Differential Equation Solver

Group Quiz : 01

EE1060: Differential Equations and Transform Techniques

Ankit Jainar	**EE24BTECH11004**
Arnav Yadnopavit	**EE24BTECH11007**
Dulla Karthik	**EE24BTECH11017**
Janjanam Kedarananda	**EE24BTECH11030**
Harsha Vardhan Reddy	**EE24BTECH11063**
Harshil Rathan Y	**EE24BTECH11064**

Contents

1 Introduction

2 Numerical Methods

2.1 Forward Euler

2.1.1 Theory

The Forward Euler method is an explicit numerical method for solving ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0 \tag{0.1}$$

- This method approximates the solution to the initial value problem (IVP) by discretizing the solution over small time steps and using the tangent slopes at the current step to update the solution.

Given the differential equation

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

and step size h, the update equation is

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{0.2}$$

Given y_0 at x_0, the goal is to approximate the solution at the next step

$$x_1 = x_0 + h \tag{0.3}$$

The key idea is to use the slope of the solution at the known point, (x_n, y_n), to update our approximations.

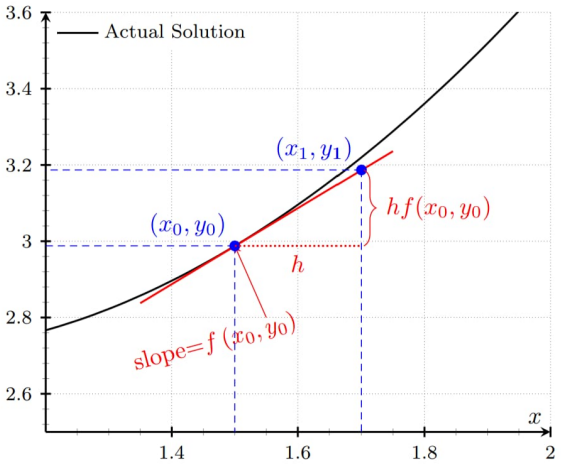


Fig. 0.1: Illustration of the Forward Euler method

Given y_0 at x_0, the goal is to approximate the solution at the next step

$$x_1 = x_0 + h \tag{0.4}$$

The key idea is to use the slope of the solution at the known point, (x_n, y_n), to update our approximations.

2.1.2 Error Analysis

- The local error in the Forward Euler method refers to the error made in a single step and can be derived by considering the Taylor series expansion of the exact solution.

Suppose the exact solution $y = g(x)$. Using the Taylor series expansion,

$$g(x_0 + h) = g(x_0) + h\frac{dg}{dx}\Big|_{x_0} + \frac{h^2}{2}\frac{d^2g}{dx^2}\Big|_{x_0} + O(h^3) \tag{0.5}$$

Substituting $\frac{dg}{dx} = f(x, y)$,

$$g(x_0 + h) = y_0 + hf(x_0, y_0) + \frac{h^2}{2}\frac{d^2g}{dx^2}\Big|_{x_0} + O(h^3) \tag{0.6}$$

The local truncation error is given by

$$\varepsilon = \frac{h^2}{2}\frac{d^2g}{dx^2}\Big|_{x_0} + O(h^3) \tag{0.7}$$

which indicates that the local error is of order $O(h^2)$.

The global truncation error in the Forward Euler method arises from error accumulation over multiple steps. It is given by

$$|y(b) - y_n| \leq Kh \tag{0.8}$$

where K is a constant, indicating that the global error is of order $O(h)$.

2.1.3 Python Code Implementation

Below is the Python code implementing the Forward Euler method for solving an ODE of the form $\frac{dy}{dx} = f(x, y)$.

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

def forward_euler(f, x0, y0, x_end, h):
    x_values = np.arange(x0, x_end + h, h)
    y_values = np.zeros(len(x_values))
    y_values[0] = y0

    for i in range(1, len(x_values)):
        y_values[i] = y_values[i - 1] + h * f(x_values[i - 1], y_values[i -
            1])

    return x_values, y_values

def f(x, y):
    return -2 * y + x # Example ODE

x0, y0 = 0, 1 # Initial conditions
```

```
x_end, h = 2, 0.1 # End point and step size
x_vals, y_vals = forward_euler(f, x0, y0, x_end, h)

plt.plot(x_vals, y_vals, label='Forward Euler')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

2.2 Backward Euler

2.2.1 Theory

The Backward Euler method is an implicit numerical method for solving ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y) \quad y(x_0) = y_0 \tag{0.9}$$

- This method approximates the solution to the IVP by discretizing the solution over small time steps and using the tangent slopes at the next step, rather than the current one, which contrasts with methods such as Forward Euler which use known values at each step to update the solution.

Given the DE

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

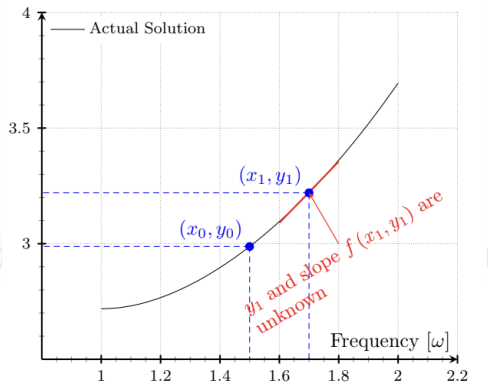
and step size h, the update equation is

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}) \tag{0.10}$$

Given y_0 at x_0 the goal is to approximate the solution at the next step

$$x_1 = x_0 + h \tag{0.11}$$

the key idea is to use the slope of the solution at the new point, (x_1, y_1) to update our approximations



- The local error in a numerical method refers to the error made in a single step of the method. The local error at each step can be derived by considering the Taylor series expansion of the exact solution.

Suppose the exact solution $y = g(x)$. Using the taylor series expansion

$$g(x_0 + h) = g(x_0) + h\frac{dg}{dx}|_{x_0} + \frac{h^2}{2}\frac{d^2g}{dx^2}|_{x_0} + O(h^3) \tag{0.12}$$

Substituting $\frac{dg}{dx} = f(x, y)$

$$g(x_0 + h) = y_0 + hf(x_0, y_0) + \frac{h^2}{2}\frac{d^2g}{dx^2}\Big|_{x_0} + O(h^3) \tag{0.13}$$

Re writing the function $f(x_1, y_1)$ using taylor series at x_0

$$f(x_1, y_1) = f(x_0, y_0) + h\frac{\partial f}{\partial x}\Big|_{(x_0,y_0)} + h\frac{\partial f}{\partial y}f(x_0, y_0) + O(h^2) \tag{0.14}$$

$$y_1 = y_0 + h\left[f(x_0, y_0) + h\frac{\partial f}{\partial x} + h\frac{\partial f}{\partial y}f(x_0, y_0)\right] + O(h^3) \tag{0.15}$$

$$y_1 = y_0 + hf(x_0, y_0) + h^2\frac{\partial f}{\partial x} + h^2\frac{\partial f}{\partial y}f(x_0, y_0) + O(h^3) \tag{0.16}$$

The local error is the difference between the exact solution and the numerical approximation, simplifying

$$\varepsilon = -\frac{h^2}{2}\left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}f\right) + O(h^3) \tag{0.17}$$

The local error in backward euler method is of order $O(h^2)$, backward euler is more stable than the forward euler

2.2.2 Python Code Implementation

Below Python code implements the Backward Euler method to solve an ordinary differential equation (ODE) of the form $\frac{dy}{dt} = f(t, y)$.

```
import numpy as np
from scipy.optimize import fsolve

def solve_backward_euler(f, t0, t_end, y0, h):
    # f is the differential equation (dy/dt = f(y, t))
    # t0 is the start time
    # t_end is the end time
    # h is the step size
    # y0 is the initial condition

    if h <= 0:
        raise ValueError("Step size must be positive.")

    N = int((t_end - t0) / h) # Number of time steps

    # Array of time values
```

```
    t_values = np.linspace(t0, t_end, N + 1)

    y_values = np.zeros_like(t_values)
    y_values[0] = y0

    for i in range(N):
        y_n = y_values[i]
        t_n = t_values[i]

        # Define equation to solve for y_{n+1}
        def equation(y_next):
            return y_next - y_n - h * f(t_n + h, y_next)

        y_next = fsolve(equation, y_n)[0]
        y_values[i + 1] = y_next

    return t_values, y_values
```

2.3 Runge-Kutta 2

2.3.1 Theory

The RK-2 method is another numerical technique for solving first-order ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y) \quad y(x_0) = y_0 \tag{0.18}$$

It is an improvement over the Euler method, providing better accuracy while maintaining a relatively simple computation process.

