МГТУ им. Н. Э. Баумана

Отчет по лабораторной работе №1 по предмету «Вычислительные алгоритмы»
«Построение и программная реализация алгоритма полиномиальной интерполяции табличных функций»

Выполнила Параскун София,
ИУ7-44Б
Проверил
Градов В. М.

Москва, 2021 г.

Цель работы: получение навыков построения алгоритма интерполяции таблично заданных функций полиномами Ньютона и Эрмита.

1. Исходные данные

1. Таблица функции и ее производных (эти данные считываются из файла).

x	y	y'
0.00	1.000000	--1.000000
0.15	0.838771	-1.14944
0.30	0.655336	-1.29552
0.45	0.450447	-1.43497
0.60	0.225336	-1.56464
0.75	-0.018310	-1.68164
0.90	-0.278390	-1.78333
1.05	-0.552430	-1.86742

2. Степень аппроксимирующего полинома — n (считывается с консоли).

3. Значение аргумента x, для которого выполняется интерполяция (считывается с консоли).

2. Код программы

Программа состоит из 5 файлов, которые собираются в app.exe. Имя файла с входными данными изначально зафиксировано в программе in.txt.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "in_out.h"
#include "polynom_func.h"

int main()
{
    int res = OK;
    FILE *f = fopen("in.txt", "r");
    struct polynom data;
    res = input_data(f, &data);
    fclose(f);

    if (!res)
    {
        sort_data(&data);
        find_config(&data);
        divided_difference_newton(&data);
        divided_difference_armit(&data);
        if (data.difference1 && data.difference2)
```

```
        {
            find_polynom_newton(&data);
            find_polynom_armit(&data);
            out_res(data, 1);
        }

        if (check_monotony(data))
        {
            swap_columns(&data);
            data.x = 0;
            sort_data(&data);
            find_config(&data);
            divided_difference_newton(&data);
            if (data.difference1)
            {
                find_polynom_newton(&data);
                out_res(data, 2);
            }
        }
        else
            printf("Function is not monotonous.\n");
    }

    return res;
}
```

in_out.h

```
#include <stdio.h>
#ifndef _IN_OUT_H_
#define _IN_OUT_H_

#define OK 0
#define ERR -1

struct polynom
{
    int count;
    float *data_x;
    float *data_y;
    float *data_y_p1;
    float x;
    int n;
    int ind1;
    int ind2;
    float *difference1;
    float *difference2;
    float y_newton;
    float y_armit;
};

int input_data(FILE *f, struct polynom *data);
void out_res(struct polynom data, int mode);

#endif
```

in_out.c

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

int file_reading(FILE *f, struct polynom *data);

int input_data(FILE *f, struct polynom *data)
{
    int res = OK;
    if (f)
    {
        res = file_reading(f, data);
    }

    if (!res)
    {
        printf("File-data is correct.\n");
        printf("Input x: ");
        int rc = 0;
        rc = scanf("%f", &data->x);
        if (rc)
        {
            printf("Input polynomal degree n (in range 1-4): ");
            rc += scanf("%d", &data->n);
            if (data->n < 1 || data->n > 4)
                res = ERR;
        }
        if (rc != 2 && !res)
        {
            printf("Wrong data!\n");
            res = ERR;
        }
        else
            printf("Correct data.\n");
    }

    return res;
}

int file_reading(FILE *f, struct polynom *data)
{
    data->count = 0;
    int res = OK, rc = 3;
    float a, b, c;

    while (rc == 3 && !feof(f))
    {
        rc = fscanf(f, "%f%f%f\n", &a, &b, &c);
        data->count++;
    }

    if (rc != 3 && feof(f))
    {
        printf("Wrong data!\n");
        res = ERR;
```

```
    }
    else
    {
        rewind(f);
        data->data_x = malloc(sizeof(float) * data->count);
        data->data_y = malloc(sizeof(float) * data->count);
        data->data_y_p1 = malloc(sizeof(float) * data->count);
        if (data->data_x &&  data->data_y && data->data_y_p1)
            for (int i = 0; i < data->count; i++)
                fscanf(f, "%f%f%f\n", &(data->data_x[i]), &(data->data_y[i]),
&(data->data_y_p1[i]));
        else
        {
            res = ERR;
            printf("Memory allocate error!\n");
        }
    }

    return res;
}

void out_res(struct polynom data, int mode)
{
    if (mode == 1)
    {
        printf("Polynoms for %d power.\nNewton: %.6f\nArmit: %.6f\n", data.n,
data.y_newton, data.y_newton);
        free(data.difference1);
        free(data.difference2);
    }
    else
    {
        printf("Function root: %.6f\n", data.y_newton);
        free(data.difference1);
        free(data.data_x);
        free(data.data_y);
        free(data.data_y_p1);
    }
}
```

