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:	Голошумов М.С.
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: 4

$y = ctg(x), a)X_i = \pi/8, 2\pi/8, 3\pi/8, 4\pi/8;)X_i = \pi/8, 5\pi/16, 3\pi/8, \pi/2; X^* = \pi/3$

2 Результаты работы

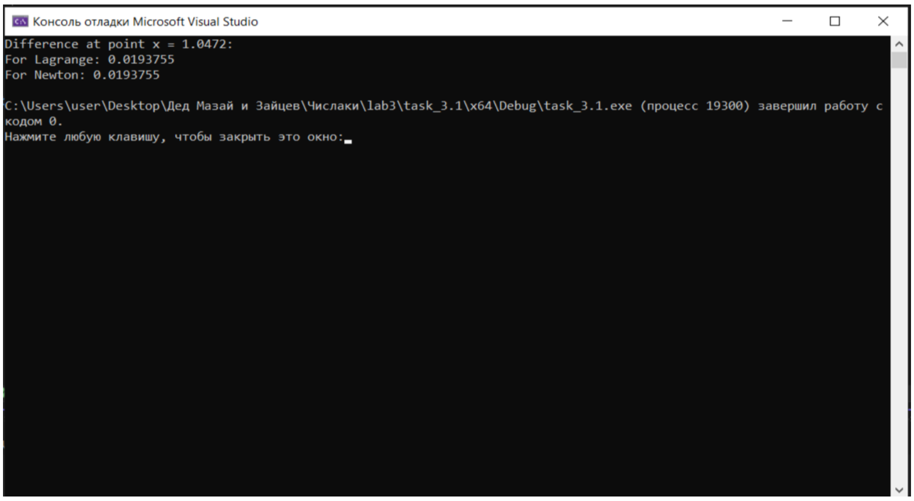


Рис. 1: Вывод программы в консоли

3 Исходный код

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

double f(double x) {
    return 1/tan(x);
}

double pi = 2 * acos(0.0);

double Lagrange(double x, std::vector<double> X_i, std::vector<double> f_i) {
    int n = X_i.size();
    double sum = 0;
    for (int i = 0; i < n; i++) {
        double composition = 1;
        for (int j = 0; j < n; j++) {
            if (i != j) {
                composition *= (x - X_i[j]) / (X_i[i] - X_i[j]);
            }
        }
        sum += f_i[i] * composition;
    }
    return sum;
}

double Newton(double x, std::vector<double> X_i, std::vector<double> f_i) {
    int n = X_i.size();
    std::vector<std::vector<double>> differences;

    auto DividedDifferences = [&]() {
        differences = { f_i };
        for (int i = 1; i < n; i++) {
            std::vector<double> temp;
            for (int k = 0; k < n - i; k++) {
                temp.push_back((differences[i - 1][k] - differences[i - 1][k + 1]) / (X_i[k] - X_i[k + i]));
            }
            differences.push_back(temp);
        }
        };

    DividedDifferences();

    double sum = f_i[0];
    for (int i = 1; i < n; i++) {
```

```
        double composition = 1;
        for (int j = 0; j < i; j++) {
            composition *= x - X_i[j];
        }
        sum += composition * differences[i][0];
    }
    return sum;
}

int main() {
    std::vector<double> X1 = { pi/8, 2* pi / 8, 3* pi / 8, 4* pi / 8 };
    std::vector<double> f1;
    for (auto x : X1) {
        f1.push_back(f(x));
    }

    std::vector<double> X2 = { pi / 8, 5*pi / 16, 3*pi / 8, pi/2 };
    std::vector<double> f2;
    for (auto x : X2) {
        f2.push_back(f(x));
    }

    double X_variation = pi/3;
    std::cout << "Difference at point x = " << X_variation << ":\n"
        << "For Lagrange: " << std::abs(f(X_variation) - Lagrange(X_variation, X1, f1)) << "\n"
        << "For Newton: " << std::abs(f(X_variation) - Newton(X_variation, X1, f1)) << "\n";

    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 $x = X^*$.