- It approximates the solution by discretizing the problem over small time steps and using tangent slopes at two distinct points: one at the current step, and the other at a point obtained by advancing a certain distance along the slope at the current step

Compute the first slope

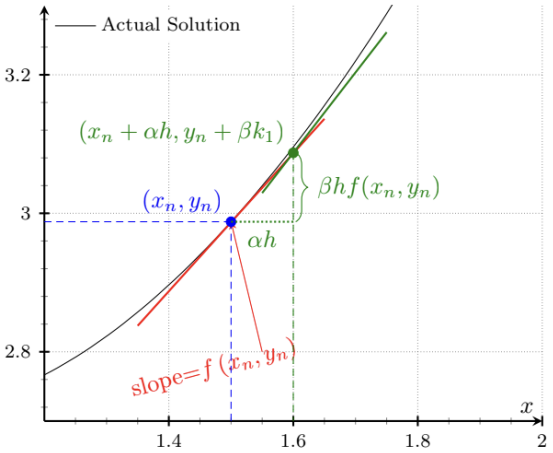
$$k_1 = f(x_n, y_n) \tag{0.19}$$

this is the slope at the beginning of the equation Compute the second slope

$$k_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{0.20}$$

The general update equation for the RK2 method is

$$y_{n+1} = y_n + ak_1 + bk_2 \tag{0.21}$$



By applying the taylor series approximation to a function $f(x_n + \alpha h, y_n + \beta k_1)$

$$f(x_n + \alpha h, y_n + \beta k_1) = f(x_n, y_n) + \alpha h \frac{\partial f(x_n, y_n)}{\partial x} + \beta h \frac{\partial f(x_n, y_n)}{\partial y} \tag{0.22}$$

Substituting and rearranging

$$y_{n+1} = y_n + (a + b)hf(x_n, y_n) + b\alpha h^2 \frac{\partial f(x_n, y_n)}{\partial x} + b\beta h^2 f(x_n, y_n) \frac{\partial f(x_n, y_n)}{\partial y} \tag{0.23}$$

The constants a, b, α, β are chosen such that the local error becomes $O(h^3)$. The resulting equation must match the first three taylor series expansion of $g(x)$

$$g(x_n + h) = g(x_n) + hg'(x_n) + \frac{h^2}{2} g''(x_n) + O(h^3) \tag{0.24}$$

under the assumption $y_n = g(x_n)$

$$g(x_n + h) = y_n + hf(x_n, y_n) + \frac{h^2}{2} f'(x_n, y_n) + O(h^3) \tag{0.25}$$

rearranging we get

$$g(x_n + h) = y_n + hf(x_n, y_n) + \frac{h^2}{2} \frac{\partial f(x_n, y_n)}{\partial x} + \frac{h^2}{2} f(x_n, y_n) \frac{\partial f(x_n, y_n)}{\partial y} + O(h^3) \tag{0.26}$$

The conditions for the constants are

$$a = b = \frac{1}{2}, \quad \alpha = \beta = 1 \tag{0.27}$$

Thus, we get the update equation to be

$$y_{n+1} = y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2 \quad k_1 = hf(x_n, y_n), \quad k_2 = hf(x_n + h, y_n + k_1) \tag{0.28}$$

2.3.2 Python Code Implementation

Below Python code implements the Runge-Kutta method to solve an ordinary differential equation (ODE) of the form $\frac{dy}{dt} = f(t, y)$.

```
import numpy as np

def solve_rk2(f, t0, t_end, y0, h):
    # This method solves an ODE using the second-order Runge-Kutta method.
    # f is the differential equation function.
    # t0 is the initial time.
    # t_end is the final time.
    # y0 is the initial condition at t0.
    # h is the step size.

    if h <= 0:
        raise ValueError("Step size must be positive.")

    # Number of time steps
    N = int((t_end - t0) / h)

    # Generate time values
    t_values = np.linspace(t0, t_end, N + 1)

    # Initialize solution array
    y_values = np.zeros_like(t_values)
    y_values[0] = y0

    for i in range(N):
        t_n = t_values[i]
        y_n = y_values[i]

        # Compute the two slopes for RK2
        k1 = f(t_n, y_n)
        k2 = f(t_n + h / 2, y_n + (h / 2) * k1)

        # Update equation to compute next value
        y_values[i + 1] = y_n + h * k2

    return t_values, y_values
```

2.4 Runge-Kutta 4

2.4.1 Theory

The Runge-Kutta 4th order (RK4) method is a numerical technique for solving first-order ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0 \tag{0.29}$$

It is a significant improvement over the Euler and RK2 methods, providing higher accuracy without drastically increasing computational complexity.

- The RK4 method approximates the solution by computing four different slope estimates at distinct points and combining them in a weighted manner.

- These slopes provide information about how the solution evolves, allowing for a more accurate update than simpler methods like Euler or RK2.

The four slope estimates are computed as follows:

$$k_1 = hf(x_n, y_n) \quad \text{(Slope at the beginning of the step)} \tag{0.30}$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \quad \text{(Slope at the midpoint using } k_1\text{)} \tag{0.31}$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \quad \text{(Slope at the midpoint using } k_2\text{)} \tag{0.32}$$

$$k_4 = hf(x_n + h, y_n + k_3) \quad \text{(Slope at the next step using } k_3\text{)} \tag{0.33}$$

The update equation for the RK4 method is:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{0.34}$$

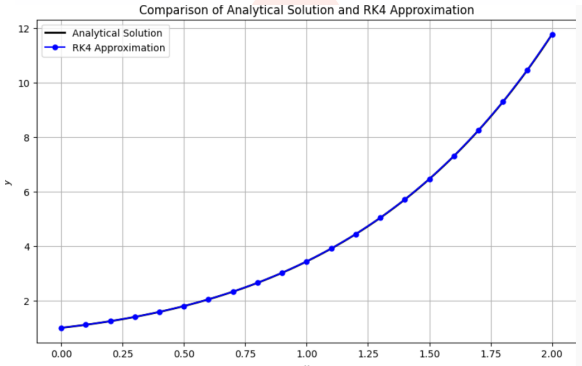


Fig. 0.2: Illustration of the RK4 method with intermediate slopes

2.4.2 Error Analysis

- The local truncation error in the RK4 method is of order $O(h^5)$, meaning each individual step introduces an error proportional to h^5.
- The global error (after multiple steps) is of order $O(h^4)$, which makes RK4 significantly more accurate than Forward Euler $O(h)$ and RK2 $O(h^2)$.

Expanding the exact solution $g(x)$ using the Taylor series,

$$g(x_n + h) = g(x_n) + hg'(x_n) + \frac{h^2}{2}g''(x_n) + \frac{h^3}{6}g'''(x_n) + \frac{h^4}{24}g''''(x_n) + O(h^5) \tag{0.35}$$

By carefully choosing the coefficients in RK4, the error terms up to $O(h^4)$ match, leaving the local truncation error proportional to $O(h^5)$.

2.4.3 Python Code Implementation

Below is a Python implementation of the RK4 method for solving an ODE of the form $\frac{dy}{dx} = f(x, y)$:

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

def runge_kutta_4(f, x0, y0, x_end, h):
    x_values = np.arange(x0, x_end + h, h)
    y_values = np.zeros(len(x_values))
    y_values[0] = y0

    for i in range(1, len(x_values)):
        x_n, y_n = x_values[i - 1], y_values[i - 1]

        k1 = h * f(x_n, y_n)
        k2 = h * f(x_n + h / 2, y_n + k1 / 2)
        k3 = h * f(x_n + h / 2, y_n + k2 / 2)
        k4 = h * f(x_n + h, y_n + k3)

        y_values[i] = y_n + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    return x_values, y_values

# Example ODE: dy/dx = -2y + x
def f(x, y):
    return -2 * y + x

x0, y0 = 0, 1
x_end, h = 2, 0.1
x_vals, y_vals = runge_kutta_4(f, x0, y0, x_end, h)

plt.plot(x_vals, y_vals, label="RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

2.5 Trapezoidal

2.5.1 Theory

The trapezoidal method, also known as the implicit Euler method or the Crank-Nicholson method, is a numerical technique for solving first-order ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0 \tag{0.36}$$

It provides an improvement over the simple Euler method by using both the current and predicted next-step values to estimate the solution, making it more stable and accurate.

- The trapezoidal method is an implicit scheme, meaning it requires solving an equation at each step.
- It computes the average of the slopes at the beginning and end of the interval for a more accurate approximation.

The formula for the trapezoidal method is given by:

$$y_{n+1} = y_n + \frac{h}{2}\left(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\right) \tag{0.37}$$

This requires solving for y_{n+1} implicitly, making it more computationally intensive than explicit methods like Euler's.

2.5.2 Application to RL Circuit

Consider an RL circuit with a time-dependent voltage input $V(t)$. The governing differential equation for the current $i(t)$ is:

$$L\frac{di}{dt} + Ri = V(t) \tag{0.38}$$

Applying the trapezoidal rule to this equation, we approximate $i(t)$ as follows:

$$i_{n+1} = i_n + \frac{h}{2}\left(\frac{V_n - Ri_n}{L} + \frac{V_{n+1} - Ri^*}{L}\right) \tag{0.39}$$

where i^* is a predictor obtained using the explicit Euler method:

$$i^* = i_n + h\frac{V_n - Ri_n}{L} \tag{0.40}$$

2.5.3 Python Code Implementation

Below is a Python implementation of the trapezoidal method for solving the RL circuit equation with a square wave voltage input:

```
import numpy as np

def square_wave(t, V0, T):
    """Generate a square wave with period T and amplitude V0."""
    return V0 * ((t % T) < (T / 2))

def trapezoidal(R, L, V0, T, h, t_max):
    t = np.arange(0, t_max + h, h)
    V = square_wave(t, V0, T)
    i = np.zeros_like(t)

    for n in range(len(t)-1):
        i_star = i[n] + h * (V[n] - R * i[n]) / L # Euler step
        i[n+1] = i[n] + (h / 2) * ((V[n] - R * i[n]) / L + (V[n+1] - R *
            i_star) / L)

    return t, i
```

2.5.4 Simulation and Plotting

To visualize the current response in the RL circuit, we use the following code:

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

def square_wave(t, T, V0, alpha):
    return V0 if (t % T) < alpha * T else 0
```

