Numerical Solution of Initial Value Problems Using the Euler Method in Rust

Biswajit Gorai

1 Problem Statement

We are given the following initial value problem (IVP):

$$\frac{dy}{dt} = \cos(t) - y, \quad y(0) = 1, \quad 0 \leq t \leq 5$$

Our objective is to solve this differential equation using:

1. The analytical method (exact solution)
2. Euler's method (numerical approximation)

2 Analytical Solution

We begin by rewriting the equation in the standard linear form:

$$\frac{dy}{dt} + y = \cos(t)$$

This is a first-order linear ordinary differential equation (ODE). We solve it using the integrating factor method.

Step 1: Compute the Integrating Factor

$$\mu(t) = e^{\int 1\,dt} = e^t$$

Step 2: Multiply Both Sides by the Integrating Factor

$$e^t \cdot \frac{dy}{dt} + e^t \cdot y = e^t \cdot \cos(t) \Rightarrow \frac{d}{dt}\left(ye^t\right) = e^t \cos(t)$$

Step 3: Integrate Both Sides

$$\int \frac{d}{dt}\left(ye^t\right) dt = \int e^t \cos(t)\, dt$$

Using the known identity:

$$\int e^t \cos(t)\, dt = \frac{1}{2}e^t(\sin(t) + \cos(t)) + C$$

Step 4: Solve for $y(t)$

$$y(t)e^t = \frac{1}{2}e^t(\sin(t) + \cos(t)) + C \Rightarrow y(t) = \frac{1}{2}(\sin(t) + \cos(t)) + Ce^{-t}$$

Step 5: Apply Initial Condition

Given $y(0) = 1$, we find C:

$$1 = \frac{1}{2}(0 + 1) + C \cdot 1 \Rightarrow C = \frac{1}{2}$$

Final Analytical Solution

$$y(t) = \frac{1}{2}(\cos(t) + \sin(t) + e^{-t})$$

3 Numerical Solution: Euler's Method

Euler's method is a simple numerical technique to approximate solutions of ODEs of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

The formula used is:

$$y_{i+1} = y_i + h \cdot f(t_i, y_i)$$

Where:

- h is the step size: $h = \frac{t_{\text{end}} - t_0}{n}$
- $f(t, y) = \cos(t) - y$
- $y_0 = 1$, $t_0 = 0$

By iteratively applying the Euler update, we obtain an approximate numerical solution to the IVP.

4 Solving the Problem Using Python

To verify the correctness of our analytical and numerical approaches, we implemented the solution in Python. The Euler method was used for the numerical approximation, and the analytical solution was also calculated for comparison. Finally, both results were plotted using `matplotlib`.

Python Code

Github link :euler$_{m}$*ethod.py*

Explanation

- The function $f(t, y) = \cos(t) - y$ defines the right-hand side of the differential equation.
- The Euler method iteratively computes approximate values of y at discrete time steps.
- The exact solution is computed using the known analytical formula:

$$y(t) = \frac{1}{2}(\cos(t) + \sin(t) + e^{-t})$$

- The output plot shows how the Euler approximation converges to the exact solution as the number of steps increases.

This Python implementation provides a reference solution to validate our Rust implementation.

5 Familiarization with Rust and Project Structure

Before implementing the Euler method in Rust, I familiarized myself with key concepts of the Rust programming language and designed a modular directory layout suitable for a numerical computation project. This section outlines the main language features and tools used.

1. Variables and Data Types Used

Rust is a statically typed language, meaning all variables must have a known type at compile time. Below are the primary data types and variables used in this project:

- **f64:** 64-bit floating-point number. Used for time (`t`), solution values (`y`), step size (`h`), and function return values. Example: `let h: f64 = 0.25;`
- **usize:** Unsigned integer type used for indexing arrays or vectors and iteration count (`n`). Example: `let n: usize = 100;`
- **Vec<f64>:** A dynamically sized growable array for storing time points, numerical solution, and exact values. Example: `let mut t: Vec<f64> = vec![0.0; n+1];`
- **String:** A UTF-8 encoded growable string type used for user input. Example: `let mut input = String::new();`
- **Result¡T, E¿:** Used for error handling when writing to files or plotting. Example: `fn write_csv(...) -> Result<(), Box<dyn Error>>`
- **bool:** Boolean values used in conditional expressions (though not used explicitly in this assignment).

2. Modularization: `mod` and `use`

Rust projects are organized into modules for better maintainability. We used the following pattern:

```
// main.rs
mod solver;
mod exact;
mod utils;
mod plot;

use solver::euler;
use exact::exact_solution;
use utils::write_csv;
use plot::generate_plot;
```

Module functionality was shared across files using Rust's `mod` and `use` declarations, promoting clarity and separation of concerns.

3. Key Concepts Learned and Used

- **Mutable vs Immutable Variables:** By default, variables in Rust are immutable. To allow mutation, the `mut` keyword is used. `let mut y = 0.0;`
- **Functions with Return Types:** Every function explicitly defines its return type. Example: `fn exact_solution(t: f64) -> f64`
- **Vector Operations and Functional Iterators:** Vectors (`Vec`) were used extensively. Rust's `map` method applies a function to each element:

  ```
  let y_exact: Vec<f64> = t.iter().map(|&ti| exact_solution(ti)).collect();
  ```

- **for Loops:** Iteration is done using ranges: `for i in 0..n`
- **Trigonometric and Exponential Functions:** Rust provides math functions as methods on `f64`, and constants via `std::f64::consts`:

  ```
  let val = t.cos() + t.sin() + (-t).exp();
  let pi = std::f64::consts::PI;
  ```