polynom_func.h

```
#include <stdio.h>
#include "in_out.h"

#ifndef _POLYNOM_FUNC_H_
#define _POLYNOM_FUNC_H_

void sort_data(struct polynom *data);
void find_config(struct polynom *data);
void divided_difference_newton(struct polynom *data);
void divided_difference_armit(struct polynom *data);
void find_polynom_newton(struct polynom *data);
void find_polynom_armit(struct polynom *data);
int check_monotony(struct polynom data);
void swap_columns(struct polynom *data);

#endif
```

polynom_func.c

```
#include <stdio.h>
#include <stdlib.h>
#include "in_out.h"
#include "polynom_func.h"

void shift(struct polynom *data, int ind, int pos);
void swap(struct polynom *data, int i, int j);

void sort_data(struct polynom *data)
{
    for (int i = 1; i < data->count; i++)
    {
        for (int j = 0; j < i; j++)
        {
            if (data->data_x[i] < data->data_x[j])
                shift(data, j, i);
        }
    }
}

void shift(struct polynom *data, int ind, int pos)
{
    for (int k = ind; k < pos; k++)
    {
        swap(data, k, pos);
    }
}

void swap(struct polynom *data, int i, int j)
{
    float a = data->data_x[i], b = data->data_y[i], c = data->data_y_p1[i];
    data->data_x[i] = data->data_x[j], data->data_y[i] = data->data_y[j], data-
>data_y_p1[i] = data->data_y_p1[j];
    data->data_x[j] = a, data->data_y[j] = b, data->data_y_p1[j] = c;
}

void find_config(struct polynom *data)
{
    data->ind1 = 0, data->ind2 = 0;
    for (int i = 0; i < data->count - 1; i++)
    {
        if (data->x >= data->data_x[i] && data->x <= data->data_x[i + 1])
            data->ind1 = i;
    }

    int step = (data->n + 1) / 2;
    if (data->count < data->n + 1)
    {
        printf("Need more data.\n");
    }
    else if ((data->n + 1) % 2 == 0)
    {
        if (data->ind1 - step >= 0)
        {
            if (data->ind1 + step <= data->count - 1)
```

```
            {
                data->ind2 = data->ind1 + step;
                data->ind1 -= step - 1;
            }
            else
            {
                data->ind2 = data->count - 1;
                data->ind1 = data->count - step * 2;
            }
        }
        else
        {
            data->ind1 = 0;
            data->ind2 = step * 2 - 1;
        }
    }
    else
    {
        if (data->ind1 - step + 1 >= 0)
        {
            if (data->ind1 + step <= data->count - 1)
            {
                data->ind2 = data->ind1 + step;
                data->ind1 -= step;
            }
            else
            {
                data->ind2 = data->count - 1;
                data->ind1 = data->count - 1 - step * 2;
            }
        }
        else
        {
            data->ind1 = 0;
            data->ind2 = step * 2;
        }
    }
    printf("Config: %f - %f\n", data->data_x[data->ind1], data->data_x[data-
>ind2]);
}

void divided_difference_newton(struct polynom *data)
{
    data->difference1 = malloc(sizeof(float) * (data->n + 1) * (data->n + 1));
    if (data->difference1)
    {
        //вычисление разностей аргументов и первых разделенных разностей
        for (int i = 0; i < data->n; i++)
        {
            data->difference1[i * (data->n + 1)] = data->x - data->data_x[i + data-
>ind1];
            data->difference1[i * (data->n + 1) + 1] = (data->data_y[i + data-
>ind1] - data->data_y[i + 1 + data->ind1]) / (data->data_x[i + data->ind1] - data-
>data_x[i + 1 + data->ind1]);
        }

        //вычисление оставшихся разделенных разностей
        for (int j = 2; j < data->n + 1; j++)
            for (int i = 0; i < data->n - j + 1; i++)
```

```
                data->difference1[i * (data->n + 1) + j] = (data->difference1[i *
(data->n + 1) + j - 1] - data->difference1[(i + 1) * (data->n + 1) + j - 1]) /
(data->data_x[i + data->ind1] - data->data_x[i + j + data->ind1]);
    }
    else
        printf("Memory allocate error!\n");
}

void divided_difference_armit(struct polynom *data)
{
    data->difference2 = malloc(sizeof(float) * 2 * (data->n + 1) * 2 * (data->n +
1));
    if (data->difference2)
    {
        //вычисление разностей аргументов
        for (int i = 0; i < data->n + 1; i++)
            data->difference2[i * 2 * (data->n + 1)] = data->x - data->data_x[i +
data->ind1];

        //вычисление первых разделенных разностей
        for (int i = 0; i < 2 * data->n + 1; i++)
            if (i % 2 == 0)
                data->difference2[i * 2 * (data->n + 1) + 1] = data->data_y_p1[i /
2 + data->ind1];
            else
                data->difference2[i * 2 * (data->n + 1) + 1] = (data->data_y[i / 2
+ data->ind1] - data->data_y[(i + 1) / 2 + data->ind1]) / (data->data_x[i / 2 +
data->ind1] - data->data_x[(i + 1) / 2 + data->ind1]);

        //вычисление оставшихся разделенных разностей
        for (int j = 2; j < 2 * (data->n + 1); j++)
            for (int i = 0; i < 2 * (data->n + 1) - j; i++)
                if (i % 2 == 0)
                    data->difference2[i * 2 * (data->n + 1) + j] = (data-
>difference2[i * 2 * (data->n + 1) + j - 1] - data->difference2[(i + 1) * 2 *
(data->n + 1) + j - 1]) / (data->data_x[i / 2 + data->ind1] - data->data_x[i / 2 +
j / 2 + data->ind1]);
                else
                    data->difference2[i * 2 * (data->n + 1) + j] = (data-
>difference2[i * 2 * (data->n + 1) + j - 1] - data->difference2[(i + 1) * 2 *
(data->n + 1) + j - 1]) / (data->data_x[i / 2 + data->ind1] - data->data_x[i / 2 +
j / 2 + j % 2 + data->ind1]);
    }
    else
        printf("Memory allocate error!\n");
}

void find_polynom_newton(struct polynom *data)
{
    data->y_newton = data->data_y[data->ind1];
    float buf_x = 1;
    for (int i = 1; i < data->n + 1; i++)
    {
        buf_x *= data->difference1[(i - 1) * (data->n + 1)];
        data->y_newton += buf_x * data->difference1[i];
    }
}
```

```
void find_polynom_armit(struct polynom *data)
{
    data->y_armit = data->data_y[data->ind1];
    float buf_x = 1;
    for (int i = 1; i < 2 * (data->n + 1); i++)
    {
        buf_x *= data->difference2[(i - 1) / 2 * 2 * (data->n + 1)];
        data->y_armit += buf_x * data->difference2[i];
    }
}

int check_monotony(struct polynom data)
{
    int res = 0;
    if (data.data_y[0] < 0)
        res = -1;
    else
        res = 1;
    for (int i = 0; i < data.count - 1; i++)
        if (res > 0 && data.data_y[i] <= 0)
            res = 2;
        else if (res < 0 && data.data_y[i] >= 0)
            res = -2;
    if (res % 2 != 0)
        res = 0;
    return res;
}

void swap_columns(struct polynom *data)
{
    float buf = 0;
    for (int i = 0; i < data->count; i++)
    {
        buf = data->data_x[i];
        data->data_x[i] = data->data_y[i];
        data->data_y[i] = buf;
    }
}
```

