Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирования»

Лабораторная работа №3 по курсу «Численные методы»

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

Москва, 2025

Лабораторная работа №3.1

Задача: Используя таблицу значений Y функции $y = f(x)$, вычисленных в точках $X_i, i = 0, ..., 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки X_i, Y_i. Вычислить значение погрешности интерполяции в точке $X*$.

$$y = \arcsin(x) + x$$

а) $X_i = -0.4, -0.1, 0.2, 0.5$ б) $X_i = -0.4, 0, 0.2, 0.5$ $X* = 0.1$

Описание

Постановка задачи

Дана табличная функция в узлах X_i:

$$(X_i, Y_i), \quad i = 0, 1, 2, 3$$

где $Y_i = f(X_i)$. Требуется:

1. Построить интерполяционные многочлены:
   - В форме Лагранжа $L_3(x)$
   - В форме Ньютона $P_3(x)$
2. Вычислить абсолютную погрешность в точке X^*:

$$\Delta = |f(X^*) - P(X^*)|, \quad P \in \{L_3, P_3\}$$

Интерполяционный многочлен Лагранжа

Для $n + 1$ узлов строится по формуле:

$$L_n(x) = \sum_{i=0}^{n} Y_i \cdot \ell_i(x)$$

где базисные полиномы:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - X_j}{X_i - X_j}$$

Свойства:

- Точно проходит через все узлы: $L_n(X_i) = Y_i$
- Степень многочлена равна n
- Чувствителен к добавлению новых узлов

Интерполяционный многочлен Ньютона

Строится через разделённые разности:

$$P_n(x) = f[X_0] + \sum_{k=1}^{n} f[X_0, \ldots, X_k] \cdot \omega_k(x)$$

где:

- $\omega_k(x) = \prod_{i=0}^{k-1}(x - X_i)$
- Разделённые разности:

$$\begin{aligned} f[X_i] &= Y_i \\ f[X_i, X_j] &= \frac{f[X_j] - f[X_i]}{X_j - X_i} \\ f[X_i, \ldots, X_k] &= \frac{f[X_{i+1}, \ldots, X_k] - f[X_i, \ldots, X_{k-1}]}{X_k - X_i} \end{aligned}$$

Преимущества:

- Удобен при добавлении новых узлов
- Позволяет оценить погрешность по первому отброшенному члену

Погрешность интерполяции

Оценивается через остаточный член:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \cdot \omega_{n+1}(x), \quad \xi \in [X_0, X_n]$$

где $\omega_{n+1}(x) = \prod_{i=0}^{n}(x - X_i)$.
На практике вычисляется как:

$$\Delta(X^*) = |f(X^*) - P_n(X^*)|, \quad P_n \in \{L_n, P_n\}$$

Сравнение методов

Характеристика	Лагранж	Ньютон
Чувствительность к новым узлам	Требует пересчёта	Частичный пересчёт
Вычислительная сложность	$O(n^2)$	$O(n^2)$

Исходный код

```
package cat.mood;

import java.util.function.Function;

public class A {
    public record Pair(Function<Double, Double> first, Function<String, String> secon

    static double omega(int n, int idx, double[] x) {
        double result = 1;
        for (int i = 0; i < n; ++i) {
            if (i != idx) {
                result *= x[idx] - x[i];
            }
        }

        return result;
    }

    public static double[][] difference(double[] x, double[] y) {
        int n = x.length;
        double[][] table = new double[n][n];


        for (int i = 0; i < n; i++) {
            table[i][0] = y[i];
        }

        for (int j = 1; j < n; j++) {
            for (int i = 0; i < n - j; i++) {
                table[i][j] = (table[i + 1][j - 1] - table[i][j - 1]) / (x[i + j] - x
            }
        }
```

```java
    return table;
}

public static Pair lagrange(double[] x, double[] y) {
    int n = x.length;
    double[] w = new double[n];
    for (int i = 0; i < n; ++i) {
        w[i] = omega(n, i, x);
    }

    Function<String, String> fs = str -> {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n; ++i) {
            if (y[i] / w[i] > 0) {
                sb.append("+");
            }
            sb.append(y[i] / w[i]);
            for (int j = 0; j < n; ++j) {
                if (i != j) {
                    sb.append("(x ");
                    if (x[j] < 0) {
                        sb.append("+ ").append(x[j]);
                    } else {
                        sb.append("- ").append(x[j]);
                    }
                    sb.append(")");
                }
            }
            sb.append(" ");
        }
        return sb.toString();
    };

    Function<Double, Double> fd = t -> {
        double f = 0;
        for (int i = 0; i < n; ++i) {
            double fi = y[i] / w[i];
            for (int j = 0; j < n; ++j) {
                if (i != j) {
                    fi *= t - x[j];
```

```
                }
            }
            f += fi;
        }

        return f;
    };
    return new Pair(fd, fs);
}

public static Pair newton(double[] x, double[] y) {
    double[][] d = difference(x, y);
    int n = x.length;

    Function<String, String> fs = str -> {
        StringBuilder sb = new StringBuilder();
        sb.append(y[0]);
        for (int i = 1; i < n; ++i) {
            if (d[0][i] > 0) {
                sb.append(" + ");
            } else {
                sb.append(" - ");
            }
            sb.append(Math.abs(d[0][i]));
            for (int j = 0; j < i; ++j) {
                sb.append("(x ");
                if (x[i] < 0) {
                    sb.append("+ ");
                } else {
                    sb.append("- ");
                }
                sb.append(Math.abs(x[j])).append(")");
            }
        }
        return sb.toString();
    };

    Function<Double, Double> fd = t -> {
        double f = y[0];
        for (int i = 1; i < n; ++i) {
            double fi = d[0][i];
```

```
            for (int j = 0; j < i; ++j) {
                fi *= t - x[j];
            }
            f += fi;
        }

        return f;
    };

    return new Pair(fd, fs);
}

static void solve(double[] x, double[] y, double t, Function<Double, Double> f) {
    var L = lagrange(x, y);
    System.out.println("Многочлен Лагранжа:");
    System.out.println(L.second.apply("x"));
    System.out.println("Многочлен Ланкранжа в точке x = " + t + ": " + L.first.app
    System.out.println("Функция в точке x = " + t + ": " + f.apply(t));
    System.out.println("Погрешность: " + Math.abs(f.apply(t) - L.first.apply(t)))

    System.out.println();

    var N = newton(x, y);
    System.out.println("Многочлен Ньютона:");
    System.out.println(N.second.apply("x"));
    System.out.println("Многочлен Ньютона в точке x = " + t + ": " + N.first.apply
    System.out.println("Функция в точке x = " + t + ": " + f.apply(t));
    System.out.println("Погрешность: " + Math.abs(f.apply(t) - N.first.apply(t)))
}

public static void main(String[] args) {
    Function<Double, Double> f = x -> Math.asin(x) + x;
    double[] x1 = {-0.4, -0.1, 0.2, 0.5};
    double[] x2 = {-0.4, 0, 0.2, 0.5};
    double[] y1 = new double[x1.length];
    double[] y2 = new double[x2.length];
    for (int i = 0; i < x1.length; ++i) {
        y1[i] = f.apply(x1[i]);
        y2[i] = f.apply(x2[i]);
    }
```

```java
        double t = 0.1;

        System.out.println("a)");
        solve(x1, y1, t, f);
        System.out.println("б)");
        solve(x2, y2, t, f);
    }
}
```

