Numerical Solution of First-Order Ordinary Differential Equations Using Numerical Methods

A Technical Project Report

Ayushmaan Kumar Verma
Department of Mathematics and Computing Technology
National Institute of Technology, Patna

Abstract

Abstract

Ordinary Differential Equations (ODEs) play a fundamental role in modeling real-world phenomena across science and engineering. Analytical solutions are not always feasible, making numerical techniques essential. This report presents a C++ implementation of three classical numerical methods—Euler Method, Modified Euler Method (Heun's Method), and the Fourth-Order Runge–Kutta Method (RK4)—to approximate solutions of first-order ODEs. The performance and accuracy of these methods are analyzed by comparing numerical results with the exact analytical solution.

Table of Contents

Contents

1 Introduction

Ordinary Differential Equations (ODEs) arise naturally in physics, engineering, biology, economics, and computational sciences. A first-order ODE is generally expressed as:

$$\frac{dy}{dx} = f(x, y) \tag{1}$$

In many practical situations, obtaining a closed-form analytical solution is difficult or impossible. Numerical methods provide approximate solutions with controllable accuracy and computational efficiency. This project focuses on implementing and analyzing three widely used numerical techniques in C++.

2 Problem Definition

The differential equation considered in this project is:

$$\frac{dy}{dx} = 2xy \tag{2}$$

with the initial condition:

$$y(0) = 1 \tag{3}$$

The objective is to compute the approximate value of $y(x)$ at $x = 1$ using different numerical methods and compare them with the exact solution.

3 Exact Analytical Solution

Separating variables,

$$\frac{1}{y}\, dy = 2x\, dx \tag{4}$$

Integrating both sides,

$$\ln y = x^2 + C \tag{5}$$

$$y = Ce^{x^2} \tag{6}$$

Using the initial condition $y(0) = 1$, we obtain $C = 1$. Therefore, the exact solution is:

$$y(x) = e^{x^2} \tag{7}$$

At $x = 1$:

$$y(1) = e \approx 2.718281828 \tag{8}$$

4 Theory of Numerical Methods

4.1 Euler Method

The Euler Method is the simplest numerical technique for solving first-order ODEs. It is based on the first-order Taylor series expansion:

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{9}$$

where h is the step size. Although computationally inexpensive, Euler's method suffers from low accuracy and error accumulation.

4.2 Modified Euler Method (Heun's Method)

The Modified Euler Method improves accuracy by averaging the slope at the beginning and end of each step:

$$k_1 = hf(x_n, y_n) \tag{10}$$

$$k_2 = hf(x_n + h, y_n + k_1) \tag{11}$$

$$y_{n+1} = y_n + \frac{k_1 + k_2}{2} \tag{12}$$

This method provides better accuracy than Euler's method while maintaining reasonable computational efficiency.

4.3 Fourth-Order Runge–Kutta Method (RK4)

The RK4 method is one of the most accurate and widely used numerical techniques. It computes a weighted average of four slopes:

$$k_1 = hf(x_n, y_n) \tag{13}$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \tag{14}$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \tag{15}$$

$$k_4 = hf(x_n + h, y_n + k_3) \tag{16}$$

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \tag{17}$$

RK4 achieves high accuracy even with relatively larger step sizes.

5 Algorithm and Pseudocode

This section presents the step-by-step algorithm and pseudocode used to numerically solve first-order ordinary differential equations using recursive implementations of Euler, Modified Euler (Heun's), and Runge–Kutta (RK4) methods.

5.1 General Algorithm

1. Define the first-order differential equation $\frac{dy}{dx} = f(x, y)$.
2. Input the initial condition (x_0, y_0).
3. Select the numerical method:
    - Method 1: Euler Method
    - Method 2: Modified Euler (Heun's) Method
    - Method 3: Fourth-Order Runge–Kutta (RK4)
4. Choose a suitable step size h.
5. Compute the number of steps $n = \frac{x-x_0}{h}$.
6. Recursively compute y_{n+1} until the base condition is reached.
7. Output the approximate value of $y(x)$.

5.2 Recursive Pseudocode

Euler Method (Recursive)

```
Euler(h, n, x, y):
    if n == 0:
        return y
    return Euler(h, n-1, x+h, y + h*f(x,y))
```

Modified Euler (Heun's Method)

```
ModifiedEuler(h, n, x, y):
    if n == 0:
        return y
    k1 = h*f(x,y)
    k2 = h*f(x+h, y+k1)
    return ModifiedEuler(h, n-1, x+h, y + (k1+k2)/2)
```

Runge–Kutta Fourth Order (RK4)

```
RK4(h, n, x, y):
    if n == 0:
        return y
    k1 = h*f(x,y)
    k2 = h*f(x+h/2, y+k1/2)
    k3 = h*f(x+h/2, y+k2/2)
    k4 = h*f(x+h, y+k3)
    return RK4(h, n-1, x+h, y + (k1+2k2+2k3+k4)/6)
```

5.3 Role of Recursion

Each numerical method is implemented using recursion, where:

- Each recursive call corresponds to one integration step.
- The base case ensures termination when the required number of steps is reached.
- Recursion provides a clean mathematical representation of iterative numerical schemes.

