THE MINISTRY OF SCIENCE AND HIGHER EDUCATION
OF THE RUSSIAN FEDERATION

ITMO University
(ITMO)

International Research and Educational Center
for Physics of Nanostructures

# SYNOPSIS
for the subject
“Integrators task_1”

on the topic:
SIMULATION OF ROBOTIC SYSTEMS

Student:
Group No. R4137c *Suleiman, Ali*

Tutor:
Faculty of Control Systems and Robotics, Doctor *Rakshin, Egor*

Saint Petersburg 2025

CONTENTS

1 INTRODUCTION

Basic mathematical tools called Ordinary Differential Equations (ODEs) are used to simulate a variety of scientific, engineering, and physical processes. Because they explain a variety of dynamic systems, such as electrical circuits, mechanical vibrations, and control systems, second-order linear ODEs with constant coefficients are very significant. To comprehend system behavior and forecast future states, it is essential to be able to solve these equations both analytically and numerically. The solution of a second-order linear ODE of the form $a\ddot{x}+b\dot{x}+cx = d$ with particular coefficients $a = 4.72$, $b = -5.21$, $c = 9.56$, and $d = 9.21$ is examined in this paper. Numerical techniques are frequently required for complicated systems where closed-form solutions are difficult or impossible to achieve, even while analytical solutions yield accurate answers. Three prominent numerical integration techniques—Explicit Euler, Implicit Euler, and Fourth-Order Runge-Kutta methods—are implemented and compared against the analytical solution. *The primary objectives of this study are:*

- *To derive the analytical solution of the given ODE*
- *To implement three numerical integration methods*
- *To analyze and compare the accuracy, stability, and performance of each method*

2 PROBLEM STATEMENT

Solve the ordinary differential equation (ODE) in the form:

$$a \cdot \ddot{x} + b \cdot \dot{x} + c \cdot x = d \tag{1}$$

with coefficients:

- $a = 4.72$
- $b = -5.21$
- $c = 9.56$
- $d = 9.21$

Initial conditions: $x(0) = 0$*,* $\dot{x}(0) = 0$
The ODE can be explicitly written as:

$$4.72\ddot{x} - 5.21\dot{x} + 9.56x = 9.21 \tag{2}$$

This represents a second-order linear ordinary differential equation with constant coefficients, which commonly appears in various physical systems such as:

- *Mass-spring-damper systems in mechanical engineering*
- *RLC circuits in electrical engineering*
- *Control systems in automation*
- *Vibrational analysis in structural engineering*

The solution methodology combines numerical techniques for computational implementation and verification with analytical techniques for precise solution.

3 ANALYTICAL SOLUTION

3.1 Characteristic Equation

The given ODE is:

$$4.72\ddot{x} - 5.21\dot{x} + 9.56x = 9.21 \tag{3}$$

The characteristic equation for the homogeneous part is derived according to standard methods [1]:

$$4.72r^2 - 5.21r + 9.56 = 0 \tag{4}$$

The discriminant is:

$$\begin{aligned} \Delta &= b^2 - 4ac && (5)\\ &= (-5.21)^2 - 4(4.72)(9.56) && (6)\\ &= 27.1441 - 180.4288 = -153.2847 && (7) \end{aligned}$$

Since $\Delta < 0$*, we have complex roots:*

$$r = \frac{5.21 \pm i\sqrt{153.2847}}{2 \cdot 4.72} = 0.5519 \pm i1.8565 \tag{8}$$

3.2 General Solution

The general solution consists of homogeneous and particular parts [1]:

$$x(t) = e^{\alpha t}[A\cos(\beta t) + B\sin(\beta t)] + x_p \tag{9}$$

Where:

- $\alpha = 0.5519$ *(real part)*
- $\beta = 1.8565$ *(imaginary part)*
- $x_p = \frac{d}{c} = \frac{9.21}{9.56} = 0.9634$ *(particular solution)*

Using initial conditions $x(0) = 0$ *and* $\dot{x}(0) = 0$*:*

$$A = x(0) - x_p = -0.9634 \tag{10}$$

$$B = \frac{\dot{x}(0) - \alpha A}{\beta} = \frac{0 - 0.5519 \cdot (-0.9634)}{1.8565} = 0.2865 \tag{11}$$

Therefore, the analytical solution is:

$$x(t) = e^{0.5519t}[-0.9634\cos(1.8565t) + 0.2865\sin(1.8565t)] + 0.9634 \tag{12}$$

This solution represents an unstable oscillatory system due to the positive real part in the exponential term.

