Федеральное государственное бюджетное образовательное учреждение «Сибирский государственный университет телекоммуникаций и информатики»

Кафедра ПМ и К

КУРСОВАЯ РАБОТА

По дисциплине «Вычислительная математика»
Вариант 6

Выполнил:
студент гр. ИВ-621
Дьяченко Д.В.
Проверил:
Чирихин К. С.

Новосибирск, 2018

Постановка задачи

Решить краевую задачу методом Рунге-Кутта II порядка с усреднением по производной.

$$\begin{cases} (y'')^5 - \cos(\mathrm{x}) * y'' = \sin(x) + 5 * \ln(\mathrm{x}) * y' + y * (x + 3) \\ y(0) = 3 \\ y'(1) = 2 \end{cases}$$

Построить графики функции y(x) и кубического сплайна S(x) (интерполяция по точкам x=0; 0.2;0.4; 0.6; 0.8; 1.0). Найти интеграл

$$\int_0^1 y(x)dx$$

Описание алгоритма

Этапы решения краевой задачи:

1. С помощью метода стрельбы находим значение первой производной.

2. Решаем задачу Коши методом Рунге-Кутта II порядка с усреднением по времени.

Метод стрельбы:

Выбираем параметры: a – y(0) из краевой задачи, а b – произвольно; для того, чтобы найти отрезок, в котором будет искомое значение; корректируем исходные параметры в зависимости от перелёта или недолёта (y(a) – y1 > 0 или y(a) – y1 < 0 соответственно). Как только по a перелет, а по b недолет, останавливаем корректировку, на данном этапе отрезок найден. Для нахождения первой производной остается решить нелинейное уравнение любым известным способом, в частности методом бисекции: y(b) = y2, где y(b) – решение задачи Коши.

Задача Коши:

Применяем метод Рунге-Кутта:

$$\begin{cases} y_{i+1} = y_i + \dfrac{h}{2}\left(f(x_i, y_i) + f(x_i + h, y_{i+1}^*)\right) \\ \text{где} \quad y_{i+1}^* = y_i + \dfrac{h}{2} f(x_i, y_i) \end{cases}$$

Так как по условию дано уравнение, которое не может быть разрешено относительно старшей производной, то каждый раз решаем нелинейное уравнение относительно старшей производной.

Интерполяция:

Для вычисления кубического сплайна на заданной сетке будем использовать формулу:

$$S_i(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + \left(y_{i-1} - \frac{M_{i-1}}{6}h_i^2\right)\frac{x_i - x}{h_i} + \left(y_i - \frac{M_i}{6}h_i^2\right)\frac{x - x_{i-1}}{h_i}$$

В данной задаче подразумевается интерполяция естественным кубическим сплайном, т. е. $M0 = Mn$. Шаг для сетки: h = const. Чтобы найти другие Mi составим СЛАУ; получим трехдиагональную матрицу, решаем систему методом прогонки и вычисляем значения сплайна в текущей точке.

Вычисление интеграла:

Численное интегрирование по формуле Симпсона; отрезок [a, b] разбивается на N = 2n частей:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n-1} f(x_{2j}) + \sum_{j=1}^{n} f(x_{2j-1}) + f(x_n)\right],$$

где $h = \frac{b-a}{N}$ – шаг, а $x_j = a + ih$ – узлы интегрирования.

Результат работы программы

Решение краевой задачи:

i	0	1	2	3	4	5
x_i	0.0	0.2	0.4	0.6	0.8	1.0
y_i	3.000	2.660	2.395	2.199	2.069	2.003
y'_i	-1.907	-1.507	-1.149	-0.813	-0.490	-0.176

Интерполяция:

