a, b] x[a, b]
    2)  q: |phi'(x)|<=q<1 x(a, b)

         :
```

```python
    q = 0.8
    [a, b] = [-1/3, (-1 + (1 + 1.5*4*q)**0.5)/3]
    x_0 = -0.25
    """
    def phi(x: float) -> float:
        return (x ** 3 + x ** 2 - 1)/2

    def eps(val1: float, val2: float) -> float:
        return q*abs(val1 - val2)/(1-q)

    if func(x_0) == 0:
        return x_0, 0

    x_next, x_curr, k = x_0 + 2 * EPS, x_0, 0
    while eps(x_curr, x_next) >= EPS:
        k += 1
        x_next, x_curr = phi(x_next), x_next
    return x_next, k


if __name__ == "__main__":
    deviation = 1e-9
    q = 0.8

    result, number_of_iterations = newton_method(x_0=2, EPS=deviation, f=func)
    print('\n ')
    print(f' (x_k): {result}\n  (k): {number_of_iterations}')
    print(f'     x_k: {func(result)}')
    print()

    parameters = {
        "x_0": -0.25,
        "a": -1 / 3,
        "b": (-1 + (1 + 1.5*4*q)**0.5)/3,
        "q": q,
        "EPS": deviation,
        "f": func
    }
    result, number_of_iterations = simple_iterations_method(**parameters)
    print('\n  ')
    print(f' (x_k): {result}\n  (k): {number_of_iterations}')
    print(f'     x_k: {func(result)}')
    print()
```

```cpp
#include <bits/stdc++.h>

using namespace std;


double func(double x){
```

```
    return pow(x, 3) + pow(x, 2) - 2 * x - 1;
}


pair<double, int> newton_method(double x_0, double EPS){
    /*
     f(x): x^3 + x^2 - 2x - 1
     : 3x^2 + 2x - 2
     : 6x + 2

     :
    f(x_0)*f''(x_0) > 0

          :
    x_0 = 2 ( )
    */

    // double dfunc(double x){
    // return 3 * pow(x, 2) + 2 * x - 2;
    // }

    // double eps(double val1, double val2){
    // return abs(val1 - val2);
    // }

    auto d_func = [](double x) {
        return 3 * pow(x, 2) + 2 * x - 2;
    };

    auto eps = [](double val1, double val2) {
        return abs(val1 - val2);
    };

    if (func(x_0) == 0)
        return make_pair(x_0, 0);

    int k = 1;
    double x_next = x_0 - func(x_0)/d_func(x_0), x_curr = x_0;

    while (eps(x_curr, x_next) >= EPS){
        k += 1;
        x_curr = x_next;
        x_next = x_next - func(x_next)/d_func(x_next);
    }
    return make_pair(x_next, k);
}



```

```
pair<double, int> simple_iterations_method(double x_0, double a, double b, double q, double EPS){

    /*
     f(x): x^3 + x^2 - 2x - 1
        (x = phi(x)): x = (x^3 + x^2 - 1)/2
    phi(x) = (x^3 + x^2 - 1)/2
    phi'(x) = 1.5*x^2 + x

     :
    1) phi(x)[a, b] x[a, b]
    2)  q: |phi'(x)|<=q<1 x(a, b)

        :
    q = 0.8
    [a, b] = [-1/3, (-1 + (1 + 1.5*4*q)**0.5)/3]
    x_0 = -0.25
    */

    auto phi = [](double x) {
        return (pow(x, 3) + pow(x, 2) - 1)/2;
    };

    auto eps = [](double val1, double val2, double q) {
        return q*abs(val1 - val2)/(1-q);
    };

    if (func(x_0) == 0)
        return make_pair(x_0, 0);

    int k = 0;
    double x_next = x_0 + 2 * EPS, x_curr = x_0;
    while (eps(x_curr, x_next, q) >= EPS){
        k += 1;
        x_curr = x_next;
        x_next = phi(x_next);
    }
    return make_pair(x_next, k);
}


int main(){
    double deviation = 1e-9, q = 0.8, result;
    int number_of_iterations;

    tie(result, number_of_iterations) = newton_method(2, deviation);
    cout << "\nNewton's method\n";
    cout << "Result (x_k): " << result << endl << "Number of iterations (k): " << number_of_iterations << endl;
```