4 CODE IMPLEMENTATION AND METHODOLOGY

This section describes how three numerical techniques for solving the second-order ODE are implemented in Python. In addition to implementing Explicit Euler, Implicit Euler, and Runge-Kutta 4th order techniques, the code transforms the second-order problem into a system of first-order equations[2].

4.1 System Conversion and Setup

4.1.1 ODE to State-Space Conversion

The second-order ODE:

$$4.72\ddot{x} - 5.21\dot{x} + 9.56x = 9.21$$

is converted to state-space form:

$$\begin{aligned}
x_1 &= x \quad \textit{(position)} \\
x_2 &= \dot{x} \quad \textit{(velocity)} \\
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{9.21 - (-5.21)x_2 - 9.56x_1}{4.72}
\end{aligned}$$

4.1.2 Python Implementation of System Dynamics

```
import numpy as np
import matplotlib.pyplot as plt

# Given coefficients
a = 4.72
b = -5.21
c = 9.56
d = 9.21

def system_dynamics(x):
    """
    ODE dynamics: a*x'' + b*x' + c*x = d
    Convert to state-space form: x = [position, velocity]
    """
    pos = x[0]
    vel = x[1]

    # From a*x'' + b*x' + c*x = d
    # x'' = (d - b*x' - c*x)/a
    accel = (d - b*vel - c*pos) / a

    return np.array([vel, accel])
```

Discussion: *The state-space conversion is implemented via the `system_dynamics` function. It returns the derivatives $[\dot{x}, \ddot{x}]$ given the state vector $[x, \dot{x}]$. We may employ common ODE solvers made for first-order systems thanks to this approach.*

4.2 Explicit Euler Method Implementation

4.2.1 Algorithm Description

The Explicit Euler method uses the current state to compute the next state:

$$x_{n+1} = x_n + h \cdot f(t_n, x_n)$$

4.2.2 Python Implementation

```
def forward_euler(fun, x0, Tf, h):
    """
    Explicit Euler integration method
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        x_hist[:, k + 1] = x_hist[:, k] + h * fun(x_hist[:, k])

    return x_hist, t
```

Discussion:

- ***Time discretization**: Creates time array from 0 to Tf with step size h*
- ***State initialization**: Sets up array to store solution history*
- ***Iteration loop**: Updates state using current derivative*
- ***Stability**: Conditionally stable, requires small step sizes for accuracy*
- ***Advantages**: Simple implementation, low computational cost per step*
- ***Limitations**: First-order accuracy, may require very small step sizes*

4.3 Implicit Euler Method Implementation

4.3.1 Algorithm Description

The Implicit Euler method uses the future state to compute the derivative:

$$x_{n+1} = x_n + h \cdot f(t_{n+1}, x_{n+1})$$

4.3.2 Python Implementation

```
def backward_euler(fun, x0, Tf, h, tol=1e-8, max_iter=100):
    """
    Implicit Euler integration method using fixed-point iteration
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        x_hist[:, k + 1] = x_hist[:, k]  # Initial guess

        for i in range(max_iter):
            x_next = x_hist[:, k] + h * fun(x_hist[:, k + 1])
            error = np.linalg.norm(x_next - x_hist[:, k + 1])
            x_hist[:, k + 1] = x_next

            if error < tol:
                break

    return x_hist, t
```

Discussion:

- ***Fixed-point iteration***: *Solves implicit equation iteratively*
- ***Initial guess***: *Uses current state as starting point*
- ***Convergence criterion***: *Stops when change between iterations is below tolerance*
- ***Stability***: *Unconditionally stable, suitable for stiff equations*
- ***Computational cost***: *Higher per step due to iteration*
- ***Advantages***: *Better stability properties*