```
R = 100 # Resistance in ohms
L = 10 # Inductance in henries
V0 = 10 # Square wave amplitude
T = 5 # Period of the square wave
alpha = 0.5 # Duty ratio

t0 = 0 # Start time
t_end = 10 # End time
h = 0.01 # Step size

t_values_trap, i_values_trap = trapezoidal(R, L, V0, T, h, t_end)

plt.figure(figsize=(10, 6))
plt.plot(t_values_trap, i_values_trap, label='Trapezoidal: Current through
    inductor')
plt.title('Current Response in RL Circuit with Square Wave Input')
plt.xlabel('Time (s)')
plt.ylabel('Current (A)')
plt.legend()
plt.grid(True)
plt.show()
```

The Trapezoidal method is the implicit second-order method:

$$i_{n+1} = i_n + \frac{h}{2}\left(\left.\frac{di}{dt}\right|_{t_n,i_n} + \left.\frac{di}{dt}\right|_{t_{n+1},i_{n+1}}\right) \tag{0.41}$$

$$= i_n + \frac{h}{2}\left(\frac{-R}{L}i_n + \frac{-R}{L}i_{n+1}\right) \tag{0.42}$$

Rearranging:

$$i_{n+1} - \frac{h}{2}\frac{R}{L}i_{n+1} = i_n\left(1 - \frac{h}{2}\frac{R}{L}\right) \tag{0.43}$$

$$i_{n+1} = \frac{i_n(1 - \frac{h}{2}\frac{R}{L})}{1 + \frac{h}{2}\frac{R}{L}} \tag{0.44}$$

For stability:

$$\left|\frac{1 - \frac{h}{2}\frac{R}{L}}{1 + \frac{h}{2}\frac{R}{L}}\right| < 1 \tag{0.45}$$

which ensures unconditional stability for all $h > 0$.

2.5.5 Conclusion

- Forward Euler: Conditionally stable ($h < 2L/R$).
- Backward Euler: Unconditionally stable.
- RK2: Improved accuracy but still constrained by stability limits.
- RK4: High accuracy and better stability.
- Trapezoidal: Unconditionally stable.

3 Stability Analysis

3.1 Forward Euler

Equation

$$V(t) = L\frac{di}{dt} + Ri \tag{0.46}$$

for Forward Euler method

$$i_{n+1} = i_n + h\frac{1}{L}(V_n - Ri_n) \tag{0.47}$$

Let $V(t) = 0$ i.e Homogeneous

$$\frac{di}{dt} = \frac{-R}{L}i \tag{0.48}$$

after solving we get

$$i = i_0 e^{\frac{Rt}{L}} \tag{0.49}$$

it decays exponentially (Stable)
Now for Forward Euler

$$i_{n+1} = i_n + h\frac{di}{dt} \tag{0.50}$$

$$= i_n + h\frac{-Ri_n}{L} \tag{0.51}$$

$$= i_n(1 + h(\frac{-Rn}{L})) \tag{0.52}$$

$$= i_n(1 - h\frac{R}{L}) \tag{0.53}$$

for stable
$|1 - h\frac{R}{L}| < 1$
$-1 < 1 - h\frac{R}{L} < 1$
$h < 2\frac{L}{R}$
if $h > 2\frac{L}{R}$ (it will produce oscillations and diverges)

3.2 Backward Euler

Equation

$$V(t) = L\frac{di}{dt} + Ri \tag{0.54}$$

For the Backward Euler method, we use

$$i_{n+1} = i_n + h\frac{1}{L}(V_{n+1} - Ri_{n+1}) \tag{0.55}$$

Let $V(t) = 0$ (Homogeneous case), then

$$\frac{di}{dt} = \frac{-R}{L} i \tag{0.56}$$

Solving this analytically, we get

$$i = i_0 e^{-\frac{R}{L}t} \tag{0.57}$$

which decays exponentially (Stable).

Now for Backward Euler:

$$i_{n+1} = i_n + h \frac{di}{dt}\Big|_{t_{n+1}} \tag{0.58}$$

$$= i_n + h \frac{-R i_{n+1}}{L} \tag{0.59}$$

Rearranging,

$$i_{n+1} - h\frac{R}{L} i_{n+1} = i_n \tag{0.60}$$

$$i_{n+1}(1 + h\frac{R}{L}) = i_n \tag{0.61}$$

$$i_{n+1} = \frac{i_n}{1 + h\frac{R}{L}} \tag{0.62}$$

For Stability: The magnitude of the amplification factor should be less than 1:

$$\left| \frac{1}{1 + h\frac{R}{L}} \right| < 1 \tag{0.63}$$

Since $1 + h\frac{R}{L} > 1$ for $h > 0$, the denominator is always greater than the numerator, ensuring

$$|i_{n+1}| < |i_n| \tag{0.64}$$

which guarantees stability for any step size $h > 0$.

3.3 RK2

Equation

$$\frac{di}{dt} = \frac{-R}{L} i \tag{0.65}$$

RK2 (Heun's Method) is given by:

$$k_1 = \frac{di}{dt}\Big|_{t_n, i_n} = \frac{-R}{L} i_n \tag{0.66}$$

$$k_2 = \frac{di}{dt}\Big|_{t_n + h, i_n + h k_1} = \frac{-R}{L}(i_n + h k_1) \tag{0.67}$$

$$i_{n+1} = i_n + \frac{h}{2}(k_1 + k_2) \tag{0.68}$$

Substituting values:

$$i_{n+1} = i_n + \frac{h}{2}\left(\frac{-R}{L}i_n + \frac{-R}{L}(i_n + h\frac{-R}{L}i_n)\right) \tag{0.69}$$

$$= i_n\left(1 - h\frac{R}{L} + \frac{h^2}{2}\left(\frac{R}{L}\right)^2\right) \tag{0.70}$$

For stability, we require:

$$\left|1 - h\frac{R}{L} + \frac{h^2}{2}\left(\frac{R}{L}\right)^2\right| < 1 \tag{0.71}$$

which provides a more relaxed stability criterion compared to Forward Euler.

3.4 RK4

Equation

$$\frac{di}{dt} = \frac{-R}{L}i \tag{0.72}$$

RK4 is given by:

$$k_1 = \frac{di}{dt}\Big|_{t_n, i_n} = \frac{-R}{L}i_n \tag{0.73}$$

$$k_2 = \frac{di}{dt}\Big|_{t_n + \frac{h}{2}, i_n + \frac{h}{2}k_1} = \frac{-R}{L}(i_n + \frac{h}{2}k_1) \tag{0.74}$$

$$k_3 = \frac{di}{dt}\Big|_{t_n + \frac{h}{2}, i_n + \frac{h}{2}k_2} = \frac{-R}{L}(i_n + \frac{h}{2}k_2) \tag{0.75}$$

$$k_4 = \frac{di}{dt}\Big|_{t_n + h, i_n + hk_3} = \frac{-R}{L}(i_n + hk_3) \tag{0.76}$$

$$i_{n+1} = i_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{0.77}$$

This method has a much higher accuracy and better stability properties compared to Forward Euler and RK2.

3.5 Trapezoidal

The Trapezoidal method is the implicit second-order method:

$$i_{n+1} = i_n + \frac{h}{2}\left(\frac{di}{dt}\Big|_{t_n, i_n} + \frac{di}{dt}\Big|_{t_{n+1}, i_{n+1}}\right) \tag{0.78}$$

$$= i_n + \frac{h}{2}\left(\frac{-R}{L}i_n + \frac{-R}{L}i_{n+1}\right) \tag{0.79}$$

Rearranging:

$$i_{n+1} - \frac{h}{2}\frac{R}{L} i_{n+1} = i_n \left(1 - \frac{h}{2}\frac{R}{L}\right) \tag{0.80}$$

$$i_{n+1} = \frac{i_n(1 - \frac{h}{2}\frac{R}{L})}{1 + \frac{h}{2}\frac{R}{L}} \tag{0.81}$$

For stability:

$$\left|\frac{1 - \frac{h}{2}\frac{R}{L}}{1 + \frac{h}{2}\frac{R}{L}}\right| < 1 \tag{0.82}$$

which ensures unconditional stability for all $h > 0$.

3.6 Conclusion

- Forward Euler: Conditionally stable ($h < 2L/R$).
- Backward Euler: Unconditionally stable.
- RK2: Improved accuracy but still constrained by stability limits.
- RK4: High accuracy and better stability.
- Trapezoidal: Unconditionally stable.

4 RLα Analysis

$T = 0.001s, V = 10V, \alpha = 0.5$

Let's consider a cycle from $t = nT$ to $t = nT + \alpha T$. Current at $t = nT$ is i_0 Current varies as:

$$i = \frac{V}{R}(1 - e^{\frac{-tR}{L}}) + i_0 e^{\frac{-tR}{L}} \tag{0.83}$$

at $t = nT + \alpha T$ let current be i_1

$$i_1 = \frac{V}{R}(1 - e^{\frac{-\alpha TR}{L}}) + i_0 e^{\frac{-\alpha TR}{L}} \tag{0.84}$$

from $t = nT + \alpha T$ to $t = (n + 1)T$ current varies as:

$$i = i_1 e^{\frac{-tR}{L}} \tag{0.85}$$

at $t = (n + 1)T$ let the current be i_2

$$i_2 = i_1 e^{\frac{-TR(1-\alpha)}{L}} \tag{0.86}$$

$$i_2 = \left(\frac{V}{R}(1 - e^{\frac{-\alpha TR}{L}}) + i_0 e^{\frac{-\alpha TR}{L}}\right) e^{\frac{-TR(1-\alpha)}{L}} \tag{0.87}$$

As $\frac{R}{L}$ varies, we can see the variation in V_R i.e Voltage across the Resistor as following:

4.1 Case 1: $e^{\frac{-\alpha TR}{L}} << 1$

This happens when $e^{\frac{-\alpha TR}{L}} < 10^{-3}$

$$\therefore \frac{R}{L} > \frac{6.909}{\alpha T} \tag{0.88}$$

$$i_1 \simeq \frac{V}{R} \tag{0.89}$$

$$i_2 \simeq 0 \tag{0.90}$$

We see a square wave nature in current.

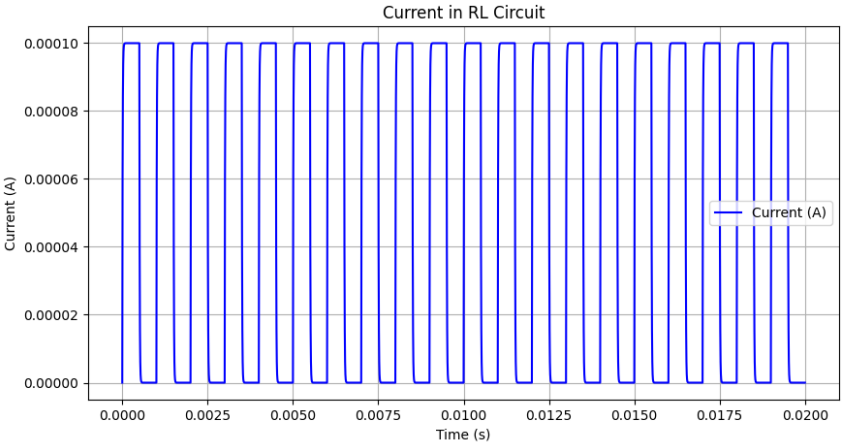


Fig. 0.3: $\frac{R}{L} = 10^5$

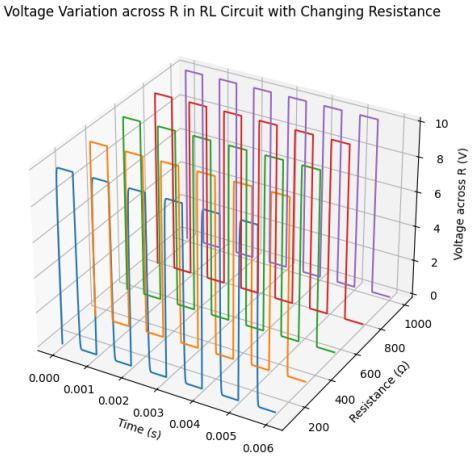


Fig. 0.4: $L = 10^{-3}$

4.2 Case 2: $\frac{R}{L}$ high

The current reaches V_0/R as max and 0 as min at steady state

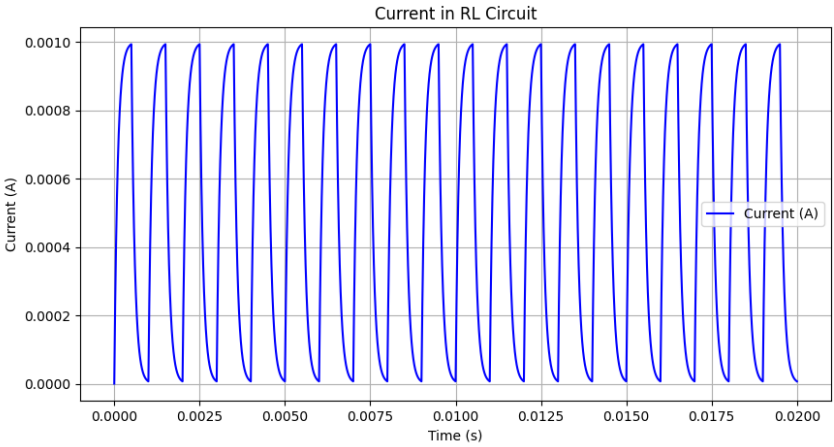


Fig. 0.5: $\frac{R}{L} = 10^4$

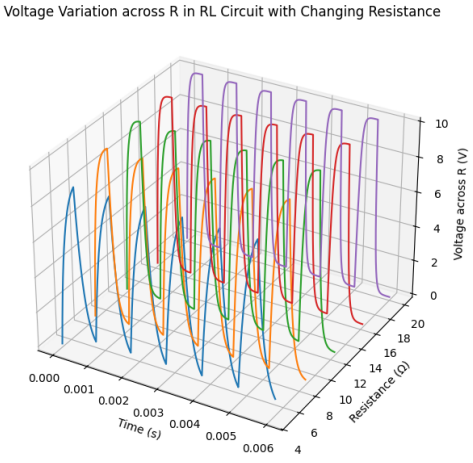


Fig. 0.6: $L = 10^{-3}$

4.3 Case 3: $\frac{R}{L}$ medium

The current doesn't reach V_0/R as max and 0 as min at steady state. But rather reaches a fixed crest and trough value.

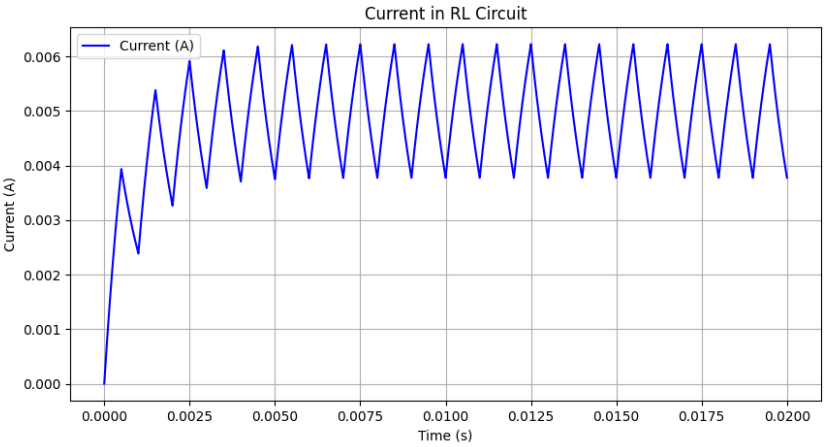


Fig. 0.7: $\frac{R}{L} = 10^3$

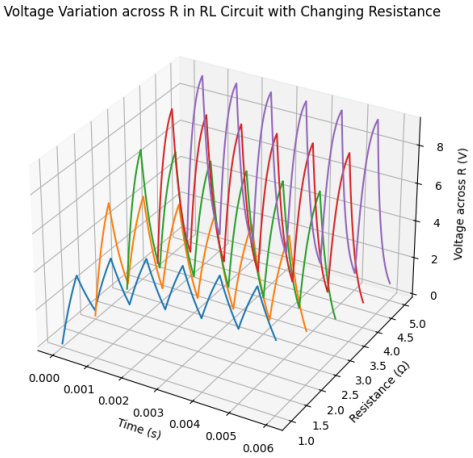


Fig. 0.8: $L = 10^{-3}$

4.4 Case 4: $e^{\frac{-\alpha TR}{L}} \simeq 1$

$$\frac{\alpha TR}{L} \simeq 0 \tag{0.91}$$

Current response tends to that of inductor-only circuit

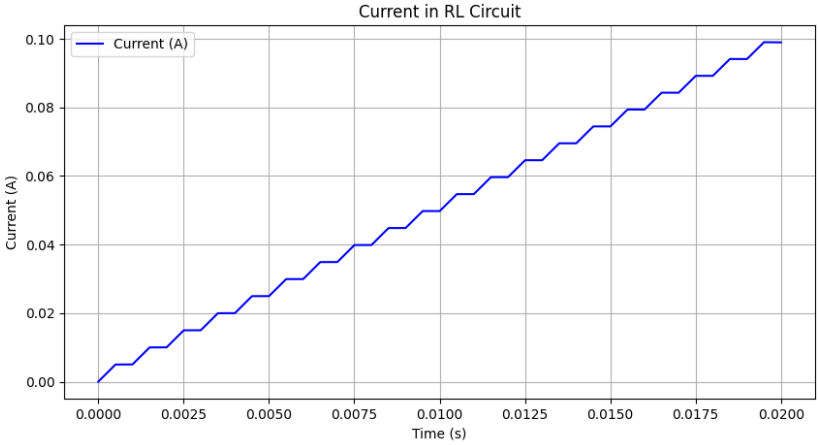


Fig. 0.9: $\frac{R}{L} = 1$

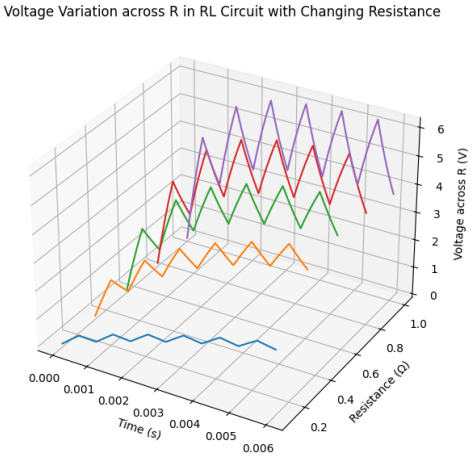


Fig. 0.10: $L = 10^{-3}$

4.5 α Analysis

