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 № 3

Тема: Построение и программная реализация алгоритма сплайн-интерполяции табличных функций.

Студент Тартыков Л.Е.

Группа ИУ7-44Б

Оценка (баллы) _____

Преподаватель Градов В.М.

2021 г.

Содержание

Цель работы: Получение навыков владения методами интерполяции таблично заданных функций с помощью кубических сплайнов.

Исходные данные

1. Таблица функции с количеством узлов N, заданная с помощью формулы $y = x^2$ в диапазоне [0..10] с шагом 1.
2. Значение аргумента x в первом интервале при x=0.5 и в середине таблицы при x= 5.5.

Код программы

Замечание: данная программа написана на языке Си (стандарт C99).

Листинг 1; main.c

```
#include <stdio.h>
#include "../inc/defines.h"
#include "../inc/struct.h"
#include "../inc/table.h"
#include "../inc/spline.h"

int main(void)
{
   int error = ERR_OK;
   double input_x = 0;

   table_t *table;
   spline_t *table_spline;
   add_coeff_t *spline_add_coeff;

   table = init_table();

   error = read_table(&table);

   if (error == ERR_OK)
      read_input_x(&input_x);

   if (error == ERR_OK)
   {
      double result = 0;
      table_spline = init_spline_coeffs();
      spline_add_coeff = init_spline_add_coeffs();
      error = spline_table(&result, &table_spline, &spline_add_coeff, table, input_x);

      if (error == ERR_OK)
         printf("Результат = %lf\n", result);
      free_table_spline(&table_spline);
   }
   return error;
}
```

Листинг 2; table.c

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

#include "../inc/table.h"
#include "../inc/struct.h"
#include "../inc/defines.h"

void read_amount_nodes(int *count);
int allocate_table(table_t **table);
```

```
void create_table(table_t **table);
void free_table(table_t **table);
void free_table_x(int **table_x);
void free_table_y(int **table_y);

table_t *init_table(void)
{
   table_t *table = NULL;
   table = malloc(sizeof(table));
   table->table_x = NULL;
   table->table_y = NULL;
   table->count = 0;

   return table;
}

int read_table(table_t **table)
{
   int error = ERR_OK;
   table_t *temp_table;
   temp_table = init_table();

   read_amount_nodes(&temp_table->count);
   error = allocate_table(&temp_table);

   if (error == ERR_OK)
      create_table(&temp_table);

   if (error == ERR_OK)
      *table = temp_table;
   return error;
}

void read_amount_nodes(int *count)
{
   printf("Введите количество узлов: ");
   while (scanf("%d", count) != 1)
   {
      scanf("%*[^\n]");
      printf("Некорректный ввод. Введите еще раз: ");
   }
}

void read_input_x(double *input_x)
{
   printf("Введите аргумент полинома: ");
   while (scanf("%lf", input_x) != 1)
   {
      scanf("%*[^\n]");
      printf("Некорректный ввод. Введите еще раз: ");
   }
```

```
}

int allocate_table(table_t **table)
{
   int error = ERR_OK;
   (*table)->table_x = malloc((*table)->count * sizeof(int));

   if (!(*table)->table_x)
   {
      free_table_x(&(*table)->table_x);
      error = ERR_ALLOCATE;
   }

   if (error == ERR_OK)
   {
      (*table)->table_y = malloc((*table)->count * sizeof(int));
      if (!(*table)->table_y)
      {
         free_table_y(&(*table)->table_y);
         error = ERR_ALLOCATE;
      }
   }
   return error;
}

void create_table(table_t **table)
{
   for (int i = 0; i < (*table)->count; i++)
   {
      (*table)->table_x[i] = i;
      (*table)->table_y[i] = i * i;
   }
}

void print_table(table_t *table)
{
   for (int i = 0; i < (*table).count; i++)
   {
      printf("%d %d\n", (*table).table_x[i], (*table).table_y[i]);
   }
}

void free_table(table_t **table)
{
   if (*table)
   {
      free_table_x(&(*table)->table_x);
      free_table_y(&(*table)->table_y);
      free(*table);
   }
}
```

