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:	Наумов Г.К.
Преподаватель:	Пивоваров Д.Е.
Группа:	М8О-303Б-21
Дата:	
Оценка:	
Подпись:	

Москва, 2024

3.1

1 Постановка задачи

Используя таблицу значений Y_i 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^*.

Вариант: 16

16. $y = \ln(x) + x$, а) $X_i = 0.1, 0.5, 0.9, 1.3$; б) $X_i = 0.1, 0.5, 1.1, 1.3$; $X^* = 0.8$.

2 Результаты работы

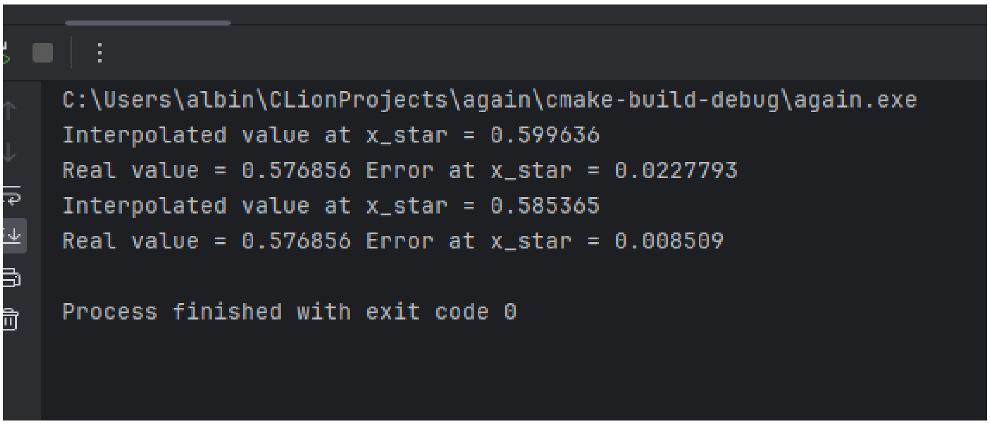

Рис. 1: Вывод программы

3 Исходный код

```
#include <iostream>
#include <vector>
#include <cmath>

double lagrange_interpolation(const std::vector<double>& x, const std::vector<double>& y, double x_star) {
    double result = 0.0;
```

```
    for (size_t i = 0; i < x.size(); ++i) {
        double term = y[i];
        for (size_t j = 0; j < x.size(); ++j) {
            if (j != i) {
                term *= (x_star - x[j]) / (x[i] - x[j]);
            }
        }
        result += term;
    }
    return result;
}

double diff(const std::vector<double>& x, const std::vector<double>& y, size_t n) {
    if (n == 0) {
        return y[0];
    } else {
        return (diff(x, y, n - 1) - diff(x, y, n - 1)) / (x[n] - x[0]);
    }
}

double newton_interpolation(const std::vector<double>& x, const std::vector<double>& y, double x_star) {
    double result = y[0];
    double term = 1.0;
    for (size_t i = 1; i < x.size(); ++i) {
        term *= (x_star - x[i - 1]);
        result += diff(x, y, i) * term;
    }
    return result;
}

int main() {
    std::vector<double> x = {0.1, 0.5, 0.9, 1.3};
    std::vector<double> y;
    for (double xi : x) {
        y.push_back(std::log(xi) + xi);
    }
    double x_star = 0.8;

    double interpolated_value = lagrange_interpolation(x, y, x_star);
    std::cout << "Interpolated value at x_star = " << interpolated_value << std::endl;

    double true_value = std::log(x_star) + x_star;
    double error = std::abs(true_value - interpolated_value);
    std::cout << "Real value = " << true_value << " Error at x_star = " << error << std::endl;

    x = {0.1, 0.5, 1.1, 1.3};
    y.clear();
```

```
    for (double xi : x) {
        y.push_back(std::log(xi) + xi);
    }
    interpolated_value = newton_interpolation(x, y, x_star);
    std::cout << "Interpolated value at x_star = " << interpolated_value << std::endl;

    true_value = std::log(x_star) + x_star;
    error = std::abs(true_value - interpolated_value);
    std::cout << "Real value = " << true_value << " Error at x_star = " << error << std::endl;

    return 0;
}
```

3.2

4 Постановка задачи

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 $x = X^*$.