As α tends to 0 maxima decreases and current goes closer to 0 A:

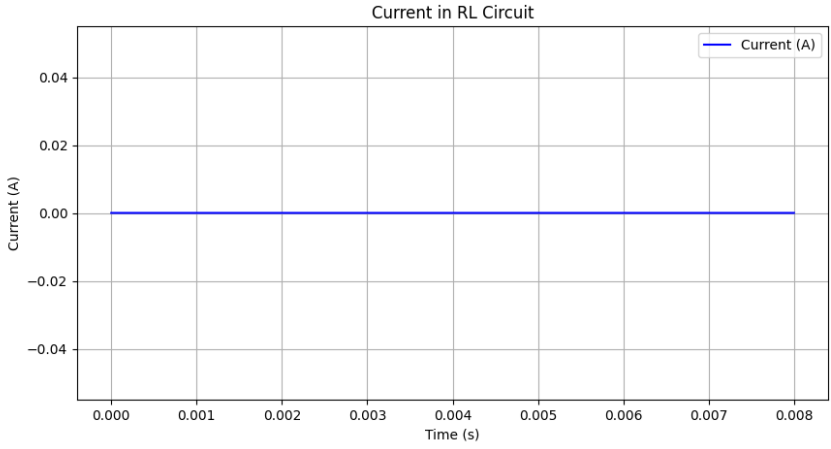


Fig. 0.11: $\alpha = 0$

As alpha grows closer to 1 the V_in tends more and more to a constant dc voltage and thus we observe current similar to that input

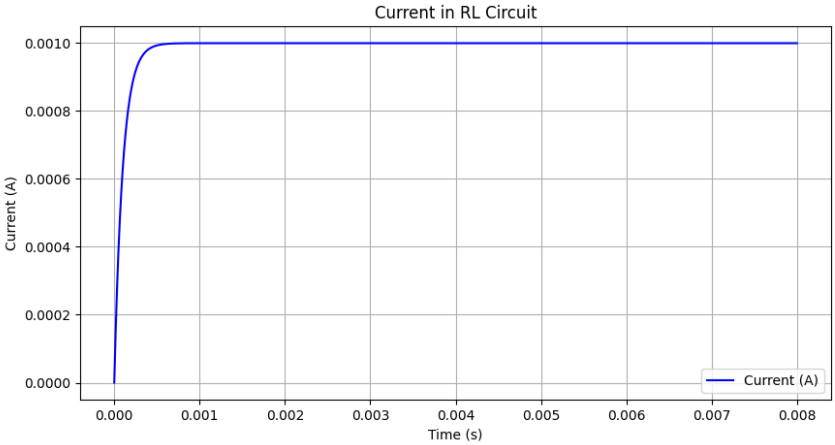


Fig. 0.12: $\alpha = 0$

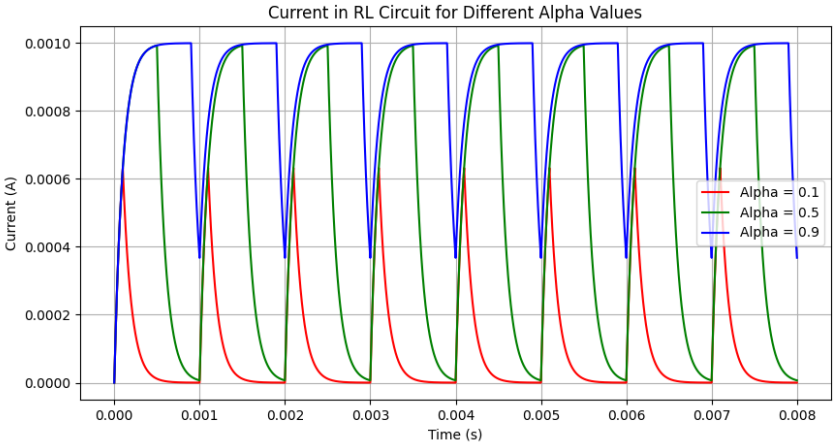


Fig. 0.13: $\alpha = 0$

5 Fourier Analysis

5.1 Problem Statement

The function `compute_current_waveform` calculates the current response of an RL circuit excited by a square wave voltage using Fourier series expansion. The given parameters are:

- Duty cycle of the square wave: α
- Resistance: R (Ohms)

- Inductance: L (Henries)
- Period of the square wave: T (seconds)
- Number of Fourier harmonics: N
- Number of time samples per period: M
- Number of cycles: C

The function returns an array containing the computed current waveform.

5.2 Compute Fundamental Angular Frequency

The fundamental angular frequency of the square wave is given by:

$$\omega_0 = \frac{2\pi}{T} \tag{0.92}$$

where T is the period of the square wave.

5.3 Compute DC Component of Voltage and Current

The DC component of the applied square wave voltage is:

$$V_{\text{DC}} = 10 \cdot \alpha \tag{0.93}$$

Since the inductor acts as a short circuit at DC, the DC component of the current is given by:

$$I_{\text{DC}} = \frac{V_{\text{DC}}}{R} = \frac{10\alpha}{R} \tag{0.94}$$

5.4 Fourier Series Representation of Voltage

The Fourier series representation of the square wave voltage is given by:

$$V_n = \frac{10}{j2\pi n}\left(1 - e^{-j2\pi n\alpha}\right) \tag{0.95}$$

where:

- n represents the harmonic index.
- j is the imaginary unit.

5.5 Compute Impedance for Each Harmonic

The impedance of the RL circuit at the n-th harmonic is:

$$Z_n = R + jn\omega_0 L \tag{0.96}$$

5.6 Compute Current for Each Harmonic

The current Fourier coefficient corresponding to each harmonic is given by:

$$I_n = \frac{V_n}{Z_n} \tag{0.97}$$

5.7 Sum Over Harmonics to Compute Total Current

The total current at time t is obtained by summing over the DC component and the harmonics:

$$I(t) = I_{\text{DC}} + 2\sum_{n=1}^{N} \text{Re}\left(I_n e^{jn\omega_0 t}\right) \tag{0.98}$$

where Re denotes the real part of the complex expression.

5.8 Compute Current for Each Time Sample

For each time sample t_i, the current is computed using:

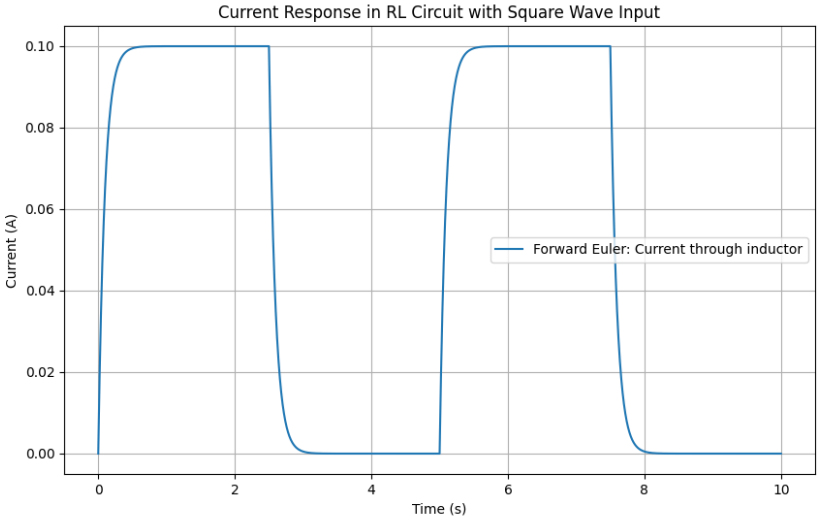
$$t_i = \frac{iT}{M}, \quad i = 0, 1, \ldots, M \cdot C \tag{0.99}$$

where $M \cdot C$ represents the total number of samples across multiple cycles.

6 Plot Analysis

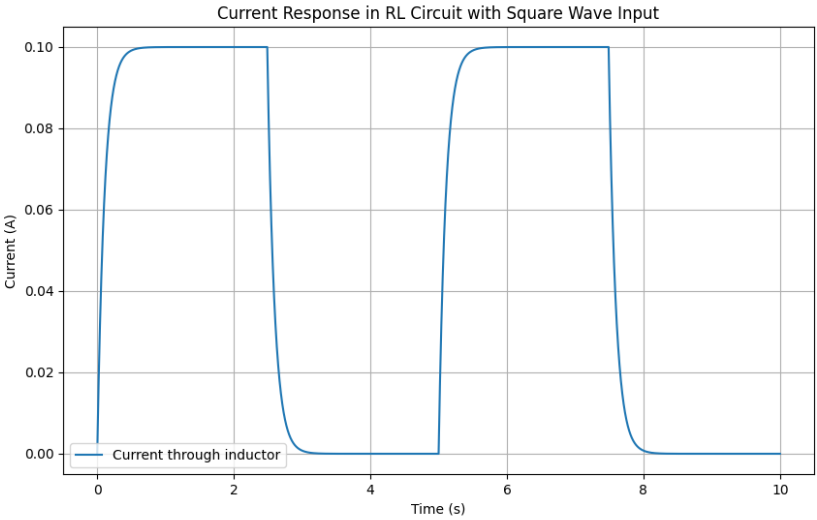
6.1 *Forward Euler :*

By using the python code mentioned in the **Numerical Methods(Forward Euler)** we get the following plot



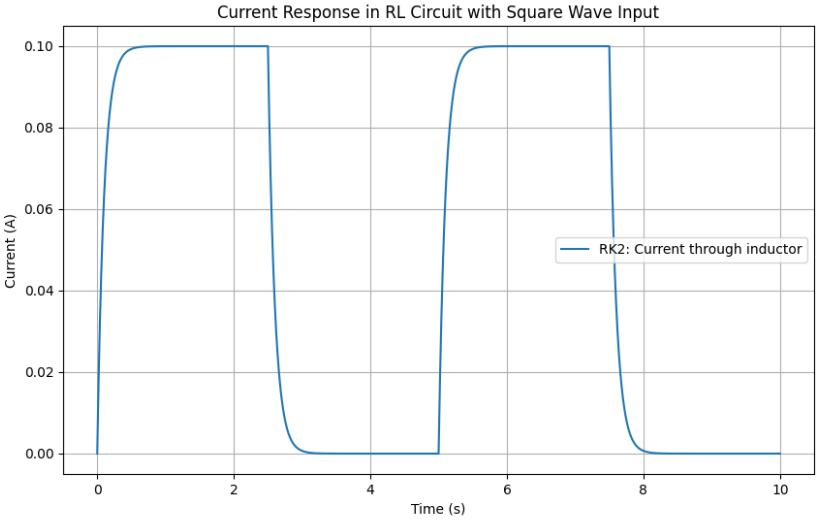
6.2 *Backward Euler :*

By using the python code mentioned in the **Numerical Methods(Backward Euler)** we get the following plot



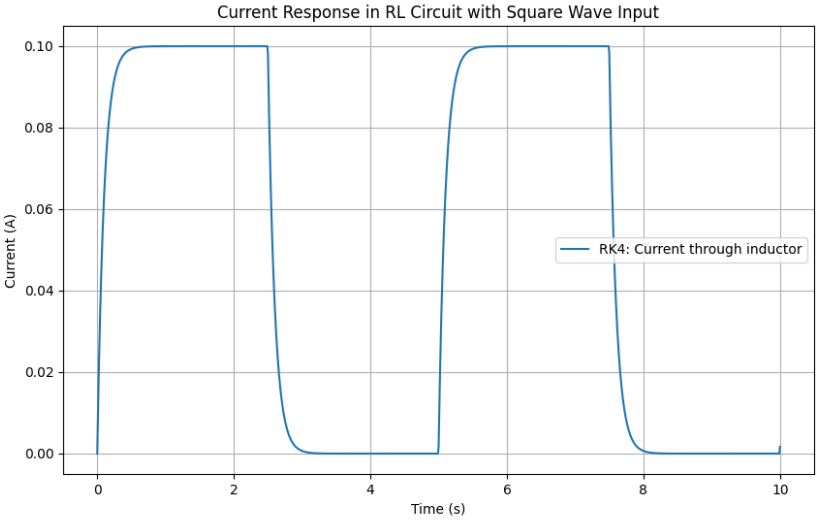
*6.3 **RK2 Euler :***

By using the python code mentioned in the **Numerical Methods(RK2)** we get the following plot



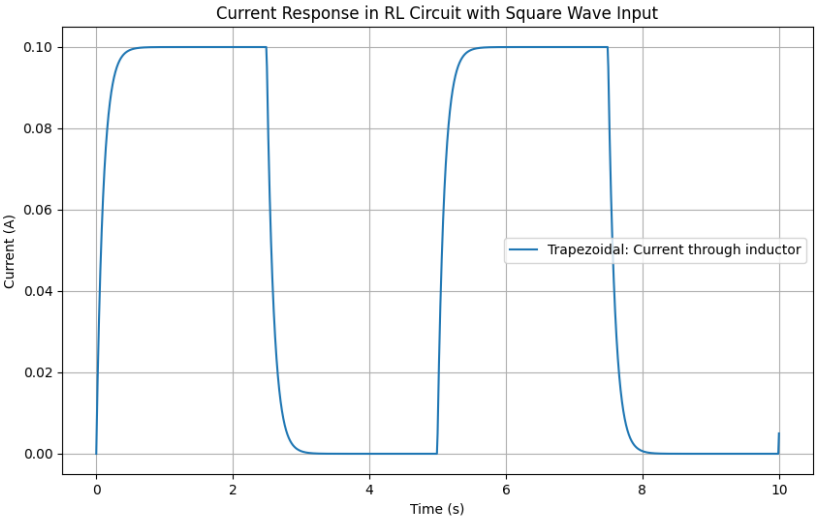
6.4 ***RK4 :***

By using the python code mentioned in the **Numerical Methods(RK4)** we get the following plot



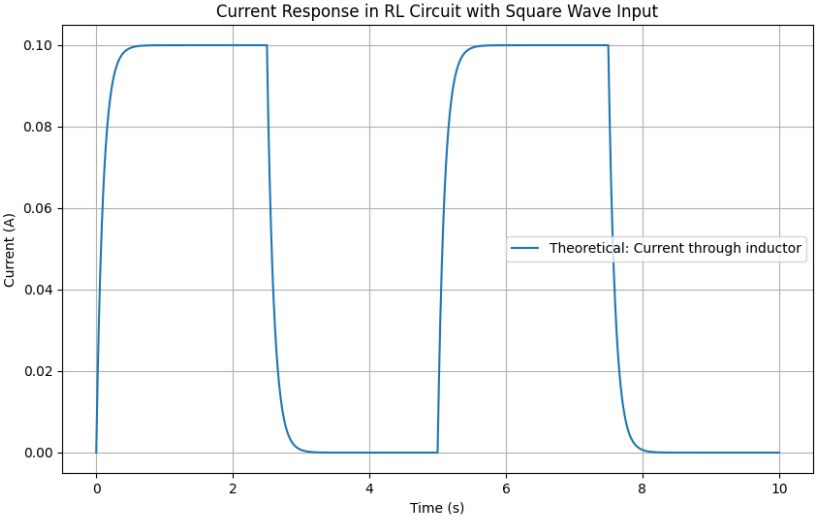
6.5 ***Trapezoidal :***

By using the python code mentioned in the **Numerical Methods(Trapezoidal)** we get the following plot



*6.6 **Theoretical :***

By using the python code mentioned in the **Numerical Methods(Theoretical Euler)** we get the following plot



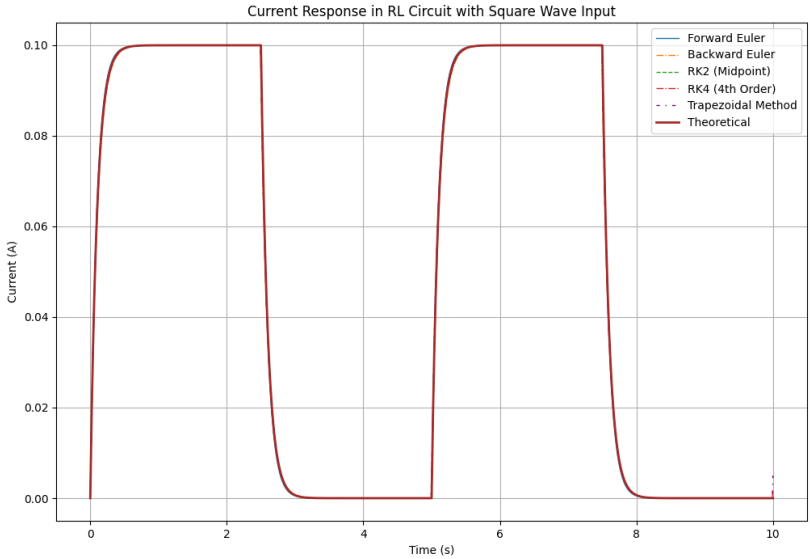
6.7 Python Code

Note : As the codes mentioned at the Numerical Methods were for generalised Differential Equations i used the below code to import above mentioned codes to plot :