```
void free_table_x(int **table_x)
{
   if (*table_x)
      free(*table_x);
}

void free_table_y(int **table_y)
{
   if (*table_y)
      free(*table_y);
}
```

Листинг 3; spline.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "../inc/spline.h"
#include "../inc/struct.h"
#include "../inc/defines.h"
#include "../inc/table.h"

void straight_walk(spline_t **table_spline, table_t *table);
void reverse_walk(spline_t **table_spline, int count);
void calculate_add_coeffs(add_coeff_t **spline_add_coeff, spline_t *table_spline, table_t *table);

void calculate_dx(double **h, int *table_x, int count);
double find_result(add_coeff_t *spline_add_coeff, double *spline_c, table_t *table, double input_x);
int find_near_x(int *table_x, double input_x, int count);

int allocate_spline(spline_t **table_spline, int count_nodes);
int allocate_spline_add_coeff(add_coeff_t **spline_add_coeff, int count_nodes);
void free_spline_c(double **table_spline_c);
void free_spline_eta(double **table_spline_eta);
void free_spline_ksi(double **table_spline_ksi);
void free_spline_f(double **table_spline_f);



spline_t *init_spline_coeffs(void)
{
   spline_t *spline = NULL;
   spline = malloc(sizeof(spline_t));
   spline->c = NULL;
   spline->ksi = NULL;
   spline->eta = NULL;
   spline->f = NULL;
   spline->h = 0;

   return spline;
```

```
}

add_coeff_t *init_spline_add_coeffs(void)
{
   add_coeff_t *spline_add = NULL;
   spline_add = malloc(sizeof(add_coeff_t));
   spline_add->a = NULL;
   spline_add->b = NULL;
   spline_add->d = NULL;

   return spline_add;
}

int spline_table(double *result, spline_t **table_spline, add_coeff_t **spline_add_coeff, table_t *table,
double input_x)
{
   int error = ERR_OK;
   error = allocate_spline(table_spline, (*table).count);
   if (error == ERR_OK)
      error = allocate_spline_add_coeff(spline_add_coeff, (*table).count);

   if (error == ERR_OK)
   {
      straight_walk(table_spline, table);
      reverse_walk(table_spline, table->count);
      calculate_add_coeffs(spline_add_coeff, *table_spline, table);
      *result = find_result(*spline_add_coeff, (*table_spline)->c, table, input_x);
   }
   return error;
}

int allocate_spline(spline_t **table_spline, int count_nodes)
{
   int error = ERR_OK;
   (*table_spline)->c = (double *)calloc(count_nodes, sizeof(double));

   if (!(*table_spline)->c)
   {
      free_spline_c(&(*table_spline)->c);
      error = ERR_ALLOCATE;
   }
   else
   {
      (*table_spline)->eta = (double *)calloc(count_nodes, sizeof(double));
      if (!(*table_spline)->eta)
      {
         free_spline_eta(&(*table_spline)->eta);
         error = ERR_ALLOCATE;
      }
      else
      {
```

```
        (*table_spline)->ksi = (double *)calloc(count_nodes, sizeof(double));
        if (!(*table_spline)->ksi)
        {
          free_spline_ksi(&(*table_spline)->ksi);
          error = ERR_ALLOCATE;
        }
      }
    }

    if (error == ERR_OK)
    {
      (*table_spline)->f = (double *)calloc(count_nodes, sizeof(double));
      if (!(*table_spline)->f)
      {
        free_spline_f(&(*table_spline)->f);
        error = ERR_ALLOCATE;
      }
    }

    if (error == ERR_OK)
    {
      (*table_spline)->h = (double *)calloc(count_nodes, sizeof(double));
      if (!(*table_spline)->h)
      {
        free_spline_f(&(*table_spline)->h);
        error = ERR_ALLOCATE;
      }
    }

    return error;
}

int allocate_spline_add_coeff(add_coeff_t **spline_add_coeff, int count_nodes)
{
    int error = ERR_OK;
    (*spline_add_coeff)->a = (double *)calloc(count_nodes, sizeof(double));
    if (!(*spline_add_coeff)->a)
    {
      free_spline_f(&(*spline_add_coeff)->a);
      error = ERR_ALLOCATE;
    }

    if (error == ERR_OK)
    {
      (*spline_add_coeff)->b = (double *)calloc(count_nodes, sizeof(double));
      if (!(*spline_add_coeff)->b)
      {
        free_spline_f(&(*spline_add_coeff)->b);
        error = ERR_ALLOCATE;
      }
    }
```

```
    if (error == ERR_OK)
    {
        (*spline_add_coeff)->d = (double *)calloc(count_nodes, sizeof(double));
        if (!(*spline_add_coeff)->d)
        {
            free_spline_f(&(*spline_add_coeff)->d);
            error = ERR_ALLOCATE;
        }
    }

    return error;
}