- **Input Parsing:** User input is read and parsed using:

  ```
  let mut input = String::new();
  std::io::stdin().read_line(&mut input).unwrap();
  let n: usize = input.trim().parse().unwrap();
  ```

- **Crate Dependencies:** Dependencies are declared in `Cargo.toml`:

  ```
  [dependencies]
  csv = "1.3"
  plotters = "0.3"
  ```

- **Error Handling:** File writing and plotting functions return `Result` types and use the `?` operator for propagation.
- **Formatting Output:** Rust's `println!()` macro allows formatted output: `println!("Value of h = :.4", h);`

4. Plotting with Plotters Crate

The `plot.rs` file generates PNG plots using the `plotters` crate. It includes:

- `BitMapBackend` – creates a PNG drawing surface.
- `ChartBuilder` – sets layout, margin, and axis descriptions.
- `LineSeries` – plots both Euler and exact values.
- `Legend and Labels` – for visual clarity and comparison.

Plot Example Snippet:

```
chart.draw_series(LineSeries::new(
    t.iter().zip(y.iter()).map(|(&x, &y)| (x, y)), &BLUE
))?.label("Euler");

chart.draw_series(LineSeries::new(
    t.iter().zip(y_exact.iter()).map(|(&x, &y)| (x, y)), &RED
))?.label("Exact");
```

This creates a clear comparison between numerical and analytical solutions.

5. Compilation and Execution

The program is compiled and run using the Cargo tool:

```
cargo build       # Compiles the project
cargo run         # Runs the project and prompts for n
```

After entering the desired number of steps (`n`), the following output files are created in the `output/` directory:

- `iteration_n.csv` – CSV file with time, Euler y, exact y, and error.
- `iteration_n.png` – Graphical plot comparing Euler and exact solutions.

Implementation Strategy

I structured the program modularly, using multiple source files:

- **main.rs** – Entry point for the program
- **solver.rs** – Implements Euler's method
- **exact.rs** – Defines the exact solution function
- **utils.rs** – Writes output to CSV
- **plot.rs** – Generates the solution plot using the `plotters` crate

Euler Method in Rust

The core function to implement Euler's method is:

```
pub fn euler(t0: f64, y0: f64, tend: f64, n: usize) -> (Vec<f64>, Vec<f64>) {
    let h = (tend - t0) / n as f64;
    let mut t = vec![0.0; n + 1];
    let mut y = vec![0.0; n + 1];
    t[0] = t0;
    y[0] = y0;

    for i in 0..n {
        t[i+1] = t[i] + h;
        y[i+1] = y[i] + h * (t[i].cos() - y[i]);
    }

    (t, y)
}
```

Exact Solution in Rust

The analytical solution function is written as:

```
pub fn exact_solution(t: f64) -> f64 {
    0.5 * (t.cos() + t.sin() + (-t).exp())
}
```

User Input and Execution (main.rs)

The user is prompted to input the number of steps, and the Euler method is invoked:

```rust
use std::io;

fn main() {
    let mut input = String::new();
    println!("Enter number of iterations: ");
    io::stdin().read_line(&mut input).unwrap();
    let n: usize = input.trim().parse().unwrap();

    let t0 = 0.0;
    let y0 = 1.0;
    let tend = 5.0;

    let (t, y) = euler(t0, y0, tend, n);
    let y_exact: Vec<f64> = t.iter().map(|&ti| exact_solution(ti)).collect();
```

CSV Output

The computed values are written to a CSV file named based on the iteration count:

```rust
let filename = format!("output/iteration_{}.csv", n);
let mut wtr = Writer::from_path(filename)?;
for i in 0..=n {
    wtr.write_record(&[
        t[i].to_string(),
        y[i].to_string(),
        y_exact[i].to_string(),
        ((y[i] - y_exact[i]).abs()).to_string()
    ])?;
}
```

Plotting in Rust

The plot is generated using the `plotters` crate. A PNG image comparing the Euler approximation and exact solution is saved:

```rust
let filename = format!("output/iteration_{}.png", n);
let root = BitMapBackend::new(&filename, (800, 600)).into_drawing_area();
let mut chart = ChartBuilder::on(&root)
    .caption("Euler vs Exact", ("Arial", 20))
    .build_cartesian_2d(t[0]..t[n], y_min..y_max)?;

chart.draw_series(LineSeries::new(
    t.iter().zip(y.iter()).map(|(&x, &y)| (x, y)), &BLUE,
))?.label("Euler");
```

```
chart.draw_series(LineSeries::new(
    t.iter().zip(y_exact.iter()).map(|(&x, &y)| (x, y)), &RED,
))?.label("Exact");
```

Output Files

Two files are generated after execution:

- `output/iteration_n.csv` – Contains time, Euler y, exact y, and absolute error
- `output/iteration_n.png` – Visualization of the two solutions

6 Conclusion

This assignment served as both a numerical analysis exercise and a practical introduction to systems programming using Rust.

From the numerical standpoint, the Euler method provided a straightforward way to approximate the solution over the interval $[0, 5]$. The accuracy of the method improved with increasing step counts (n), which was evident in both the plotted curves and the computed absolute errors.

Despite Rust's steeper learning curve compared to high-level scripting languages like Python, it offered significant benefits in terms of speed, memory safety, and predictable behavior.

Overall, this assignment not only reinforced theoretical concepts in ordinary differential equations and numerical methods but also provided practical skills in system-level programming with Rust. It opened the door to exploring more advanced numerical schemes and encouraged using Rust for scientific computing applications, where both safety and performance are critical.