Рисунок 1 Интерполяция Кубическим Сплайном

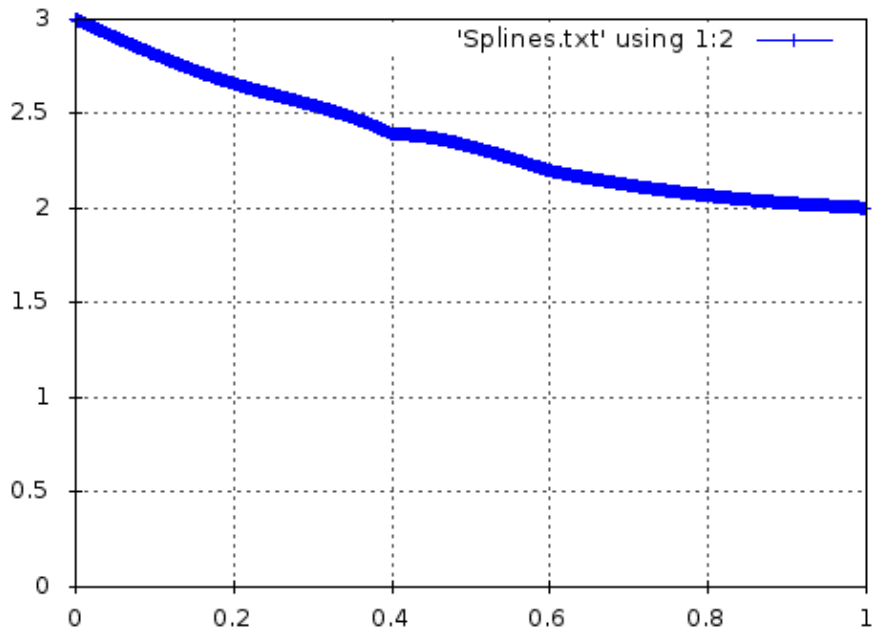


Рисунок 2 Результат функции Рунге-Кутты для y(x)

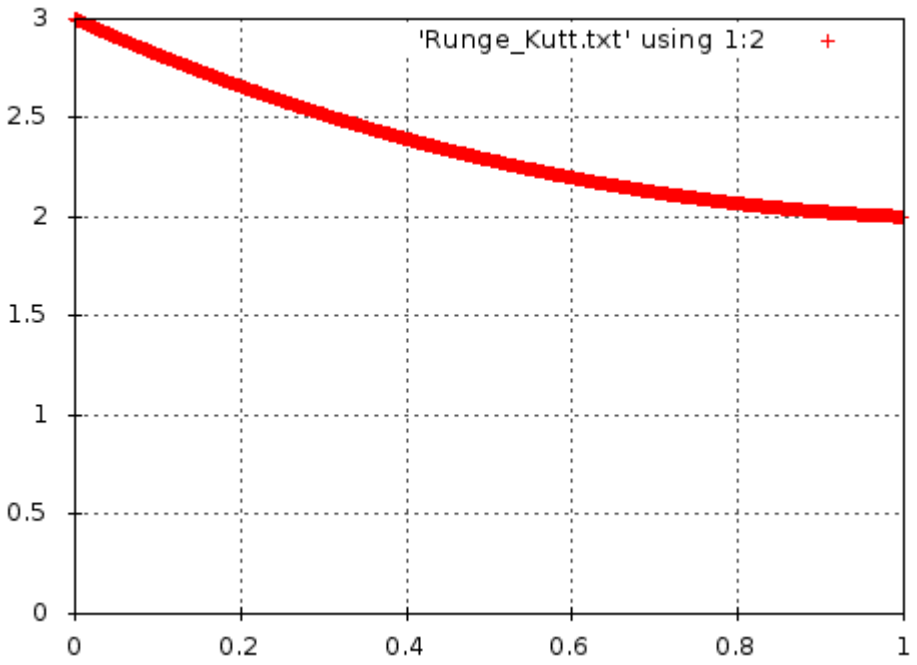
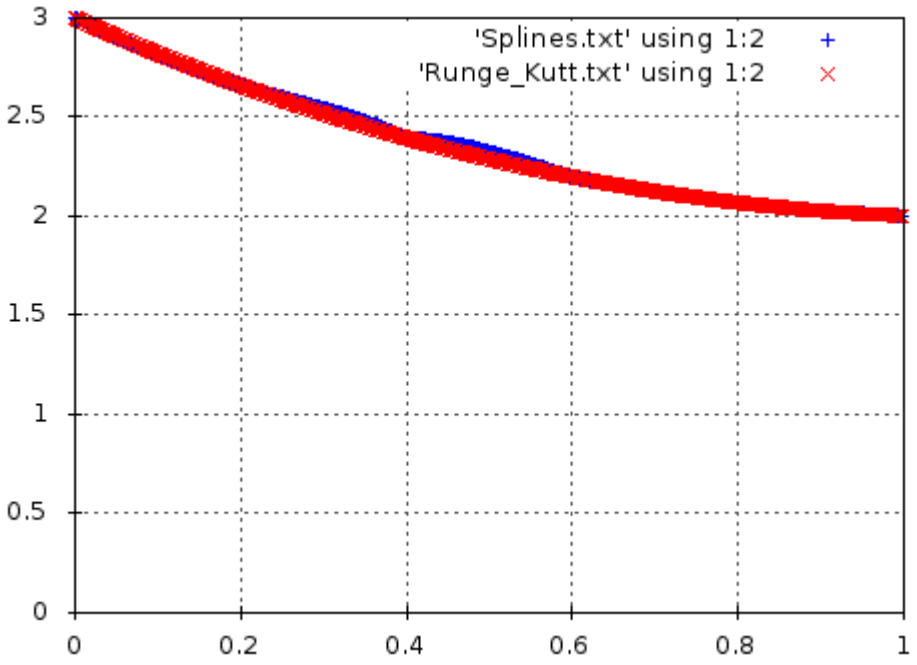