```
import numpy as np
import matplotlib.pyplot as plt
from BackwardEuler import solve_backward_euler #(by changing this we can get
    for each plot)

def square_wave(t, T, V0, alpha):
    # Square wave generator with duty ratio alpha
    return V0 if (t % T) < alpha * T else 0

# Parameters
R = 100 # Resistance in ohms
L = 10 # Inductance in henries
V0 = 10 # Square wave amplitude
T = 5 # Period of the square wave
alpha = 0.5 # Duty ratio (0 < alpha <= 1)

# Define the differential equation for the RL circuit
def rl_circuit(t, i):
    v_in = square_wave(t, T, V0, alpha)
    return (v_in - R * i) / L

# Time parameters
t0 = 0 # Start time
t_end = 10 # End time
h = 0.01 # Step size
i0 = 0 # Initial current

# Solve the differential equation
t_values, i_values = solve_backward_euler(rl_circuit, t0, t_end, i0, h)

# Plot the current response
plt.figure(figsize=(10, 6))
plt.plot(t_values, i_values, label='Current through inductor')
plt.title('Current Response in RL Circuit with Square Wave Input')
plt.xlabel('Time (s)')
plt.ylabel('Current (A)')
plt.legend()
plt.grid(True)
plt.show()
```

6.8 *ANALYSIS OF ALL NUMERICAL METHODS wrt THEORETICAL :*

The below plot consists of all the simulation methods and theoretical method in a single plot.



6.8.1 *error analysis :*

For this we have implemented a Python program that systematically evaluates and compares the errors produced by different simulation methods against the theoretical method. The program does this by calculating errors at every 0.0001 seconds along the x-axis.

At each time step, it ranks the simulation methods in descending order based on their error values. Whenever there is a change in the method with the least error, the program detects this shift and prints the updated ranking, displaying the simulation method names along with their respective error values. This approach ensures a continuous and detailed analysis of how each method performs over time, highlighting the most accurate method at different instances.

6.8.2 *Python code :*

```
   import numpy as np
import matplotlib.pyplot as plt
from ForwardEuler import solve_forward_euler
from BackwardEuler import solve_backward_euler
from rk2 import solve_rk2
from rk4 import solve_rk4
from trapezoidal import trapezoidal
from theoretical import theoretical_i
```

```
# Square wave generator
def square_wave(t, T, V0, alpha):
    return V0 if (t % T) < alpha * T else 0

# Parameters
R = 100 # Resistance in ohms
L = 10 # Inductance in henries
V0 = 10 # Square wave amplitude
T = 5 # Period of the square wave
alpha = 0.5 # Duty ratio (0 < alpha <= 1)

# Time parameters
t0 = 0 # Start time
t_end = T # End time (one cycle)
h = 0.01 # Step size

# Differential equation for RL circuit
def rl_circuit(t, i):
    return (square_wave(t, T, V0, alpha) - R * i) / L

# Time array
t_values = np.arange(t0, t_end + h, h)

# Solve using different methods
_, i_forward = solve_forward_euler(rl_circuit, t0, t_end, 0, h)
_, i_backward = solve_backward_euler(rl_circuit, t0, t_end, 0, h)
_, i_rk2 = solve_rk2(rl_circuit, t0, t_end, 0, h)
_, i_rk4 = solve_rk4(rl_circuit, t0, t_end, 0, h)
_, i_trapezoidal = trapezoidal(R, L, V0, T, h, t_end)
i_theoretical = theoretical_i(R, L, T, V0, 0, alpha, t_values)

# Resample for error calculation
fine_t = np.arange(t0, t_end, 0.0001)
i_forward_interp = np.interp(fine_t, t_values, i_forward)
i_backward_interp = np.interp(fine_t, t_values, i_backward)
i_rk2_interp = np.interp(fine_t, t_values, i_rk2)
i_rk4_interp = np.interp(fine_t, t_values, i_rk4)
i_trapezoidal_interp = np.interp(fine_t, t_values, i_trapezoidal)
i_theoretical_interp = np.interp(fine_t, t_values, i_theoretical)