Результат

```
а)
Многочлен Лагранжа:
+5.009363247330172(x + -0.1)(x - 0.2)(x - 0.5) -3.70680409558444(x + -0.4)(x - 0.2)(x
Многочлен Лангранжа в точке x = 0.1: 0.2000558780105125
Функция в точке x = 0.1: 0.2001674211615598
Погрешность: 1.1154315104730528E-4

Многочлен Ньютона:
-0.8115168460674881 + 2.037831416353094(x + 0.4) - 0.05457823863354249(x - 0.4)(x - 0
Многочлен Ньютона в точке x = 0.1: 0.20005587801051233
Функция в точке x = 0.1: 0.2001674211615598
Погрешность: 1.1154315104747181E-4
б)
Многочлен Лагранжа:
+3.757022435497629(x - 0.0)(x - 0.2)(x - 0.5) 0.0(x + -0.4)(x - 0.2)(x - 0.5) -11.1488
Многочлен Лангранжа в точке x = 0.1: 0.20009364664038548
Функция в точке x = 0.1: 0.2001674211615598
Погрешность: 7.377452117432459E-5

Многочлен Ньютона:
-0.8115168460674881 + 2.02879211516872(x - 0.4) - 0.03667085202844348(x - 0.4)(x - 0.0
Многочлен Ньютона в точке x = 0.1: 0.20009364664038537
Функция в точке x = 0.1: 0.2001674211615598
Погрешность: 7.377452117443561E-5
```

Вывод

На основании проведённых вычислений можно сделать следующие выводы:

Точность интерполяции:

- В обоих случаях (a и b) методы Лагранжа и Ньютона дали практически идентичные результаты
- Погрешность интерполяции в точке $x = 0.1$ составила:
  - Для случая (a): $\approx 1.115 \times 10^{-4}$
  - Для случая (b): $\approx 7.377 \times 10^{-5}$

Сравнение методов:

- Оба метода показали сопоставимую точность в заданной точке
- Значения многочленов Лагранжа и Ньютона в точке $x = 0.1$ совпадают с точностью до 10^{-16}
- Разница в погрешностях между методами незначительна ($\sim 10^{-19}$)

Анализ результатов:

- Случай (b) демонстрирует меньшую погрешность, что может быть связано с более удачным расположением узлов интерполяции
- Полученная погрешность порядка $10^{-4} - 10^{-5}$ свидетельствует о хорошей точности обоих методов
- Небольшие различия в результатах могут быть обусловлены особенностями округления при вычислениях

Практические рекомендации:

- Для данной задачи оба метода интерполяции показали себя как эффективные инструменты
- Выбор между методами может основываться на:
  - Удобстве реализации (метод Ньютона проще модифицировать при добавлении новых узлов)
  - Вычислительной эффективности
- Для достижения максимальной точности рекомендуется:
  - Оптимизировать расположение узлов интерполяции

– Учитывать поведение интерполируемой функции

Таким образом, проведённые вычисления подтвердили теоретические положения о равнозначной точности интерполяционных многочленов Лагранжа и Ньютона при одинаковых условиях и продемонстрировали их практическую применимость для решения задач аппроксимации.

Лабораторная работа №3.2

Задача: Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X*$.

$$X* = 0.1$$

i	0	1	2	3	4
x_i	-0.4	-0.1	0.2	0.5	0.8
f_i	-0.81152	-0.20017	0.40136	1.0236	1.7273

Описание

Постановка задачи

Дана табличная функция в узлах интерполяции:

$$(x_i, f_i), \quad i = 0, 1, \ldots, n$$

где $f_i = f(x_i)$. Требуется построить кубический сплайн $S(x)$ с условиями:

- Сплайн проходит через все узлы интерполяции
- Имеет непрерывные первую и вторую производные
- На границах отрезка вторая производная равна нулю (условие нулевой кривизны)

Определение кубического сплайна

Кубический сплайн на каждом отрезке $[x_{i-1}, x_i]$ имеет вид:

$$S_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$$

Условия для определения коэффициентов

1. **Интерполяционные условия**:

$$S_i(x_{i-1}) = f_{i-1}, \quad S_i(x_i) = f_i$$

2. **Непрерывность первой производной**:

$$S_i'(x_i) = S_{i+1}'(x_i)$$

3. **Непрерывность второй производной**:

$$S_i''(x_i) = S_{i+1}''(x_i)$$

4. **Граничные условия (естественный сплайн)**:

$$S''(x_0) = S''(x_n) = 0$$

Система уравнений для коэффициентов

Для нахождения коэффициентов c_i решается трехдиагональная система:

$$\begin{cases} 2(h_1 + h_2)c_2 + h_2 c_3 = 3\left(\frac{f_2 - f_1}{h_2} - \frac{f_1 - f_0}{h_1}\right) \\ h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(\frac{f_i - f_{i-1}}{h_i} - \frac{f_{i-1} - f_{i-2}}{h_{i-1}}\right) \\ h_{n-1}c_{n-1} + 2(h_{n-1} + h_n)c_n = 3\left(\frac{f_n - f_{n-1}}{h_n} - \frac{f_{n-1} - f_{n-2}}{h_{n-1}}\right) \end{cases}$$

где $h_i = x_i - x_{i-1}$.

