Министерство цифрового развития связи и массовых коммуникаций Российский Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

“Сибирский государственный университет телекоммуникаций и информатики” (СибГУТИ)

Кафедра прикладной математики и кибернетики

Курсовая работа

по дисциплине «Вычислительная математика»

Вариант 15

Выполнил:

Студент группы ИП-316

Кудря И. А.

Проверил:

Ассистент кафедры ПМиК

Истомина А. С.

Новосибирск

2025

Оглавление

[1. Постановка задачи 3](#_Toc198069324)

[2. Теоретические сведения 4](#_Toc198069325)

[3. Описание алгоритма 7](#_Toc198069326)

[4. Результат работы программы 1](#_Toc198069327)1

[5. Листинг 1](#_Toc198069328)2

# Постановка задачи

1. Решить дифференциальное уравнение на интервале [0,1] методами Рунге кутта 2 и 4 порядка с точностью стартовый шаг h=0,1

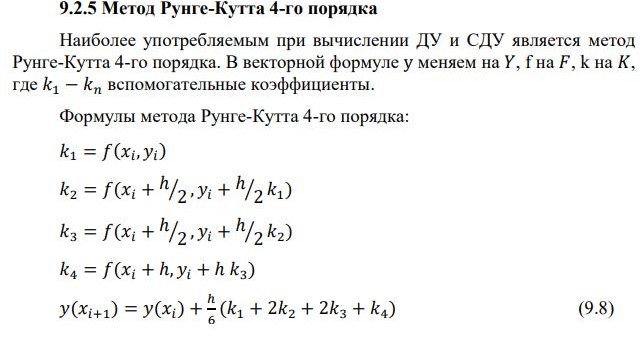
+

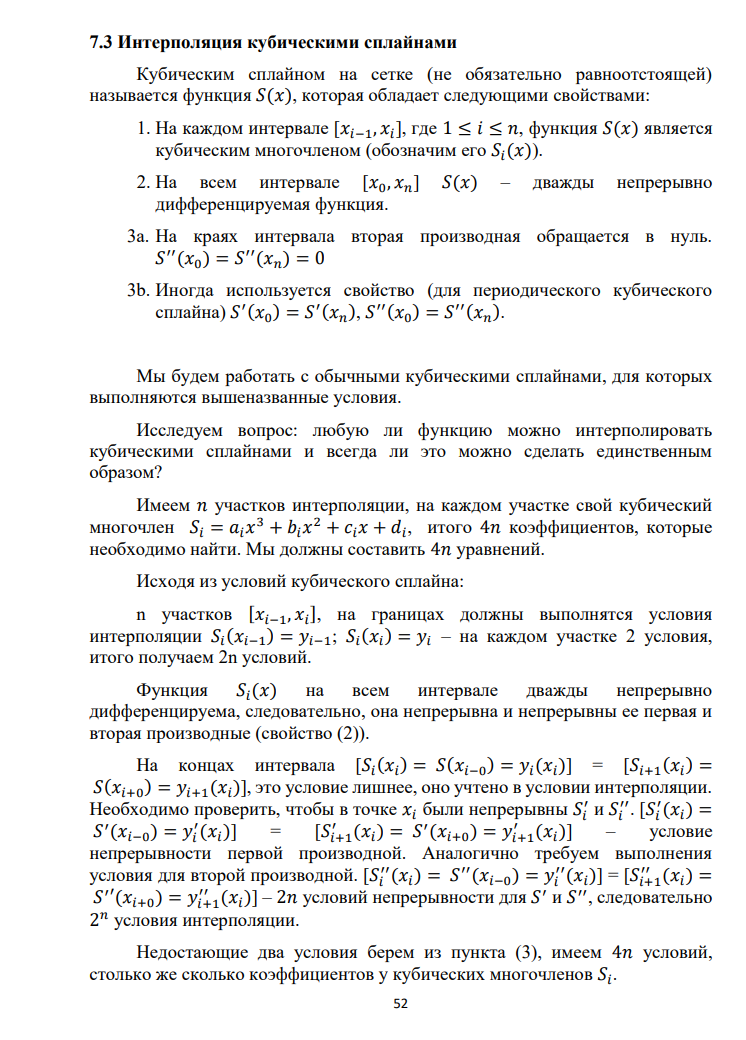
y(0) = 1

y(1) = 2,718

Про интерполировать найденное решение с помощью кубического сплайна по узлам интерполяции 0,0, 0,2, 0,4, 0,6, 0,8, 1,0

# Теоретические сведения





# Описание алгоритма

**equation(double x, SystemState s)**

SystemState equation(double x, SystemState s) {

double dderiv = exp(2 \* x) + (s.val + s.deriv) / 2;

return {s.deriv, dderiv};

}

Назначение: Вычисляет правую часть дифференциального уравнения второго порядка.

