Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирования»

Лабораторная работа №2 по курсу «Численные методы»

Студент: Т. Д. Голубев
Преподаватель: И. Э. Иванов
Группа: М8О-306Б-22
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №2.1

Задача: 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.

$$x^4 - 2x - 1 = 0$$

Описание

Численное решение нелинейных (алгебраических или трансцендентных) уравнений вида

$$f(x) = 0 \tag{2.1}$$

заключается в нахождении значений x, удовлетворяющих (с заданной точностью) данному уравнению и состоит из следующих основных этапов:

1. Отделение (изоляция, локализация) корней уравнения.

2. Уточнение с помощью некоторого вычислительного алгоритма конкретного выделенного корня с заданной точностью.

Целью первого этапа является нахождение отрезков из области определения функции $f(x)$, внутри которых содержится только один корень решаемого уравнения. Иногда ограничиваются рассмотрением лишь какой-нибудь части области определения, вызывающей по тем или иным соображениям интерес. Для реализации данного этапа используются графические или аналитические способы.

При завершении первого этапа, должны быть определены промежутки, на каждом из которых содержится только один корень уравнения.

Для уточнения корня с требуемой точностью обычно применяется какой-либо итерационный метод, заключающийся в построении числовой последовательности $x^{(k)}$ $(k = 0, 1, 2, ...)$, сходящейся к искомому корню $x^{(*)}$ уравнения (2.1)

Метод простой итерации

При использовании метода простой итерации уравнение

$$f(x) = 0 \tag{2.1}$$

заменяется эквивалентным уравнением с выделенным линейным членом:

$$x = \varphi(x). \tag{2.5}$$

Решение ищется путём построения последовательности:

$$x^{(k+1)} = \varphi(x^{(k)}), \quad k = 0, 1, 2, \ldots \tag{2.6}$$

начиная с некоторого заданного значения $x^{(0)}$.

Если $\varphi(x)$ — непрерывная функция, а последовательность $\{x^{(k)}\}$ сходится, то значение $x^{(s)} = \lim_{k\to\infty} x^{(k)}$ является решением уравнения (2.5).

Условия сходимости метода и оценка его погрешности определяются теоремой:

Теорема 1 (2.3). *Пусть функция $\varphi(x)$ определена и дифференцируема на отрезке $[a, b]$. Тогда если выполняются условия:*

1. $\varphi(x) \in [a, b] \quad \forall x \in [a, b]$,
2. $\exists q : |\varphi'(x)| \le q < 1 \quad \forall x \in (a, b)$,

то уравнение (2.5) имеет единственный корень $x^{(s)}$ на $[a, b]$, и последовательность $\{x^{(k)}\}$ сходится к $x^{(s)}$ при любом $x^{(0)} \in [a, b]$. При этом справедливы оценки погрешности:

$$\left|x^{(s)} - x^{(k+1)}\right| \le \frac{q}{1-q}\left|x^{(k+1)} - x^{(k)}\right|,$$

$$\left|x^{(s)} - x^{(k+1)}\right| \le \frac{q^{k+1}}{1-q}\left|x^{(1)} - x^{(0)}\right|. \tag{2.7}$$

Метод Ньютона (метод касательных)

При нахождении корня уравнения

$$f(x) = 0 \tag{2.1}$$

методом Ньютона итерационный процесс определяется формулой:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad k = 0, 1, 2, \ldots \tag{2.2}$$

Для начала вычислений требуется задание начального приближения $x^{(0)}$.

Условия сходимости метода определяются следующей теоремой:

Теорема 2 (2.2). *Пусть на отрезке* $[a, b]$ *функция* $f(x)$ *имеет первую и вторую производные постоянного знака и пусть* $f(a)f(b) < 0$. *Тогда если точка* $x^{(0)}$ *выбрана на* $[a, b]$ *так, что*

$$f(x^{(0)})f''(x^{(0)}) > 0, \tag{2.3}$$

то начатая с неё последовательность $\{x^{(k)}\}$ $(k = 0, 1, 2, \ldots)$, *определяемая методом Ньютона (2.2), монотонно сходится к корню* $x^{(v)} \in (a, b)$ *уравнения (2.1).*