# Calculate errors
def calculate_errors():
    errors = {
        'Forward Euler': np.abs(i_forward_interp - i_theoretical_interp),
        'Backward Euler': np.abs(i_backward_interp - i_theoretical_interp),
        'RK2 (Midpoint)': np.abs(i_rk2_interp - i_theoretical_interp),
        'RK4 (4th Order)': np.abs(i_rk4_interp - i_theoretical_interp),
        'Trapezoidal': np.abs(i_trapezoidal_interp - i_theoretical_interp)
    }
    return errors

errors = calculate_errors()
```

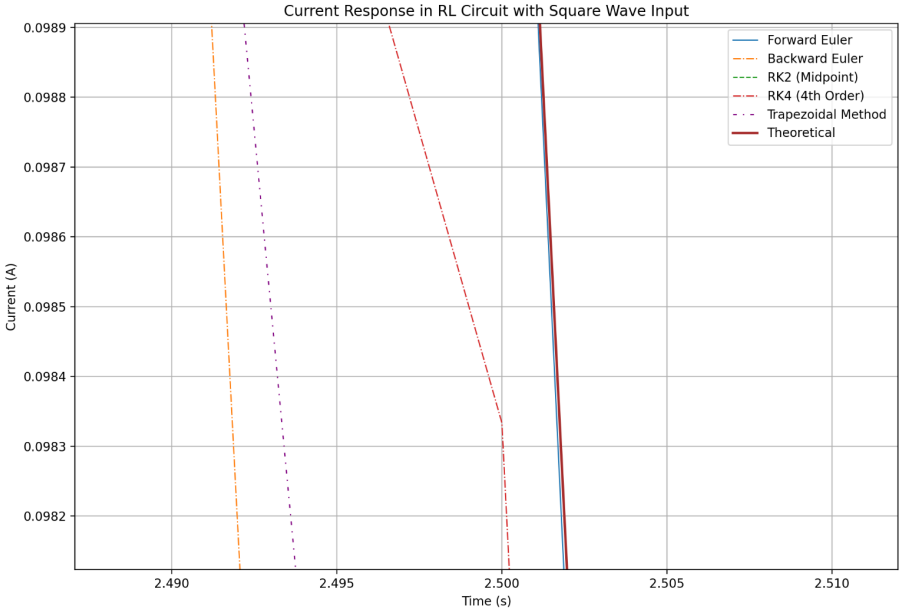
```
# Track least error method
prev_least_error = None
for i, t in enumerate(fine_t):
    current_order = sorted(errors.items(), key=lambda x: x[1][i], reverse=
        True)
    least_error_method = current_order[-1][0]

    if least_error_method != prev_least_error:
        print(f"At t = {t:.4f}s, error order (desc):")
        for method, error in current_order:
            print(f"{method}: {error[i]:.6f}")
        prev_least_error = least_error_method

# Plot all results
plt.figure(figsize=(12, 8))
plt.plot(t_values, i_forward, label='Forward Euler', linestyle='-', linewidth
    =1)
plt.plot(t_values, i_backward, label='Backward Euler', linestyle='dashdot',
    linewidth=1)
plt.plot(t_values, i_rk2, label='RK2 (Midpoint)', linestyle='--', linewidth
    =1)
plt.plot(t_values, i_rk4, label='RK4 (4th Order)', linestyle='-.', linewidth
    =1)
plt.plot(t_values, i_trapezoidal, label='Trapezoidal Method', linestyle=(0,
    (3, 5, 1, 5)), linewidth=1, color='purple')
plt.plot(t_values, i_theoretical, label='Theoretical', linestyle='-',
    linewidth=2, color='brown')