Алгоритм:

* Принимает текущие значения *x*, *y* и *y*′.
* Возвращает производные *y*′ и *y*′′=*e^*2*x*+(*y*+*y*′)/2.

**step\_rk2(double x, SystemState s, double h)**

SystemState step\_rk2(double x, SystemState s, double h) {

SystemState k1 = equation(x, s);

SystemState k2 = equation(x + h, {s.val + h \* k1.val, s.deriv + h \* k1.deriv});

return {

s.val + h \* (k1.val + k2.val) / 2,

s.deriv + h \* (k1.deriv + k2.deriv) / 2

};

}

Назначение: Реализует один шаг метода Рунге–Кутты второго порядка (RK2).

Алгоритм:

* Вычисляет два промежуточных значения k и Обновляет значения *y* и *y*′ с использованием среднего из двух шагов: *yn*+1​=*yn*​+*h*​(*k*1​.*y*+*k*2​.*y*)/2

**step\_rk4(double x, SystemState s, double h)**

SystemState step\_rk4(double x, SystemState s, double h) {

SystemState k1 = equation(x, s);

SystemState k2 = equation(x + h/2, {s.val + h\*k1.val/2, s.deriv + h\*k1.deriv/2});

SystemState k3 = equation(x + h/2, {s.val + h\*k2.val/2, s.deriv + h\*k2.deriv/2});

SystemState k4 = equation(x + h, {s.val + h\*k3.val, s.deriv + h\*k3.deriv});

return {

s.val + h \* (k1.val + 2\*k2.val + 2\*k3.val + k4.val) / 6,

s.deriv + h \* (k1.deriv + 2\*k2.deriv + 2\*k3.deriv + k4.deriv) / 6

};

}

Назначение: Реализует один шаг метода Рунге–Кутты четвёртого порядка (RK4).

Алгоритм:

* Вычисляет четыре промежуточных приближения: *k*1​,*k*2​,*k*3​,*k*4​.
* Обновляет значения *y* и *y*′ с использованием взвешенного среднего:*yn*+1​=*yn*​+6*h*​(*k*1​.*y*+2*k*2​.*y*+2*k*3​.*y*+*k*4​.*y*)

**solve\_ode(double init\_deriv)**

pair<vector<pair<double, double>>, vector<pair<double, double>>> solve\_ode(double init\_deriv) {

vector<pair<double, double>> rk2\_sol, rk4\_sol;

SystemState state\_rk2 = {1.0, init\_deriv};

SystemState state\_rk4 = {1.0, init\_deriv};

double x = START\_X;

double h = INIT\_STEP;

while (x <= END\_X + ACCURACY) {

rk2\_sol.emplace\_back(x, state\_rk2.val);

rk4\_sol.emplace\_back(x, state\_rk4.val);

state\_rk2 = step\_rk2(x, state\_rk2, h);

state\_rk4 = step\_rk4(x, state\_rk4, h);

x += h;

}

return {rk2\_sol, rk4\_sol};

}

Назначение: Численно решает ОДУ двумя методами (RK2 и RK4) на заданном интервале.

Алгоритм:

* Инициализирует начальные условия *y*(0)=1, *y*′(0)=*initd*​*eriv*.
* Пошагово вычисляет значения *y*(*x*) на сетке.
* Сохраняет результаты для обоих методов

**find\_initial\_derivative()**

double find\_initial\_derivative() {

double left = -10.0;

double right = 10.0;

const int MAX\_ITER = 100;

for (int iter = 0; iter < MAX\_ITER; ++iter) {

double mid = (left + right) / 2;

auto [\_, sol] = solve\_ode(mid);

double final\_y = sol.back().second;

if (abs(final\_y - TARGET\_FINAL\_Y) < ACCURACY)

break;

if (final\_y < TARGET\_FINAL\_Y)

left = mid;

else

right = mid;

}

return (left + right) / 2;

}

Назначение: Метод стрельбы — поиск начального значения y ′

(0), чтобы удовлетворить краевому условию y(1)=e.

Алгоритм:

Использует бинарный поиск в диапазоне [−10,10].

На каждой итерации:

Решает ОДУ с предполагаемым значением y′(0),

Сравнивает конечное значение с целевым y(1)=e,

Корректирует диапазон поиска.

**get\_values\_at(const vector<pair<double, double>>& solution, const vector<double>& points)**

Назначение: Извлекает значения решения в заданных точках.

Алгоритм:

* Для каждой точки из списка **points** находит ближайшее значение в массиве решений.
* Возвращает соответствующие значения *y*(*x*).

