# Task Report
# Flax & Teal - Internship Application
# (solving IVP by Euler's method in rust)


by

Satheesh D M

MA24M023


05/07/25

---

## Contents

# 1 Problem Statement

Solve the following initial value problem (IVP) using the Euler's method in Rust:

$$y^{'} = \cos(t) - y; \qquad 0 \leq t \leq 5; \quad y(0) = 1 \tag{1.1}$$

with the different n values (10, 25 and 1000). Then compare with the analytic solution and plot.

# 2 Analytical Solution

Given the Initial Value Problem (IVP),

$$y^{'} = \cos(t) - y; \qquad 0 \leq t \leq 5; \quad y(0) = 1 \tag{2.1}$$

$$\frac{dy}{dt} + y = \cos(t) \tag{2.2}$$

This is a standard First order ODE with variable coefficients. The general form of this IVP is,

$$\frac{dy}{dt} + P(t)\ y = Q(t) \tag{2.3}$$

on comparing equations (2.2) and (2.3) we have, $P(t) = 1$ and $Q(t) = \cos(t)$. The general soultion is of the form,

$$y * (I.F) = \int (I.F) * Q(t)dt + C \tag{2.4}$$

here, $(I.F)$ is called the integrating factor and is given by, $(I.F) = e^{\int P(t)dt}$. And $C$ be the real valued integration constant. Now, calculating the integrating factor for the given IVP,

$$(I.F) = e^{\int P(t)dt} = e^{\int (1)dt} = e^{t} \tag{2.5}$$

Hence the solution for the IVP is given by the following equation, from equation (2.4)

$$y\ e^{t} = \int \cos(t)\ e^{t}dt + C \tag{2.6}$$

Solving the RHS of the equation (2.12),

$$\int \cos(t)\ e^t dt = \int \frac{(e^{it} + e^{-it})}{2}\ e^t dt \tag{2.7}$$

$$= \int \frac{(e^{(1+i)t} + e^{(1-i)t})}{2} dt \tag{2.8}$$

$$= \frac{e^{(1+i)t}}{2(1+i)} + \frac{e^{(1-i)t}}{2(1-i)} \qquad \text{(on simplification...)} \tag{2.9}$$

$$= \frac{e^t}{2}(\cos(t) + \sin(t)) \tag{2.10}$$

from equations (2.12) and (2.10); the general analytical solution is,

$$y(t) = \frac{1}{2}(\cos(t) + \sin(t)) + e^{-t}C \tag{2.11}$$

substituting the given initial condition from (1.1), we get $C = 1/2$. Hence the analytical solution for the given IVP is,

$$\boxed{y(t) = \frac{1}{2}(\cos(t) + \sin(t)) + \frac{e^{-t}}{2}} \tag{2.12}$$

# 3 Numerical Solution - Euler's Method

Euler's method is a first-order numerical procedure for solving ODEs with a given initial value. Now consider a general first-order ODE with its initial conditions,

$$\frac{dy}{dt} = f(t, y(t)); \qquad y(t = t_o) = y_o \tag{3.1}$$

We want to approximate the solution near the initial point $t = t_o$. So at this point we have the ODE.

$$\left.\frac{dy}{dt}\right|_{t=t_o} = f(t_o, y(t_o)); \tag{3.2}$$

By using the definition of the derivative, the above can be written in difference form as,

$$y = y_o + f(t_o, y(t_o))(t - t_o) \tag{3.3}$$

On creating a 1D mesh (discretizing the independent variable domain), with uniform spacing (say, $h$) between $n+1$ points. Then the above definition can be modified to obtain the Euler's solution.

$$\boxed{y_{k+1} = y_k + f(t_k, y(t_k)) * h; \qquad h = (t_{k+1} - t_k); \quad k \in \{0, 1, 2, \ldots, n\}} \tag{3.4}$$