Исходный код

```
package cat.mood;

import java.util.function.Function;

public class A {
    public static double derivative(Function<Double, Double> f, double x, double eps)
        double dy = f.apply(x + eps) - f.apply(x);
        return dy / eps;
    }

    public static double secondDerivative(Function<Double, Double> f, double x, doubl
        double fPlus = f.apply(x + eps);
        double fMinus = f.apply(x - eps);
        double fCenter = f.apply(x);

        return (fPlus - 2 * fCenter + fMinus) / (eps * eps);
    }

    public static boolean checkFunction(Function<Double, Double> phi, double eps, dou
        double x = a;
        while (x < b) {
            double y = phi.apply(x);
            if (y < a || y > b) {
                return false;
            }
            if (Math.abs(derivative(phi, x, eps)) >= 1) {
                return false;
            }
            x += eps;
        }
```

```
    return true;
}

public static double iteration(Function<Double, Double> phi, double eps, double a
    boolean check = checkFunction(phi, eps, a, b);
    if (!check) {
        throw new RuntimeException("Не выполнено условие сходимости");
    }

    double prev = a;
    double cur = phi.apply(prev);
    int iters = 1;
    while (Math.abs(cur - prev) > eps) {
        prev = cur;
        cur = phi.apply(prev);
        ++iters;
    }

    System.out.println("Количество итераций: " + iters);

    return cur;
}

public static double newton(Function<Double, Double> f, double eps, double a, doul
    if (f.apply(a) * f.apply(b) >= 0) {
        throw new RuntimeException("Не выполнено условие сходимости");
    }

    double prev = b;
    while (prev > a) {
        if (f.apply(prev) * secondDerivative(f, prev, eps) > 0) {
            break;
        }
        prev -= eps;
    }
    if (f.apply(prev) * secondDerivative(f, prev, eps) <= 0) {
        throw new RuntimeException("Не выполнено условие сходимости");
    }
    int iters = 1;
    double cur = prev - f.apply(prev) / derivative(f, prev, eps);
```

```
        while (Math.abs(cur - prev) > eps) {
            prev = cur;
            cur = prev - f.apply(prev) / derivative(f, prev, eps);
            ++iters;
        }

        System.out.println("Количество итераций: " + iters);

        return cur;
    }

    public static void main(String[] args) {
        System.out.println("Метод простой итерации:");
        System.out.println(iteration(x -> (Math.pow(2 * x + 1, 0.25)), 0.000001, 0, 2
        System.out.println("Метод Ньютона:");
        System.out.println(newton(x -> (Math.pow(x, 4) - 2 * x - 1), 0.000001, 0, 2))
    }
}
```

Результат

```
Метод простой итерации:
Количество итераций: 10
1.3953368880468564
Метод Ньютона:
Количество итераций: 6
1.3953369944670735
```

Вывод

В ходе выполнения работы были успешно реализованы и протестированы два численных метода решения нелинейных уравнений: метод простой итерации и метод Ньютона. Проведенные вычисления позволили сделать следующие выводы:

Сходимость методов:

- Метод Ньютона продемонстрировал более быструю сходимость (6 итераций) по сравнению с методом простой итерации (10 итераций)
- Оба метода сошлись к близким значениям корня: 1.395336888 (простая итерация) и 1.395336994 (метод Ньютона)

Точность результатов: Различие между полученными значениями составляет около 1.06×10^{-7}, что свидетельствует о хорошей точности обоих методов

Эффективность методов:

- Метод Ньютона оказался более эффективным по количеству требуемых итераций
- Метод простой итерации, хотя и потребовал больше вычислений, проще в реализации и не требует вычисления производной

Результаты работы подтвердили теоретические положения о скорости сходимости рассматриваемых методов и продемонстрировали их практическую применимость для решения нелинейных уравнений.

Лабораторная работа №2.2

Задача: 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.