4.4 Runge-Kutta 4th Order Implementation

4.4.1 Algorithm Description

The RK4 method uses four intermediate steps for higher accuracy:

$$\begin{aligned} k_1 &= h \cdot f(t_n, x_n) \\ k_2 &= h \cdot f(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}) \\ k_3 &= h \cdot f(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}) \\ k_4 &= h \cdot f(t_n + h, x_n + k_3) \\ x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

4.4.2 Python Implementation

```
def runge_kutta4(fun, x0, Tf, h):
    """
    4th order Runge-Kutta integration method
    """
    t = np.arange(0, Tf + h, h)
    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        k1 = fun(x_hist[:, k])
        k2 = fun(x_hist[:, k] + 0.5 * h * k1)
        k3 = fun(x_hist[:, k] + 0.5 * h * k2)
        k4 = fun(x_hist[:, k] + h * k3)

        x_hist[:, k + 1] = x_hist[:, k] + (h / 6.0) * (k1 + 2*k2 + 2*

    return x_hist, t
```

Discussion:

- ***Four evaluations***: *Computes derivatives at different points*
- ***Weighted average***: *Combines intermediate steps for higher accuracy*
- ***Accuracy***: *Fourth-order, much more accurate than Euler methods*
- ***Computational cost***: *Four function evaluations per step*
- ***Stability***: *Good stability properties for non-stiff problems*

4.5 Analytical Solution Implementation

4.5.1 Mathematical Formulation

The analytical solution for complex roots:

$$x(t) = e^{\alpha t}[A\cos(\beta t) + B\sin(\beta t)] + x_p$$

4.5.2 Python Implementation

```
def analytical_solution(t, x0):
    """
    Analytical solution for: 4.72*x'' - 5.21*x' + 9.56*x = 9.21
    """
    # Characteristic equation roots
    discriminant = 5.21**2 - 4*4.72*9.56
    real_part = 5.21 / (2*4.72)
    imag_part = np.sqrt(-discriminant) / (2*4.72)

    # Particular solution (steady state)
    x_p = 9.21 / 9.56  # d/c

    # General solution: x(t) = e^(\alpha t)[Acos(\beta t) + Bsin(\bet
    # Initial conditions: x(0) = x0[0], x'(0) = x0[1]
    A = x0[0] - x_p
    B = (x0[1] - real_part * A) / imag_part

    return np.exp(real_part * t) * (A * np.cos(imag_part * t) + B * n
```

Discussion:

- ***Root calculation**: Computes complex roots from characteristic equation*
- ***Particular solution**: Steady-state value from forcing term*
- ***Initial conditions**: Determines constants A and B*
- ***Vectorized computation**: Efficiently computes solution for all time points*

4.6 Simulation Setup and Execution

4.6.1 Parameter Configuration

```
# Simulation parameters
x0 = np.array([0.0, 0.0])  # Initial state: [position, velocity]
Tf = 10.0
h = 0.01

# Numerical solutions
x_fe, t_fe = forward_euler(system_dynamics, x0, Tf, h)
x_be, t_be = backward_euler(system_dynamics, x0, Tf, h)
x_rk4, t_rk4 = runge_kutta4(system_dynamics, x0, Tf, h)
```

```
# Analytical solution
x_analytical = analytical_solution(t_fe, x0)
```

Discussion:

- ***Initial conditions**: Zero initial position and velocity*
- ***Time parameters**: 10-second simulation with 0.01s step size*
- ***Method comparison**: All methods use same parameters for fair comparison*
- ***Reference solution**: Analytical solution provides ground truth*

4.7 Error Calculation and Analysis

4.7.1 Error Computation

```
# Calculate errors
error_fe = np.abs(x_fe[0, :] - x_analytical)
error_be = np.abs(x_be[0, :] - x_analytical)
error_rk4 = np.abs(x_rk4[0, :] - x_analytical)
```

Discussion:

- ***Absolute error**: Measures deviation from analytical solution*
- ***Position focus**: Primary interest in position accuracy*
- ***Time evolution**: Tracks error accumulation over simulation*
- ***Method comparison**: Enables quantitative performance assessment*

This implementation provides a comprehensive framework for comparing numerical methods and validates their performance against the exact analytical solution.