```
    cout << "The value of the function at the given root x_k: " << func(result) << endl << endl;

    tie(result, number_of_iterations) = simple_iterations_method(-0.25, -1/3, (-1 + sqrt(1 + 1.5*4*q))/3, q, deviation);
    cout << "\nThe method of simple iterations\n";
    cout << "Result (x_k): " << result << endl << "Number of iterations (k): " << number_of_iterations << endl;
    cout << "The value of the function at the given root x_k: " << func(result) << endl << endl;

    return 1;
}
```

2.2 Методы простой итерации и Ньютона

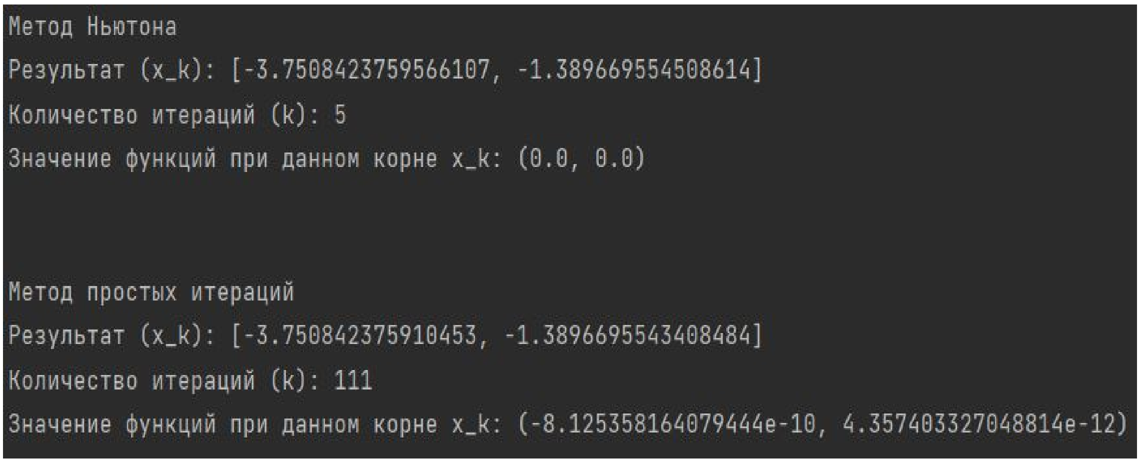
4 Постановка задачи

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

```
from typing import Callable
import math

f1 = lambda x: x[0]**2 + x[1]**2 - 16
f2 = lambda x: x[0] - math.e**x[1] + 4


def newton_method(x_0: list[float], EPS: float) -> tuple[list[float], int]:
    """
     f1(x1, x2): x1^2 + x2^2 - 16
     f2(x1, x2): x1 - e^(x2) + 4

      f1  x1: 2*x1
      f1  x2: 2*x2

      f2  x1: 1
      f2  x2: -e^(x2)

      (  ):
    x1 = -3.5
    x2 = -2
    """

    def eps(vect1: list[float], vect2: list[float]) -> float:
        return max((abs(vect1[i]-vect2[i]) for i in range(len(vect1))))

    def det(x: list[float], matrix: list[list[Callable]]) -> float:
        return matrix[0][0](x) * matrix[1][1](x) - matrix[0][1](x) * matrix[1][0](x)

    df1_x1 = lambda x: 2*x[0]
    df1_x2 = lambda x: 2*x[1]
    df2_x1 = lambda x: 1
    df2_x2 = lambda x: -math.e**x[1]

    A1 = [
        [f1, df1_x2],
        [f2, df2_x2]
    ]

    A2 = [
        [df1_x1, f1],
        [df2_x1, f2]
    ]

    J = [
        [df1_x1, df1_x2],
        [df2_x1, df2_x2]
```

```
    ]