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

Описание

Систему нелинейных уравнений с n неизвестными можно записать в виде

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0 \\ f_2(x_1, x_2, ..., x_n) = 0 \\ ... \\ f_n(x_1, x_2, ..., x_n) = 0 \end{cases} \tag{1}$$

или, более коротко, в векторной форме

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{2}$$

где x — вектор неизвестных величин, f — вектор-функция

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad f = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{pmatrix} \quad \mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

В редких случаях для решения такой системы удается применить метод последовательного исключения неизвестных и свести решение исходной задачи к решению одного нелинейного уравнения с одним неизвестным. Значения других неизвестных величин находятся соответствующей подстановкой в конкретные выражения. Однако в подавляющем большинстве случаев для решения систем нелинейных уравнений используются итерационные методы.

Метод Ньютона

Если определено начальное приближение $\mathbf{x^{(0)}} = (\mathbf{x_1^{(0)}}, \mathbf{x_2^{(0)}}, ..., \mathbf{x_n^{(0)}})^{\mathbf{T}}$, итерационный процесс нахождения решения системы методом Ньютона можно представить в виде

$$\begin{pmatrix} x_1^{(k+1)} = x_1^{(k)} + \Delta x_1^{(k)} \\ x_2^{(k+1)} = x_2^{(k)} + \Delta x_2^{(k)} \\ ... \\ x_n^{(k+1)} = x_n^{(k)} + \Delta x_n^{(k)} \end{pmatrix} \tag{3}$$

$k = 0, 1, 2, ...$

где значения приращений $\Delta x_1^{(k)}, \Delta x_2^{(k)}, ..., \Delta x_n^{(k)}$ определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)})^T$

$$\begin{cases} f_1(x^{(k)}) + \frac{\partial f_1(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_1(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_1(x^{(k)})}{\partial x_n}\Delta x_n(k) = 0 \\ f_2(x^{(k)}) + \frac{\partial f_2(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_2(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_2(x^{(k)})}{\partial x_n}\Delta x_n(k) = 0 \\ ... \\ f_n(x^{(k)}) + \frac{\partial f_n(x^{(k)})}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_n(x^{(k)})}{\partial x_2}\Delta x_2^{(k)} + ... + \frac{\partial f_n(x^{(k)})}{\partial x_n}\Delta x_n(k) = 0 \end{cases} \tag{4}$$

В векторно-матричной форме расчетные формулы имеют вид

$$x^{(k+1)} = x^{(k)} + \Delta x^{(k)} \tag{5}$$

$k = 0, 1, 2, ...$

где вектор приращений $\Delta x^{(k)} = \begin{pmatrix} \Delta x_1^{(k)} \\ \Delta x_2^{(k)} \\ \vdots \\ \Delta x_n^{(k)} \end{pmatrix}$ находится из решения уравнения

$$f(x^{(k)}) + J(x^{(k)})\Delta x^{(k)} = 0 \tag{6}$$

Здесь $J(x) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & ... & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & ... & \frac{\partial f_2(x)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(x)}{\partial x_1} & \frac{\partial f_n(x)}{\partial x_2} & ... & \frac{\partial f_n(x)}{\partial x_n} \end{pmatrix}$ — матрица Якоби первых производных вектор-функции $f(x)$.

Выражая из (6) вектор приращений $\Delta x^{(k)}$ и подставляя его в (5), итерационный процесс нахождения решения можно записать в виде

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})f(x^{(k)}) \tag{7}$$

$k = 0, 1, 2, ...$

где $J^{-1}(x)$ — матрица, обратная матрице Якоби. Формула (7) есть обобщение формулы (2.2) на случай систем нелинейных уравнений.

При реализации алгоритма метода Ньютона в большинстве случаев предпочтительным является не вычисление обратной матрицы $J^{-1}(x^{(k)})$, а нахождение из системы (4) значений приращений $\Delta x_1^{(k)}, \Delta x_2^{(k)}, ..., \Delta x_n^{(k)}$ и вычисление нового приближения по (3). Для решения таких линейных систем можно привлекать самые разные методы, как прямые, так и итерационные, с учетом размерности n решаемой задачи и специфики матриц Якоби $J(x)$ (например, симметрии, разреженности и т.п.).