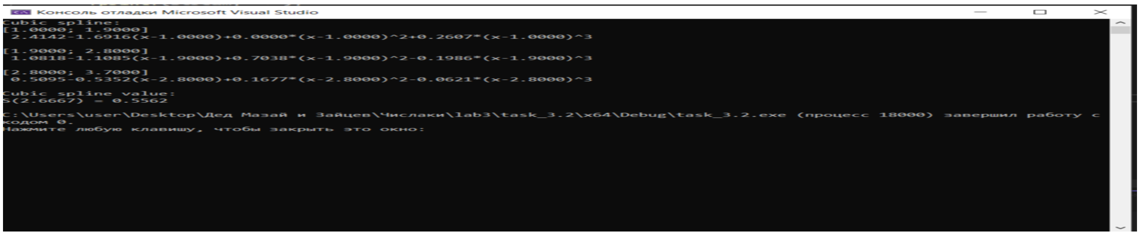
Вариант: 4

4. $X^* = 2.66666667$

i	0	1	2	3	4
x_i	1.0	1.9	2.8	3.7	4.6
f_i	2.4142	1.0818	0.50953	.11836	-0.24008

Рис. 2: Условие

5 Результаты работы



```
Консоль отладки Microsoft Visual Studio
Cubic spline:
[1.0000; 1.9000]
 2.4142-1.6916(x-1.0000)+0.0000*(x-1.0000)^2+0.2607*(x-1.0000)^3

[1.9000; 2.8000]
 1.0818-1.1085(x-1.9000)+0.7038*(x-1.9000)^2-0.1986*(x-1.9000)^3

[2.8000; 3.7000]
 0.5095-0.5352(x-2.8000)+0.1677*(x-2.8000)^2-0.0621*(x-2.8000)^3

Cubic spline value:
S(2.6667) = 0.5562

C:\Users\user\Desktop\Дед Мазай и Зайцев\Числаки\lab3\task_3.2\x64\Debug\task_3.2.exe (процесс 18000) завершил работу с
кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Рис. 3: Вывод программы в консоли

6 Исходный код

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "read.h"

int DEBUG = 0;

Matrix* create_cubic_spline(Matrix* matrix) {
    Matrix* spline, * tridiagonal_matrix, * vector, * solve;
    int i;
    if (!matrix || matrix->width != 2) {
        fprintf(stderr, "Invalid size of matrix\n");
    }
    spline = create_matrix();
    resize_matrix(spline, matrix->height, 5);

    tridiagonal_matrix = create_matrix();
    resize_matrix(tridiagonal_matrix, matrix->height - 2, 3);

    vector = create_matrix();
    resize_matrix(vector, matrix->height - 2, 1);

    for (i = 0; i < vector->height; i++) {
        tridiagonal_matrix->data[i][0] = (i == vector->height - 1 ? 0 : matrix->data[i + 1][0] - matrix->data[i][0]);
        tridiagonal_matrix->data[i][1] = 2 * (matrix->data[i + 2][0] - matrix->data[i][0]);
        tridiagonal_matrix->data[i][2] = (i == 0 ? 0 : matrix->data[i + 2][0] - matrix->data[i + 1][0]);
        vector->data[i][0] = 3 * ((matrix->data[i + 2][1] - matrix->data[i + 1][1]) / (matrix->data[i + 2][0] - matrix->data[i + 1][0]) -
            (matrix->data[i + 1][1] - matrix->data[i][1]) / (matrix->data[i + 1][0] - matrix->data[i][0]));
    }
    if (DEBUG) {
        fprintf(stderr, "Tridiagonal matrix:\n");
        print_matrix(tridiagonal_matrix, stderr);
        fprintf(stderr, "Vector of right part:\n");
        print_matrix(vector, stderr);
    }
    solve = TDMA(tridiagonal_matrix, vector);
    if (!solve) {
        fprintf(stderr, "Singular matrix\n");
        return NULL;
    }
    remove_matrix(tridiagonal_matrix);
    remove_matrix(vector);
```

```
    free(tridiagonal_matrix);
    free(vector);

    spline->data[0][3] = 0;
    for (i = 0; i < spline->height - 1; i++) {
        spline->data[i][0] = matrix->data[i][0];
        spline->data[i][1] = matrix->data[i][1];
        if (i < spline->height - 2)
            spline->data[i + 1][3] = solve->data[i][0];
        spline->data[i][2] = (matrix->data[i + 1][1] - matrix->data[i][1]) / (matrix->
            data[i + 1][0] - matrix->data[i][0]) -
            1.0 / 3.0 * (matrix->data[i + 1][0] - matrix->data[i][0]) * (2 * spline->
                data[i][3] + (i < spline->height - 2 ? spline->data[i + 1][3] : 0));
        spline->data[i][4] = ((i < spline->height - 2 ? spline->data[i + 1][3] : 0) -
            spline->data[i][3]) /
            (3 * (matrix->data[i + 1][0] - matrix->data[i][0]));
    }

    spline->data[spline->height - 1][0] = matrix->data[spline->height - 1][0];

    if (DEBUG) {
        fprintf(stderr, "Spline matrix\n");
        print_matrix(spline, stderr);