Вариант: 16

16. $X^* = 0.8$

i	0	1	2	3	4
x_i	0.1	0.5	0.9	1.3	1.7
f_i	-2.2026	-0.19315	0.79464	1.5624	2.2306

Рис. 2: Условие

5 Результаты работы

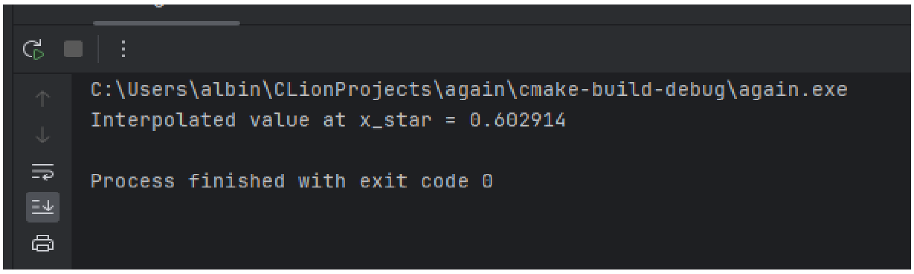

Рис. 3: Вывод программы

6 Исходный код

```
#include <iostream>
#include <vector>
#include <cmath>

void cubic_spline(const std::vector<double>& x, const std::vector<double>& f, double
    x_star) {
    size_t n = x.size();

```

```cpp
    std::vector<double> h(n - 1);
    std::vector<double> alpha(n - 1);
    std::vector<double> l(n);
    std::vector<double> mu(n - 1);
    std::vector<double> z(n);

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

    for (size_t i = 1; i < n - 1; ++i) {
        alpha[i] = 3 * (f[i + 1] - f[i]) / h[i] - 3 * (f[i] - f[i - 1]) / h[i - 1];
    }

    l[0] = 1;
    mu[0] = 0;
    z[0] = 0;

    for (size_t i = 1; i < n - 1; ++i) {
        l[i] = 2 * (x[i + 1] - x[i - 1]) - h[i - 1] * mu[i - 1];
        mu[i] = h[i] / l[i];
        z[i] = (alpha[i] - h[i - 1] * z[i - 1]) / l[i];
    }

    l[n - 1] = 1;
    z[n - 1] = 0;
    std::vector<double> c(n);
    std::vector<double> b(n - 1);
    std::vector<double> d(n - 1);

    for (int j = n - 2; j >= 0; --j) {
        c[j] = z[j] - mu[j] * c[j + 1];
        b[j] = (f[j + 1] - f[j]) / h[j] - h[j] * (c[j + 1] + 2 * c[j]) / 3;
        d[j] = (c[j + 1] - c[j]) / (3 * h[j]);
    }

    size_t interval_index = 0;
    for (size_t i = 0; i < n - 1; ++i) {
        if (x[i] <= x_star && x_star <= x[i + 1]) {
            interval_index = i;
            break;
        }
    }

    double A = f[interval_index];
    double B = b[interval_index];
    double C = c[interval_index];
    double D = d[interval_index];
```

```
    double interpolated_value = A + B * (x_star - x[interval_index]) + C * pow((x_star - x[interval_index]), 2) + D * pow((x_star - x[interval_index]), 3);

    std::cout << "Interpolated value at x_star = " << interpolated_value << std::endl;
}

int main() {
    std::vector<double> x = {0.1, 0.5, 0.9, 1.3, 1.7};
    std::vector<double> f = {-2.2026, -0.19315, 0.79464, 1.5624, 2.2306};
    double x_star = 0.8;

    cubic_spline(x, f, x_star);

    return 0;
}
```

3.3

7 Постановка задачи

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.