3. Результаты работы

1. Значения y(x) при степенях полиномов Ньютона и Эрмита n = 1, 2, 3, 4 при фиксированном x. Результаты свести в таблицу для сравнения полиномов.

За основу возьмем данные, указанные в исходных. X примем за 0.6217.

Полином	*n = 1*	*n = 2*	*n = 3*	*n = 4*
Ньютона	0.190089	0.191235	0.191186	0.191194
Эрмита	0.191190	0.191190	0.191191	0.191191

2. Найти корень заданной выше табличной функции с помощью обратной интерполяции, используя полином Ньютона.

Степень	*n = 1*	*n = 2*	*n = 3*	*n = 4*
Корень	0.738728	0.739046	0.739095	0.739088

4. Вопросы при защите лабораторной работы

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

При n = 0 не будет работать, так как установлено ограничение от 1 до 4, однако такой полином можно будет построить — таким образом при расчете значения функции мы получим константу — табличное значение у наиболее близкого к заданному x.

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

Погрешность можно оценить по формуле

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

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

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

Однако, трудность использования теоретической оценки состоит в том, что производные интерполируемой функции обычно неизвестны, поэтому для определения погрешности удобнее воспользоваться оценкой первого отброшенного члена.

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

Минимальная степень полинома будет 3, так как мы имеем 4 узла — x_0, x_0, x_1, x_1.

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

При построении конфигурации узлов немаловажным является упорядоченность аргумента функции. Если он убывает или возрастает, поиск конфигурации сводится к выбору аргументов около заданного x.

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

Выравнивающие переменные являются производными функциями от исходных переменных, с их использованием можно добиться того, чтобы в новых переменных график был близок к прямой хотя бы на отдельных участках, что значительно упрощает вычисление для быстроменяющихся функций. Чтобы произвести интерполяцию необходимо задать новые n = n(y), e = e(x), провести интерполяцию по переменным (n, e), а затем обратным интерполированием найти $y_i = y(n_i)$.