Использование метода Ньютона предполагает дифференцируемость функций $f_1(x), f_2(x), ..., f_n(x)$ и невырожденность матрицы Якоби $\det J(x^{(k)}) \neq 0$. В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная. В практических вычислениях в качестве условия окончания итераций обычно используется критерий

$$\|x^{(k+1)} - x^{(k)} \leq \varepsilon\| \tag{8}$$

где ε — заданная точность.

Метод итераций

При использовании метода простой итерации система уравнений (1) приводится к эквивалентной системе специального вида

$$\begin{cases} x_1 = \varphi_1(x_1, x_2, ..., x_n) \\ x_2 = \varphi_2(x_1, x_2, ..., x_n) \\ ... \\ x_n = \varphi_n(x_1, x_2, ..., x_n) \end{cases} \tag{9}$$

или, в векторной форме

$$x = \varphi(x),\ \varphi(x) = \begin{pmatrix} \varphi_1(x) \\ \varphi_2(x) \\ \vdots \\ \varphi_n(x) \end{pmatrix} \tag{10}$$

где функции $\varphi_1(x), \varphi_2(x), ..., \varphi_n(x)$ — определены и непрерывны в некоторой окрестности искомого изолированного решения $x^{(*)} = (x_1^{(*)}, x_2^{(*)}, ..., x_n^{(*)})^T$.

Если выбрано некоторое начальное приближение $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, ..., x_n^{(0)})^T$, последующие приближения в методе простой итерации находятся по формулам

$$\begin{cases} x_1^{(k+1)} = \varphi_1(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}) \\ x_2^{(k+1)} = \varphi_2(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}) \\ ... \\ x_n^{(k+1)} = \varphi_n(x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}) \end{cases} \tag{11}$$

или, в векторной форме

$$x^{(k+1)} = \varphi(x^{(k)}) \tag{12}$$

$k = 0, 1, 2, ...$

Если последовательность векторов $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)})^T$ сходится, то она сходится к решению $x^{(*)} = (x_1^{(*)}, x_2^{(*)}, ..., x_n^{(*)})^T$. Достаточное условие сходимости итерационного процесса (11) формулируется следующим образом

Теорема 3. *Пусть вектор-функция $\varphi(x)$ непрерывна, вместе со своей производной*

$$\varphi'(x) = \begin{bmatrix} \frac{\partial \varphi_1(x)}{\partial x_1} & \frac{\partial \varphi_1(x)}{\partial x_2} & \dots & \frac{\partial \varphi_1(x)}{\partial x_n} \\ \frac{\partial \varphi_2(x)}{\partial x_1} & \frac{\partial \varphi_2(x)}{\partial x_2} & \dots & \frac{\partial \varphi_2(x)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \varphi_n(x)}{\partial x_1} & \frac{\partial \varphi_n(x)}{\partial x_2} & \dots & \frac{\partial \varphi_n(x)}{\partial x_n} \end{bmatrix},$$

в ограниченной выпуклой замкнутой области G и

$$\max_{x \in G} \|\varphi'(x)\| \le q < 1$$

где q — постоянная. Если $x^{(0)} \in G$ и все последовательные приближения

$$x^{(k+1)} = \varphi(x^{(k)}), \quad k = 0, 1, 2, \dots$$

также содержатся в G, то процесс итерации сходится к единственному решению уравнения

$$x = \varphi(x)$$

в области G и справедливы оценки погрешности ($\forall k \in \mathbb{N}$):

