



Министерство науки и высшего образования Российской  
Федерации  
Федеральное государственное бюджетное образовательное  
учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы  
управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные  
технологии»

**Лабораторная работа № 1**

**Тема** Интерполяция таблично заданных функций полиномом  
Ньютона и Эрмита

**Студент** Андреев А.А.

**Группа** ИУ7-44Б

**Оценка (баллы)** \_\_\_\_\_

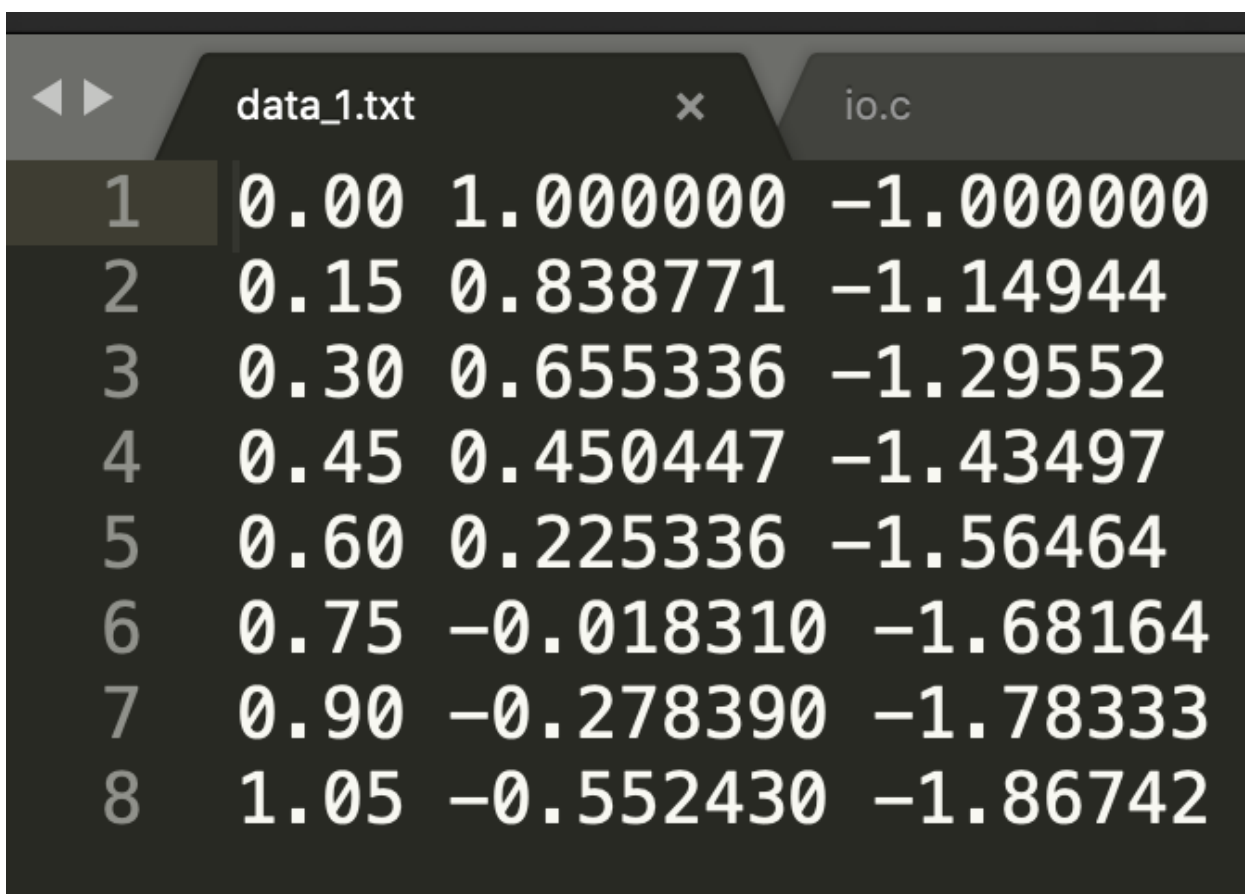
**Преподаватель** \_\_\_\_\_

Москва.  
2021 г.