    }
    remove_matrix(solve);
    free(solve);
    return spline;
}
double cubic_spline(Matrix* spline, double x) {
    int i;
    for (i = 0; i < spline->height - 1; i++)
        if (spline->data[i][0] <= x && x <= spline->data[i + 1][0])
            return spline->data[i][1] + spline->data[i][2] * (x - spline->data[i][0]) +
            spline->data[i][3] * (x - spline->data[i][0]) * (x - spline->data[i][0]) +
            spline->data[i][4] * (x - spline->data[i][0]) * (x - spline->data[i][0]) *
            (x - spline->data[i][0]);
    if (DEBUG)
        fprintf(stderr, "Incorrect value of argument\n");
    return 0;
}
void print_cubic_spline(Matrix* spline, FILE* stream) {
    int i;
    for (i = 0; i < spline->height - 1; i++) {
        fprintf(stream, "[%.4f; %.4f]\n", spline->data[i][0], spline->data[i + 1][0]);
        if (spline->data[i][1] >= 0)
            fprintf(stream, " ");
        fprintf(stream, "%.4f", spline->data[i][1]);
        if (spline->data[i][2] >= 0)
            fprintf(stream, "+");
```

```
        fprintf(stream, "%.4f(x", spline->data[i][2]);
        if (spline->data[i][0] < 0)
            fprintf(stream, "+");
        fprintf(stream, "%.4f)", -spline->data[i][0]);
        if (spline->data[i][3] >= 0)
            fprintf(stream, "+");
        fprintf(stream, "%.4f*(x", spline->data[i][3]);
        if (spline->data[i][0] < 0)
            fprintf(stream, "+");
        fprintf(stream, "%.4f)^2", -spline->data[i][0]);
        if (spline->data[i][4] >= 0)
            fprintf(stream, "+");
        fprintf(stream, "%.4f*(x", spline->data[i][4]);
        if (spline->data[i][0] < 0)
            fprintf(stream, "+");
        fprintf(stream, "%.4f)^3\n\n", -spline->data[i][0]);
    }
}

int main(void) {
    float x = 2.66666667;
    Matrix* result, * matrix;
    FILE* fmatrix;

    fmatrix = fopen("task_3.2matrix.txt", "r");
    if (!fmatrix) {
        fprintf(stderr, "Incorrect name of file\n");
        return 0;
    }

    matrix = create_matrix();
    scan_matrix(matrix, fmatrix);
    fclose(fmatrix);

    if (result = create_cubic_spline(matrix)) {
        printf("Cubic spline:\n");
        print_cubic_spline(result, stdout);
        printf("Cubic spline value:\nS(%.4f) = %.4f\n", x, cubic_spline(result, x));
        remove_matrix(result);
        free(result);
    }
    remove_matrix(matrix);
    free(matrix);
    return 0;
}
```

```
#include "read.h"
#include <stdlib.h>

Matrix* create_matrix(void) {
```

```
    Matrix* new_matrix = (Matrix*)malloc(sizeof(Matrix));
    new_matrix->width = new_matrix->height = 0;
    new_matrix->data = NULL;
    return new_matrix;
}
void remove_matrix(Matrix* matrix) {
    int i;
    if (!matrix)
        return;
    for (i = 0; i < matrix->height; i++)
        free(matrix->data[i]);
    free(matrix->data);
    matrix->data = NULL;
    matrix->height = matrix->width = 0;
}
void resize_matrix(Matrix* matrix, const int height, const int width) {
    if (height > 0) {
        matrix->data = (double**)realloc(matrix->data, sizeof(double*) * height);
        for (int i = matrix->height; i < height; i++) {
            matrix->data[i] = (double*)malloc(sizeof(double) * (width <= 0 ? matrix->width : width));
            for (int j = 0; j < width; j++)
                matrix->data[i][j] = 0;
        }
    }
    if (width > 0)