Рисунок 3 Сравнение результата функции Рунге-Кутты для y(x) и интерполяции Кубическим Сплайном



Численное интегрирование:

$$\int_0^1 y(x)dx \approx 2.36764$$

Заключение

В рамках курсовой работы была решена краевая задача, результаты которой удовлетворяют заданным граничным условиям в концах интервала. Проведена интерполяция кубическими сплайнами, построен график сеточной функции, который иллюстрирует решение дифференциального уравнения. По формуле Симпсона вычислено приближенное значение интеграла для заданной подынтегральной функции.

Листинг

main.c

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

double eps = 1e-4;

double diff(double x, double y, double D1, double D2)
{
	if (x == 0) {
		x = 0.0001;
	}
	return pow(D2, 5) - cos(x) * D2 - sin(x) - 5 * log(x) * D1 - y * (x + 3);
}
double *addition_of_vectors(double *v1, double *v2, int n)
{
	double *v3 = malloc(sizeof(double) * n);

	for (int i = 0; i < n; i++) {
		v3[i] = v1[i] + v2[i];
	}

	return v3;
}
double *multiple_dig_by_vector(double a, double *v, int n)
{
	double *v2 = malloc(sizeof(double) * n);

	for (int i = 0; i < n; i++) {
		v2[i] = a * v[i];
	}

	return v2;
}
double *f(double x, double *y)
{
	double a = 0, b = 2;
	double fa = 0, fb = 0;
	do {
		fa = diff(x, y[0], y[1], a--);
		fb = diff(x, y[0], y[1], b++);
	} while (fa * fb > 0);
	double c = 0;
	while(fabs(b - a) >= eps) {
		c = (a + b) / 2;
		if(diff(x, y[0], y[1], a) * diff(x, y[0], y[1], c) < 0)
			b = c;
		else if (diff(x, y[0], y[1], c) * diff(x, y[0], y[1], b) < 0)
```

```
                a = c;
        }
        double *tmp_y = malloc(sizeof(double) * 2);
        tmp_y[0] = y[1];
        tmp_y[1] = (a + b) / 2;
        return  tmp_y;
}
double *Runge_Kutt(double a, double b, double h, double *y0)
{
        double *tmp_y;
        double *y = y0;
        for (double i = a + h; i <= b; i += h) {
                tmp_y = addition_of_vectors(y, multiple_dig_by_vector(h, f(i, y), 2), 2);
                y = addition_of_vectors(y, multiple_dig_by_vector(h / 2, addition_of_vectors(f(i, y), f(i + h, tmp_y), 2), 2),
2);
        }
        return y;
}
double MethodShooting(double x0, double x1, double y0, double y1, double h)
{
        double al = 1.0;
        double bt = 0.0;
        double fa = 0.0;
        double fb = 0.0;
        double tmp[2];
        double *vt;
        do {
                tmp[0] = y0;
                tmp[1] = al;
                vt = Runge_Kutt(x0, x1, h, tmp);
                fa = vt[0] - y1;
                tmp[1] = bt;
                vt = Runge_Kutt(x0, x1, h, tmp);
                fb = vt[0] - y1;
                al -= h;
                bt += h;
        } while (fa * fb > 0);
        double c = 0.0;
        while (fabs(bt - al) >= eps) {
                c = (al + bt) / 2;

                tmp[1] = al;
                double *tmp1 = Runge_Kutt(x0, x1, h, tmp);
                tmp[1] = c;
                double *tmp2 = Runge_Kutt(x0, x1, h, tmp);
                tmp[1] = bt;
                double *tmp4 = Runge_Kutt(x0, x1, h, tmp);

                if ((((tmp1[0] - y1) * (tmp2[0] - y1)) < 0)) {
                        bt = c;
                } else if ((((tmp2[0] - y1) * (tmp4[0] - y1)) < 0)) {
                        al = c;
                }
        }
        return (al + bt) / 2;
}
void set_h(double *h, double *X, int n)
{
        for (int i = 1; i < n; i++)
                h[i] = X[i] - X[i - 1];
}
void set_d(double *d, double *h, double *Y, int n)
{
        for (int i = 1; i < n - 1; i++)
                d[i] = ((Y[i + 1] - Y[i]) / h[i + 1]) - ((Y[i] - Y[i - 1]) / h[i]);
```