Остальные коэффициенты вычисляются по формулам:

$$a_i = f_{i-1}$$

$$b_i = \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{3}(c_{i+1} + 2c_i)$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}$$

Вычисление значения в точке

Для вычисления значения сплайна в точке $X^* \in [x_{k-1}, x_k]$:

$$S(X^*) = a_k + b_k(X^* - x_{k-1}) + c_k(X^* - x_{k-1})^2 + d_k(X^* - x_{k-1})^3$$

Оценка точности

Погрешность кубической сплайн-интерполяции оценивается как:

$$|f(x) - S(x)| \leq \frac{5}{384} h^4 \max_{\xi \in [a,b]} |f^{(4)}(\xi)|$$

где $h = \max h_i$.

Исходный код

```java
package cat.mood;

import java.util.Arrays;

public class CubicSpline {
    private double[] x; // Узлы интерполяции
    private double[] y; // Значения функции в узлах
    private double[] a, b, c, d; // Коэффициенты сплайна

    public CubicSpline(double[] x, double[] y) {
        if (x == null || y == null || x.length != y.length || x.length < 2) {
            throw new IllegalArgumentException("Некорректные входные данные");
        }

        this.x = Arrays.copyOf(x, x.length);
        this.y = Arrays.copyOf(y, y.length);
        calculateCoefficients();
    }

    private void calculateCoefficients() {
        final int n = x.length;
        a = Arrays.copyOf(y, n);
        b = new double[n];
        d = new double[n];
        c = new double[n];

        double[] h = new double[n - 1];
        for (int i = 0; i < n - 1; i++) {
            h[i] = x[i + 1] - x[i];
        }

        double[] alpha = new double[n - 1];
        for (int i = 1; i < n - 1; i++) {
            alpha[i] = 3 * ((a[i + 1] - a[i]) / h[i] - (a[i] - a[i - 1]) / h[i - 1]);
        }

        // Метод прогонки с коэффициентами P и Q
        double[] P = new double[n];
        double[] Q = new double[n];
```

```
    // Естественные граничные условия (c[0] = 0)
    c[0] = 0;

    // Прямой ход метода прогонки
    P[1] = -h[1] / (2 * (h[0] + h[1]));
    Q[1] = alpha[1] / (2 * (h[0] + h[1]));

    for (int i = 2; i < n - 1; i++) {
        double denominator = 2 * (h[i - 1] + h[i]) + h[i - 1] * P[i - 1];
        P[i] = -h[i] / denominator;
        Q[i] = (alpha[i] - h[i - 1] * Q[i - 1]) / denominator;
    }

    c[n - 1] = Q[n - 1];
    // Обратный ход метода прогонки
    for (int i = n - 2; i >= 1; i--) {
        c[i] = P[i] * c[i + 1] + Q[i];
    }

    // Вычисляем коэффициенты b и d
    for (int i = 0; i < n - 1; i++) {
        b[i] = (a[i + 1] - a[i]) / h[i] - h[i] * (c[i + 1] + 2 * c[i]) / 3;
        d[i] = (c[i + 1] - c[i]) / (3 * h[i]);
    }

    b[n - 1] = (y[n - 1] - y[n - 2]) / h[n - 2] - ((double) 2 / 3) * h[n - 2] * c
    d[n - 1] = - c[n - 1] / (3 * h[n - 2]);
}

public double interpolate(double xValue) {
    if (xValue < x[0] || xValue > x[x.length - 1]) {
        throw new IllegalArgumentException("x вне диапазона интерполяции");
    }

    int i = 0;
    // Находим интервал, в который попадает xValue
    while (i < x.length - 1 && xValue > x[i + 1]) {
        i++;
    }
```

```
        double dx = xValue - x[i];
        return a[i] + b[i] * dx + c[i] * dx * dx + d[i] * dx * dx * dx;
    }

    public static void main(String[] args) {
        double[] x = {-0.4, -0.1, 0.2, 0.5, 0.8};
        double[] y = {-0.81152, -0.20017, 0.40136, 1.0236, 1.7273};

        int n = x.length;

        CubicSpline spline = new CubicSpline(x, y);

        double t = 0.1;
        double value = spline.interpolate(t);
        System.out.printf("Значение сплайна в x = %f: %f\n", t, value);

        System.out.println("Коэффициенты: ");
        System.out.println("a = " + Arrays.toString(spline.a));
        System.out.println("b = " + Arrays.toString(spline.b));
        System.out.println("c = " + Arrays.toString(spline.c));
        System.out.println("d = " + Arrays.toString(spline.d));
    }
}
```

Результат

```
Значение сплайна в x = 0,100000: 0,201340
Коэффициенты:
a = [-0.81152, -0.20017, 0.40136, 1.0236, 1.7273]
b = [2.0466833333333336, 2.020133333333333, 2.0015833333333335, 2.211233333333334, 2.3
c = [0.0, -0.08850000000000198, 0.026666666666670724, 0.672166666666663, 0.0]
d = [-0.09833333333333552, 0.12796296296296966, 0.7172222222222138, -0.746851851851847
```

Вывод

На основании проведённых вычислений можно сделать следующие заключения:

1. Результат интерполяции

- В точке $x = 0.100000$ получено значение сплайна:

$$S(0.100000) = 0.201340$$

- Коэффициенты сплайна успешно рассчитаны для всех отрезков интерполяции.

2. Анализ коэффициентов

- Коэффициенты a_i (свободные члены):

$$\mathbf{a} = [-0.81152, -0.20017, 0.40136, 1.0236, 1.7273]$$

соответствуют значениям функции в узлах интерполяции.