        for (int i = 0; i < matrix->height; i++) {
            matrix->data[i] = (double*)realloc(matrix->data[i], sizeof(double) * width);
            for (int j = matrix->width; j < width; j++)
                matrix->data[i][j] = 0;
        }
    if (width > 0)
        matrix->width = width;
    if (height > 0)
        matrix->height = height;
}
void print_matrix(Matrix* matrix, FILE* stream) {
    int i, j;
    if (!matrix->data)
        return;
    for (i = 0; i < matrix->height; i++) {
        fputc('[', stream);
        for (j = 0; j < matrix->width; j++)
            fprintf(stream, "%.5f ", matrix->data[i][j]);
        fprintf(stream, "\b\b]\n");
    }
    fprintf(stream, "size: %d x %d\n", matrix->height, matrix->width);
}
```

```
void scan_matrix(Matrix* matrix, FILE* stream) {
    int i, j, c = 0;
    float a;
    for (i = 0; c != EOF; i++) {
        resize_matrix(matrix, i + 1, -1);
        c = 0;
        for (j = 0; c != '\n'; j++) {
            if (!i)
                resize_matrix(matrix, -1, j + 1);
            fscanf_s(stream, "%f", &a);
            matrix->data[i][j] = a;
            c = getc(stream);
            if (c == EOF) {
                resize_matrix(matrix, i, -1);
                return;
            }
        }
    }
}

static inline float absolute(float a) {
    return a > 0 ? a : -a;
}

Matrix* tridiagonal_matrix_algorithm(Matrix* matrix, Matrix* vector) {
    Matrix* PQ, * result;
    int i;
    if (!matrix || !vector || matrix->width != 3 || matrix->height != vector->height ||
         vector->width != 1)
        return NULL;
    PQ = create_matrix();
    resize_matrix(PQ, matrix->height - 1, 2);
    result = create_matrix();
    resize_matrix(result, matrix->height, 1);
    PQ->data[0][0] = -matrix->data[0][2] / matrix->data[0][1];
    PQ->data[0][1] = vector->data[0][0] / matrix->data[0][1];
    if (absolute(matrix->data[0][1]) < absolute(matrix->data[0][2]) ||
        absolute(matrix->data[matrix->height - 1][1]) < absolute(matrix->data[matrix->
            height - 1][2]))
        return NULL;
    for (i = 1; i < PQ->height; i++) {
        if (absolute(matrix->data[i][1]) < absolute(matrix->data[i][0]) + absolute(
            matrix->data[i][2]))
            return NULL;
        double temp = matrix->data[i][0] * PQ->data[i - 1][0] + matrix->data[i][1];
        PQ->data[i][0] = -matrix->data[i][2] / temp;
        PQ->data[i][1] = (vector->data[i][0] - matrix->data[i][0] * PQ->data[i - 1][1])
             / temp;
    }
```

```
    i = result->height - 1;
    result->data[i][0] = (vector->data[i][0] - matrix->data[i][0] * PQ->data[i - 1][1])
         /
       (matrix->data[i][0] * PQ->data[i - 1][0] + matrix->data[i][1]);
    for (i = result->height - 2; i >= 0; i--)
       result->data[i][0] = PQ->data[i][0] * result->data[i + 1][0] + PQ->data[i][1];
    remove_matrix(PQ);
    free(PQ);
    return result;
}
Matrix* (* const TDMA)(Matrix*, Matrix*) = tridiagonal_matrix_algorithm;
```

```
#ifndef _LAB3_
#define _LAB3_

#include <stdio.h>

typedef struct Matrix {
    double** data;
    unsigned int width;
    unsigned int height;
} Matrix;

Matrix* create_matrix(void);
void remove_matrix(Matrix*);
void resize_matrix(Matrix*, const int, const int);
void print_matrix(Matrix*, FILE*);
void scan_matrix(Matrix*, FILE*);
Matrix* (* const TDMA)(Matrix*, Matrix*);

#endif /* _LAB3_ */
```