    A = [A1, A2]

    x_next, x_curr, k = [x_0[i] - det(x_0, A[i])/det(x=x_0, matrix=J) for i in range(len(x_0))], x_0, 1
    while eps(x_curr, x_next) >= EPS:
        k += 1
        x_next, x_curr = [x_next[i] - det(x_next, A[i])/det(x=x_next, matrix=J) for i in range(len(x_next))], x_next
    return x_next, k


def simple_iterations_method(x_0: list[float], q: float, EPS: float) -> tuple[list[float], int]:
    """
     f1(x1, x2): x1^2 + x2^2 - 16
     f2(x1, x2): x1 - e^(x2) + 4

        (x1 = phi1(x1, x2)): x1 = e^(x2) - 4
        (x2 = phi2(x1, x2)): x2 = sqrt(16 - x1^2)

    phi1(x1, x2) = e^(x2) - 4
    phi1_dx1(x1, x2) = 0
    phi1_dx2(x1, x2) = e^(x2)

    phi2(x1, x2) = sqrt(16 - x1^2)
    phi1_dx1(x1, x2) = -x1/sqrt(16 - x1^2)
    phi1_dx2(x1, x2) = 0
    """

    def eps(vect1: list[float], vect2: list[float]) -> float:
        return q*max((abs(vect1[i] - vect2[i]) for i in range(len(vect1))))/(1-q)

    phi1 = lambda x: math.e ** x[1] - 4
    phi2 = lambda x: -math.sqrt(16-x[0]**2)

    x_next, x_curr, k = x_0, [3*i for i in x_0], 0
    while eps(x_curr, x_next) >= EPS:
        k += 1
        x_next, x_curr = [phi1(x_next), phi2(x_next)], x_next
    return x_next, k


if __name__ == "__main__":
    deviation = 1e-9

    result, number_of_iterations = newton_method(x_0=[-3.5, -2], EPS=deviation)
    print('\n ')
```

```
    print(f' (x_k): {result}\n  (k): {number_of_iterations}')
    print(f'     x_k: {f1(result), f2(result)}')
    print()

    result, number_of_iterations = simple_iterations_method(x_0=[-3.5, -2], q=0.9, EPS=deviation)
    print('\n  ')
    print(f' (x_k): {result}\n  (k): {number_of_iterations}')
    print(f'     x_k: {f1(result), f2(result)}')
    print()
```

```
#include <bits/stdc++.h>

using namespace std;


auto f1 = [](vector<double> x) {
    return pow(x[0], 2) + pow(x[1], 2) - 16;
};

auto f2 = [](vector<double> x) {
    return x[0] - exp(x[1]) + 4;
};

pair<vector<double>, int> newton_method(vector<double> x_0, double EPS) {
    /*
        f1(x1, x2): x1^2 + x2^2 - 16
        f2(x1, x2): x1 - e^(x2) + 4

         f1  x1: 2*x1
         f1  x2: 2*x2

         f2  x1: 1
         f2  x2: -e^(x2)

         (  ):
       x1 = -3.5
       x2 = -2
    */
    auto eps = [](const vector<double>& vect1, const vector<double>& vect2) {
        double max_diff = 0.0;
        for (size_t i = 0; i < vect1.size(); ++i) {
            max_diff = max(max_diff, abs(vect1[i] - vect2[i]));
        }
        return max_diff;
    };

    auto det = [](const vector<double>& x, const vector<vector<function<double(vector<double>)>>>& matrix) {
        return matrix[0][0](x) * matrix[1][1](x) - matrix[0][1](x) * matrix[1][0](x);
```

```
    };

    auto df1_x1 = [](const vector<double>& x) { return 2 * x[0]; };
    auto df1_x2 = [](const vector<double>& x) { return 2 * x[1]; };
    auto df2_x1 = [](const vector<double>& x) { return 1; };
    auto df2_x2 = [](const vector<double>& x) { return -exp(x[1]); };

    vector<vector<function<double(vector<double>)>>> A1 = {
        {f1, df1_x2},
        {f2, df2_x2}
    };

    vector<vector<function<double(vector<double>)>>> A2 = {
        {df1_x1, f1},
        {df2_x1, f2}
    };