Класс **CubicSpline**

class CubicSpline {

vector<double> knots\_x, knots\_y, b, c, d;

public:

CubicSpline(const vector<double>& x, const vector<double>& y) : knots\_x(x), knots\_y(y) {

size\_t n = x.size();

vector<double> h(n-1), alpha(n-1);

b.resize(n); c.resize(n); d.resize(n);

for (size\_t i = 0; i < n-1; ++i)

h[i] = x[i+1] - x[i];

for (size\_t i = 1; i < n-1; ++i)

alpha[i] = 3\*(y[i+1]-y[i])/h[i] - 3\*(y[i]-y[i-1])/h[i-1];

vector<double> l(n), mu(n), z(n);

l[0] = 1; mu[0] = z[0] = 0;

for (size\_t i = 1; i < n-1; ++i) {

l[i] = 2\*(x[i+1]-x[i-1]) - h[i-1]\*mu[i-1];

mu[i] = h[i]/l[i];

z[i] = (alpha[i] - h[i-1]\*z[i-1])/l[i];

}

l[n-1] = 1; z[n-1] = 0; c[n-1] = 0;

for (int i = n-2; i >= 0; --i) {

c[i] = z[i] - mu[i]\*c[i+1];

b[i] = (y[i+1]-y[i])/h[i] - h[i]\*(c[i+1]+2\*c[i])/3;

d[i] = (c[i+1]-c[i])/(3\*h[i]);

}

}

double interpolate(double xq) const {

auto it = upper\_bound(knots\_x.begin(), knots\_x.end(), xq);

size\_t i = max(int(it - knots\_x.begin()) - 1, 0);

double dx = xq - knots\_x[i];

return knots\_y[i] + b[i]\*dx + c[i]\*dx\*dx + d[i]\*dx\*dx\*dx;

}

};

Назначение: Построение кубического сплайна для гладкой интерполяции решения.

Алгоритм:

Инициализируется узловыми точками и значениями.

Вычисляются коэффициенты многочленов на каждом интервале.

Реализуется метод **interpolate**, который возвращает значение сплайна в любой точке внутри области определения.

**main()**

int main() {

double init\_deriv = find\_initial\_derivative();

auto [sol\_rk2, sol\_rk4] = solve\_ode(init\_deriv);

vector<double> interp\_points = {0.0, 0.2, 0.4, 0.6, 0.8, 1.0};

CubicSpline spline\_rk2(interp\_points, get\_values\_at(sol\_rk2, interp\_points));

CubicSpline spline\_rk4(interp\_points, get\_values\_at(sol\_rk4, interp\_points));

cout << "y(1) по RK4: " << sol\_rk4.back().second << "\n";

cout << fixed << setprecision(6);

cout << "x\t\ty\_RK2\t\ty\_RK4\n";

for (double x = 0.0; x <= 1.0; x += 0.1) {

double y\_rk2 = spline\_rk2.interpolate(x);

double y\_rk4 = spline\_rk4.interpolate(x);

cout << x << "\t" << y\_rk2 << "\t" << y\_rk4 << "\n";

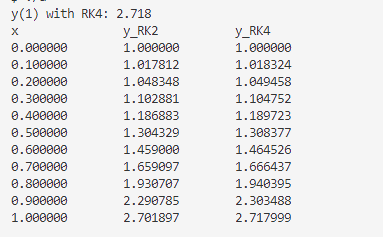
}

return 0;

}

Назначение: Основная функция программы — управление процессом решения и вывод результата

