Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент:	Кучмистов Д.Р.
Преподаватель:	Пивоваров Д.Е.
Группа:	М8О-303Б-21
Дата:	
Оценка:	
Подпись:	

Москва, 2024

4.1 Методы Эйлера, Рунге-Кутты и Адамса

1 Постановка задачи

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Вариант: 14

14	$y''+2y'ctgx+3y=0$, $y(1)=1$, $y'(1)=1$, $x\in[1,2], h=0.1$	$y=\dfrac{-0.9783\cos 2x+0.4776\sin 2x}{\sin x}$

Рис. 1: Входные данные

2 Результаты работы

```
x Euler RK4 Adams Exact Error (Euler) Error (RK4) Error (Adams)
1.000000 1.000000 1.000000 1.000000 0.999912 0.000088 0.000088 0.000088

1.100000 1.100000 1.079386 1.079386 1.079287 0.020713 0.000099 0.000099

1.200000 1.157158 1.120224 1.120224 1.120118 0.037040 0.000106 0.000106

1.300000 1.175498 1.125624 1.125624 1.125513 0.049984 0.000110 0.000110

1.400000 1.157697 1.097850 1.097886 1.097739 0.059958 0.000111 0.000147

1.500000 1.105619 1.038619 1.038670 1.038510 0.067109 0.000109 0.000160

1.600000 1.020607 0.949261 0.949317 0.949157 0.071450 0.000104 0.000160

1.700000 0.903633 0.830793 0.830845 0.830696 0.072936 0.000096 0.000149

1.800000 0.755356 0.683920 0.683965 0.683835 0.071521 0.000086 0.000130

1.900000 0.576118 0.508981 0.509012 0.508909 0.067209 0.000072 0.000103

```

```
2.000000 0.365855 0.305798 0.305812 0.305742 0.060113 0.000055 0.000069


Runge-Romberg Error Estimation for Euler's Method
x Euler (h=0.1) Euler (h=0.05) Error Estimate (Euler)
1.000000 1.000000 1.000000 0.000000
1.100000 1.100000 1.089290 0.010710
1.200000 1.157158 1.137957 0.019201
1.300000 1.175498 1.149515 0.025983
1.400000 1.157697 1.126413 0.031284
1.500000 1.105619 1.070433 0.035187
1.600000 1.020607 0.982913 0.037694
1.700000 0.903633 0.864855 0.038777
1.800000 0.755356 0.716953 0.038403
1.900000 0.576118 0.539559 0.036559
2.000000 0.365855 0.332573 0.033282

Runge-Romberg Error Estimation for RK4 Method
x RK4 (h=0.1) RK4 (h=0.05) Error Estimate (RK4)
1.000000 1.000000 1.000000 0.000000
1.100000 1.079386 1.079385 0.000000
1.200000 1.120224 1.120221 0.000000
1.300000 1.125624 1.125620 0.000000
1.400000 1.097850 1.097845 0.000000
1.500000 1.038619 1.038612 0.000000
1.600000 0.949261 0.949252 0.000001
1.700000 0.830793 0.830782 0.000001
1.800000 0.683920 0.683908 0.000001
1.900000 0.508981 0.508967 0.000001
2.000000 0.305798 0.305782 0.000001

Runge-Romberg Error Estimation for Adams Method
x Adams (h=0.1) Adams (h=0.05) Error Estimate (Adams)
1.000000 1.000000 1.000000 0.000000
1.100000 1.079386 1.079385 0.000000
1.200000 1.120224 1.120223 0.000000
1.300000 1.125624 1.125622 0.000000
1.400000 1.097886 1.097847 0.000003
1.500000 1.038670 1.038614 0.000004
1.600000 0.949317 0.949253 0.000004
1.700000 0.830845 0.830782 0.000004
1.800000 0.683965 0.683907 0.000004
1.900000 0.509012 0.508964 0.000003
2.000000 0.305812 0.305777 0.000002
```