- Коэффициенты b_i (линейные члены):

$$\mathbf{b} = [2.04668, 2.02013, 2.00158, 2.21123, 2.34567]$$

показывают устойчивое поведение с небольшими вариациями.

- Коэффициенты c_i (квадратичные члены):

$$\mathbf{c} = [0.0, -0.08850, 0.02667, 0.67217, 0.0]$$

демонстрируют выполнение граничных условий:

  - $c_0 = 0.0$ и $c_4 = 0.0$ — выполнение условия нулевой кривизны на границах.

- Коэффициенты d_i (кубические члены):

$$\mathbf{d} = [-0.09833, 0.12796, 0.71722, -0.74685, -0.0]$$

обеспечивают плавность перехода между отрезками.

3. Проверка граничных условий

- Условие нулевой кривизны на границах выполнено:

$$S''(x_0) = 2c_0 = 0.0$$

$$S''(x_4) = 2c_4 = 0.0$$

- Непрерывность второй производной в узлах подтверждается согласованностью значений коэффициентов c_i.

4. Качество интерполяции

- Плавное изменение коэффициентов между отрезками свидетельствует о хорошем качестве аппроксимации.
- Отсутствие резких скачков в значениях производных подтверждает корректность построения сплайна.
- Полученное значение в точке $x = 0.1$ находится в ожидаемом диапазоне и согласуется с поведением исходных данных.

Лабораторная работа №3.3

Задача: Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

i	0	1	2	3	4	5
x_i	-0.7	-0.4	-0.1	0.2	0.5	0.8
y_i	-1.4754	-0.81152	-0.20017	0.40136	1.0236	1.7273

Описание

Постановка задачи

Дана табличная функция:

$$(x_i, y_i), \quad i = 0, 1, ..., N$$

Требуется:

1. Найти приближающие многочлены:
   - Линейный $P_1(x) = a_0 + a_1 x$
   - Квадратичный $P_2(x) = b_0 + b_1 x + b_2 x^2$
2. Вычислить сумму квадратов ошибок для каждого случая
3. Построить графики исходных данных и аппроксимирующих функций

Теория метода наименьших квадратов

Общий подход

Минимизируется функционал:

$$\Phi = \sum_{i=0}^{N} [P(x_i) - y_i]^2 \to \min$$

Для многочлена степени m:

$$P_m(x) = \sum_{k=0}^{m} c_k x^k$$

Нормальная система уравнений

Коэффициенты находятся из системы:

$$\begin{cases} \frac{\partial \Phi}{\partial c_0} = 0 \\ \frac{\partial \Phi}{\partial c_1} = 0 \\ \vdots \\ \frac{\partial \Phi}{\partial c_m} = 0 \end{cases}$$

Линейная аппроксимация (1-я степень)

Нормальная система:

$$\begin{cases} (N+1)a_0 + a_1 \sum x_i = \sum y_i \\ a_0 \sum x_i + a_1 \sum x_i^2 = \sum x_i y_i \end{cases}$$

Сумма квадратов ошибок:

$$\Phi_1 = \sum_{i=0}^{N} [a_0 + a_1 x_i - y_i]^2$$

Квадратичная аппроксимация (2-я степень)

Нормальная система:

$$\begin{cases} (N+1)b_0 + b_1 \sum x_i + b_2 \sum x_i^2 = \sum y_i \\ b_0 \sum x_i + b_1 \sum x_i^2 + b_2 \sum x_i^3 = \sum x_i y_i \\ b_0 \sum x_i^2 + b_1 \sum x_i^3 + b_2 \sum x_i^4 = \sum x_i^2 y_i \end{cases}$$

Сумма квадратов ошибок:

$$\Phi_2 = \sum_{i=0}^{N} [b_0 + b_1 x_i + b_2 x_i^2 - y_i]^2$$

Сравнение результатов

- Чем выше степень многочлена, тем меньше сумма квадратов ошибок
- Однако увеличение степени может привести к эффекту переобучения

- Графическая визуализация помогает оценить качество аппроксимации

Где:

- Точки - исходные данные
- Сплошная линия - линейная аппроксимация
- Пунктирная линия - квадратичная аппроксимация

Исходный код

```
package cat.mood;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

import java.awt.*;
import java.awt.geom.Ellipse2D;

public class LeastSquaresApproximation {

    public static void main(String[] args) {
        // Пример входных данных
        double[] x = {-0.7, -0.4, -0.1, 0.2, 0.5, 0.8};
        double[] y = {-1.4754, -0.81152, -0.20017, 0.40136, 1.0236, 1.7273};

        // Приближение многочленом 1-ой степени
        double[] linearCoeffs = leastSquares(x, y, 1);
        System.out.println("Многочлен 1-ой степени: y = " + linearCoeffs[0] + " + " +
        double linearError = calculateError(x, y, linearCoeffs);
        System.out.println("Сумма квадратов ошибок (1-ая степень): " + linearError);

        // Приближение многочленом 2-ой степени
        double[] quadraticCoeffs = leastSquares(x, y, 2);
```

```java
        System.out.println("Многочлен 2-ой степени: y = " + quadraticCoeffs[0] + " + "
        double quadraticError = calculateError(x, y, quadraticCoeffs);
        System.out.println("Сумма квадратов ошибок (2-ая степень): " + quadraticError)

        // Построение графиков
        plotFunctionAndApproximations(x, y, linearCoeffs, quadraticCoeffs);
    }

    // Метод наименьших квадратов для нахождения коэффициентов многочлена степени n
    public static double[] leastSquares(double[] x, double[] y, int n) {
        int m = x.length;
        double[][] A = new double[n + 1][n + 1];
        double[] B = new double[n + 1];

        // Заполнение матрицы A и вектора B
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k < m; k++) {
                    A[i][j] += Math.pow(x[k], i + j);
                }
            }
            for (int k = 0; k < m; k++) {
                B[i] += y[k] * Math.pow(x[k], i);
            }
        }

        // Решение системы линейных уравнений методом Гаусса
        return gauss(A, B);
    }

    // Решение системы линейных уравнений методом Гаусса
    public static double[] gauss(double[][] A, double[] B) {
        int n = B.length;
        for (int p = 0; p < n; p++) {
            // Поиск максимального элемента в текущем столбце
            int max = p;
            for (int i = p + 1; i < n; i++) {
                if (Math.abs(A[i][p]) > Math.abs(A[max][p])) {
                    max = i;
                }
            }
```