# Описание работы

**Цель работы:** восстановить функцию для нахождения её значения любой точке.

**Входные данные:** таблица точек, степень вычисляемого полинома, аргумент функции.



	data_1.txt	io.c
1	0.00	1.000000 -1.000000
2	0.15	0.838771 -1.14944
3	0.30	0.655336 -1.29552
4	0.45	0.450447 -1.43497
5	0.60	0.225336 -1.56464
6	0.75	-0.018310 -1.68164
7	0.90	-0.278390 -1.78333
8	1.05	-0.552430 -1.86742

**Выходные данные:** результат интерполяции, точное значение функции, корень функции, найденный методом половинного деления, корень функции, найденный методом обратной интерполяции.

## Алгоритм выполнения

### *Линейная интерполяция.*

Для нахождения полинома Ньютона первым делом выполняется сортировка входной таблицы по возрастанию аргументов. Данное действие необходимо для правильного выбора узлов, где  $X$  является центром конфигурации. Количество узлов равно степени полинома + 1. Для нахождения полинома  $n$ -ой степени необходимо найти разделенные разности.

Формулы вычисления разделенных разностей и интерполяционный многочлен Ньютона:

$$f(x_0; x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0},$$

$$f(x_0; x_1; x_2) = \frac{f(x_1; x_2) - f(x_0; x_1)}{x_2 - x_0} = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0},$$

$$f(x_0; x_1; \dots; x_{n-1}; x_n) = \frac{f(x_1; \dots; x_{n-1}; x_n) - f(x_0; x_1; \dots; x_{n-1})}{x_n - x_0}.$$

$$L_n(x) = f(x_0) + f(x_0; x_1) \cdot (x - x_0) + f(x_0; x_1; x_2) \cdot (x - x_0) \cdot (x - x_1) + \dots + f(x_0; \dots; x_n) \cdot \prod_{k=0}^{n-1} (x - x_k)$$

После вычисления разделенных разностей, подставив аргумент в интерполяционный многочлен Ньютона можно найти значение функции.

В программе предусмотрено вычисление точного значения и относительной ошибки вычисления.

## ***Нахождение корня используя методы половинного деления и обратной интерполяции.***

Метод половинного деления позволяет исключать в точности половину интервала на каждой итерации. При использовании метода считается, что функция непрерывна и имеет на концах интервала разный знак. После вычисления значения функции в середине интервала одна часть интервала отбрасывается так, чтобы функция имела разный знак на концах оставшейся части. Итерации метода прекращаются если интервал становится достаточно мал (используется относительная погрешность)

Метод обратной интерполяции предусматривает перестановку столбцов аргументов и значений функции для нахождения методом линейной интерполяции значения аргумента при значении функции равном нулю.

# Код программы

Программа состоит из 8 файлов. Репозиторий проекта содержит makefile, папки src, inc, out, data, где в src находится исходный код программы, в inc заголовки функций с внешним связыванием, в out складывается на время компиляции файлы формата \*.o, в data находится входной файл с таблицей данных для  $x$ ,  $y$ ,  $y'$ .

*main.c*

```
#include <stdio.h>
#include <stdlib.h>

#include "../inc/io.h"

#include "../inc/base_functions.h"

#include "../inc/newton_polynom.h"
#include "../inc/armit_polynom.h"

#include "../inc/check_functions.h"

#include "../inc/config.h"

#include "../inc/comparators.h"

int main()
{
    interpolation_operation data;

    if (get_opening_file_status(&data) == SUCCESS_STATUS)
    {
        data_transmission_comparator(&data);
        dif_polynom_comparator(&data, check_monotony(data));
    }

    return SUCCESS_STATUS;
}
```