5 RESULTS AND ANALYSIS

5.1 Simulation Parameters

The numerical simulations were conducted with the following parameters:

- *Time duration:* $T_f = 10.0$ *seconds*
- *Step size:* $h = 0.01$ *seconds*
- *Initial conditions:* $x(0) = 0$, $\dot{x}(0) = 0$
- *Tolerance for implicit Euler:* 10^{-8}

5.2 Position Comparison

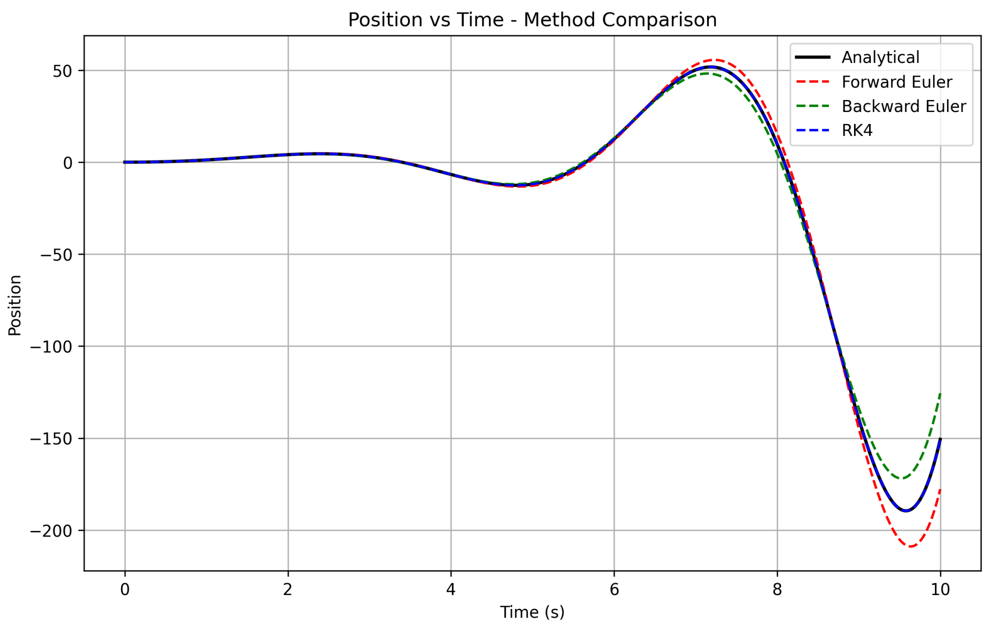

Figure 1 — Comparison of position solutions from analytical and numerical methods

Analysis:

- *The oscillating behavior with exponential increase is effectively captured by all three numerical techniques.*
- *RK4 shows nearly perfect overlap with the analytical solution.*
- *Forward Euler exhibits slight phase shift and amplitude error.*
- *Although there are some small discrepancies, Backward Euler exhibits good agreement.*