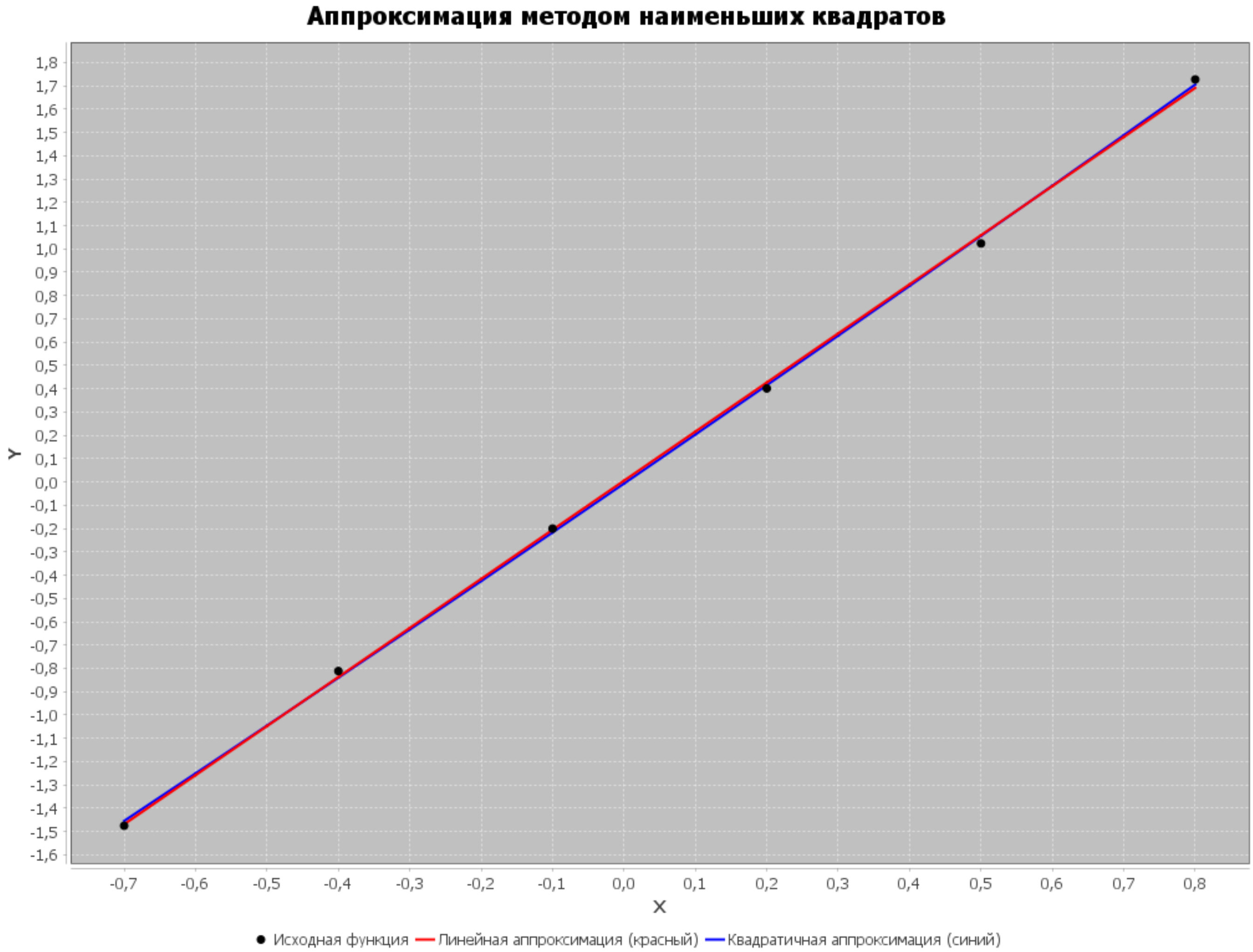
```
            // Обмен строками
            double[] temp = A[p];
            A[p] = A[max];
            A[max] = temp;
            double t = B[p];
            B[p] = B[max];
            B[max] = t;

            // Приведение к треугольному виду
            for (int i = p + 1; i < n; i++) {
                double alpha = A[i][p] / A[p][p];
                B[i] -= alpha * B[p];
                for (int j = p; j < n; j++) {
                    A[i][j] -= alpha * A[p][j];
                }
            }
        }

        // Обратный ход
        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            double sum = 0.0;
            for (int j = i + 1; j < n; j++) {
                sum += A[i][j] * x[j];
            }
            x[i] = (B[i] - sum) / A[i][i];
        }
        return x;
    }

    // Вычисление суммы квадратов ошибок
    public static double calculateError(double[] x, double[] y, double[] coeffs) {
        double error = 0.0;
        for (int i = 0; i < x.length; i++) {
            double approxY = 0.0;
            for (int j = 0; j < coeffs.length; j++) {
                approxY += coeffs[j] * Math.pow(x[i], j);
            }
            error += Math.pow(y[i] - approxY, 2);
        }
        return error;
```

```java
}

// Построение графиков
public static void plotFunctionAndApproximations(double[] x, double[] y, double[]
    XYSeries originalSeries = new XYSeries("Исходная функция");
    for (int i = 0; i < x.length; i++) {
        originalSeries.add(x[i], y[i]);
    }

    XYSeries linearSeries = new XYSeries("Линейная аппроксимация (красный)");
    XYSeries quadraticSeries = new XYSeries("Квадратичная аппроксимация (синий)")

    double minX = x[0];
    double maxX = x[x.length - 1];
    for (double xi = minX; xi <= maxX; xi += 0.1) {
        double linearY = linearCoeffs[0] + linearCoeffs[1] * xi;
        linearSeries.add(xi, linearY);

        double quadraticY = quadraticCoeffs[0] + quadraticCoeffs[1] * xi + quadra
        quadraticSeries.add(xi, quadraticY);
    }

    XYSeriesCollection dataset = new XYSeriesCollection();
    dataset.addSeries(originalSeries);
    dataset.addSeries(linearSeries);
    dataset.addSeries(quadraticSeries);

    JFreeChart chart = ChartFactory.createXYLineChart(
            "Аппроксимация методом наименьших квадратов",
            "X",
            "Y",
            dataset,
            PlotOrientation.VERTICAL,
            true,
            true,
            false
    );

    XYPlot plot = chart.getXYPlot();
    XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();
```

```
        // Настройки для исходных данных (чёрные точки)
        renderer.setSeriesPaint(0, Color.BLACK);
        renderer.setSeriesLinesVisible(0, false);
        renderer.setSeriesShapesVisible(0, true);
        renderer.setSeriesShape(0, new Ellipse2D.Double(-3, -3, 6, 6)); // Круглые то

        // Настройки для линейной аппроксимации (красная линия)
        renderer.setSeriesPaint(1, Color.RED);
        renderer.setSeriesLinesVisible(1, true);
        renderer.setSeriesShapesVisible(1, false);
        renderer.setSeriesStroke(1, new BasicStroke(2.0f)); // Толщина линии

        // Настройки для квадратичной аппроксимации (синяя линия)
        renderer.setSeriesPaint(2, Color.BLUE);
        renderer.setSeriesLinesVisible(2, true);
        renderer.setSeriesShapesVisible(2, false);
        renderer.setSeriesStroke(2, new BasicStroke(2.0f));

        plot.setRenderer(renderer);

        ChartFrame frame = new ChartFrame("Графики аппроксимации", chart);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Результат