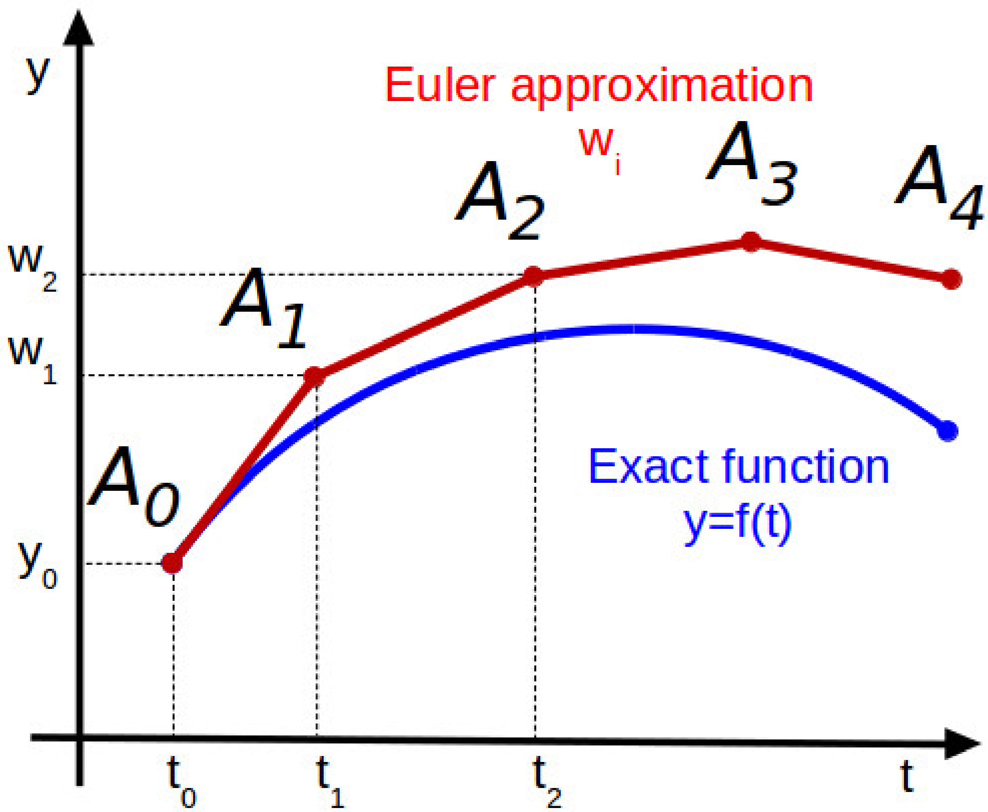



Figure 3.1: Illustration. Image Credits: https://x-engineer.org/euler-integration/

**Algorithm 1** Euler's Method for Solving an Initial Value Problem

**Require:** Derivative function $f(t, y)$, start time $t_0$, end time $t_n$, initial value $y_0$, number of steps $n$

```
Compute step size h ← (t_n − t_0)/n
Generate mesh: T ← [t_0 + i · h | i = 0, ..., n]
Initialize solution array: Y ← array of size n + 1
Set initial condition: Y[0] ← y_0
for k = 0 to n − 1 do
    Y[k + 1] ← Y[k] + h · f(T[k], Y[k])
end for
return Arrays T, Y (mesh and solution values)
```

# 4 Python Implementation

Though the aim is to solve the IVP in rust programming language, I solved this first in python to get an approach and project structure. Then the project and all functionalities are migrated into rust language.