Условия:

```
1.0 2.4142
1.9 1.0818
2.8 0.50953
3.7 0.11836
4.6 -0.24008
```

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: 4

i	0	1	2	3	4	5
x_i	1.0	1.9	2.8	3.7	4.6	5.5
y_i	2.4142	1.0818	0.50953	0.11836	-0.24008	-0.66818

Рис. 4: Условия

8 Результаты работы

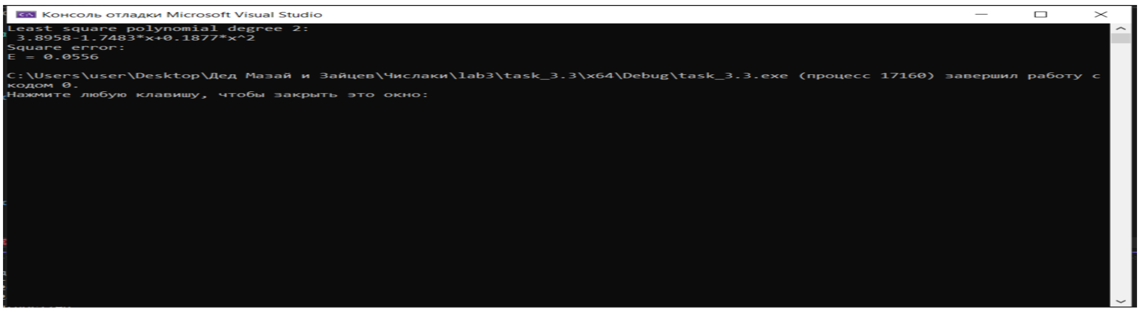


Рис. 5: Вывод программы в консоли

9 Исходный код

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "read.h"

int FIRST = 0;
int SECOND = 1;

double square_error(Matrix* least_squares, Matrix* vector) {
    double result = 0;
    int i, j;
    if (!least_squares || !vector || vector->width != 2 || least_squares->width != 1) {
        fprintf(stderr, "Invalid size of matrix");
        return 0;
    }
    for (i = 0; i < vector->height; i++) {
        double temp = least_squares->data[0][0] - vector->data[i][1];
        for (j = 1; j < least_squares->height; j++)
            temp += least_squares->data[j][0] * pow(vector->data[i][0], j);
        result += temp * temp;
    }
    return result;
}

Matrix* least_square_polynomial_degree_2(Matrix* vector) {
    Matrix* result;
    int i;
    double a11 = vector->height, a12 = 0, a22 = 0, a23 = 0, a33 = 0, b1 = 0, b2 = 0, b3 = 0;
    double c11, c12, c22, d1, d2;
    if (!vector || vector->width != 2) {
        fprintf(stderr, "Invalid size of matrix");
        return 0;
    }
    result = create_matrix();
    resize_matrix(result, 3, 1);
    for (i = 0; i < vector->height; i++) {
        a12 += vector->data[i][0];
        a22 += vector->data[i][0] * vector->data[i][0];
        a23 += vector->data[i][0] * vector->data[i][0] * vector->data[i][0];
        a33 += vector->data[i][0] * vector->data[i][0] * vector->data[i][0] * vector->data[i][0];
        b1 += vector->data[i][1];
        b2 += vector->data[i][1] * vector->data[i][0];
        b3 += vector->data[i][1] * vector->data[i][0] * vector->data[i][0];
    }
```

```
    c11 = a22 * a11 - a12 * a12;
    c12 = a23 * a11 - a12 * a22;
    c22 = a11 * a33 - a22 * a22;
    d1 = b2 * a11 - b1 * a12;
    d2 = b3 * a11 - b1 * a22;
    result->data[1][0] = (d1 * c22 - d2 * c12) / (c11 * c22 - c12 * c12);
    result->data[2][0] = (d2 * c11 - d1 * c12) / (c11 * c22 - c12 * c12);
    result->data[0][0] = (b1 - result->data[1][0] * a12 - result->data[2][0] * a22) / a11;
    return result;
}

Matrix* least_square_polynomial_degree_1(Matrix* vector) {
    Matrix* result;
    int i;
    double a11 = vector->height, a12 = 0, a22 = 0, b1 = 0, b2 = 0;
    if (!vector || vector->width != 2) {
        fprintf(stderr, "Invalid size of matrix");
        return 0;
    }
    result = create_matrix();
    resize_matrix(result, 2, 1);
    for (i = 0; i < vector->height; i++) {
        a12 += vector->data[i][0];
        a22 += vector->data[i][0] * vector->data[i][0];
        b1 += vector->data[i][1];
        b2 += vector->data[i][0] * vector->data[i][1];
    }
    result->data[0][0] = (b1 * a22 - b2 * a12) / (a11 * a22 - a12 * a12);
    result->data[1][0] = (b2 * a11 - b1 * a12) / (a11 * a22 - a12 * a12);
    return result;
}