Вариант: 16

16.

i	0	1	2	3	4	5
x_i	0.1	0.5	0.9	1.3	1.7	2.1
y_i	-2.2026	-0.19315	0.79464	1.5624	2.2306	2.8419

Рис. 4: Условия

8 Результаты работы

```
C:\Users\albin\CLionProjects\again\cmake-build-debug\again.exe
Coefficients for degree 1 polynomial: -1.77444 2.37582
Sum of squared errors for degree 1 polynomial: 0.985412
Coefficients for degree 2 polynomial: -2.46046 4.40618 -0.922892
Sum of squared errors for degree 2 polynomial: 0.171386

Process finished with exit code 0
```

Рис. 5: Вывод программы

9 Исходный код

```
#include <iostream>
#include <vector>
#include <cmath>
```

```


void solve_system(std::vector<std::vector<double>>& A, std::vector<double>& b, std::
    vector<double>& x) {
    int n = A.size();
    for (int i = 0; i < n; ++i) {
        int pivot = i;
        for (int j = i + 1; j < n; ++j) {
            if (abs(A[j][i]) > abs(A[pivot][i])) {
                pivot = j;
            }
        }
        std::swap(A[i], A[pivot]);
        std::swap(b[i], b[pivot]);
        for (int j = i + 1; j < n; ++j) {
            double factor = A[j][i] / A[i][i];
            for (int k = i; k < n; ++k) {
                A[j][k] -= factor * A[i][k];
            }
            b[j] -= factor * b[i];
        }
    }
    x.assign(n, 0);
    for (int i = n - 1; i >= 0; --i) {
        double sum = 0;
        for (int j = i + 1; j < n; ++j) {
            sum += A[i][j] * x[j];
        }
        x[i] = (b[i] - sum) / A[i][i];
    }
}

std::vector<double> least_squares_polynomial(const std::vector<double>& x, const std::
    vector<double>& f, int degree) {
    int n = x.size();
    std::vector<std::vector<double>> A(degree + 1, std::vector<double>(degree + 1, 0));
    std::vector<double> b(degree + 1, 0);
    for (int i = 0; i <= degree; ++i) {
        for (int j = 0; j <= degree; ++j) {
            for (int k = 0; k < n; ++k) {
                A[i][j] += pow(x[k], i + j);
            }
        }
        for (int k = 0; k < n; ++k) {
            b[i] += pow(x[k], i) * f[k];
        }
    }
    std::vector<double> coefficients;
    solve_system(A, b, coefficients);
```

```
    return coefficients;
}

double evaluate_polynomial(const std::vector<double>& coefficients, double x) {
    double result = 0;
    for (size_t i = 0; i < coefficients.size(); ++i) {
        result += coefficients[i] * pow(x, i);
    }
    return result;
}

double sum_of_squared_errors(const std::vector<double>& x, const std::vector<double>&
    f, const std::vector<double>& coefficients) {
    double sum = 0;
    for (size_t i = 0; i < x.size(); ++i) {
        double error = f[i] - evaluate_polynomial(coefficients, x[i]);
        sum += error * error;
    }
    return sum;
}

int main() {
    std::vector<double> x = {0.1, 0.5, 0.9, 1.3, 1.7, 2.1};
    std::vector<double> f = {-2.2026, -0.19315, 0.79464, 1.5624, 2.2306, 2.8419};

    std::vector<double> coefficients_degree_1 = least_squares_polynomial(x, f, 1);
    double sum_of_squared_errors_degree_1 = sum_of_squared_errors(x, f,
        coefficients_degree_1);
    std::cout << "Coefficients for degree 1 polynomial: ";
    for (size_t i = 0; i < coefficients_degree_1.size(); ++i) {
        std::cout << coefficients_degree_1[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Sum of squared errors for degree 1 polynomial: " <<
        sum_of_squared_errors_degree_1 << std::endl;

    std::vector<double> coefficients_degree_2 = least_squares_polynomial(x, f, 2);
    double sum_of_squared_errors_degree_2 = sum_of_squared_errors(x, f,
        coefficients_degree_2);
    std::cout << "Coefficients for degree 2 polynomial: ";
    for (size_t i = 0; i < coefficients_degree_2.size(); ++i) {
        std::cout << coefficients_degree_2[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Sum of squared errors for degree 2 polynomial: " <<
        sum_of_squared_errors_degree_2 << std::endl;

    return 0;
}
```