3 Исходный код

```
#include <iostream>
#include <vector>
```

```cpp
#include <cmath>
#include <iomanip>

double exact_solution(double x) {
    return (-0.9783 * cos(2 * x) + 0.4776 * sin(2 * x)) / sin(x);
}

std::vector<double> f(double x, double y, double y_prime) {
    double dy = y_prime;
    double dy_prime = -2 * y_prime * (cos(x) / sin(x)) - 3 * y;
    return {dy, dy_prime};
}

void euler_method(double h, const std::vector<double>& x, std::vector<double>& y, std::vector<double>& y_prime) {
    int n = x.size();
    for (int i = 1; i < n; ++i) {
        std::vector<double> f_val = f(x[i-1], y[i-1], y_prime[i-1]);
        y[i] = y[i-1] + h * f_val[0];
        y_prime[i] = y_prime[i-1] + h * f_val[1];
    }
}

void runge_kutta_method(double h, const std::vector<double>& x, std::vector<double>& y, std::vector<double>& y_prime) {
    int n = x.size();
    for (int i = 1; i < n; ++i) {
        std::vector<double> k1 = f(x[i-1], y[i-1], y_prime[i-1]);
        std::vector<double> k2 = f(x[i-1] + h/2, y[i-1] + h/2 * k1[0], y_prime[i-1] + h/2 * k1[1]);
        std::vector<double> k3 = f(x[i-1] + h/2, y[i-1] + h/2 * k2[0], y_prime[i-1] + h/2 * k2[1]);
        std::vector<double> k4 = f(x[i-1] + h, y[i-1] + h * k3[0], y_prime[i-1] + h * k3[1]);

        y[i] = y[i-1] + h/6 * (k1[0] + 2*k2[0] + 2*k3[0] + k4[0]);
        y_prime[i] = y_prime[i-1] + h/6 * (k1[1] + 2*k2[1] + 2*k3[1] + k4[1]);
    }
}

void adams_bashforth_moulton_method(double h, const std::vector<double>& x, std::vector<double>& y, std::vector<double>& y_prime) {
    int n = x.size();
    std::vector<double> f0, f1, f2, f3;

    for (int i = 1; i <= 3; ++i) {
        std::vector<double> k1 = f(x[i-1], y[i-1], y_prime[i-1]);
        std::vector<double> k2 = f(x[i-1] + h/2, y[i-1] + h/2 * k1[0], y_prime[i-1] + h/2 * k1[1]);
```

```
        std::vector<double> k3 = f(x[i-1] + h/2, y[i-1] + h/2 * k2[0], y_prime[i-1] + h
            /2 * k2[1]);
        std::vector<double> k4 = f(x[i-1] + h, y[i-1] + h * k3[0], y_prime[i-1] + h *
            k3[1]);

        y[i] = y[i-1] + h/6 * (k1[0] + 2*k2[0] + 2*k3[0] + k4[0]);
        y_prime[i] = y_prime[i-1] + h/6 * (k1[1] + 2*k2[1] + 2*k3[1] + k4[1]);
    }

    for (int i = 4; i < n; ++i) {
        f0 = f(x[i-1], y[i-1], y_prime[i-1]);
        f1 = f(x[i-2], y[i-2], y_prime[i-2]);
        f2 = f(x[i-3], y[i-3], y_prime[i-3]);
        f3 = f(x[i-4], y[i-4], y_prime[i-4]);

        double yp = y[i-1] + h/24 * (55*f0[0] - 59*f1[0] + 37*f2[0] - 9*f3[0]);
        double y_prime_p = y_prime[i-1] + h/24 * (55*f0[1] - 59*f1[1] + 37*f2[1] - 9*f3
            [1]);

        std::vector<double> fp = f(x[i], yp, y_prime_p);
        y[i] = y[i-1] + h/24 * (9*fp[0] + 19*f0[0] - 5*f1[0] + f2[0]);
        y_prime[i] = y_prime[i-1] + h/24 * (9*fp[1] + 19*f0[1] - 5*f1[1] + f2[1]);
    }
}

double runge_romberg(double y_h1, double y_h2, int p) {
    return (y_h1 - y_h2) / (std::pow(2, p) - 1);
}

int main() {
    double h = 0.1;