*io.c*

```
#include <stdio.h>
#include <stdlib.h>

#include "../inc/io.h"

#include "../inc/config.h"

#include "../inc/comparators.h"
```

```

int getting_file_data(FILE *operation_file, interpolation_operation **data)
{
    (*data)->total = 0;
    float x_values, y_values, first_derivative_values;

    int number_of_values = NUMBER_OF_VALUES;

    while (number_of_values == NUMBER_OF_VALUES && !feof(operation_file))
    {
        number_of_values = fscanf(operation_file, "%f%f%f\n", &x_values, &y_values,
&first_derivative_values);
        (*data)->total++;
    }

    if (number_of_values != NUMBER_OF_VALUES && feof(operation_file))
        return ERROR_STATUS;
    return SUCCESS_STATUS;
}

void write_read_data(FILE *operation_file, interpolation_operation *data)
{
    for (int temp_number_of_string = 0; temp_number_of_string < data->total;
temp_number_of_string++)
        fscanf(operation_file, "%f%f%f\n", &(data->x_values[temp_number_of_string]),
&(data->y_values[temp_number_of_string]),
&(data->first_derivative_values[temp_number_of_string]));
}

int file_reading(FILE *operation_file, interpolation_operation *data)
{
    if (getting_file_data(operation_file, &data) == ERROR_STATUS)
        return ERROR_STATUS;

    rewind(operation_file);
    data->x_values = malloc(sizeof(float) * data->total);
    data->y_values = malloc(sizeof(float) * data->total);
    data->first_derivative_values = malloc(sizeof(float) * data->total);

    write_read_data(operation_file, data);

    return SUCCESS_STATUS;
}

int get_x_reading_status(interpolation_operation *data)
{
    printf("Введите значение x: ");
    return scanf("%f", &data->x);
}

int get_number_of_polynomial_degree(interpolation_operation *data)
{

```

```

        printf("Введите значение степени полинома от 1 до 4: ");
        return scanf("%d", &data->number_of_polynomial_degree);
    }

int is_right_input_data(interpolation_operation *data)
{
    if (data->number_of_polynomial_degree < 1 ||
        data->number_of_polynomial_degree > 4)
        return ERROR_STATUS;
    return SUCCESS_STATUS;
}

int input_data(FILE *operation_file, interpolation_operation *data)
{
    int temp_error_status = file_reading(operation_file, data);

    if (temp_error_status == SUCCESS_STATUS)
    {
        if (get_x_reading_status(data) != 1)
            return ERROR_STATUS;

        if (get_number_of_polynomial_degree(data) != 1)
            return ERROR_STATUS;

        if (is_right_input_data(data) == ERROR_STATUS)
            return ERROR_STATUS;
    }
    return temp_error_status;
}

void output_polynom(interpolation_operation data)
{
    printf("\nЗначения полиномов:\n\tНьютон:    %.6f\n\tЭрмит:    %.6f\n",
        data.y_newton_value, data.y_newton_value);
    free(data.first_data_inequality);
    free(data.second_data_inequality);
}

void output_function_root(interpolation_operation data)
{
    printf("\nЗначение корня: %.6f\n", data.y_newton_value);
    free(data.first_data_inequality);
    free(data.x_values);
    free(data.y_values);
    free(data.first_derivative_values);
}

void update_first_indicator(interpolation_operation *data)
{
    data->first_indicator = 0, data->second_indicator = 0;
}

```

```

    for (int i = 0; i < data->total - 1; i++)
    {
        if (data->x >= data->x_values[i] && data->x <= data->x_values[i + 1])
            data->first_indicator = i;
    }
}

int is_data_enough(interpolation_operation *data)
{
    if (data->total < data->number_of_polynomial_degree + 1)
        return SUCCESS_STATUS;
    return ERROR_STATUS;
}

void out_need_data()
{
    printf("Необходимо больше данных.\n");
}

void out_config(interpolation_operation *data)
{
    printf("\nУстановка: %f - %f\n", data->x_values[data->first_indicator],
data->x_values[data->second_indicator]);
}

void get_config(interpolation_operation *data)
{
    update_first_indicator(data);

    if (is_data_enough(data) == ERROR_STATUS)
    {
        out_need_data();
        return ;
    }

    data_transmission_comparator(data);

    out_config(data);
}

int get_opening_file_status(interpolation_operation *data)
{
    FILE *f = fopen(INPUT_DATA_FILE, "r");

    int temp_opening_file_status = input_data(f, data);

    fclose(f);

    return temp_opening_file_status;
}