$$\|x^{(v)} - x^{(k+1)}\| \leq \frac{q^{k+1}}{1-q}\|x^{(1)} - x^{(0)}\|,$$

$$\|x^{(v)} - x^{(k+1)}\| \leq \frac{q}{1-q}\|x^{(k+1)} - x^{(k)}\|.$$

Исходный код

```
package cat.mood;

import java.util.Arrays;
import java.util.function.Function;

public class B {
    public static void main(String[] args) {
        Function<double[], double[]> phi = x -> new double[]{
                Math.sqrt(2 * Math.log10(x[1]) + 1),
                (x[0] * x[0] + 1) / x[0]
        };

        double[] initialGuess = {1.5, 2};
        int maxIterations = 100;
        double eps = 1e-6;
        double checkRadius = 0.5;

        double[] solution = iterations(phi, initialGuess, maxIterations, eps, checkRa

        System.out.println("Метод итераций:");
        if (solution != null) {
            System.out.println("Решение найдено:");
            for (int i = 0; i < solution.length; i++) {
                System.out.printf("x%d = %.6f\n", i, solution[i]);
            }
        } else {
            System.out.println("Решение не найдено (нарушено условие сходимости).");
```

```
    }

    Function<double[], double[]> f = x -> new double[]{
            x[0] * x[0] - 2 * Math.log10(x[1]) - 1,
            x[0] * x[0] - x[0] * x[1] + 1
    };

    solution = newton(f, initialGuess, maxIterations, eps);
    System.out.println("Метод Ньютона:");
    System.out.println("Решение найдено:");
    for (int i = 0; i < solution.length; i++) {
        System.out.printf("x%d = %.6f\n", i, solution[i]);
    }
}

public static double[] iterations(
        Function<double[], double[]> phi,
        double[] initialGuess,
        int maxIterations,
        double eps,
        double checkRadius) {

    int n = initialGuess.length;
    double[] current = Arrays.copyOf(initialGuess, n);

    if (!checkConvergence(phi, current, checkRadius, eps)) {
        return null;
    }

    for (int iter = 0; iter < maxIterations; iter++) {
        double[] next = phi.apply(current);
        double error = 0;

        for (int i = 0; i < n; i++) {
            error = Math.max(error, Math.abs(next[i] - current[i]));
        }

        if (error < eps) {
            System.out.printf("Сходимость достигнута за %d итераций.\n", iter + 1
            return next;
        }
```

```
            current = Arrays.copyOf(next, n);
        }

        System.out.println("Достигнуто максимальное число итераций.");
        return current;
    }

    // Проверка условия сходимости (||J||_inf < 1)
    public static boolean checkConvergence(
            Function<double[], double[]> phi,
            double[] point,
            double radius,
            double eps) {

        int n = point.length;
        double[][] testPoints = generateTestPoints(point, radius);

        for (double[] p : testPoints) {
            double[][] J = computeJacobian(phi, p, eps);
            double norm = 0;

            for (int i = 0; i < n; i++) {
                double rowSum = 0;
                for (int j = 0; j < n; j++) {
                    rowSum += Math.abs(J[i][j]);
                }
                norm = Math.max(norm, rowSum);
            }

            if (norm >= 1.0) {
                System.out.printf("Норма Якоби = %.4f в точке %s\n", norm, Arrays.toS
                return false;
            }
        }

        return true;
    }

    // Генерация тестовых точек в окрестности
    private static double[][] generateTestPoints(double[] center, double radius) {
```

```
    int n = center.length;
    int numPoints = 1 << n; // 2^n точек (все комбинации +-radius)
    double[][] points = new double[numPoints][n];

    for (int i = 0; i < numPoints; i++) {
        for (int j = 0; j < n; j++) {
            points[i][j] = center[j] + (((i >> j) & 1) == 1 ? radius : -radius);
        }
    }

    return points;
}

public static double determinant(double[][] A) {
    int n = A.length;
    if (n == 2) {
        return A[0][0] * A[1][1] - A[0][1] * A[1][0];
    } else if (n == 3) {
        return A[0][0] * (A[1][1] * A[2][2] - A[1][2] * A[2][1])
                - A[0][1] * (A[1][0] * A[2][2] - A[1][2] * A[2][0])
                + A[0][2] * (A[1][0] * A[2][1] - A[1][1] * A[2][0]);
    } else {
        throw new UnsupportedOperationException("n > 3");
    }
}

public static double[] newton(
        Function<double[], double[]> F,
        double[] initialGuess,
        int maxIterations,
        double eps) {

    int n = initialGuess.length;
    double[] x = Arrays.copyOf(initialGuess, n);

    double[][] J = computeJacobian(F, x, eps);
    if (Math.abs(determinant(J)) < eps) {
        System.out.println("Ошибка: Якобиан вырожден в начальной точке.");
        return null;
    }
```

```
    for (int iter = 0; iter < maxIterations; iter++) {
        double[] Fx = F.apply(x);
        J = computeJacobian(F, x, eps); // Численный Якобиан

        // Решаем линейную систему J * deltaX = -Fx
        double[] deltaX = solveLinearSystem(J, Fx);

        for (int i = 0; i < n; i++) {
            x[i] += deltaX[i];
        }

        // Проверка на сходимость
        double error = 0;
        for (double d : deltaX) {
            error = Math.max(error, Math.abs(d));
        }

        if (error < eps) {
            System.out.printf("Сходимость за %d итераций.\n", iter + 1);
            return x;
        }
    }

    System.out.println("Достигнут максимум итераций.");
    return null;
}

// Численное вычисление Якобиана
public static double[][] computeJacobian(
        Function<double[], double[]> F,
        double[] x,
        double eps) {

    int n = x.length;
    double[][] J = new double[n][n];
    double[] Fx = F.apply(x);

    for (int j = 0; j < n; j++) {
        double[] xPlusH = Arrays.copyOf(x, n);
        xPlusH[j] += eps;
```

```
            double[] FxPlusH = F.apply(xPlusH);

            for (int i = 0; i < n; i++) {
                J[i][j] = (FxPlusH[i] - Fx[i]) / eps;
            }
        }

        return J;
    }

    public static double[] solveLinearSystem(double[][] J, double[] Fx) {
        int n = Fx.length;
        double[][] A = new double[n][n + 1];

        for (int i = 0; i < n; i++) {
            System.arraycopy(J[i], 0, A[i], 0, n);
            A[i][n] = -Fx[i];
        }

        for (int k = 0; k < n; k++) {
            int maxRow = k;
            for (int i = k + 1; i < n; i++) {
                if (Math.abs(A[i][k]) > Math.abs(A[maxRow][k])) {
                    maxRow = i;
                }
            }

            double[] temp = A[k];
            A[k] = A[maxRow];
            A[maxRow] = temp;

            for (int i = k + 1; i < n; i++) {
                double factor = A[i][k] / A[k][k];
                for (int j = k; j <= n; j++) {
                    A[i][j] -= factor * A[k][j];
                }
            }
        }

        double[] deltaX = new double[n];
        for (int i = n - 1; i >= 0; i--) {
```

```
            double sum = 0;
            for (int j = i + 1; j < n; j++) {
                sum += A[i][j] * deltaX[j];
            }
            deltaX[i] = (A[i][n] - sum) / A[i][i];
        }

        return deltaX;
    }
}
```