3.4

10 Постановка задачи

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

Вариант: 16

16. $X^* = 2.0$

i	0	1	2	3	4
x_i	0.0	1.0	2.0	3.0	4.0
y_i	0.0	2.0	3.4142	4.7321	6.0

Рис. 6: Условия

11 Результаты работы

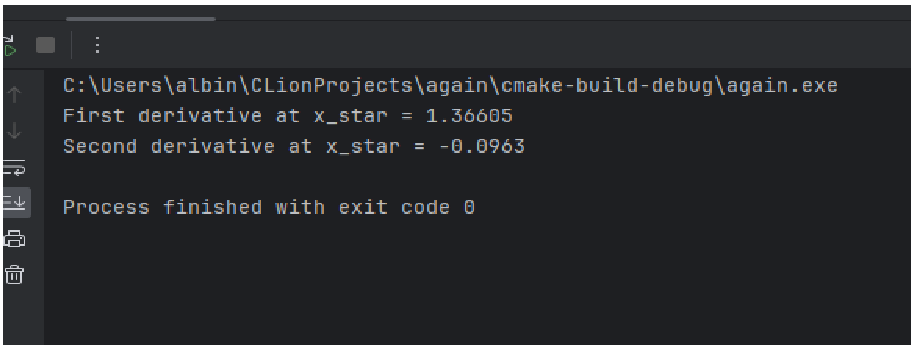

Рис. 7: Вывод программы

12 Исходный код

```
#include <iostream>
#include <vector>

double first_derivative(const std::vector<double>& x, const std::vector<double>& y,
    double x_star) {
    int idx = -1;
    for (int i = 0; i < x.size(); ++i) {
        if (x[i] <= x_star)
            idx = i;
```

```
        else
            break;
    }
    if (idx == -1 || idx == x.size() - 1) {
        std::cerr << "x_star is outside the range of the table." << std::endl;
        return 0.0;
    }
    double h = x[1] - x[0];
    double derivative = (y[idx + 1] - y[idx - 1]) / (2 * h);

    return derivative;
}

double second_derivative(const std::vector<double>& x, const std::vector<double>& y,
    double x_star) {
    int idx = -1;
    for (int i = 0; i < x.size(); ++i) {
        if (x[i] <= x_star)
            idx = i;
        else
            break;
    }

    if (idx == -1 || idx == x.size() - 1) {
        std::cerr << "x_star is outside the range of the table." << std::endl;
        return 0.0;
    }

    double h = x[1] - x[0];
    double derivative = (y[idx + 1] - 2 * y[idx] + y[idx - 1]) / (h * h);

    return derivative;
}

int main() {
    std::vector<double> x = {0.0, 1.0, 2.0, 3.0, 4.0};
    std::vector<double> y = {0.0, 2.0, 3.4142, 4.7321, 6.0};
    double x_star = 2.0;

    double first_deriv = first_derivative(x, y, x_star);
    std::cout << "First derivative at x_star = " << first_deriv << std::endl;

    double second_deriv = second_derivative(x, y, x_star);
    std::cout << "Second derivative at x_star = " << second_deriv << std::endl;

    return 0;
}
```

3.5

13 Постановка задачи

Вычислить определенный интеграл $\int\limits_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга:
Вариант: 16

$$16. \quad y = \frac{x^2}{x^4 + 256}, \qquad X_0 = 0, \quad X_k = 2, \quad h_1 = 0.5, \quad h_2 = 0.25;$$