```

## *comprators.c*

```
#include <stdio.h>
#include <stdlib.h>

#include "../inc/io.h"

#include "../inc/base_functions.h"

#include "../inc/newton_polynom.h"
#include "../inc/armit_polynom.h"

#include "../inc/check_functions.h"

#include "../inc/config.h"

#include "../inc/comparators.h"

void data_transmission_comparator(interpolation_operation *data)
{
    sort_data(data);
    get_config(data);
}

void getting_dif_polynom_comparator(interpolation_operation *data)
{
    divided_difference_newton(data);
    divided_difference_armit(data);
}

void dif_polynom_comparator(interpolation_operation *data, int
temp_monotony_status)
{
    getting_dif_polynom_comparator(data);

    if (data->first_data_inequality && data->second_data_inequality)
    {
        get_newton_polynom(data);
        get_armit_polynom(data);
        output_polynom(*data);
    }
    if (temp_monotony_status)
    {
        swap_columns(data);

        if (data->first_data_inequality)
        {
            get_newton_polynom(data);
            output_function_root(*data);
        }
    }
}
```



```

    }
    else
        printf("Function is not monotonous.\n");
}

void data_tranmission_comparator(interpolation_operation *data)
{
    int step = (data->number_of_polynomial_degree + 1) / 2;

    if ((data->number_of_polynomial_degree + 1) % 2 == 0)
    {
        if (data->first_indicator - step >= 0)
        {
            if (data->first_indicator + step <= data->total - 1)
            {
                data->second_indicator = data->first_indicator + step;
                data->first_indicator -= step - 1;
            }
            else
            {
                data->second_indicator = data->total - 1;
                data->first_indicator = data->total - step * 2;
            }
        }
        else
        {
            data->first_indicator = 0;
            data->second_indicator = step * 2 - 1;
        }
    }
    else
    {
        if (data->first_indicator - step + 1 >= 0)
        {
            if (data->first_indicator + step <= data->total - 1)
            {
                data->second_indicator = data->first_indicator + step;
                data->first_indicator -= step;
            }
            else
            {
                data->second_indicator = data->total - 1;
                data->first_indicator = data->total - 1 - step * 2;
            }
        }
        else
        {
            data->first_indicator = 0;
            data->second_indicator = step * 2;
        }
    }
}

```

```
}  
}
```

#### *check\_functions.c*

```
#include <stdio.h>  
#include <stdlib.h>  
#include "../inc/io.h"  
#include "../inc/check_functions.h"  
  
int check_monotony(interpolation_operation data)  
{  
    int res = 0;  
    if (data.y_values[0] < 0)  
        res = -1;  
    else  
        res = 1;  
    for (int i = 0; i < data.total - 1; i++)  
        if (res > 0 && data.y_values[i] <= 0)  
            res = 2;  
        else if (res < 0 && data.y_values[i] >= 0)  
            res = -2;  
    if (res % 2 != 0)  
        res = 0;  
    return res;  
}
```

#### *base\_functions.c*

```
#include <stdio.h>  
#include <stdlib.h>  
  
#include "../inc/io.h"  
  
#include "../inc/base_functions.h"  
  
#include "../inc/newton_polynom.h"  
#include "../inc/armit_polynom.h"  
  
#include "../inc/check_functions.h"  
  
#include "../inc/config.h"  
  
#include "../inc/comparators.h"  
  
void swap(interpolation_operation *data, int starting_position, int ending_position)  
{  
    float a = data->x_values[starting_position], b = data->y_values[starting_position], c  
    = data->first_derivative_values[starting_position];
```

```

    data->x_values[starting_position] = data->x_values[ending_position],
    data->y_values[starting_position] = data->y_values[ending_position],
    data->first_derivative_values[starting_position] =
    data->first_derivative_values[ending_position];
    data->x_values[ending_position] = a, data->y_values[ending_position] = b,
    data->first_derivative_values[ending_position] = c;
}

void displacement_data(interpolation_operation *data, int min_position, int
max_position)
{
    for (int temp_position = min_position; temp_position < max_position;
temp_position++)
        swap(data, temp_position, max_position);
}

void sort_data(interpolation_operation *data)
{
    for (int right_search_cur = 1; right_search_cur < data->total; right_search_cur++)
        for (int left_search_cur = 0; left_search_cur < right_search_cur;
left_search_cur++)
            if (data->x_values[right_search_cur] < data->x_values[left_search_cur])
                displacement_data(data, left_search_cur, right_search_cur);
}

void swap_columns(interpolation_operation *data)
{
    float buf = 0;
    for (int i = 0; i < data->total; i++)
    {
        buf = data->x_values[i];
        data->x_values[i] = data->y_values[i];
        data->y_values[i] = buf;
    }

    data->x = 0;

    data_transmission_comparator(data);
    divided_difference_newton(data);
}