Результат

```
Сходимость достигнута за 11 итераций.
Метод итераций:
Решение найдено:
x0 = 1,275762
x1 = 2,059607
Сходимость за 4 итераций.
Метод Ньютона:
Решение найдено:
x0 = 1,275762
x1 = 2,059607
```

Вывод

В ходе выполнения работы были успешно реализованы и протестированы два численных метода решения систем нелинейных уравнений: метод простой итерации и метод Ньютона. Проведенные вычисления позволили сделать следующие выводы:

Результаты вычислений:

- Оба метода пришли к идентичному решению:

$$x_0 = 1.275762$$
$$x_1 = 2.059607$$

- Метод Ньютона показал более быструю сходимость (4 итерации) по сравнению с методом простой итерации (11 итераций)

Эффективность методов:

- Метод Ньютона продемонстрировал ожидаемо более высокую скорость сходимости (квадратичная сходимость против линейной)
- Несмотря на большее количество итераций, метод простой итерации может быть предпочтительнее в случаях, когда вычисление матрицы Якоби затруднительно

Точность результатов:

- Совпадение результатов, полученных разными методами, подтверждает корректность реализации алгоритмов
- Оба метода обеспечили требуемую точность решения

Практические рекомендации:

- Для систем с легко вычисляемым Якобианом рекомендуется использовать метод Ньютона
- В случаях сложного аналитического дифференцирования целесообразно применять метод простой итерации
- Начальное приближение, определенное графическим методом, оказалось удачным для обоих методов

Результаты работы подтвердили теоретические положения о скорости сходимости рассматриваемых методов и продемонстрировали их практическую применимость для решения систем нелинейных уравнений. Особенно показательным является факт совпадения результатов, полученных принципиально разными численными методами.