void print_least_square(Matrix* least_square, FILE* stream) {
    int i;
    if (!least_square || least_square->width != 1) {
        fprintf(stderr, "Invalid size of matrix");
        return;
    }
    if (least_square->data[0][0] >= 0)
        fputc(' ', stream);
    fprintf(stream, "%.4f", least_square->data[0][0]);
    if (least_square->data[1][0] >= 0)
        fputc('+', stream);
    fprintf(stream, "%.4f*x", least_square->data[1][0]);
    for (i = 2; i < least_square->height; i++) {
        if (least_square->data[i][0] >= 0)
            fputc('+', stream);
        fprintf(stream, "%.4f*x^%d", least_square->data[i][0], i);
```

```c
    }
    fputc('\n', stream);
}

int main(void) {
    int i;
    Matrix* result, * vector;
    FILE* fvector;

    fvector = fopen("task_3.3matrix.txt", "r");
    if (!fvector) {
        fprintf(stderr, "Incorrect name of file\n");
        return 0;
    }
    vector = create_matrix();
    scan_matrix(vector, fvector);
    fclose(fvector);

    if (FIRST && (result = least_square_polynomial_degree_1(vector))) {
        printf("Least square polynomial degree 1:\n");
        print_least_square(result, stdout);
        printf("Square error:\nE = %.4f\n", square_error(result, vector));
        remove_matrix(result);
        free(result);
    }
    else if (SECOND && (result = least_square_polynomial_degree_2(vector))) {
        printf("Least square polynomial degree 2:\n");
        print_least_square(result, stdout);
        printf("Square error:\nE = %.4f\n", square_error(result, vector));
        remove_matrix(result);
        free(result);
    }
    remove_matrix(vector);
    free(vector);
    return 0;
}
```

Условия:

```
1.0 2.4142
1.9 1.0818
2.8 0.50953
3.7 0.11836
4.6 -0.24008
5.5 -0.66818
```

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: 4

$X^* = 0.2$

i	0	1	2	3	4
x_i	0.0	0.1	0.2	0.3	0.4
y_i	1.0	1.1052	1.2214	1.3499	1.4918

Рис. 6: Условия

11 Результаты работы

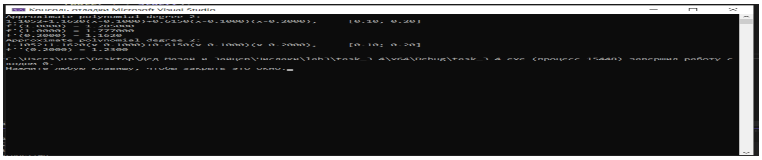


Рис. 7: Вывод программы в консоли

12 Исходный код

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "read.h"

int DEBUG = 1;

double first_derivative(Matrix* vector, double x) {
    int i;
    double b, c, b_1;
    if (!vector || vector->width != 2) {
        fprintf(stderr, "Invalid size of matrix\n");
        return 0;
    }
    for (i = 1; i < vector->height; i++)
        if (vector->data[i - 1][0] <= x && x <= vector->data[i][0])
            break;
    if (i > vector->height - 2) {
        fprintf(stderr, "Too few information about the function\n");
        return 0;
    }

    b = (vector->data[i][1] - vector->data[i - 1][1]) / (vector->data[i][0] - vector->
        data[i - 1][0]);
    c = ((vector->data[i + 1][1] - vector->data[i][1]) / (vector->data[i + 1][0] -
        vector->data[i][0]) - b) /
        (vector->data[i + 1][0] - vector->data[i - 1][0]);
    if (DEBUG) {
        fprintf(stderr, "Approximate polynomial degree 2:\n%.4f", vector->data[i -
            1][1]);
        if (b >= 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f(x", b);
        if (vector->data[i - 1][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f)", -vector->data[i - 1][0]);
        if (c >= 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f(x", c);
        if (vector->data[i - 1][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f)(x", -vector->data[i - 1][0]);
        if (vector->data[i][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f),\t[%.2f; %.2f]\n",
            -vector->data[i][0], vector->data[i - 1][0], vector->data[i][0]);
    }
```

```
    b_1 = (vector->data[i + 1][1] - vector->data[i][1]) / (vector->data[i + 1][0] - vector->data[i][0]);
    printf("f'(1.0000) = %f\n", b_1);
    printf("f'(1.0000) = %f\n", c + b);
    return b;
}
double second_derivative(Matrix* vector, double x) {
    int i;
    double b, c;
    if (!vector || vector->width != 2) {
        fprintf(stderr, "Invalid size of matrix\n");
        return 0;
    }
    for (i = 1; i < vector->height; i++)
        if (vector->data[i - 1][0] <= x && x <= vector->data[i][0])
            break;
    if (i > vector->height - 2) {
        fprintf(stderr, "Too few information about the function\n");
        return 0;
    }
    b = (vector->data[i][1] - vector->data[i - 1][1]) / (vector->data[i][0] - vector->data[i - 1][0]);
    c = ((vector->data[i + 1][1] - vector->data[i][1]) / (vector->data[i + 1][0] - vector->data[i][0]) - b) /
        (vector->data[i + 1][0] - vector->data[i - 1][0]);
    if (DEBUG) {
        fprintf(stderr, "Approximate polynomial degree 2:\n%.4f", vector->data[i - 1][1]);
        if (b >= 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f(x", b);
        if (vector->data[i - 1][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f)", -vector->data[i - 1][0]);
        if (c >= 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f(x", c);
        if (vector->data[i - 1][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f)(x", -vector->data[i - 1][0]);
        if (vector->data[i][0] < 0)
            fputc('+', stderr);
        fprintf(stderr, "%.4f),\t[%.2f; %.2f]\n",
            -vector->data[i][0], vector->data[i - 1][0], vector->data[i][0]);
    }
    return 2 * c;
}

int main(void) {
```

```
    int i;
    Matrix* vector;
    float x = 0.2;
    FILE* fvector;

    fvector = fopen("task_3.4matrix.txt", "r");
    if (!fvector) {
        fprintf(stderr, "Incorrect name of file\n");
        return 0;
    }
    vector = create_matrix();
    scan_matrix(vector, fvector);
    fclose(fvector);

    printf("f'(%.4f) = %.4f\n", x, first_derivative(vector, x));
    printf("f''(%.4f) = %.4f\n", x, second_derivative(vector, x));
    remove_matrix(vector);
    free(vector);
    return 0;
}
```