```
}
void set_C(double *C, double *h, int n)
{
	for (int i = 1; i < n - 1; i++) {
		for (int j = 1; j < n - 1; j++) {
			if (i == j) {
				C[i * n + j] = (h[i] + h[i + 1]) / 3;
			} else if (j == i + 1) {
				C[i * n + j] = h[i + 1] / 6;
			} else if (j == i - 1) {
				C[i * n + j] = h[i] / 6;
			} else {
				C[i * n + j] = 0;
			}
		}
	}
}
int Matrix_max_first_elem(double *a, int j, int n)
{
	double num = 0;
	int num_ind = 0;
	for (int z = j; z < n; z++) {
		if (fabsf(a[z * (n + 1) + j]) > num) {
			num = a[z * (n + 1) + j];
			num_ind = z;
		}
	}
	return num_ind;
}
void Matrix_swap_line(double *a, int j, int k, int n)
{
	double buf;
	for (int i = 1; i < n + 1; i++) {
		buf = a[j * (n + 1) + i];
		a[j * (n + 1) + i] = a[k * (n + 1) + i];
		a[k * (n + 1) + i] = buf;
	}
}
void Matrix_answer(double *arr_arg, double *a, int n)
{
	for (int i = n - 1; i > 0; i--) {
		for (int j = n - 1; j != i; j--) {
			a[i * (n + 1) + n] = a[i * (n + 1) + n] - (a[i * (n + 1) + j] * arr_arg[j]);
			a[i * (n + 1) + j] = 0;
		}

		arr_arg[i] = a[i * (n + 1) + n] / a[i * (n + 1) + i];
	}
}
void set_M(double *M, double *C, double *d, int n)
{
	double *arr = malloc(sizeof(double) * n * (n + 1));
	for (int i = 1; i < n - 1; i++) {
		for (int j = 1; j < n; j++) {
			arr[i * n + j] = C[i * n + j];
		}
		arr[i * n + (n - 1)] = d[i];
	}
	double mult;
	for (int j = 1; j < n - 2; j++) {
		int num_ind = Matrix_max_first_elem(arr, j, n - 1);
		Matrix_swap_line(arr, j, num_ind, n - 1);
		for (int i = j + 1; i < (n - 1); i++) {
			if (arr[i * (n - 1) + j] != 0) {
				mult = -(arr[i * (n - 1) + j] / arr[j * (n - 1) + i]);
```

```
                                } else {
                                        break;
                                }

                                for (int k = j; k < n; k++) {
                                        arr[i * (n - 1) + k] += mult * arr[j * (n - 1) + k];
                                }
                        }
                }
                Matrix_answer(M, arr, (n - 1));
}
int set_i(double *X, double x, int n)
{
                int i = 0;
                for (int j = 1; j < n; j++) {
                        if (X[j - 1] <= x && x <= X[j])
                                i = j;
                }
                return i;
}
double set_s(double *X, double *Y, double x, double *h, double *M, int i)
{
                double s = M[i - 1] * (pow((X[i] - x), 3) / (6 * h[i]));
                s += M[i] * (pow((x - X[i - 1]), 3) / (6 * h[i]));
                s += (Y[i - 1] - ((M[i - 1] * pow(h[i], 2)) / 6)) * ((X[i] - x) / h[i]);
                s += (Y[i] - ((M[i] * pow(h[i], 2)) / 6)) * ((x - X[i - 1]) / h[i]);
                return s;
}
double Splines(double *X, double *Y, double x, int n)
{
                double *h = malloc(sizeof(double) * n);
                double *d = malloc(sizeof(double) * (n - 1));
                double *C = malloc(sizeof(double) * n * n);
                double *M = malloc(sizeof(double) * n);
                set_h(h, X, n);
                set_d(d, h, Y, n);
                set_C(C, h, n);
                set_M(M, C, d, n);
                int i = set_i(X, x, n);
                double s = set_s(X, Y, x, h, M, i);
                free(M);
                free(C);
                free(d);
                free(h);
                return s;
}
double Form_of_Simpson(double a, double b, double h, double *y0)
{
                double res = 0.0;
                double j = a;
                double *tmp;
                for (int i = 1; j <= b - h; i++, j += h) {
                        tmp = Runge_Kutt(a, j, h, y0);
                        res += (i % 2 ? 4 : 2) * tmp[0];
                }
                tmp = Runge_Kutt(a, 0, h, y0);
                res += tmp[0];
                tmp = Runge_Kutt(a, b, h, y0);
                res += tmp[0];
                res = (res * h) / 3;
                return res;
}
double double_counting(double (*method)(double, double, double, double *), double a, double b, double h, double Eps,
double *y)
{
```