void free_table_spline(spline_t **table_spline)
{
    if (table_spline)
    {
        free_spline_c(&(*table_spline)->c);
        free_spline_eta(&(*table_spline)->eta);
        free_spline_ksi(&(*table_spline)->ksi);
        free_spline_f(&(*table_spline)->f);
        free(*table_spline);
    }
}

void free_spline_add_coeff(add_coeff_t **spline_add_coeff)
{
    if (*spline_add_coeff)
    {
        free_spline_f(&(*spline_add_coeff)->a);
        free_spline_f(&(*spline_add_coeff)->b);
        free_spline_f(&(*spline_add_coeff)->d);
        free(*spline_add_coeff);
    }
}

void free_spline_c(double **table_spline_c)
{
    if (table_spline_c)
    {
        free(*table_spline_c);
        *table_spline_c = NULL;
    }
}

void free_spline_eta(double **table_spline_eta)
{
    if (table_spline_eta)
    {
        free(*table_spline_eta);
```

```
        *table_spline_eta = NULL;
    }
}

void free_spline_ksi(double **table_spline_ksi)
{
    if (table_spline_ksi)
    {
        free(*table_spline_ksi);
        *table_spline_ksi = NULL;
    }
}

void free_spline_f(double **table_spline_f)
{
    if (table_spline_f)
    {
        free(*table_spline_f);
        *table_spline_f = NULL;
    }
}

void straight_walk(spline_t **table_spline, table_t *table)
{
    calculate_dx(&(*table_spline)->h, table->table_x, table->count);
    for (int i = 3; i < table->count; i++)
    {
        (*table_spline)->ksi[i] = -(*table_spline)->h[i - 1] / ((*table_spline)->h[i - 2] * (*table_spline)->ksi[i - 1]
+
                                2 * ((*table_spline)->h[i - 2] + (*table_spline)->h[i - 1]));
        (*table_spline)->f[i - 1] = 3 * ((table->table_y[i - 1] - table->table_y[i - 2]) / (*table_spline)->h[i - 1] -
                    (table->table_y[i - 2] - table->table_y[i - 3]) / (*table_spline)->h[i - 2]);
        (*table_spline)->eta[i] = ((*table_spline)->f[i - 1] - (*table_spline)->h[i - 2] * (*table_spline)->eta[i - 1])
/
                    ((*table_spline)->h[i - 2] * (*table_spline)->ksi[i - 1] + 2 * ((*table_spline)->h[i - 2] +
(*table_spline)->h[i - 1]));
    }
}

void calculate_dx(double **h, int *table_x, int count)
{
    for (int i = 0; i < count; i++)
    {
        (*h)[i] = table_x[i + 1] - table_x[i];
    }
}

void reverse_walk(spline_t **table_spline, int count)
{
    for (int i = count - 2 ; i >= 0; i--)
    {
```

```
      (*table_spline)->c[i] = (*table_spline)->ksi[i + 1] * (*table_spline)->c[i + 1] + (*table_spline)->eta[i +
1];
   }
}

void calculate_add_coeffs(add_coeff_t **spline_add_coeff, spline_t *table_spline, table_t *table)
{
   for (int i = 1 ; i < table->count - 1; i++)
   {
      (*spline_add_coeff)->a[i] = table->table_y[i - 1];
      (*spline_add_coeff)->b[i] = (table->table_y[i] - table->table_y[i - 1]) / table_spline->h[i] -
                        (table_spline->h[i] / 3 * (table_spline->c[i + 1] + 2 * table_spline->c[i]));
      (*spline_add_coeff)->d[i] = (table_spline->c[i + 1] - table_spline->c[i]) / (3 * table_spline->h[i]);
   }

   (*spline_add_coeff)->b[table->count - 1] = (table->table_y[table->count - 1] - table->table_y[table->count
- 2] / table_spline->h[table->count - 1]) -
                        (table_spline->h[table->count - 1] * (2 * table_spline->c[table->count - 1]) / 3);
   (*spline_add_coeff)->d[table->count - 1] = -(table_spline->c[table->count - 1]) / (3 * table_spline->h[table-
>count - 1]);
}

double find_result(add_coeff_t *spline_add_coeff, double *spline_c, table_t *table, double input_x)
{
   double result = 0;
   int i_near = find_near_x(table->table_x, input_x, table->count);
   result = spline_add_coeff->a[i_near] + spline_add_coeff->b[i_near] * (input_x - table->table_x[i_near - 1])
                  + spline_c[i_near] * pow((input_x - table->table_x[i_near - 1]), 2)
                  + spline_add_coeff->d[i_near] * pow((input_x - table->table_x[i_near - 1]), 3);
   return result;
}

int find_near_x(int *table_x, double input_x, int count)
{
   int i_near = 0;
   double min = fabs(table_x[0] - input_x);
   double delta = 0;
   for (int i = 1; i < count; i++)
   {
      delta = fabs(table_x[i] - input_x);
      if (delta < min)
      {
         i_near = i;
         min = delta;
      }
   }

   return i_near;
}
}
```