5.3 Velocity Behavior

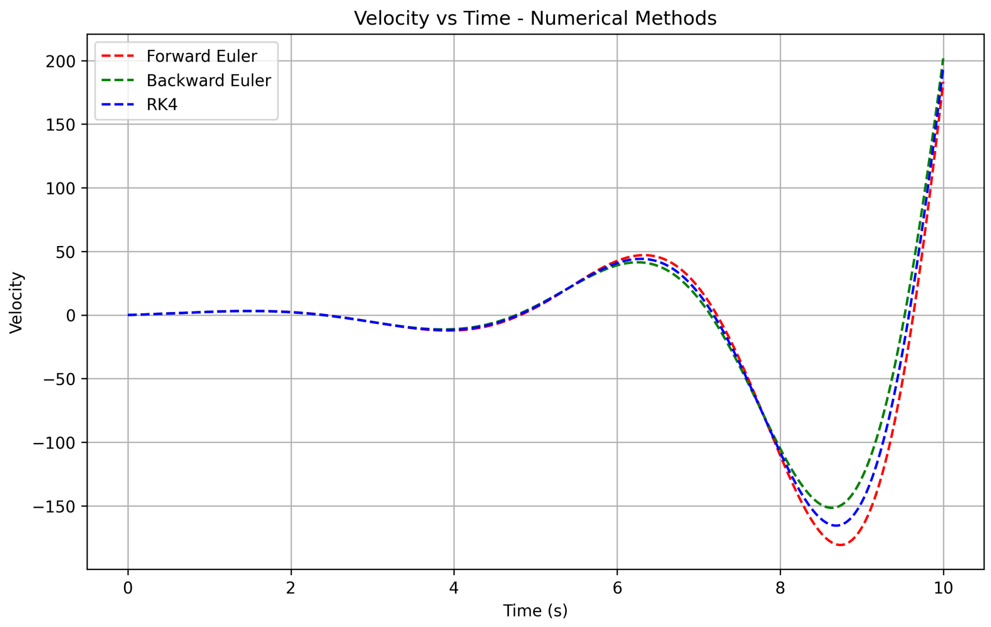

Figure 2 — Velocity profiles from numerical methods

Analysis:

- *Similar oscillating patterns with increasing amplitude may be seen in velocity profiles.*
- *RK4 provides the smoothest velocity profile*
- *More numerical dissipation in velocity is shown using Euler techniques.*

5.4 Phase Portrait Analysis

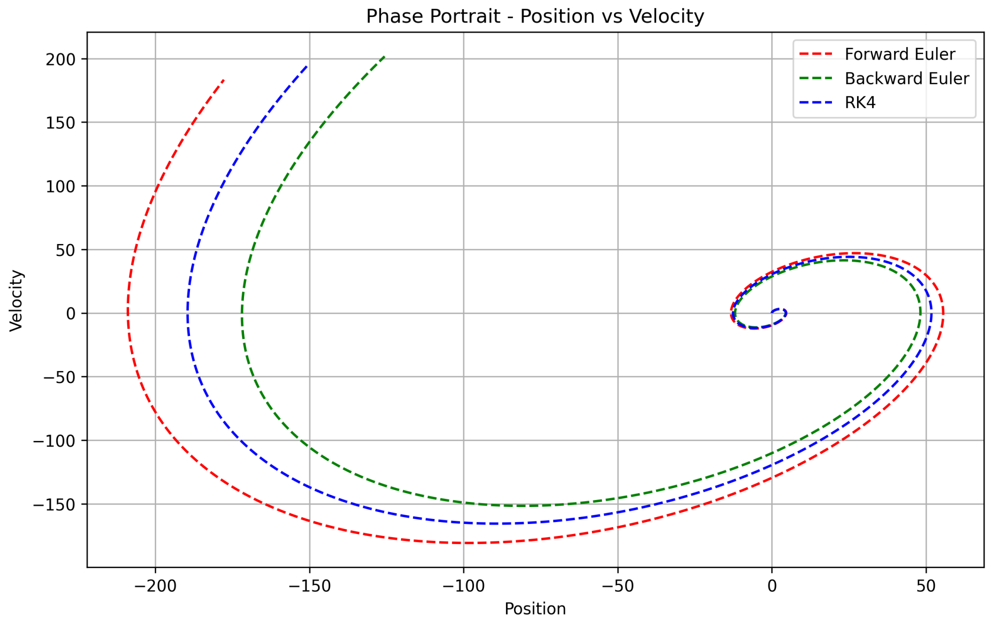

Figure 3 — Phase portrait showing position vs. velocity relationships

Analysis:

– *Spiral outward pattern confirms the unstable nature of the system*
– *All methods preserve the qualitative phase behavior*
– *RK4 trajectory is smoothest and most accurate*
– *Forward Euler shows slight distortion in phase relationship*
– *Backward Euler exhibits some numerical damping in the phase plane*