# Результат работы программы



# Листинг

# #include <iostream>

# #include <vector>

# #include <cmath>

# #include <iomanip>

# #include <algorithm>

# using namespace std;

# const double ACCURACY = 1e-6;

# const double INIT\_STEP = 0.1;

# const double START\_X = 0.0;

# const double END\_X = 1.0;

# const double TARGET\_FINAL\_Y = 2.718; // e^1

# struct SystemState {

# double val;

# double deriv;

# };

# SystemState equation(double x, SystemState s) {

# double dderiv = exp(2 \* x) + (s.val + s.deriv) / 2;

# return {s.deriv, dderiv};

# }

# SystemState step\_rk2(double x, SystemState s, double h) {

# SystemState k1 = equation(x, s);

# SystemState k2 = equation(x + h, {s.val + h \* k1.val, s.deriv + h \* k1.deriv});

# return {

# s.val + h \* (k1.val + k2.val) / 2,

# s.deriv + h \* (k1.deriv + k2.deriv) / 2

# };

# }

# SystemState step\_rk4(double x, SystemState s, double h) {

# SystemState k1 = equation(x, s);

# SystemState k2 = equation(x + h/2, {s.val + h\*k1.val/2, s.deriv + h\*k1.deriv/2});

# SystemState k3 = equation(x + h/2, {s.val + h\*k2.val/2, s.deriv + h\*k2.deriv/2});

# SystemState k4 = equation(x + h, {s.val + h\*k3.val, s.deriv + h\*k3.deriv});

# return {

# s.val + h \* (k1.val + 2\*k2.val + 2\*k3.val + k4.val) / 6,

# s.deriv + h \* (k1.deriv + 2\*k2.deriv + 2\*k3.deriv + k4.deriv) / 6

# };

# }

# pair<vector<pair<double, double>>, vector<pair<double, double>>> solve\_ode(double init\_deriv) {

# vector<pair<double, double>> rk2\_sol, rk4\_sol;

# SystemState state\_rk2 = {1.0, init\_deriv};

# SystemState state\_rk4 = {1.0, init\_deriv};

# double x = START\_X;

# double h = INIT\_STEP;

# while (x <= END\_X + ACCURACY) {

# rk2\_sol.emplace\_back(x, state\_rk2.val);

# rk4\_sol.emplace\_back(x, state\_rk4.val);

# state\_rk2 = step\_rk2(x, state\_rk2, h);

# state\_rk4 = step\_rk4(x, state\_rk4, h);

# x += h;

# }

# return {rk2\_sol, rk4\_sol};

# }

# double find\_initial\_derivative() {

# double left = -10.0;

# double right = 10.0;

# const int MAX\_ITER = 100;

# for (int iter = 0; iter < MAX\_ITER; ++iter) {

# double mid = (left + right) / 2;

# auto [\_, sol] = solve\_ode(mid);

# double final\_y = sol.back().second;

# if (abs(final\_y - TARGET\_FINAL\_Y) < ACCURACY)

# break;

# if (final\_y < TARGET\_FINAL\_Y)

# left = mid;

# else

# right = mid;

# }

# return (left + right) / 2;

# }

# vector<double> get\_values\_at(const vector<pair<double, double>>& solution,

# const vector<double>& points) {

# vector<double> result;

# for (double x : points) {

# auto it = find\_if(solution.begin(), solution.end(),

# [x](auto& p) { return abs(p.first - x) < ACCURACY; });

# result.push\_back(it->second);

# }

# return result;

# }

# class CubicSpline {

# vector<double> knots\_x, knots\_y, b, c, d;

# public:

# CubicSpline(const vector<double>& x, const vector<double>& y) : knots\_x(x), knots\_y(y) {

# size\_t n = x.size();

# vector<double> h(n-1), alpha(n-1);

# b.resize(n); c.resize(n); d.resize(n);

# for (size\_t i = 0; i < n-1; ++i)

# h[i] = x[i+1] - x[i];

# for (size\_t i = 1; i < n-1; ++i)

# alpha[i] = 3\*(y[i+1]-y[i])/h[i] - 3\*(y[i]-y[i-1])/h[i-1];

# vector<double> l(n), mu(n), z(n);

# l[0] = 1; mu[0] = z[0] = 0;

# for (size\_t i = 1; i < n-1; ++i) {

# l[i] = 2\*(x[i+1]-x[i-1]) - h[i-1]\*mu[i-1];

# mu[i] = h[i]/l[i];

# z[i] = (alpha[i] - h[i-1]\*z[i-1])/l[i];

# }

# l[n-1] = 1; z[n-1] = 0; c[n-1] = 0;

# for (int i = n-2; i >= 0; --i) {

# c[i] = z[i] - mu[i]\*c[i+1];

# b[i] = (y[i+1]-y[i])/h[i] - h[i]\*(c[i+1]+2\*c[i])/3;

# d[i] = (c[i+1]-c[i])/(3\*h[i]);

# }

# }

# double interpolate(double xq) const {

# auto it = upper\_bound(knots\_x.begin(), knots\_x.end(), xq);

# size\_t i = max(int(it - knots\_x.begin()) - 1, 0);

# double dx = xq - knots\_x[i];

# return knots\_y[i] + b[i]\*dx + c[i]\*dx\*dx + d[i]\*dx\*dx\*dx;

# }

# };

# int main() {

# double init\_deriv = find\_initial\_derivative();

# auto [sol\_rk2, sol\_rk4] = solve\_ode(init\_deriv);

# vector<double> interp\_points = {0.0, 0.2, 0.4, 0.6, 0.8, 1.0};

# CubicSpline spline\_rk2(interp\_points, get\_values\_at(sol\_rk2, interp\_points));

# CubicSpline spline\_rk4(interp\_points, get\_values\_at(sol\_rk4, interp\_points));

# cout << "y(1) with RK4: " << sol\_rk4.back().second << "\n";

# cout << fixed << setprecision(6);

# cout << "x\t\ty\_RK2\t\ty\_RK4\n";

# for (double x = 0.0; x <= 1.0; x += 0.1) {

# double y\_rk2 = spline\_rk2.interpolate(x);

# double y\_rk4 = spline\_rk4.interpolate(x);

# cout << x << "\t" << y\_rk2 << "\t" << y\_rk4 << "\n";

# }

# return 0;

# }