5.4 Flowchart of Recursive Numerical Method

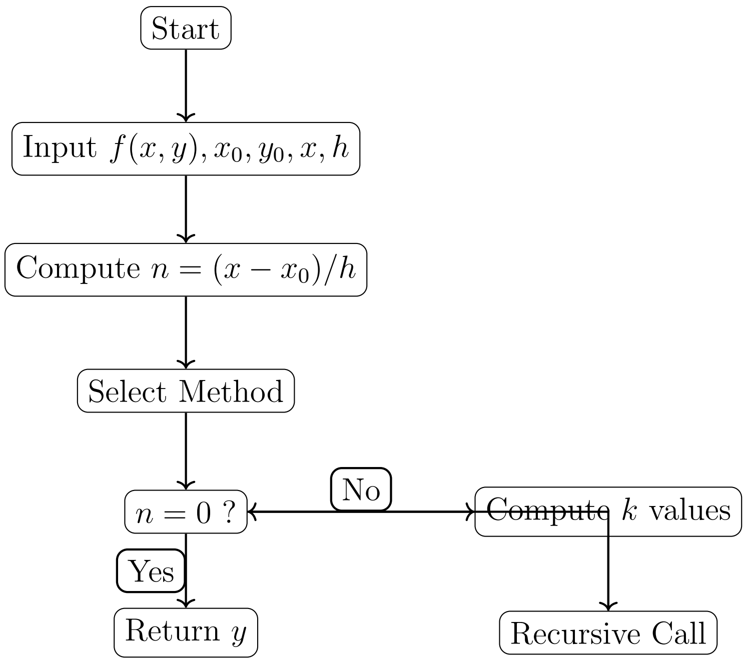

Figure 1: Flowchart of Recursive Numerical Solution

5.5 Recursion Tree Representation

The recursive structure of the numerical methods can be visualized as a linear recursion tree. For example, Euler's method follows:

$$Euler(n) \to Euler(n-1) \to Euler(n-2) \to \cdots \to Euler(0)$$

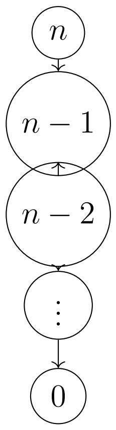

Figure 2: Recursion Tree for Numerical Integration

This recursion tree highlights that the depth of recursion is directly proportional to the number of integration steps, emphasizing the importance of step size selection.

6 Program Description

The C++ program allows users to define any first-order differential equation in the form $f(x, y)$. Recursive implementations of Euler, Modified Euler, and RK4 methods are provided. The function `approx()` selects the numerical method based on user input and computes the solution using a fixed step size of $h = 10^{-5}$.

7 Results and Comparison

For $x = 1$ and $y(0) = 1$, the numerical results obtained are compared with the exact value.

Method	**Approximate Value of $y(1)$**	**Exact Value**	**Error**
Euler Method	≈ 2.71	2.71828	Higher
Modified Euler Method	≈ 2.718	2.71828	Moderate
Runge–Kutta (RK4)	≈ 2.71828	2.71828	Very Low

Table 1: Comparison of Numerical Methods with Exact Solution

The results clearly show that RK4 provides the highest accuracy, followed by the Modified Euler method, while the Euler method is the least accurate.

8 Conclusion

This project demonstrates the effectiveness of numerical methods for solving first-order ODEs. While Euler's method is simple, its accuracy is limited. The Modified Euler method offers a balance between accuracy and computational cost, whereas the RK4 method provides excellent accuracy even with larger step sizes. The modular structure of the program allows easy extension to other differential equations.

9 Learning Outcomes

Through this project, the following outcomes and insights were achieved:

- Developed a clear understanding of first-order ordinary differential equations and their numerical treatment.
- Gained theoretical and practical knowledge of Euler, Modified Euler (Heun's), and Fourth-Order Runge–Kutta (RK4) methods.
- Understood the role of step size in controlling accuracy and numerical stability.
- Learned to compare numerical approximations with exact analytical solutions and interpret errors.
- Improved proficiency in implementing mathematical algorithms using C++ with a modular and extensible design.
- Appreciated why higher-order methods like RK4 provide superior accuracy with comparable computational effort.

Source Code

```
#include<iostream>
using namespace std;

double diff_eqn(double x, double y){
    return 2*x*y;
}

double euler(double step_size, double step, double x0, double y0){
    if(step == 0){
        return y0;
    }
    return euler(step_size, step - 1,
                 x0 + step_size,
                 y0 + step_size*diff_eqn(x0,y0));
}

double modified_euler(double step_size, double step, double x0, double y0){
    if(step == 0){
        return y0;
    }
    double k1 = step_size*diff_eqn(x0,y0);
    double k2 = step_size*diff_eqn(x0 + step_size, y0 + k1);
    return modified_euler(step_size, step - 1,
                          x0 + step_size,
                          y0 + (k1 + k2)/2);
}

double rk4(double step_size, double step, double x0, double y0){
    if(step == 0){
        return y0;
    }
    double k1 = step_size*diff_eqn(x0,y0);
    double k2 = step_size*diff_eqn(x0 + step_size/2, y0 + k1/2);
    double k3 = step_size*diff_eqn(x0 + step_size/2, y0 + k2/2);
    double k4 = step_size*diff_eqn(x0 + step_size, y0 + k3);
    return rk4(step_size, step - 1,
               x0 + step_size,
               y0 + (k1 + 2*k2 + 2*k3 + k4)/6);
}

double approx(int method, double x, double x0, double y0){
    double step_size = 0.00001;
    int step = (x - x0) / step_size;

    switch(method){
        case 1: return euler(step_size, step, x0, y0);
        case 2: return modified_euler(step_size, step, x0, y0);
        case 3: return rk4(step_size, step, x0, y0);
        default: return 0;
    }
}

int main(){
    cout << "EULER = " << approx(1,1,0,1) << endl;
    cout << "MODIFIED EULER = " << approx(2,1,0,1) << endl;
    cout << "RK4 = " << approx(3,1,0,1) << endl;
}
```