	Министерство образования и науки 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 (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

Лабораторная работа №1

Тема: Построение и программная реализация алгоритма полиномиальной интерполяции табличных функций.

Студент Сучкова Т.М.
Группа ИУ7-42Б
Оценка (баллы) _____________
Преподаватель Градов В.М.

Москва
2021 г.

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

Исходные данные.

1. Таблица функции и её производных

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.

3. Значение аргумента, для которого выполняется интерполяция.

Алгоритм

Полином Ньютона.
Удобно использовать для практических вычислений.
Вводится понятие разделенных разностей функции y(x), заданной в узлах xi:

$$y(x_i,x_j)=[y(x_i)-y(x_j)]/(x_i-x_j),$$

$$y(x_i,x_j,x_k)=[y(x_i,x_j)-y(x_j,x_k)]/(x_i-x_k), \qquad (1.5)$$

$$y(x_i,x_j,x_k,x_l)=[y(x_i,x_j,x_k)-y(x_j,x_k,x_l)]/(x_i-x_l),$$

Интерполяционный многочлен Ньютона:

$$y(x)\approx y(x_0)+\sum_{k=1}^{n}(x-x_0)(x-x_1)..(x-x_{k-1})\; y(x_0,...,x_k). \qquad (1.7)$$

Алгоритм построения полинома Ньютона:
В качестве исходных данных задаются: таблично интерполируемая функция, значение аргумента x, для которого выполняется интерполяция, и степень полинома.
1. Формируется конфигурация из (n+1)-го узлов, по возможности симметрично расположенных относительно значения x (понятно, что вблизи краев таблицы это невозможно).
2. На сформированной конфигурации узлов строится вспомогательная таблица, аналогичная той, которая представлена в вышеприведенном примере.
3. Строится полином Ньютона с разделенными разностями, взятыми из верхней строки таблицы.

Обратная интерполяция применяется для приближенного нахождения корня монотонных функций.

Полином Эрмита.
В практике вычислений находит применение еще один полином, называемый полиномом Эрмита. Его используют, если в узлах задана не только функция, но и ее производные различного порядка.

В общем случае полином Эрмита:

$$H_n(x) = P_n(x, \underbrace{x_0, x_0, \ldots, x_0}_{n_0}, x_1, x_1, \ldots, x_1, x_2, x_2, \ldots, x_2, \underbrace{x_k, x_k, \ldots, x_k}_{n_k}),$$

$$\sum_{l=0}^{k} n_l = n + 1, \qquad (1.9)$$

При кратности узлов не выше двух формулы для разделенных разностей получают предельным переходом:

$$y(x_0, x_0) = \lim_{x_1 \to x_0} \frac{y(x_0) - y(x_1)}{x_0 - x_1} = y'(x_0),$$

$$y(x_0, x_0, x_1) = \frac{y(x_0, x_0) - y(x_0, x_1)}{x_0 - x_1} = \frac{y'(x_0) - y(x_0, x_1)}{x_0 - x_1},$$

$$y(x_0, x_0, x_1, x_1) = \frac{y(x_0, x_0, x_1) - y(x_0, x_1, x_1)}{x_0 - x_1} = \frac{y'(x_0) - 2y(x_0, x_1) + y'(x_1)}{(x_0 - x_1)^2}.$$

Выражения для разделенных разностей в случае узлов кратности выше второй удобнее находить дифференцированием полинома Ньютона. В итоге общее выражение для разделенных разностей при кратности узлов m имеет вид:

$$\underbrace{y(x_0, x_0, \ldots, x_0)}_{m} = \frac{1}{(m-1)!} y^{(m-1)}(x_0).$$

Код программы

__**main.c:**__

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

```
#define OK 0
#define ERR -1

typedef struct point
{
    double x;
    double y;
    double y_der;
} point_t;

double f(double x)
{
    return cos(x) - x;
}

// Initial values table
int initial_table_create(point_t **table, int n)
{
    *table = malloc(n * sizeof(point_t));
    if (!*table)
    {
        return ERR;
    }

    for (int i = 0; i < n; i++)
    {
        printf("Input %d point data (x y y'): ", i + 1);
        scanf("%lf%lf%lf", &((*table)[i].x), &((*table)[i].y), &((*table)[i].y_der));
    }

    return OK;
}

void initial_table_print(point_t *table, int n)
{
    printf("\n%10s | %10s | %10s\n", "X", "Y", "Y'");

    for (int i = 0; i < n; i++)
    {
        printf("%10.6lf | %10.6lf | %10.6lf\n", table[i].x, table[i].y, table[i].y_der);
    }
}

void points_sort_x(point_t *table, int n)
{
    int idx = 0;
    double val = 0.0;
    point_t cur_dot = {0, 0, 0};

    for (int i = 1; i < n; i++)
    {
        val = table[i].x;
```

```
        idx = i - 1;
        cur_dot = table[i];

        while (idx >= 0 && table[idx].x > val)
        {
            table[idx + 1] = table[idx];
            idx--;
        }

        table[idx + 1] = cur_dot;
    }
}

int min_point_i(point_t *points, int n_init, double x)
{
    int min_i = 0, min;
    int flag = 0;

    for (int i = 0; i < n_init; i++)
    {
        if (!flag)
        {
            min = abs(points[i].x - x);
            flag = 1;
        }
        else if (abs(points[i].x - x) < min)
        {
            min = abs(points[i].x - x);
            min_i = i;
        }
    }

    return min_i;
}

// Points selection
point_t *choose_points(point_t *points, int n_init, double x, int n)
{
    point_t *selected_points = malloc(n * sizeof(point_t));
    if (!selected_points)
    {
        return NULL;
    }

    int i_beg;
    int n_required = ceil(n / 2);
    int i_near = min_point_i(points, n_init, x);

    if (i_near + n_required + 1 > n_init)
    {
        i_beg = n_init - n;
    }
```

```
    else if (i_near < n_required)
    {
        i_beg = 0;
    }
    else
    {
        i_beg = i_near - n_required + 1;
    }

    for (int i = 0; i < n; i++)
    {
        selected_points[i] = points[i_beg + i];
    }

    return selected_points;
}

// Divided difference matrix
void dif_matrix_free(double **dif_matrix, int n)
{
    for (int i = 0; i < n; i++)
    {
        free(dif_matrix[i]);
    }

    free(dif_matrix);
}

int dif_matrix_allocate(double ***dif_matrix, int n)
{
    *dif_matrix = calloc(n, sizeof(double *));
    if (!*dif_matrix)
    {
        return ERR;
    }

    for (int i = 0; i < n; i++)
    {
        (*dif_matrix)[i] = malloc((n - i) * sizeof(double));
        if (!(*dif_matrix)[i])
        {
            dif_matrix_free(*dif_matrix, i);
            return ERR;
        }
    }

    return OK;
}

void arr_null(double *arr, int n)
{
    for (int  i = 0; i < n; i++)
```

```
    {
        arr[i] = 0;
    }
}
```

void arr_copy(double *destination, double *source, int n)

```
{
    for (int i = 0; i < n; i++)
    {
        destination[i] = source[i];
    }
}
```

int dif_matrix_fill(double **dif_matrix, point_t *points, int n, int hermit_flag)

```
{
    double *tmp_arr = malloc(n *sizeof(double));
    if (!tmp_arr)
    {
        return ERR;
    }

    int cur_j = 0, cur_count;

    for (int i = 0; i < n; i++)
    {
        arr_null(tmp_arr, n);

        for (int j = 0; j < n - i; j++)
        {
            if (i == 0)
            {
                if (!hermit_flag)
                    tmp_arr[j] = (points[j].y - points[j + 1].y) /
                                        (points[j].x - points[j + i + 1].x);
                else
                {
                    cur_j = j / 2;

                    if (j % 2 == 0)
                    {
                        tmp_arr[j] = points[cur_j].y_der;
                    }
                    else
                    {
                        tmp_arr[j] = (points[cur_j].y - points[cur_j + 1].y) /
                                        (points[cur_j].x - points[cur_j + i + 1].x);
                    }
                }

            }
            else
            {
```

```
            if (!hermit_flag)
            {
                tmp_arr[j] = (dif_matrix[i - 1][j] - dif_matrix[i - 1][j + 1]) /
                             (points[j].x - points[j + i + 1].x);
            }
            else
            {
                tmp_arr[j] = (dif_matrix[i - 1][j] - dif_matrix[i - 1][j + 1]) /
                             (points[j / 2].x - points[(j + i - 1) / 2 + 1].x);
            }
        }
    }

    if (!hermit_flag)
        cur_count = n - i - 1;
    else
        cur_count = n - i;

    arr_copy(dif_matrix[i], tmp_arr, cur_count);
  }

  free(tmp_arr);

  return OK;
}
```