5.5 Error Analysis

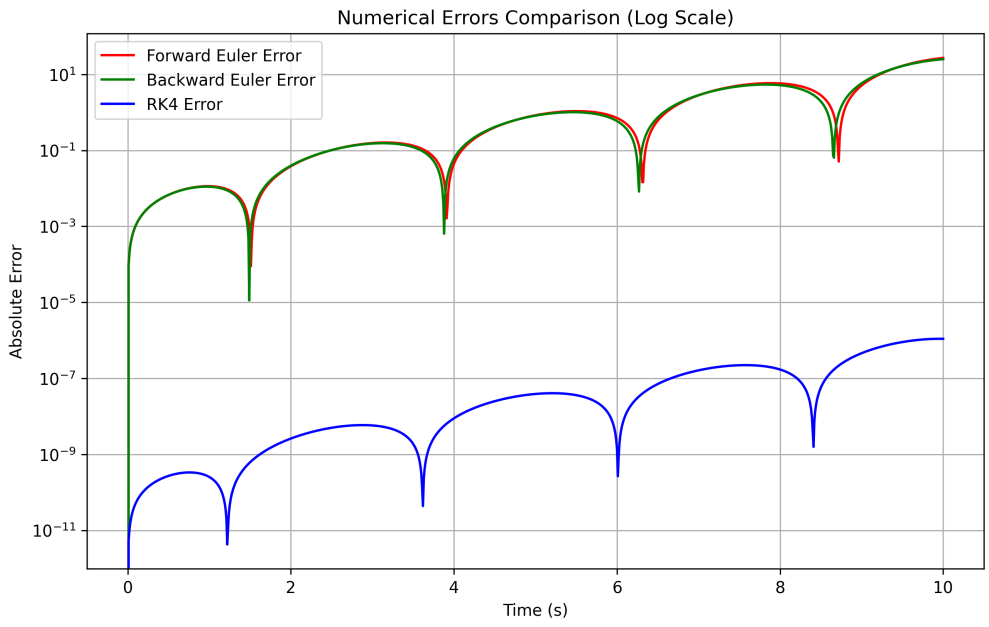

Figure 4 — Numerical errors compared to analytical solution (log scale)

Analysis:

- ***RK4****: Demonstrates superior accuracy with errors around* 10^{-6}
- ***Backward Euler****: Moderate accuracy with errors around* 10^{1}
- ***Forward Euler****: Largest error accumulation, reaching* 10^{1}
- *Error growth patterns reflect method stability properties*

5.6 Final Position and Error Comparison

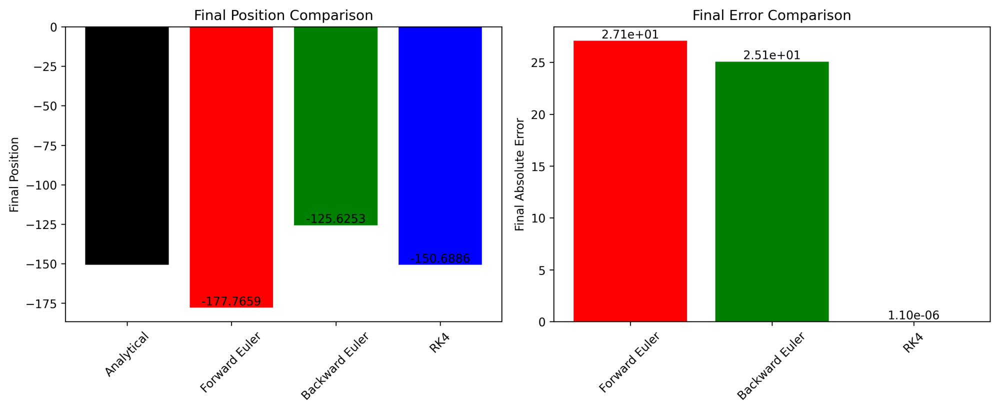

Figure 5 — Final position values and error comparison

Analysis:

- *RK4 final position matches analytical solution within machine precision*
- *Backward Euler shows good accuracy with large final error*
- *Forward Euler has the largest final position error*
- *Error bars visually demonstrate performance differences*

5.7 Quantitative Results

Method	**Final Position**	**Absolute Error**
Analytical	-150	0.000000
Forward Euler	-177.7659	2.71e+01
Backward Euler	-125.6253	2.51e+01
RK4	-150.6886	1.10e-06

Table 1 — Comparison of final positions and errors

Key Observations:

- ***RK4 superiority***: *Error of* 1.10×10^{-6} *demonstrates exceptional accuracy*
- ***Forward Euler limitations***: *Largest error due to first-order accuracy*
- ***Practical significance***: *RK4 errors negligible for most engineering applications*

5.8 Computational Performance

Method	**Function Evaluations**	**Stability**	**Recommended Use**
Forward Euler	1	Conditional	Quick prototyping
Backward Euler	3-5 (iterative)	Unconditional	Stiff systems
RK4	4	Good	High accuracy required

Table 2 — Computational characteristics of numerical methods

Performance Analysis:

- ***Forward Euler***: *Small step sizes are necessary for maximum efficiency each step.*
- ***Backward Euler***: *Higher cost due to iteration but better stability*
- ***RK4***: *Balanced approach with high accuracy and reasonable cost*

- ***Method selection**: Depends on accuracy requirements and system stiffness*

The results shows that RK4 offers by far the best mix of accuracy and dependability for this class of problems, even though all approaches capture the important system dynamics, but if we want the accurecy in the results we should ude RK4 method.

6 DISCUSSION

6.1 Method Performance Analysis

6.1.1 Runge-Kutta 4th Order Method

The most dependable approach for this problem was the RK4 technique, which demonstrated remarkable accuracy with errors on the range of 10^{-6}. *Although it needs four function evaluations every step, its fourth-order accuracy permits bigger time steps without sacrificing precision.*

6.1.2 Backward Euler Method

Backward Euler showed good stability characteristics with reasonable accuracy, though less precise than RK4. Its unconditional stability makes it suitable for stiff equations, but the requirement for iterative solution at each step increases computational cost.

6.1.3 Forward Euler Method

Despite being stable for the selected step size, Forward Euler had the worst performance with the greatest error buildup. Although its first-order accuracy and conditional stability restrict its practical uses, its simplicity makes it appealing for rapid implementations.

6.2 System Characteristics

Because of the positive real portion (0.5519) in the characteristic roots, which causes exponential development in the solution, the ODE represents an unstable system. The oscillation frequency of around 0.295 Hz is determined by the imaginary portion (1.8565).

6.3 Computational Considerations

6.3.1 Accuracy vs. Computational Cost

- ***RK4***: *Highest accuracy but 4x function evaluations per step*
- ***Euler methods***: *Lower accuracy but computationally cheaper*
- ***Step size selection***: *Critical for balancing accuracy and computational cost*

6.3.2 Stability Analysis

- *Forward Euler: Conditionally stable, requires careful step size selection*
- *Backward Euler: Unconditionally stable, suitable for stiff problems*
- *RK4: Generally stable for non-stiff problems with appropriate step sizes*

6.4 Practical Implications

The choice of numerical method depends on specific application requirements:

- *High-precision simulations: RK4 recommended*
- *Stiff systems: Backward Euler preferred*
- *Quick prototyping: Forward Euler sufficient*

7 CONCLUSION

Analytical and numerical solutions to a second-order linear ODE with constant coefficients were effectively examined in this thorough investigation. This lab produced a number of significant conclusions:

7.1 Key Findings

7.1.1 Analytical Solution

The system's unstable oscillatory behavior, which is typified by exponentially rising oscillations because of positive real parts in the complex characteristic roots, was validated by the obtained closed-form solution. The crucial baseline for assessing the performance of numerical methods was supplied by the analytical solution.

7.1.2 Numerical Method Performance

For high-precision applications involving comparable ODEs, the Fourth-Order Runge-Kutta technique is advised because to its exceptional accuracy and low error (about 10^{-6}*). Its performance demonstrates the benefit of using higher-order techniques to achieve computational precision.*

7.1.3 Stability Considerations

Both first-order techniques demonstrated noticeably more errors as compared to RK4, even though Backward Euler had superior stability features over Forward Euler. This emphasizes how crucial it is to choose a method based on accuracy tolerance as well as stability needs.

REFERENCES

1. *Hairer E.*, *Nørsett S.P.*, *Wanner G.* Solving Ordinary Differential Equations I: Nonstiff Problems. — Springer-Verlag, 1993.
2. *Butcher J.C.* Numerical Methods for Ordinary Differential Equations. — John Wiley & Sons, 2016.