```

armit\_polynom.c

```

#include <stdio.h>
#include <stdlib.h>

#include "../inc/io.h"

#include "../inc/base_functions.h"

```

```

#include "../inc/newton_polynom.h"
#include "../inc/armit_polynom.h"

#include "../inc/check_functions.h"

#include "../inc/config.h"

#include "../inc/comparators.h"

void swap(interpolation_operation *data, int starting_position, int ending_position)
{
    float a = data->x_values[starting_position], b = data->y_values[starting_position], c
= data->first_derivative_values[starting_position];
    data->x_values[starting_position] = data->x_values[ending_position],
data->y_values[starting_position] = data->y_values[ending_position],
data->first_derivative_values[starting_position] =
data->first_derivative_values[ending_position];
    data->x_values[ending_position] = a, data->y_values[ending_position] = b,
data->first_derivative_values[ending_position] = c;
}

void displacement_data(interpolation_operation *data, int min_position, int
max_position)
{
    for (int temp_position = min_position; temp_position < max_position;
temp_position++)
        swap(data, temp_position, max_position);
}

void sort_data(interpolation_operation *data)
{
    for (int right_search_cur = 1; right_search_cur < data->total; right_search_cur++)
        for (int left_search_cur = 0; left_search_cur < right_search_cur;
left_search_cur++)
            if (data->x_values[right_search_cur] < data->x_values[left_search_cur])
                displacement_data(data, left_search_cur, right_search_cur);
}

void swap_columns(interpolation_operation *data)
{
    float buf = 0;
    for (int i = 0; i < data->total; i++)
    {
        buf = data->x_values[i];
        data->x_values[i] = data->y_values[i];
        data->y_values[i] = buf;
    }

    data->x = 0;

    data_transmission_comparator(data);
}

```

```

    divided_difference_newton(data);
}

```

### *newton\_polynom.c*

```

#include <stdio.h>
#include <stdlib.h>

#include "../inc/io.h"

#include "../inc/base_functions.h"
#include "../inc/newton_polynom.h"

void
get_calculated_inequality_of_arguments_and_first_separated_inequality(interpolation
_operation *data)
{
    for (int i = 0; i < data->number_of_polynomial_degree; i++)
    {
        data->first_data_inequality[i * (data->number_of_polynomial_degree + 1)] =
data->x - data->x_values[i + data->first_indicator];
        data->first_data_inequality[i * (data->number_of_polynomial_degree + 1) + 1] =
(data->y_values[i + data->first_indicator] - data->y_values[i + 1 +
data->first_indicator]) / (data->x_values[i + data->first_indicator] - data->x_values[i + 1
+ data->first_indicator]);
    }
}

void get_calculated_staying_separated_inequality(interpolation_operation *data)
{
    for (int j = 2; j < data->number_of_polynomial_degree + 1; j++)
        for (int i = 0; i < data->number_of_polynomial_degree - j + 1; i++)
            data->first_data_inequality[i * (data->number_of_polynomial_degree + 1) +
j] = (data->first_data_inequality[i * (data->number_of_polynomial_degree + 1) + j - 1] -
data->first_data_inequality[(i + 1) * (data->number_of_polynomial_degree + 1) + j - 1])
/ (data->x_values[i + data->first_indicator] - data->x_values[i + j +
data->first_indicator]);
}

void divided_difference_newton(interpolation_operation *data)
{
    data->first_data_inequality = malloc(sizeof(float) *
(data->number_of_polynomial_degree + 1) * (data->number_of_polynomial_degree +
1));

    get_calculated_inequality_of_arguments_and_first_separated_inequality(data);
    get_calculated_staying_separated_inequality(data);
}

void get_newton_polynom(interpolation_operation *data)