Условия:

```
0.0 1.0
0.1 1.1052
0.2 1.2214
0.3 1.3499
0.4 1.4918
```

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: 4

$y = \frac{3x+4}{2x+7}$

$X_0 = -2, X_k = 2, h_1 = 1.0, h_2 = 0.5$

14 Результаты работы

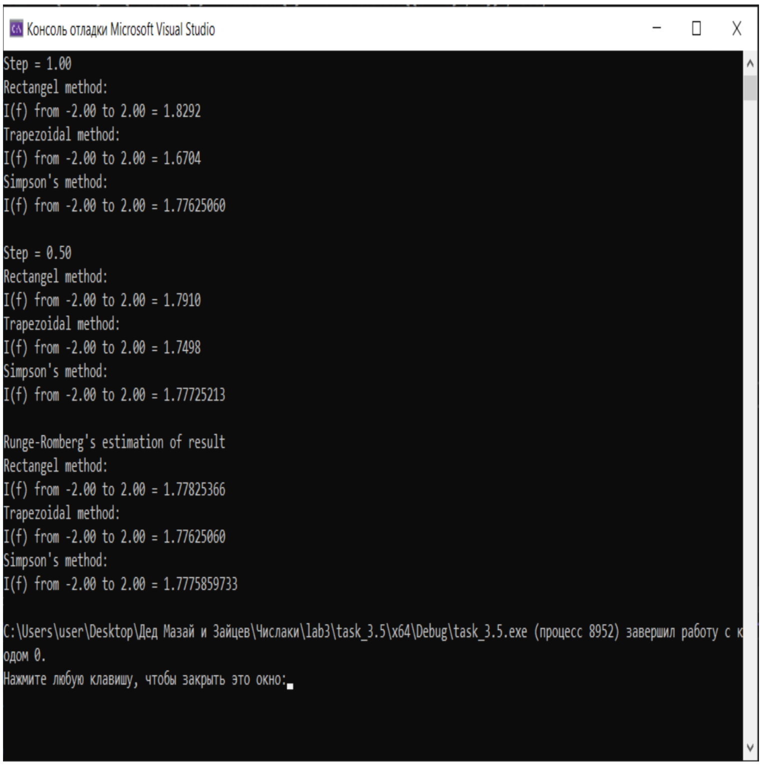


Рис. 8: Вывод программы в консоли

15 Исходный код

```c
#include <stdio.h>
#include <string.h>

double Function(double x) {
    return (3*x + 4) / (2 * x + 7);
}

double rectangle_method(double (*Function)(double), double a, double b, double step) {
    double x, sum = 0;
    if (a >= b) {
        fprintf(stderr, "Incorrect limits of interval\n");
        return 0;
    }
    for (x = a + step; x < b; x += step)
        sum += step * Function(x - step / 2);
    x -= step;
    sum += (b - x) * Function((x + b) / 2);
    return sum;
}

double trapezoidal_method(double (*Function)(double), double a, double b, double step) {
    double x, sum = 0;
    if (a >= b) {
        fprintf(stderr, "Incorrect limits of interval\n");
        return 0;
    }
    for (x = a + step; x < b; x += step)
        sum += 0.5 * step * (Function(x) + Function(x - step));
    x -= step;
    sum += 0.5 * (b - x) * (Function(x) + Function(b));
    return sum;
}

double simpson_method(double (*Function)(double), double a, double b, double step) {
    double x, sum = 0;
    if (a >= b) {
        fprintf(stderr, "Incorrect limits of interval\n");
        return 0;
    }
    for (x = a + step; x < b; x += step)
        sum += step * (Function(x) + Function(x - step) + 4 * Function(x - step / 2)) / 6.0;
    x -= step;
    sum += (b - x) * (Function(x) + Function(b) + 4 * Function((x + b) / 2)) / 6.0;
    return sum;
}
```

```
double runge_romberg_method(double first_estimate, double first_step, double
    second_estimate, double second_step) {
    double k = second_step / first_step;
    if (first_step == second_step) {
        fprintf(stderr, "Equal step of estimates\n");
        return 0;
    }
    return first_estimate + (first_estimate - second_estimate) / (k * k - 1);
}