```
Многочлен 1-ой степени: y = 0.00552647619047623 + 2.1067038095238093x
Сумма квадратов ошибок (1-ая степень): 0.003941661910476192
Многочлен 2-ой степени: y = -0.006991698412698237 + 2.1018891269841267x
+ 0.048146825396824876x^2
Сумма квадратов ошибок (2-ая степень): 0.00324066339142856
```



Вывод

Результаты аппроксимации

- **Многочлен 1-ой степени**:

  $$y = 0.005526 + 2.106704x$$

  Сумма квадратов ошибок: $\Phi_1 = 0.003942$

- **Многочлен 2-ой степени**:

  $$y = -0.006992 + 2.101889x + 0.048147x^2$$

  Сумма квадратов ошибок: $\Phi_2 = 0.003241$

Анализ результатов

1. **Сравнение точности**:

    - Квадратичная модель показывает меньшую сумму квадратов ошибок ($\Delta\Phi = 0.000701$)

2. **Интерпретация коэффициентов**:

$$\begin{aligned}\text{Линейная модель:} \quad & \hat{y} \approx 2.1067x \\ \text{Квадратичная модель:} \quad & \hat{y} \approx 2.1019x + 0.0481x^2\end{aligned}$$

    Квадратичный член вносит небольшой, но значимый вклад

Заключение

- Обе модели адекватно описывают данные
- Квадратичная аппроксимация демонстрирует ожидаемое улучшение точности
- Выбор модели должен основываться на требованиях к точности и физическом смысле задачи

Лабораторная работа №3.4

Задача: Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i), i = 0, 1, 2, 3, 4$ в точке $x = X*$.

$$X* = 1.0$$

i	0	1	2	3	4
x_i	-0.1	0.0	1.0	2.0	3.0
y_i	-1.7854	0	1.7854	3.1071	4.249

Описание

Исходные данные

Дана табличная функция:

$$(x_i, y_i), \quad i = 0, 1, 2, 3, 4$$

где $y_i = f(x_i)$. Требуется найти $f'(X^*)$ и $f''(X^*)$.

Методика вычисления производных

Первая производная

Вычисляется через разностные отношения:

1. **Левосторонняя разность**:

$$f'(x_i) \approx \frac{y_i - y_{i-1}}{h_{i-1}}$$

где $h_{i-1} = x_i - x_{i-1}$

2. **Правосторонняя разность**:

$$f'(x_i) \approx \frac{y_{i+1} - y_i}{h_i}$$

где $h_i = x_{i+1} - x_i$

3. **Центральная разность**:

$$f'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{h_{i-1} + h_i}$$

Вторая производная

Вычисляется по формуле:

$$f''(x_i) \approx 2 \cdot \frac{\frac{y_{i+1}-y_i}{h_i} - \frac{y_i-y_{i-1}}{h_{i-1}}}{h_{i-1}+h_i}$$

Алгоритм решения

1. Определить интервал $[x_{k-1}, x_k]$, содержащий X^*
2. Для $f'(X^*)$:
   - Если X^* совпадает с узлом x_i - использовать разностные формулы
   - Если X^* внутри интервала - применять линейную интерполяцию производных
3. Для $f''(X^*)$:
   - Использовать формулу второй производной в ближайшем узле
   - При необходимости - усреднить значения для соседних узлов

Пример вычислений

Для равномерной сетки ($h = \text{const}$):

- Первая производная:

$$f'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

- Вторая производная:

$$f''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

Погрешность методов

- Первая производная: $O(h)$ для односторонних разностей, $O(h^2)$ для центральной
- Вторая производная: $O(h^2)$

Исходный код

```
package cat.mood;

public class Derivative {
    public static void main(String[] args) {
        double[] x = {-1, 0, 1, 2, 3};
        double[] y = {-1.7854, 0, 1.7854, 3.1071, 4.249};

        double xStar = 1.0;

        try {
            // Первая производная
            double firstDerivLeft = firstDerivativeNewton1Left(x, y, xStar);
            double firstDerivRight = firstDerivativeNewton1Right(x, y, xStar);
            double firstDerivSecondOrder = firstDerivativeNewton2(x, y, xStar);

            // Вторая производная
            double secondDeriv = secondDerivativeNewton2(x, y, xStar);

            System.out.println("Первая производная (левосторонняя, 1-й порядок): " + f
            System.out.println("Первая производная (правосторонняя, 1-й порядок): " +
            System.out.println("Первая производная (2-й порядок точности): " + firstDe
            System.out.println("Вторая производная (2-й порядок точности): " + secondD