```

```

{
    data->y_newton_value = data->y_values[data->first_indicator];
    float buf_x = 1;
    for (int i = 1; i < data->number_of_polynomial_degree + 1; i++)
    {
        buf_x *= data->first_data_inequality[(i - 1) *
(data->number_of_polynomial_degree + 1)];
        data->y_newton_value += buf_x * data->first_data_inequality[i];
    }
}
}

```

### *makefile*

```

all: app.exe clear

# APP.EXE
app.exe : main.o io.o base_functions.o check_functions.o newton_polynom.o
armit_polynom.o comparators.o
    gcc -o app.exe out/main.o out/io.o out/base_functions.o out/check_functions.o
out/newton_polynom.o out/armit_polynom.o out/comparators.o

io.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/io.o src/io.c

base_functions.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/base_functions.o
src/base_functions.c

main.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/main.o src/main.c

check_functions.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/check_functions.o
src/check_functions.c

newton_polynom.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/newton_polynom.o
src/newton_polynom.c

armit_polynom.o :
    mkdir -p out
    gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/armit_polynom.o
src/armit_polynom.c

comparators.o :

```

```

mkdir -p out
gcc -std=c99 -Wall -Werror -ggdb -pedantic -c -o out/comparators.o
src/comparators.c

clear :
    rm out/*.o

clear_all :
    rm *.exe

```

## Результат работы программы в таблице

n	1	2	3	4
Ньютона	0.190089	0.191235	0.191186	0.191194
Эрмита	0.191190	0.191190	0.191191	0.191191
Корень	0.738728	0.739046	0.739095	0.739088

## Контрольные вопросы:

### 1. Будет ли работать программа при степени полинома $n = 0$ ?

У меня установлено ограничение от 1 до 4 на значение  $n$ , поэтому программа работать не будет, но существует возможность построить полином при использовании табличного значения  $y$  наиболее близкого к значению  $X$ .

### 2. Как практически оценить погрешность интерполяции? Почему сложно применить для этих целей теоретическую оценку?

Производные интерполируемой функции как правило изначально неизвестны, поэтому лучше использовать оценку первого отброшенного члена.

Погрешность оценивается:

$$|y(x) - P_n(x)| \leq \frac{M_{n+1}}{(n+1)!} |\varpi_n(x)|,$$

где  $M_{n+1} = \max |y^{(n+1)}(\xi)|$  - максимальное значение производной интерполируемой функции на отрезке между наименьшим и наибольшим из значений  $x_0, x_1, x_2, \dots, x_n$ , а полином

$$\varpi_n(x) = \prod_{i=0}^n (x - x_i).$$

**3. Если в двух точках заданы значения функции и ее первых производных, то полином какой минимальной степени может быть построен в этих точках?**

У меня 4 узла, поэтому минимальная степень полинома - три.

**4. В каком месте алгоритма построения полинома существенна информация об упорядоченности аргумента функции (возрастает, убывает)?**

В алгоритме построения полинома информация об упорядоченности аргумента функции нужна во время построения узлов, при убывании или возрастании выбираю аргументы около изначального X.

**5. Что такое выравнивающие переменные и как их применить для повышения точности интерполяции?**

Выравнивающие переменные - это производные функции от исходных переменных, их использую для того, чтобы упростить подсчет для быстро изменяющихся функций на отдельных промежутках переменный график был близок к прямой.

Для интерполяции задаются новые  $n = n(y)$ ,  $e = e(x)$ , далее проводится интерполяция по переменным  $(n, e)$ , а после обратным интерполированием находится  $y_i = y(n_i)$ .