    vector<vector<function<double(vector<double>)>>> J = {
        {df1_x1, df1_x2},
        {df2_x1, df2_x2}
    };

    vector<vector<vector<function<double(vector<double>)>>>> A = {A1, A2};

    vector<double> x_next, x_curr = x_0;
    x_next = {
        x_0[0] - det(x_0, A[0])/det(x_0, J),
        x_0[1] - det(x_0, A[1])/det(x_0, J)
    };
    int k = 1;

    while (eps(x_curr, x_next) >= EPS){
        k += 1;
        x_curr = x_next;
        x_next = {
            x_next[0] - det(x_next, A[0])/det(x_next, J),
            x_next[1] - det(x_next, A[1])/det(x_next, J)
        };
    }
    return make_pair(x_next, k);
}





pair<vector<double>, int> simple_iterations_method(vector<double> x_0, double q,
    double EPS) {
    /*
```

```
        f1(x1, x2): x1^2 + x2^2 - 16
        f2(x1, x2): x1 - e^(x2) + 4

            (x1 = phi1(x1, x2)): x1 = e^(x2) - 4
            (x2 = phi2(x1, x2)): x2 = sqrt(16 - x1^2)

        phi1(x1, x2) = e^(x2) - 4
        phi1_dx1(x1, x2) = 0
        phi1_dx2(x1, x2) = e^(x2)

        phi2(x1, x2) = sqrt(16 - x1^2)
        phi1_dx1(x1, x2) = -x1/sqrt(16 - x1^2)
        phi1_dx2(x1, x2) = 0
    */

    auto eps = [](const vector<double>& vect1, const vector<double>& vect2, double q) {
        double max_diff = 0.0;
        for (size_t i = 0; i < vect1.size(); ++i) {
            max_diff = max(max_diff, abs(vect1[i] - vect2[i]));
        }
        return q*max_diff/(1-q);
    };

    auto phi1 = [](const vector<double>& x) {
        return exp(x[1]) - 4;
    };

    auto phi2 = [](const vector<double>& x) {
        return -sqrt(16 - pow(x[0], 2));
    };

    vector<double> x_next = x_0, x_curr = {x_0[0]*3, x_0[1]*3};
    int k = 1;

    while (eps(x_curr, x_next, q) >= EPS){
        k += 1;
        x_curr = x_next;
        x_next = {
            phi1(x_next),
            phi2(x_next)
        };
    }
    return make_pair(x_next, k);
}


int main(){
    double deviation = 1e-9;
    vector<double> result, x_0 = {-3.5, -2};
```

```
    int number_of_iterations;

    tie(result, number_of_iterations) = newton_method(x_0, deviation);
    cout << "\nNewton's method\n";
    cout << "Result (x_k): [" << result[0] << ", " << result[1] << "]" << endl << "
        Number of iterations (k): " << number_of_iterations << endl;
    cout << "The value of the function at the given root x_k: " << f1(result) << " " <<
         f2(result) << endl << endl;

    tie(result, number_of_iterations) = simple_iterations_method(x_0, 0.9, deviation);
    cout << "\nhe method of simple iterations\n";
    cout << "Result (x_k): [" << result[0] << ", " << result[1] << "]" << endl << "
        Number of iterations (k): " << number_of_iterations << endl;
    cout << "The value of the function at the given root x_k: " << f1(result) << " " <<
         f2(result) << endl << endl;

    return 1;
}
```

```
import numpy as np
import matplotlib.pyplot as plt


figure, axes = plt.subplots(1)

angle = np.linspace(0, 2 * np.pi, 200)
x1 = 4 * np.cos(angle)
x2 = 4 * np.sin(angle)
axes.plot(x1, x2)

x2 = np.linspace(-5, 2.5, 500)
x1 = list(map(lambda i: np.e**i - 4, x2))
axes.plot(x1, x2)


plt.xticks(np.arange(min(*x1, *x2)-1, max(*x1, *x2)+1, 1.0))
plt.yticks(np.arange(min(*x1, *x2)-1, max(*x1, *x2)+1, 1.0))
axes.set_aspect(1)
plt.grid()
plt.show()
```