    double h2 = h / 2;
    double a = 1.0;
    double b = 2.0;
    int n = static_cast<int>((b - a) / h) + 1;
    int n2 = static_cast<int>((b - a) / h2) + 1;

    std::vector<double> x(n), y_euler(n), y_prime_euler(n);
    std::vector<double> y_rk(n), y_prime_rk(n);
    std::vector<double> y_adams(n), y_prime_adams(n);

    std::vector<double> x2(n2), y_euler2(n2), y_prime_euler2(n2);
    std::vector<double> y_rk2(n2), y_prime_rk2(n2);
    std::vector<double> y_adams2(n2), y_prime_adams2(n2);

    x[0] = x2[0] = a;
    y_euler[0] = y_rk[0] = y_adams[0] = y_euler2[0] = y_rk2[0] = y_adams2[0] = 1.0;
    y_prime_euler[0] = y_prime_rk[0] = y_prime_adams[0] = y_prime_euler2[0] =
        y_prime_rk2[0] = y_prime_adams2[0] = 1.0;
```

```cpp

    for (int i = 1; i < n; ++i) {
        x[i] = x[i-1] + h;
    }

    for (int i = 1; i < n2; ++i) {
        x2[i] = x2[i-1] + h2;
    }

    euler_method(h, x, y_euler, y_prime_euler);
    runge_kutta_method(h, x, y_rk, y_prime_rk);
    adams_bashforth_moulton_method(h, x, y_adams, y_prime_adams);

    euler_method(h2, x2, y_euler2, y_prime_euler2);
    runge_kutta_method(h2, x2, y_rk2, y_prime_rk2);
    adams_bashforth_moulton_method(h2, x2, y_adams2, y_prime_adams2);

    std::cout << std::fixed << std::setprecision(6);
    std::cout << "x\tEuler\t\tRK4\t\tAdams\t\tExact\t\tError (Euler)\tError (RK4)\tError (Adams)\n";

    for (int i = 0; i < n; ++i) {
        double exact = exact_solution(x[i]);
        std::cout << x[i] << "\t"
                  << y_euler[i] << "\t" << y_rk[i] << "\t" << y_adams[i] << "\t" << exact << "\t"
                  << std::abs(y_euler[i] - exact) << "\t" << std::abs(y_rk[i] - exact) << "\t" << std::abs(y_adams[i] - exact) << "\n";
    }

    std::cout << "\nRunge-Romberg Error Estimation for Euler's Method\n";

    std::cout << "x\tEuler (h=0.1)\tEuler (h=0.05)\tError Estimate (Euler)\n";
    for (int i = 0; i < n; ++i) {
        double y_h2 = y_euler2[i * 2];
        double error_estimate = runge_romberg(y_euler[i], y_h2, 1);
        std::cout << x[i] << "\t" << y_euler[i] << "\t\t" << y_h2 << "\t\t" << error_estimate << "\n";
    }

    std::cout << "\nRunge-Romberg Error Estimation for RK4 Method\n";

    std::cout << "x\tRK4 (h=0.1)\tRK4 (h=0.05)\tError Estimate (RK4)\n";
    for (int i = 0; i < n; ++i) {
        double y_h2 = y_rk2[i * 2];
        double error_estimate = runge_romberg(y_rk[i], y_h2, 4);
        std::cout << x[i] << "\t" << y_rk[i] << "\t\t" << y_h2 << "\t\t" << error_estimate << "\n";
    }
```

```
    std::cout << "\nRunge-Romberg Error Estimation for Adams Method\n";

    std::cout << "x\tAdams (h=0.1)\tAdams (h=0.05)\tError Estimate (Adams)\n";
    for (int i = 0; i < n; ++i) {
        double y_h2 = y_adams2[i * 2];
        double error_estimate = runge_romberg(y_adams[i], y_h2, 4);
        std::cout << x[i] << "\t" << y_adams[i] << "\t\t" << y_h2 << "\t\t" <<
            error_estimate << "\n";
    }


    return 0;
}
```