```
	h = b - a;
	double prev = method(a, b, h, y);
	h /= 2;
	double cur = method(a, b, h, y);
	int count = 0;
	while (fabs(prev - cur) >= Eps) {
		prev = cur;
		h /= 2;
		cur = method(a, b, h, y);

		count++;
	}
	printf("Count iteration = %d\n", count);
	return cur;
}
double **dbl_counting_Runge(double a, double b, double h, double Eps, double *y, int *count, double *h_)
{
	double **y_prev;
	double **y_cur;
	double max;
	int count_elem;
	do {
		count_elem = (b - a) / h;
		y_prev = malloc(sizeof(double*) * count_elem);
		double t = a;
		for (int i = 0; fabs(t - b) >= 1e-8; i++, t += h) {
			y_prev[i] = malloc(sizeof(double) * 2);
			y_prev[i] = Runge_Kutt(a, t, h, y);
		}
		h /= 2;
		count_elem = (b - a) / h;
		t = a;
		y_cur = malloc(sizeof(double*) * count_elem);
		for (int i = 0; fabs(t - b) >= 1e-8; i++, t += h) {
			y_cur[i] = malloc(sizeof(double) * 2);
			y_cur[i] = Runge_Kutt(a, t, h, y);
		}
		double mod = 0.0;
		max = 0;
		int j = 0;
		int del = count_elem;
		int i;
		for (i = 0; j < del + (del % 2 ? 1 : 0); i++, j += 2) {
			mod = fabs(y_prev[i][0] - y_cur[j][0]);
			if (mod > max) {
				max = mod;
			}
		}
	} while (fabs(max) >= Eps);
	*count = count_elem;
	*h_ = h;
	for (int i = 0, j = 0; i < count_elem; i += 2, j++) {
		y_cur[i][0] -= (y_cur[i][0] - y_prev[j][0]) / 3;
		y_cur[i][1] -= (y_cur[i][1] - y_prev[j][1]) / 3;
	}
	return y_cur;
}
double dbl_count_D1(double x0, double x1, double y0, double y1, double h, double Eps)
{
	double prev;
	double cur;
	do {
		prev = MethodShooting(x0, x1, y0, y1, h);
		h /= 2;
		cur = MethodShooting(x0, x1, y0, y1, h);
```

```
	} while (fabs(prev - cur) >= Eps);
	return cur;
}
int main()
{
	double a = 0.0;
	double b = 1.0;
	double h = 0.2;
	int size = (a + b) / h;
	double y[size + 1];
	double x0 = 0.0;
	double y0 = 3.0;
	double x1 = 1.0;
	double y1 = 2.0;
	double D1 = dbl_count_D1(x0, x1, y0, y1, h, 1e-3);
	printf("D1 = %.3lf\n", D1);
	FILE *out = fopen("Runge_Kutt.txt", "w");
	printf("x\ty(x)\ty\'(x)\n");
	double Eps = 1e-3;
	double tmp[2] = { y0, D1 };
	int count_elem;
	double h_;
	double **yt = dbl_counting_Runge(a, b, h, Eps, tmp, &count_elem, &h_);
	int i_count[6];
	i_count[0] = 0;
	double x[6] = { 0.0, 0.2, 0.4, 0.6, 0.8, 1.0 };
	int t = count_elem / 5;
	for (int j = 1; j < 6; j++) {
		i_count[j] = t * j;
	}
	i_count[5]--;
	double ta = a;
	for (int i = 0; i < count_elem; i++, ta += h_) {
		fprintf(out, "%.4lf %lf\n", ta, yt[i][0]);
	}
	double m = 0.0;
	for (int i = 0; i < 6; i++, m += h) {
		printf("%.2lf\t", m);
		printf("%.3lf\t", yt[i_count[i]][0]);
		printf("%.3lf\n", yt[i_count[i]][1]);
		y[i] = yt[i_count[i]][0];
	}
	printf("\n");
	fclose(out);
	printf("Spleins interpolation:\n");
	FILE *splines_out = fopen("Splines.txt", "w");
	for (double i = a; i <= b; i += h_) {
		double tmp = Splines(x, y, i, 6);
		fprintf(splines_out, "%lf %lf\n", i, tmp);
	}
	printf("\n");
	fclose(splines_out);
	Eps = 1e-2;
	printf("Integration = %.10lf\n", double_counting(Form_of_Simpson, a, b, h, Eps, tmp));
	return 0;
}
```