## Project structure :

```
python_code
├── README.md
├── environment.yml
├── jupyter_nbooks
│   ├── IVP_Euler_Solver.ipynb
│   └── report_plots.ipynb
├── requirements.txt
├── src
│   ├── __init__.py
│   ├── config.ini
│   ├── main.py             # entry point (command line executable)
│   ├── solution.csv
│   └── solvers.py
└── tests
    ├── __init__.py
    └── test_solvers.py     # tested the code logic
```

## Standard coding practices followed:

- **Code Modularization :** The code is split into 3 files inside the "**`python_code/src/.`**"
  - a command line executable python script. (entry point - contains main function). (**`main.py`**)
  - a library script with the helper class, OOP approach is used. (**`solvers.py`**)
  - a file to get inputs from the user, (**`config.ini`**). Solves a general $1^{st}$ order ODE, (IVP).
- **Docstrings :** Detailed comments and method or function summaries are included.
- **Type Hinting :** To capture errors early. (using `mypy`, a static type checker).
- **Exception handling :** Uses "`if not ... or ...`" to check parsed values from `config.ini`
- **Logging :** Uses `logging` package to log every individual runs at `python_code/src/logs/`
- **Testing :** tested all the library functions or methods in the `solvers.py` file.
- **Virtual environment :** environment.yml (conda env) & requirements.txt (pip dependencies)

```python
def solve(self) -> np.ndarray:
    """
    Solve the IVP using Euler's method.

    Returns:
        np.ndarray: Array of y values at each mesh point
    """
    y = np.zeros(self.num_steps + 1)
    y[0] = self.y_0

    for k in range(self.num_steps):
        t = self.mesh[k]
        y[k + 1] = y[k] + self.h * self.f(t, y[k])

    return y
```

Figure 4.1: Euler's method logic inside the Class.

## User Inputs : (`config.ini`)

```ini
# Mesh configuration
[mesh_1D]
n = 10
domain_start = 0.0
domain_end = 5.0

; Initial values for the ODE
[initial_conditions]
y_0 = 1.0

# ODE function definition
#   dy/dt = f(t, y)
#   expression = f(t, y)
[ode_function]
expression = np.cos(t) - y
; use numpy convention to define the function
```

Figure 4.2: All the features of a general first order ODE (IVP) is configurable through this file.

## Outputs :

The output is exported both as a "`.csv`" file and as a line plot.

## Approach :

- Started with a "`.ipynb`" file. Made a (**trial & error, not so neat**) working code logic of Euler's method in procedural programming approach. This code served as a reference while modularizing.
- Then split the logic into the above-mentioned 3 source code files.

# 5 Rust Implementation

## Project structure :

```
rust_code
├── Cargo.lock
├── Cargo.toml
├── README.md
├── config.ini          # user inputs
├── solution.csv
└── src
    ├── lib.rs          # library crate
    └── main.rs         # binary crate (command line executable)
```

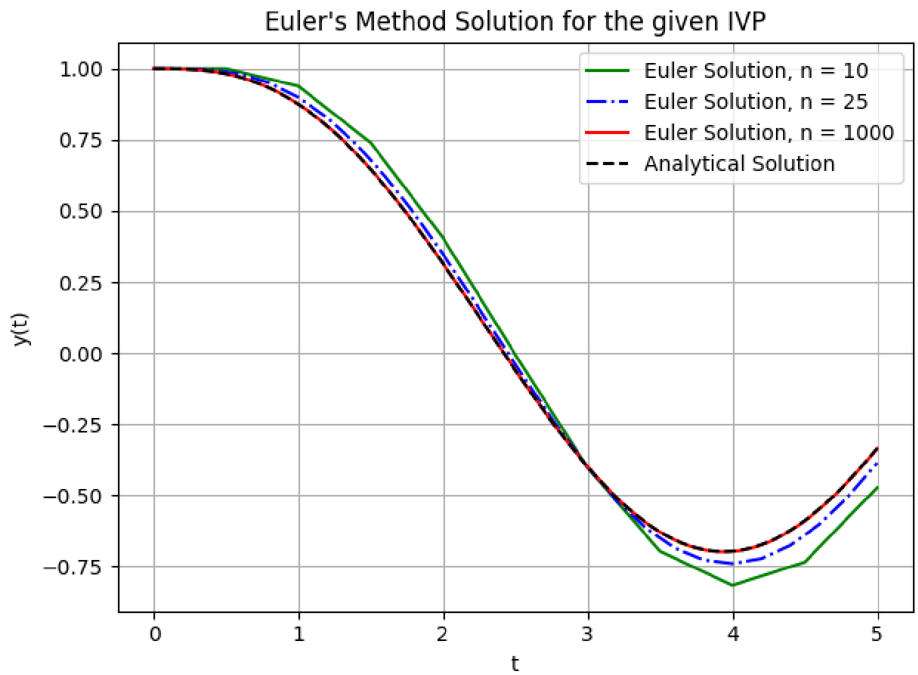
## Comments :