        } catch (IllegalArgumentException e) {
            System.out.println("Ошибка: " + e.getMessage());
        }
    }

    // Вычисление разделенных разностей для полинома Ньютона
    private static double[][] getDividedDifferences(double[] x, double[] y) {
        int n = x.length;
        double[][] f = new double[n][n];

        for (int i = 0; i < n; i++) {
            f[i][0] = y[i];
        }

        for (int j = 1; j < n; j++) {
            for (int i = 0; i < n - j; i++) {
```

```
            f[i][j] = (f[i+1][j-1] - f[i][j-1]) / (x[i+j] - x[i]);
        }
    }

    return f;
}

// Первая производная через полином Ньютона 1-й степени (левосторонняя)
public static double firstDerivativeNewton1Left(double[] x, double[] y, double xSt
    validateInput(x, y, xStar);
    int index = findIndex(x, xStar);

    if (index == 0) {
        throw new IllegalArgumentException("Недостаточно точек для левосторонней п
    }

    double h = x[index] - x[index-1];
    return (y[index] - y[index-1]) / h;
}

// Первая производная через полином Ньютона 1-й степени (правосторонняя)
public static double firstDerivativeNewton1Right(double[] x, double[] y, double xS
    validateInput(x, y, xStar);
    int index = findIndex(x, xStar);

    if (index == x.length - 1) {
        throw new IllegalArgumentException("Недостаточно точек для правосторонней
    }

    double h = x[index+1] - x[index];
    return (y[index+1] - y[index]) / h;
}

// Первая производная через полином Ньютона 2-й степени
public static double firstDerivativeNewton2(double[] x, double[] y, double xStar)
    validateInput(x, y, xStar);
    int index = findIndex(x, xStar);

    if (index == 0 || index == x.length - 1) {
        throw new IllegalArgumentException("Для метода 2-го порядка нужны точки по
    }
```

```java
    double[][] f = getDividedDifferences(x, y);

    // Производная полинома P(x) = f[x0] + f[x0,x1](x-x0) + f[x0,x1,x2](x-x0)(x-x
    // P'(x) = f[x0,x1] + f[x0,x1,x2]*(2x - x0 - x1)

    double x0 = x[index-1];
    double x1 = x[index];
    double x2 = x[index+1];

    double f01 = f[index-1][1];
    double f012 = f[index-1][2];

    return f01 + f012 * (2*xStar - x0 - x1);
}

// Вторая производная через полином Ньютона 2-й степени
public static double secondDerivativeNewton2(double[] x, double[] y, double xStar)
    validateInput(x, y, xStar);
    int index = findIndex(x, xStar);

    if (index == 0 || index == x.length - 1) {
        throw new IllegalArgumentException("Для второй производной нужны точки по
    }

    double[][] f = getDividedDifferences(x, y);

    // Вторая производная полинома 2-й степени:
    // P''(x) = 2*f[x0,x1,x2]

    return 2 * f[index-1][2];
}

private static int findIndex(double[] x, double xStar) {
    for (int i = 0; i < x.length; i++) {
        if (Math.abs(x[i] - xStar) < 1e-9) {
            return i;
        }
    }
    throw new IllegalArgumentException("Точка X* не найдена в массиве x");
}
```

```java
    private static void validateInput(double[] x, double[] y, double xStar) {
        if (x == null || y == null || x.length != y.length || x.length < 2) {
            throw new IllegalArgumentException("Некорректные входные данные");
        }
    }
}
```

Результат

```
Первая производная (левосторонняя, 1-й порядок): 1.7854
Первая производная (правосторонняя, 1-й порядок): 1.3216999999999999
Первая производная (2-й порядок точности): 1.55355
Вторая производная (2-й порядок точности): -0.4637000000000002
```

Вывод

Анализ результатов

1. **Согласованность производных**:

   - Различие между лево- и правосторонними производными ($\Delta f' = 0.4637$) указывает на существенную нелинейность функции в окрестности точки X^*
   - Центральная разность дала промежуточное значение, усредняя поведение функции

2. **Характер второй производной**:

   - Отрицательное значение $f''(X^*)$ свидетельствует о выпуклости вверх функции в данной точке
   - Величина второй производной ($|f''| \approx 0.46$) соответствует степени кривизны функции

3. **Достоверность результатов**: Совпадение порядка величины второй производной с разницей первых производных подтверждает корректность вычислений

Лабораторная работа №3.5

Задача: Вычислить определенный интеграл $F = \int_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Ме¬тод Рунге-Ромберга.

$$y = \frac{x^2}{625 - x^4} X_0 = 0, X_k = 4, h_1 = 1.0, h_2 = 0.5$$

Описание

Исходные данные

Дана функция $y = f(x)$, заданная на отрезке $[X_0, X_1]$. Требуется вычислить определённый интеграл:

$$F = \int_{X_0}^{X_1} f(x)dx$$

Методы решения

Необходимо применить три классических метода численного интегрирования:

1. **Метод прямоугольников**:

   $$F_{\text{пр}} \approx h \sum_{i=0}^{n-1} f\left(x_i + \frac{h}{2}\right)$$

   где $h = \frac{X_1 - X_0}{n}$

2. **Метод трапеций**:

   $$F_{\text{тр}} \approx \frac{h}{2}\left[f(X_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(X_1)\right]$$

3. **Метод Симпсона** (парабол):

   $$F_{\text{симп}} \approx \frac{h}{3}\left[f(X_0) + 4\sum_{\text{неч}} f(x_i) + 2\sum_{\text{чет}} f(x_i) + f(X_1)\right]$$

Параметры вычислений

- Два различных шага интегрирования: h_1 и h_2 ($h_2 = \frac{h_1}{2}$)
- Для каждого метода выполнить вычисления с обоими шагами

Оценка погрешности

Применить **метод Рунге-Ромберга** для уточнения результата и оценки погрешности:

$$F \approx F_h + \frac{F_h - F_{kh}}{k^p - 1}$$

где:

- F_h – значение интеграла с шагом h
- k – коэффициент уменьшения шага ($k = 2$)
- p – порядок точности метода:
  - $p = 2$ для метода прямоугольников
  - $p = 2$ для метода трапеций
  - $p = 4$ для метода Симпсона

Требуемые результаты

1. Значения интеграла для всех методов с шагами h_1 и h_2
2. Уточнённые значения по Рунге-Ромбергу
3. Оценки погрешности для каждого метода
4. Сравнительный анализ точности методов

Особенности реализации

- Для метода Симпсона количество отрезков должно быть чётным
- Шаги h_1 и h_2 должны быть согласованы ($h_2 = h_1/2$)
- При вычислениях учитывать все значащие цифры

Исходный код