int main(int argc, char* argv[]) {
    int i;
    float left_limit = -2, right_limit = 2, step1 = 1.0, step2 = 0.5;
    double rectangle1, rectangle2, trapezoidal1, trapezoidal2, simpsons1, simpsons2;

    printf("Step = %.2f\nRectangel method:\n", step1);
    printf("I(f) from %.2f to %.2f = %.4f\n",
        left_limit, right_limit, rectangle1 = rectangle_method(Function, left_limit,
            right_limit, step1));
    printf("Trapezoidal method:\n");
    printf("I(f) from %.2f to %.2f = %.4f\n",
        left_limit, right_limit, trapezoidal1 = trapezoidal_method(Function, left_limit
            , right_limit, step1));
    printf("Simpson's method:\n");
    printf("I(f) from %.2f to %.2f = %.8f\n",
        left_limit, right_limit, simpsons1 = simpson_method(Function, left_limit,
            right_limit, step1));

    printf("\nStep = %.2f\nRectangel method:\n", step2);
    printf("I(f) from %.2f to %.2f = %.4f\n",
        left_limit, right_limit, rectangle2 = rectangle_method(Function, left_limit,
            right_limit, step2));
    printf("Trapezoidal method:\n");
    printf("I(f) from %.2f to %.2f = %.4f\n",
        left_limit, right_limit, trapezoidal2 = trapezoidal_method(Function, left_limit
            , right_limit, step2));
    printf("Simpson's method:\n");
    printf("I(f) from %.2f to %.2f = %.8f\n",
        left_limit, right_limit, simpsons2 = simpson_method(Function, left_limit,
            right_limit, step2));

    printf("\nRunge-Romberg's estimation of result\nRectangel method:\n");
    printf("I(f) from %.2f to %.2f = %.8f\n",
        left_limit, right_limit, runge_romberg_method(rectangle1, step1, rectangle2,
            step2));
    printf("Trapezoidal method:\n");
    printf("I(f) from %.2f to %.2f = %.8f\n",
```

```
        left_limit, right_limit, runge_romberg_method(trapezoidal1, step1, trapezoidal2, step2));
    printf("Simpson's method:\n");
    printf("I(f) from %.2f to %.2f = %.10f\n",
        left_limit, right_limit, runge_romberg_method(simpsons1, step1, simpsons2, step2));
    return 0;
}
```