int newton_interpolation(point_t *points, int n_init, double x, int n, double *result)

```
{
  point_t *selected_points = calloc(n + 1, sizeof(point_t));
  selected_points = choose_points(points, n_init, x, n + 1);
  if (!selected_points)
  {
    return ERR;
  }

  double **dif_matrix;
  dif_matrix_allocate(&dif_matrix, n);
  dif_matrix_fill(dif_matrix, selected_points, n + 1, 0);

  double cur_k = 1;

  for (int i = 0; i < n + 1; i++)
  {
    if (i == 0)
    {
      *result += selected_points[i].y;
    }
    else
    {
      *result += cur_k * dif_matrix[i - 1][0];
    }
```

```
        cur_k *= x - selected_points[i].x;
    }

    free(selected_points);
    dif_matrix_free(dif_matrix, n);

    return OK;
}
```

int hermit_interpolation(point_t *points, int n_init, double x, int n, double *result)

```
{
    point_t *selected_points = calloc(n + 1, sizeof(point_t));
    selected_points = choose_points(points, n_init, x, n + 1);
    if (!selected_points)
    {
        return ERR;
    }

    double **dif_matrix;
    dif_matrix_allocate(&dif_matrix, 2 * n - 1);
    dif_matrix_fill(dif_matrix, selected_points, 2 * n - 1, 1);

    double cur_k = 1;

    for (int i = 0; i < 2 * n - 1; i++)
    {
        if (i == 0)
        {
            *result += selected_points[i].y;
        }
        else
        {
            *result += cur_k * dif_matrix[i - 1][0];
        }

        cur_k *= x - selected_points[i / 2].x;
    }

    free(selected_points);
    dif_matrix_free(dif_matrix, n);

    return OK;
}
```

void change_y_x(point_t *points, int n_init)

```
{
    double tmp;

    for (int i = 0; i < n_init; i++)
    {
        tmp = points[i].x;
        points[i].x = points[i].y;
```

```
        points[i].y = tmp;
    }
}
```

int cmp_table_print(point_t *initial_table, int n_init, double x, int n_max)

```
{
    double cur_res_1 = 0.0, cur_res_2 = 0.0;
    int n_cur = 1;

    if (n_max == 0)
    {
        n_cur = 0;
        n_max = 1;
    }

    printf("\n|%3s |%27s |%27s |\n", "N", "Newton intorpolated value", "Hermit intorpolated value");

    for (int i = 0; i < n_max; i++)
    {
        cur_res_1 = 0.0;
        cur_res_2 = 0.0;

        if (n_cur != 0)
        {
            n_cur = i + 1;
        }

        if (newton_interpolation(initial_table, n_init, x, n_cur, &cur_res_1) != OK)
        {
            free(initial_table);
            return ERR;
        }

        if (hermit_interpolation(initial_table, n_init, x, n_cur, &cur_res_2) != OK)
        {
            free(initial_table);
            return ERR;
        }

        printf("|%3d |", i +1);
        printf("%27lf|", cur_res_1);
        printf("%27lf|\n", cur_res_2);
    }

    return OK;
}
```