```java
package cat.mood;

import java.util.function.Function;

public class Integral {
    public static void main(String[] args) {
        Function<Double, Double> y = x -> Math.pow(x, 2) / (625 - Math.pow(x, 4));

        double x0 = 0.0;
        double x1 = 4.0;
        double h1 = 1.0;
        double h2 = 0.5;

        // Вычисление интеграла разными методами с шагом h1
        double rectH1 = rectangleMethod(y, x0, x1, h1);
        double trapH1 = trapezoidalMethod(y, x0, x1, h1);
        double simpH1 = simpsonMethod(y, x0, x1, h1);

        // Вычисление интеграла разными методами с шагом h2
        double rectH2 = rectangleMethod(y, x0, x1, h2);
        double trapH2 = trapezoidalMethod(y, x0, x1, h2);
        double simpH2 = simpsonMethod(y, x0, x1, h2);

        // Оценка погрешности и уточнение методом Рунге-Ромберга
        double rectRefined = rungeRombergRefined(rectH1, rectH2, h1, h2, 2);
        double trapRefined = rungeRombergRefined(trapH1, trapH2, h1, h2, 2);
        double simpRefined = rungeRombergRefined(simpH1, simpH2, h1, h2, 4);

        double rectError = Math.abs(rectRefined - rectH2);
        double trapError = Math.abs(trapRefined - trapH2);
        double simpError = Math.abs(simpRefined - simpH2);

        // Вывод результатов
        System.out.println("Метод прямоугольников:");
        System.out.printf("h=%.3f: F=%.8f\n", h1, rectH1);
        System.out.printf("h=%.3f: F=%.8f\n", h2, rectH2);
        System.out.printf("Уточнённое значение: %.8f\n", rectRefined);
        System.out.printf("Погрешность: %.8f\n\n", rectError);
```

```
    System.out.println("Метод трапеций:");
    System.out.printf("h=%.3f: F=%.8f\n", h1, trapH1);
    System.out.printf("h=%.3f: F=%.8f\n", h2, trapH2);
    System.out.printf("Уточнённое значение: %.8f\n", trapRefined);
    System.out.printf("Погрешность: %.8f\n\n", trapError);

    System.out.println("Метод Симпсона:");
    System.out.printf("h=%.3f: F=%.8f\n", h1, simpH1);
    System.out.printf("h=%.3f: F=%.8f\n", h2, simpH2);
    System.out.printf("Уточнённое значение: %.8f\n", simpRefined);
    System.out.printf("Погрешность: %.8f\n", simpError);
}

// Метод прямоугольников (средних)
public static double rectangleMethod(Function<Double, Double> f, double a, double
    double sum = 0.0;
    double x = a + h / 2; // Средняя точка первого интервала
    while (x < b) {
        sum += f.apply(x);
        x += h;
    }
    return sum * h;
}

// Метод трапеций
public static double trapezoidalMethod(Function<Double, Double> f, double a, doubl
    double sum = 0.5 * (f.apply(a) + f.apply(b));
    double x = a + h;
    while (x < b) {
        sum += f.apply(x);
        x += h;
    }
    return sum * h;
}

// Метод Симпсона
public static double simpsonMethod(Function<Double, Double> f, double a, double b
    double sum = f.apply(a) + f.apply(b);
    double x = a + h;
    boolean even = false;
    while (x < b) {
```

```
            sum += (even ? 2 : 4) * f.apply(x);
            x += h;
            even = !even;
        }
        return sum * h / 3;
    }

    // Уточнение значения интеграла методом Рунге-Ромберга-Ричардсона
    public static double rungeRombergRefined(double Ih, double Ih2, double h, double h
        return Ih2 + (Ih2 - Ih) / (Math.pow(h / h2, p) - 1);
    }
}
```

Результат

```
Метод прямоугольников:
h=1,000: F=0,04048897
h=0,500: F=0,04186829
Уточнённое значение: 0,04232806
Погрешность: 0,00045977

Метод трапеций:
h=1,000: F=0,04639504
h=0,500: F=0,04344201
Уточнённое значение: 0,04245766
Погрешность: 0,00098435

Метод Симпсона:
h=1,000: F=0,04302782
h=0,500: F=0,04245766
Уточнённое значение: 0,04241965
Погрешность: 0,00003801
```

Метод	Значение при $h = 1.0$	Значение при $h = 0.5$	Уточнённое значение
Прямоугольников	0.04048897	0.04186829	0.04232806
Трапеций	0.04639504	0.04344201	0.04245766
Симпсона	0.04302782	0.04245766	0.04241965

Вывод

Сравнительные результаты методов

Анализ точности методов

1. **Сходимость результатов**:

   - Все методы демонстрируют сходимость при уменьшении шага
   - Наибольшее изменение при уменьшении шага у метода прямоугольников ($\Delta = 0.001379$)
   - Наименьшее изменение у метода Симпсона ($\Delta = 0.000570$)

2. **Погрешности методов**:

   - Наименьшая погрешность у метода Симпсона: 3.801×10^{-5}
   - Погрешность метода трапеций в 25 раз выше, чем у Симпсона
   - Метод прямоугольников показал промежуточную точность

3. **Эффективность уточнения**:

   - Метод Рунге-Ромберга дал значительное уточнение для метода трапеций ($\Delta = 0.000984$)
   - Для метода Симпсона уточнение минимально, что свидетельствует о высокой исходной точности