- All the standard practices mentioned in the python implementation is also followed here almost.
- The project structure is also similar. (`config.ini`, `src/main.rs` and `src/lib.rs` files).
- But certain things are implemented in rusty manner.
  - **Package Managing :** It is done using `cargo`. It comes with standard rust installation.
  - **Environment :** Dependencies are maintained automatically by `Cargo.lock` and `Cargo.toml`
  - **Type hinting :** Rust is a statically typed language. Hence it becomes mandatory.
  - **Logging :** Rust already capture all errors while compilation. So, avoided purposefully.
  - Other practices like **Docstings, Exception & error handling, Testing** are also followed.
- Outputs are exported only as a ”`.csv`” file.
- **Approch :** The Python implementation made it easier while migrating. Followed it.
- Didn’t follow the OOP paradigm. But the rust features `struc`, `fn`, `impl` were used to group the data and their associated methods.

## Repository Link :

https://github.com/SATHEESH-D-M/flax-teal_assignment

# 6 Results



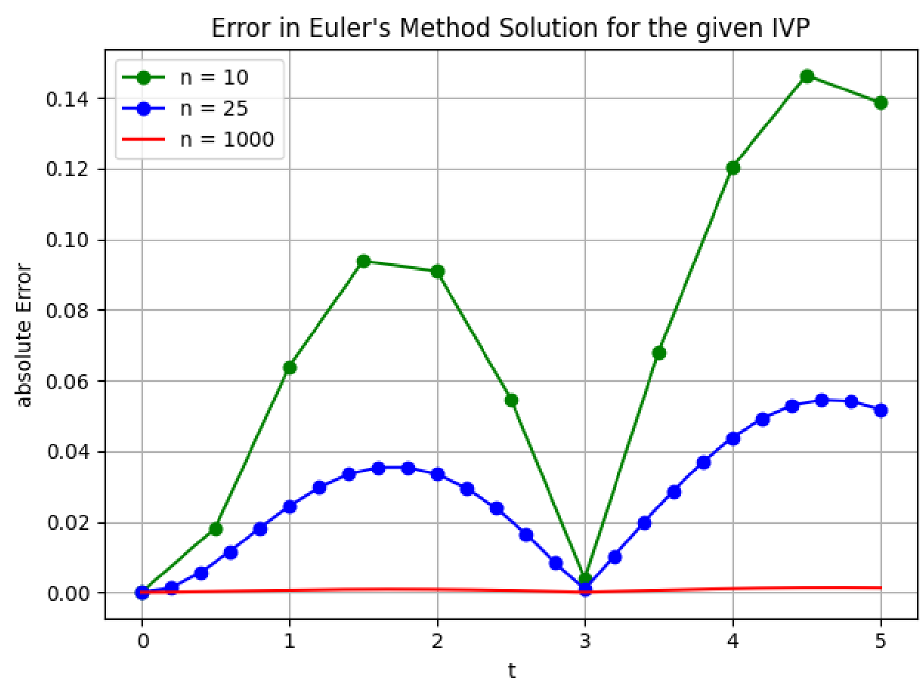




Figure 6.1: Absolute error, because Euler's method uses only the first derivative (linear approximation) for solving. It misses the curvature of original y(t). To reduce this error, increase step count (n).