Листинг 4; defines.h

```
#ifndef DEFINES_H
#define DEFINES_H

#define ERR_OK 0
#define ERR_ALLOCATE 1

#endif
```

Листинг 5; spline.h

```
#ifndef SPLINE_H
#define SPLINE_H

#include "../inc/struct.h"

spline_t *init_spline_coeffs(void);
add_coeff_t *init_spline_add_coeffs(void);

int spline_table(double *result, spline_t **table_spline, add_coeff_t **spline_add_coeff, table_t *table,
double input_x);
void free_table_spline(spline_t **table_spline);
void free_spline_add_coeff(add_coeff_t **spline_add_coeff);

#endif
```

Листинг 6; struct.h

```
#ifndef STRUCT_H
#define STRUCT_H

typedef struct table table_t;
struct table
{
   int *table_x;
   int *table_y;
   int count;
};

typedef struct spline spline_t;
struct spline
{
   double *c;
   double *ksi;
   double *eta;
   double *f;
   double *h;
};

typedef struct add_coeff add_coeff_t;
struct add_coeff
```

```
{
   double *a;
   double *b;
   double *d;
};

#endif
```

Листинг 7; read.h

```
#ifndef TABLE_H
#define TABLE_H

#include "../inc/struct.h"

table_t *init_table(void);
int read_table(table_t **table);
void read_input_x(double *input_x);
void print_table(table_t *table);

#endif
```

Результаты работы

Значение функции «y» для аргумента:
Кубический сплайн:
1. x = 0.5; y = 0.0
2. x = 5.5; y = 42.261321

Полином Ньютона третьей степени:
1. x = 0.5; y = 0.25
2. x = 5.5; y = 42.25

Точное значение:
1. x = 0.5; y = 0.25
2. x = 5.5; y = 42.25

Ответы на вопросы при защите лабораторной работы

1. Получить выражения для коэффициентов кубического сплайна, построенного на двух точках.

Сплайн, построенный на двух точках, есть прямая линия. Тогда коэффициенты «c» и «d» будут равны нулю. Следовательно, коэффициенты «a» и «b» примут следующий вид:

$a_i = y_{i-1}$

$b = (y_i — y_{i-1}) / h_i$

$0 <= i <= 1$

2. Выписать все условия для определения коэффициентов сплайна, построенного на 3-х точках.

Сплайн по трем точкам будет состоять из двух интервалов, следовательно, из двух функций (ϕ_1, ϕ_2).

Таким образом, необходимо восемь условий:

$\phi_1(x_1)=y_1$ $\quad$ $\phi'_1(x_2)=\phi'_2(x_2)$

$\phi_1(x_2)=y_1$ $\quad$ $\phi''_1(x_2)=\phi''_2(x_2)$

$\phi_1(x_2)=y_2$ $\quad$ $\phi''_1(x_1)=0$

$\phi_1(x_3)=y_2$ $\quad$ $\phi''_2(x_3)=0$

3. Определить начальные значения прогоночных коэффициентов, если принять, что для коэффициентов сплайна справедливо $C_1 = C_2$.

Воспользуемся формулой $c_i = \xi_{i+1} * c_{i+1} + \eta_{i+1}$. Получим выражением для C1

$c_1 = \xi_2 * c_2 + \eta_2$

$c_1 = c_2 \Rightarrow c_1 = \xi_2 * c_1 + \eta_2 \Rightarrow \xi_2 = 1;\ \eta_2 = 0$

4. Написать формулу для определения последнего коэффициента сплайна C_N , чтобы можно было выполнить обратный ход метода прогонки, если в качестве граничного условия задано $kC_{N-1} + mC_N = p$, где k,m и p - заданные числа.

Воспользуемся формулой $c_i = \xi_{i+1} * c_{i+1} + \eta_{i+1}$ и преобразуем ее к виду $c_{N-1} = \xi_N * c_N + \eta_N$. (1)

Преобразуем $kC_{N-1} + mC_N = p$ к виду $kC_{N-1} = -mC_N + p$ (2).

Подставим (1) в (2)

$k(\xi_N * C_N + \eta_N) = -m * C_N + p$

Полученное значение $C_N = \dfrac{p - k * \xi_N}{k * \eta_N + m}$