14 Результаты работы

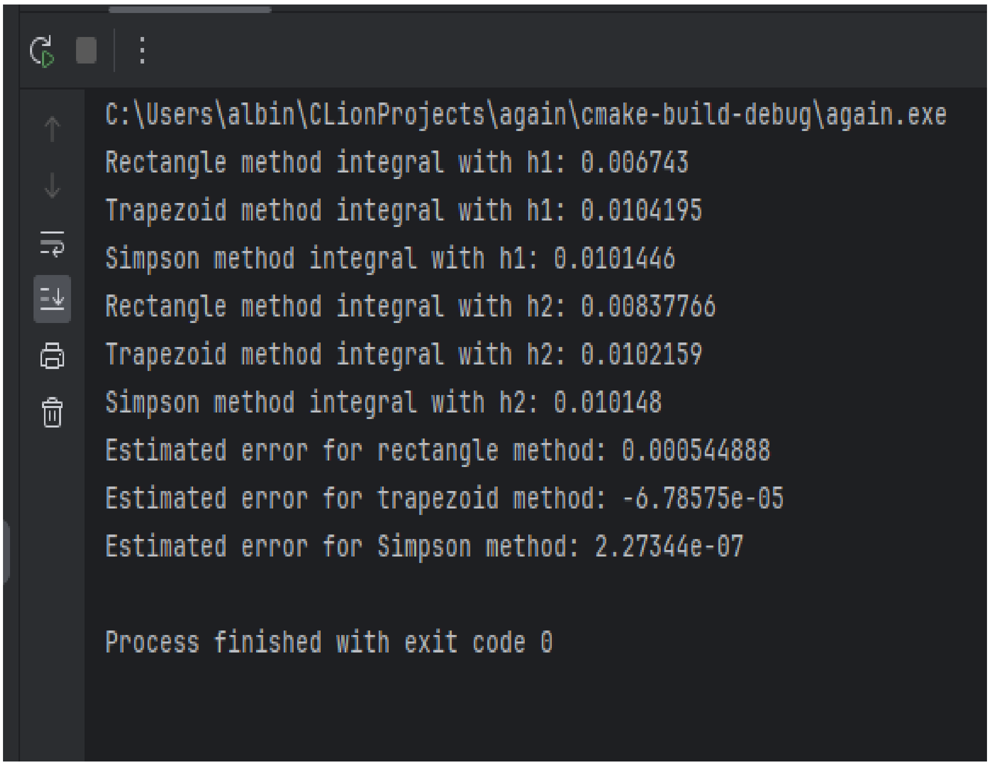

Рис. 8: Вывод программы

15 Исходный код

```
#include <iostream>
#include <cmath>

double y(double x) {
    return x * x / (pow(x, 4) + 256);
}

double rectangle_method(double x0, double x1, double h) {
    double integral = 0.0;
    for (double x = x0; x < x1; x += h) {
        integral += y(x) * h;
    }
    return integral;
}

double trapezoid_method(double x0, double x1, double h) {
    double integral = 0.0;
    for (double x = x0; x < x1; x += h) {
        integral += (y(x) + y(x + h)) * h / 2.0;
    }
    return integral;
}

double simpson_method(double x0, double x1, double h) {
    double integral = 0.0;
    for (double x = x0; x < x1; x += 2 * h) {
        integral += (y(x) + 4 * y(x + h) + y(x + 2 * h)) * h / 3.0;
    }
    return integral;
}

double runge_romberg_method(double I1, double I2, double p) {
    return (I2 - I1) / (pow(2, p) - 1);
}

int main() {
    double x0 = 0.0;
    double x1 = 2.0;
    double h1 = 0.5;
    double h2 = 0.25;

    double integral_rect_h1 = rectangle_method(x0, x1, h1);
    double integral_rect_h2 = rectangle_method(x0, x1, h2);

    double integral_trap_h1 = trapezoid_method(x0, x1, h1);
    double integral_trap_h2 = trapezoid_method(x0, x1, h2);

```

```
    double integral_simpson_h1 = simpson_method(x0, x1, h1);
    double integral_simpson_h2 = simpson_method(x0, x1, h2);

    double error_rect = runge_romberg_method(integral_rect_h1, integral_rect_h2, 2);
    double error_trap = runge_romberg_method(integral_trap_h1, integral_trap_h2, 2);
    double error_simpson = runge_romberg_method(integral_simpson_h1,
        integral_simpson_h2, 4);

    std::cout << "Rectangle method integral with h1: " << integral_rect_h1 << std::endl
        ;
    std::cout << "Trapezoid method integral with h1: " << integral_trap_h1 << std::endl
        ;
    std::cout << "Simpson method integral with h1: " << integral_simpson_h1 << std::
        endl;

    std::cout << "Rectangle method integral with h2: " << integral_rect_h2 << std::endl
        ;
    std::cout << "Trapezoid method integral with h2: " << integral_trap_h2 << std::endl
        ;
    std::cout << "Simpson method integral with h2: " << integral_simpson_h2 << std::
        endl;


    std::cout << "Estimated error for rectangle method: " << error_rect << std::endl;
    std::cout << "Estimated error for trapezoid method: " << error_trap << std::endl;
    std::cout << "Estimated error for Simpson method: " << error_simpson << std::endl;

    return 0;
}
```