int main(void)

```
{
    int n_init = 0;
    setbuf(stdout, NULL);
```

```
printf("\nINITIAL TABLE DATA\n");

printf("Input dots amount: ");
if (scanf("%d", &n_init) != 1 || n_init < 1)
{
    return ERR;
}

point_t *initial_table;
if (initial_table_create(&initial_table, n_init) != OK)
{
    return ERR;
}

//initial_table_print(initial_table, n_init);
points_sort_x(initial_table, n_init);
//initial_table_print(initial_table, n_init);

// Task conditions
int n;
double x;

printf("\nInput n: ");
if (scanf("%d", &n) != 1 || n < 0)
{
    free(initial_table);
    return ERR;
}

printf("Input x: ");
if (scanf("%lf", &x) != 1)
{
    free(initial_table);
    return ERR;
}

// Task solution
double root = 0;

cmp_table_print(initial_table, n_init, x, n);

change_y_x(initial_table, n_init);
if (newton_interpolation(initial_table, n_init, 0, n, &root) != OK)
{
    free(initial_table);
    return ERR;
}

printf("\ny(x)              : %lf", f(x));
printf("\nThis function root: %lf", root);

free(initial_table);
```

```
    return OK;
}
```

Результаты работы

1. Значения y(x) при степенях полиномов Ньютона и Эрмита n= 1, 2, 3 и 4 при фиксированном x, например, x=0.525 (середина интервала 0.45- 0.60). Результаты свести в таблицу для сравнения полиномов

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

```
INITIAL TABLE DATA
Input dots amount: 8
Input 1 point data (x y y'): 0.00 1.000000 -1.000000
Input 2 point data (x y y'): 0.15 0.838771 -1.14944
Input 3 point data (x y y'): 0.30 0.655336 -1.29552
Input 4 point data (x y y'): 0.45 0.450447 -1.43497
Input 5 point data (x y y'): 0.60 0.225336 -1.56464
Input 6 point data (x y y'): 0.75 -0.018310 -1.68164
Input 7 point data (x y y'): 0.90 -0.278390 -1.78333
Input 8 point data (x y y'): 1.05 -0.552430 -1.86742

Input n: 4
Input x: 0.525

|  N |  Newton intorpolated value|  Hermit intorpolated value|
|  1 |                  0.435699|                  1.000000|
|  2 |                  0.338547|                  0.337445|
|  3 |                  0.340192|                  0.340339|
|  4 |                  0.340324|                  0.340320|

y(x)               : 0.340324
This function root: 0.735955
```

Вопросы при защите лабораторной работы.

Ответы на вопросы дать письменно в Отчете о лабораторной работе.

1. Будет ли работать программа при степени полинома n=0?
Программа работать будет, но вычисления будут очень неточными, т. к. не хватает данных для точных вычислений (при нулевой степени полинома).

Input n: 0
Input x: 0.525

N	Newton intorpolated value	Hermit intorpolated value
0	0.838771	0.000000

y(x) : 0.340324
This function root: 0.300000

2. Как практически оценить погрешность интерполяции? Почему сложно применить для этих целей теоретическую оценку?
Погрешность многочлена Ньютона можно оценить по формуле

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

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

а полином

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

Трудность использования указанных теоретических оценок на практике состоит в том, что производные интерполируемой функции обычно неизвестны, поэтому для определения погрешности **на практике** 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 на этих точках?

В этом случае мы имеем 4 условия (заданы 2 значения функции и 2 значения её производной), значит, полином будет иметь ***3-ю степень***. Формально строим полином Ньютона по четырем узлам, каждый из которых повторяется дважды.

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

Эта информация существенна для выбора точек, работа с которыми будет произведена при интерполяции. Следует брать (n+1) узлов, которые окружают x. Желательно, чтобы значение x находилось посередине интервала первой и последней из выбранных точек. Если такое невозможно, сдвигаются в одну из сторон, в зависимости оттого, сверху или снизу x не хватает точек.

Чтобы x оказался в середине интервала (при достаточном количестве точек нужного диапазона), выполняется сортировка значений по аргументу функции, чтобы было удобнее осуществлять отбор нужных для интервала точек.

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

Выравнивающие переменные: $\eta = \eta(y)$ и $\xi = \xi(x)$

В каждом конкретном случае приходится специально подбирать вид этих функций.

Преобразованием этих переменных можно добиться того, чтобы в новых переменных график $\eta(\xi)$ был близок к прямой хотя бы на отдельных участках.

В этом случае интерполяцию проводят в переменных (η, ξ),

а затем обратным интерполированием находят $y_i = y(\eta_i)$.