4.2 Метод стрельбы и конечно-разностный метод

4 Постановка задачи

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Вариант: 14

14	$(e^x+1)\,y''-2y'-e^x y=0$, $y'(0)=1$, $y'(1)-y(1)=1$	$y(x)=e^x-1$

Рис. 2: Входные данные

5 Результаты работы

```
x       Exact    Shooting        Finite Difference
0.000000        0.000000        0.000000                0.000000
0.100000        0.105171        0.000062                0.285510
0.200000        0.221403        0.000227                0.562489
0.300000        0.349859        0.000667                0.822772
0.400000        0.491825        0.001859                1.058921
0.500000        0.648721        0.005210                1.264553
0.600000        0.822119        0.015048                1.434632
0.700000        1.013753        0.045393                1.565715
0.800000        1.225541        0.144064                1.656156
0.900000        1.459603        0.483200                1.706238
1.000000        1.718282        1.718282                1.718282

Runge-Romberg Error (Shooting Method): 0.057853
Runge-Romberg Error (Finite Difference Method): 0.207464
```

Рис. 3: Вывод программы

6 Исходный код

```cpp
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

double f(double x, double y, double yp) {
    return ((exp(x) + 1) * yp - 2 * y - exp(x) * y);
}

void rungeKutta(double h, double x0, double y0, double yp0, double xf, vector<double>& x_vals, vector<double>& y_vals) {
    double k1, k2, k3, k4;
    double y = y0, yp = yp0;
    for (double x = x0; x < xf; x += h) {
        x_vals.push_back(x);
        y_vals.push_back(y);
        k1 = h * yp;
        k2 = h * (yp + 0.5 * k1);
        k3 = h * (yp + 0.5 * k2);
        k4 = h * (yp + k3);
        y = y + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
        yp = yp + (f(x, y, yp) + f(x + h, y + k1, yp + k1)) / 2;
    }
}

double shootingMethod(double h, double x0, double y0, double x_end, double y_end, double initial_guess) {
    double tolerance = 1e-6;
    double guess1 = initial_guess;
    double guess2 = initial_guess + 0.1;

    double f1, f2;

    while (true) {
        vector<double> x_vals, y_vals1, y_vals2;
        rungeKutta(h, x0, y0, guess1, x_end, x_vals, y_vals1);
        rungeKutta(h, x0, y0, guess2, x_end, x_vals, y_vals2);

        f1 = y_vals1.back() - y_end;
        f2 = y_vals2.back() - y_end;

        if (fabs(f1) < tolerance) {
            return guess1;
        }
        if (fabs(f2) < tolerance) {
            return guess2;
```

```
        }

        double guess_new = guess1 - f1 * (guess2 - guess1) / (f2 - f1);
        guess1 = guess2;
        guess2 = guess_new;
    }
}

void finiteDifferenceMethod(double h, double x0, double y0, double x_end, double y_end, vector<double>& x_vals, vector<double>& y_vals) {
    int n = (x_end - x0) / h;
    vector<double> a(n + 1), b(n + 1), c(n + 1), d(n + 1), y(n + 1);

    for (int i = 0; i <= n; ++i) {
        x_vals.push_back(x0 + i * h);
    }

    a[0] = 0;
    b[0] = 1;
    c[0] = 0;
    d[0] = y0;

    for (int i = 1; i < n; ++i) {
        double x = x0 + i * h;
        a[i] = 1 / (h * h) - tan(x) / (2 * h);
        b[i] = -2 / (h * h) + 2;
        c[i] = 1 / (h * h) + tan(x) / (2 * h);
        d[i] = 0;
    }

    a[n] = 0;
    b[n] = 1;
    c[n] = 0;
    d[n] = y_end;

    for (int i = 1; i <= n; ++i) {
        double m = a[i] / b[i - 1];
        b[i] -= m * c[i - 1];
        d[i] -= m * d[i - 1];
    }

    y[n] = d[n] / b[n];
    for (int i = n - 1; i >= 0; --i) {
        y[i] = (d[i] - c[i] * y[i + 1]) / b[i];
    }

    for (int i = 0; i <= n; ++i) {
        y_vals.push_back(y[i]);
    }
```

```
}

int main() {
    double x0 = 0.0, xf = 1.0;
    double y0 = 0.0, yf = exp(1) - 1;
    double h = 0.1; // Step size
    double epsilon = 1e-6;

vector<double> exact_solution_x, exact_solution_y;
for (double x = x0; x <= xf; x += h) {
    exact_solution_x.push_back(x);
    exact_solution_y.push_back(exp(x) - 1);
}

    vector<double> shooting_solution_x, shooting_solution_y;
    double initial_guess = (yf - y0) / (xf - x0);
    double yp0 = shootingMethod(h, x0, y0, xf, yf, initial_guess);
    rungeKutta(h, x0, y0, yp0, xf, shooting_solution_x, shooting_solution_y);

    vector<double> fd_solution_x, fd_solution_y;
    finiteDifferenceMethod(h, x0, y0, xf, yf, fd_solution_x, fd_solution_y);

    cout << "x\tExact\tShooting\tFinite Difference\n";
    cout.precision(6);
    cout.setf(ios::fixed);
    for (int i = 0; i < exact_solution_x.size(); ++i) {
        cout << exact_solution_x[i] << "\t" << exact_solution_y[i] << "\t" <<
            shooting_solution_y[i] << "\t\t" << fd_solution_y[i] << endl;
    }

double shooting_error = 0.0;
vector<double> shooting_solution_x_half, shooting_solution_y_half;
rungeKutta(h / 2, x0, y0, yp0, xf, shooting_solution_x_half, shooting_solution_y_half)
    ;
for (int i = 0; i < shooting_solution_x.size(); ++i) {
    double error = abs(shooting_solution_y_half[i] - shooting_solution_y[i]) / 15;
    if (error > shooting_error) {
        shooting_error = error;
    }
}

double fd_error = 0.0;
vector<double> fd_solution_x_half, fd_solution_y_half;
finiteDifferenceMethod(h / 2, x0, y0, xf, yf, fd_solution_x_half, fd_solution_y_half);
for (int i = 0; i < fd_solution_x.size(); ++i) {
    double error = abs(fd_solution_y_half[i] - fd_solution_y[i]) / 3;
    if (error > fd_error) {
        fd_error = error;
    }
```

```
}

cout << "\nRunge-Romberg Error (Shooting Method): " << shooting_error << endl;
cout << "Runge-Romberg Error (Finite Difference Method): " << fd_error << endl;

    return 0;
}
```