# Customize plot
plt.title('Current Response in RL Circuit with Square Wave Input')
plt.xlabel('Time (s)')
plt.ylabel('Current (A)')
plt.legend()
plt.grid(True)
plt.show()
```

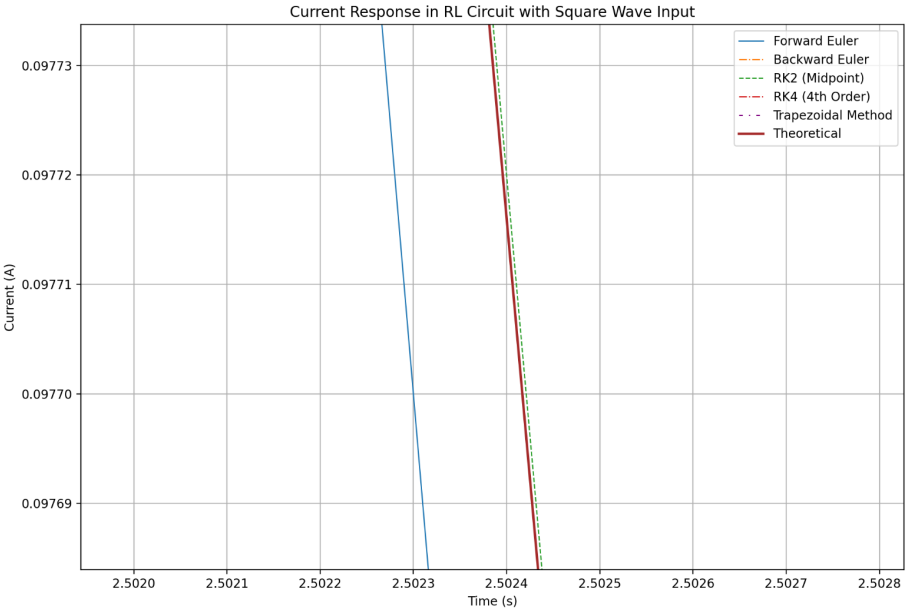

6.8.3 ***Data Obtained :***

At t = 2.4901

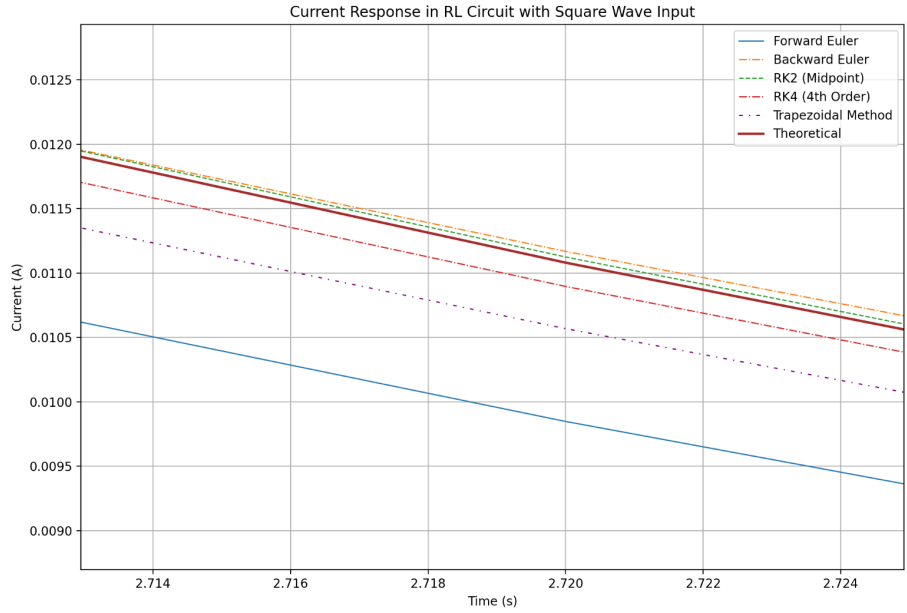
Time (s)	Highest Error		2nd Highest		3rd Highest		4th Highest		Least Error	
	Method	Error	Method	Error	Method	Error	Method	Error	Method	Error
0.0000	Forward Euler	0.000000	Backward Euler	0.000000	RK2 (Midpoint)	0.000000	RK4 (4th Order)	0.000000	Trapezoidal	0.000000
0.0001	Forward Euler	0.000005	Backward Euler	0.000004	RK2 (Midpoint)	0.000000	Trapezoidal	0.000000	RK4 (4th Order)	0.000000
2.4901	Backward Euler	0.000091	Trapezoidal	0.000050	RK4 (4th Order)	0.000017	Forward Euler	0.000000	RK2 (Midpoint)	0.000000
2.6960	Forward Euler	0.001405	Trapezoidal	0.000658	RK4 (4th Order)	0.000235	RK2 (Midpoint)	0.000050	Backward Euler	0.000049
2.7115	Forward Euler	0.001293	Trapezoidal	0.000560	RK4 (4th Order)	0.000201	Backward Euler	0.000046	RK2 (Midpoint)	0.000046
3.4196	Backward Euler	0.000004	Forward Euler	0.000004	Trapezoidal	0.000000	RK2 (Midpoint)	0.000000	RK4 (4th Order)	0.000000
4.4205	Backward Euler	0.000000	Forward Euler	0.000000	RK2 (Midpoint)	0.000000	RK4 (4th Order)	0.000000	Trapezoidal	0.000000
4.9901	Trapezoidal	0.000050	RK4 (4th Order)	0.000017	Backward Euler	0.000000	Forward Euler	0.000000	RK2 (Midpoint)	0.000000
4.9937	Trapezoidal	0.001850	RK4 (4th Order)	0.000617	Backward Euler	0.000000	RK2 (Midpoint)	0.000000	Forward Euler	0.000000

TABLE 0: Error order at different time steps

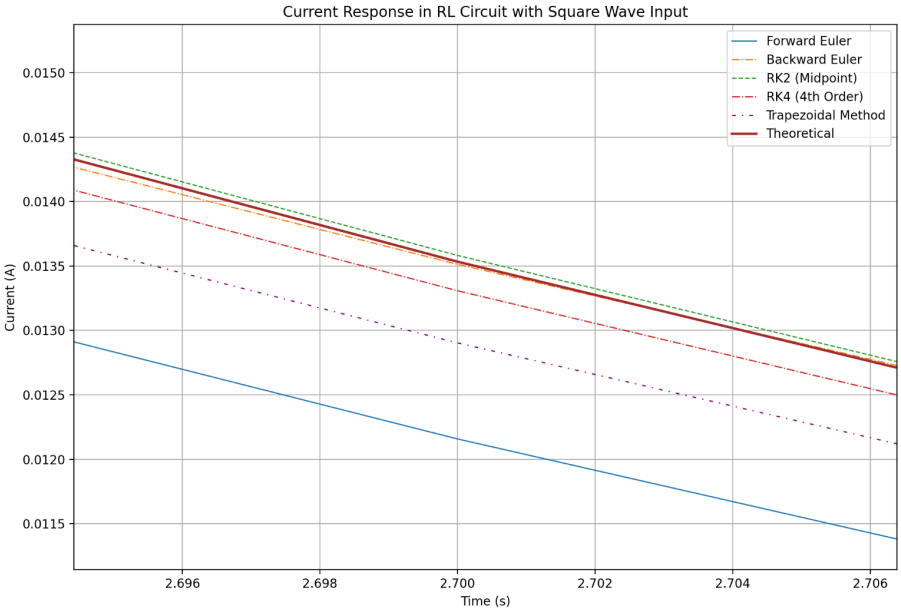



Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Current (A)
0.09773
0.09772
0.09771
0.09770
0.09769
2.5020
2.5021
2.5022
2.5023
2.5024
2.5025
2.5026
2.5027
2.5028
Time (s)

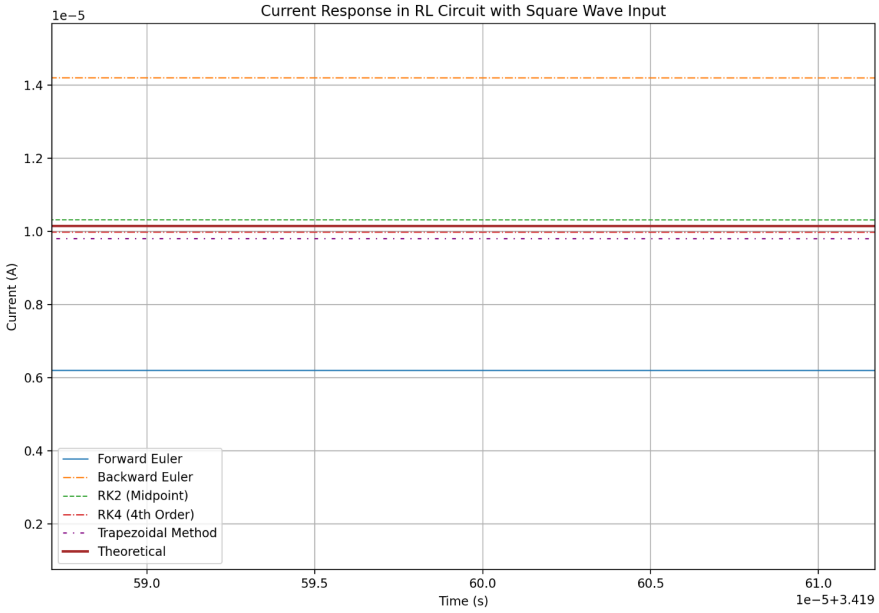
At t = 2.6960


Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Current (A)
0.0125
0.0120
0.0115
0.0110
0.0105
0.0100
0.0095
0.0090
2.714
2.716
2.718
2.720
2.722
2.724
Time (s)

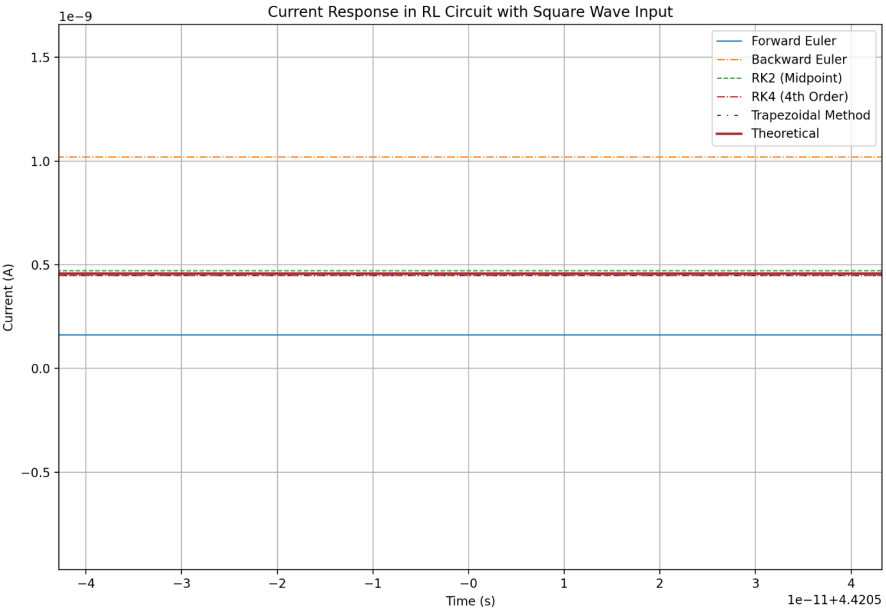
At t = 2.7115

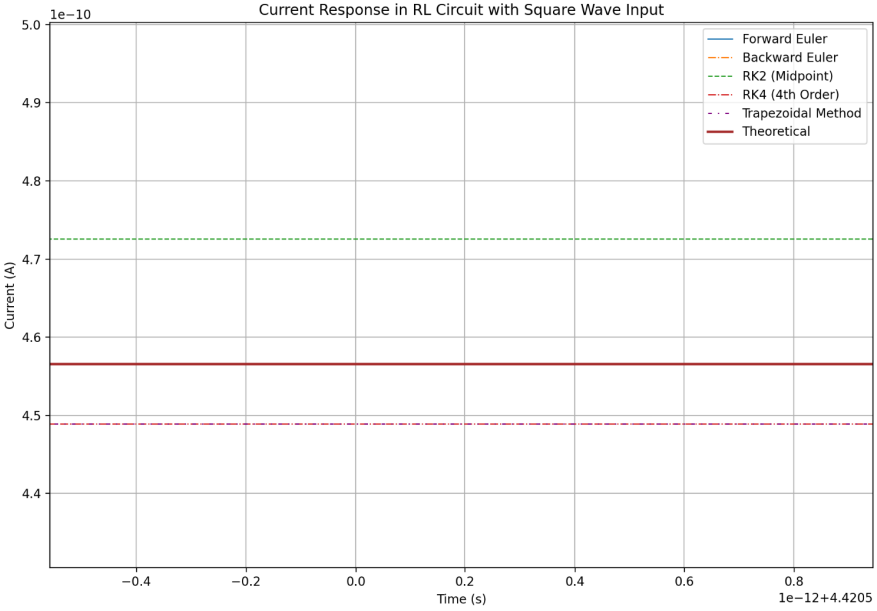


At t = 3.4196

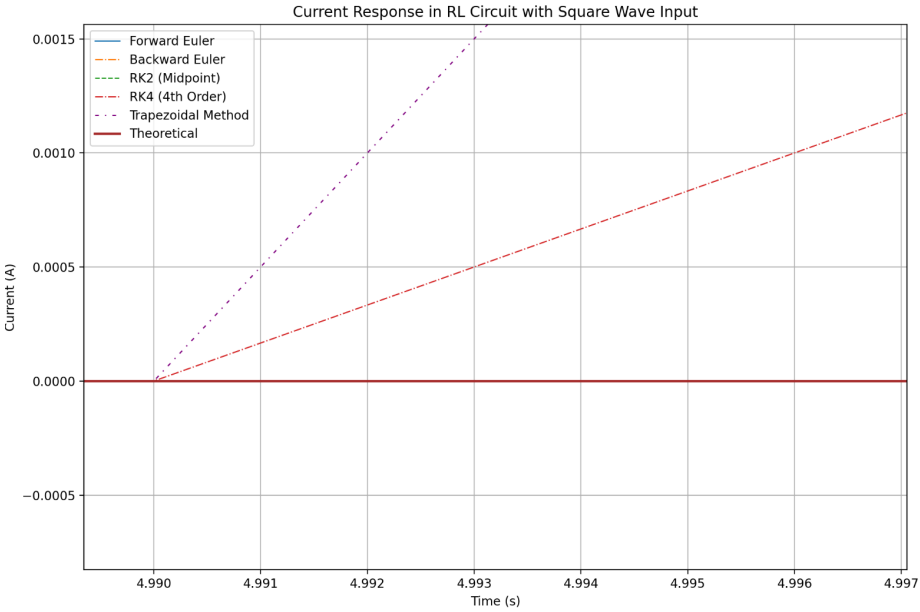


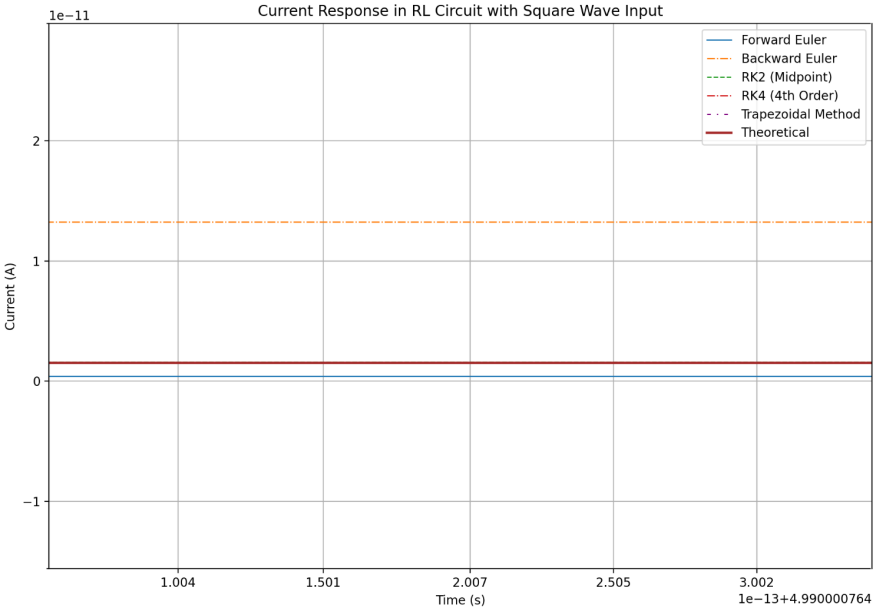
At t = 4.4205

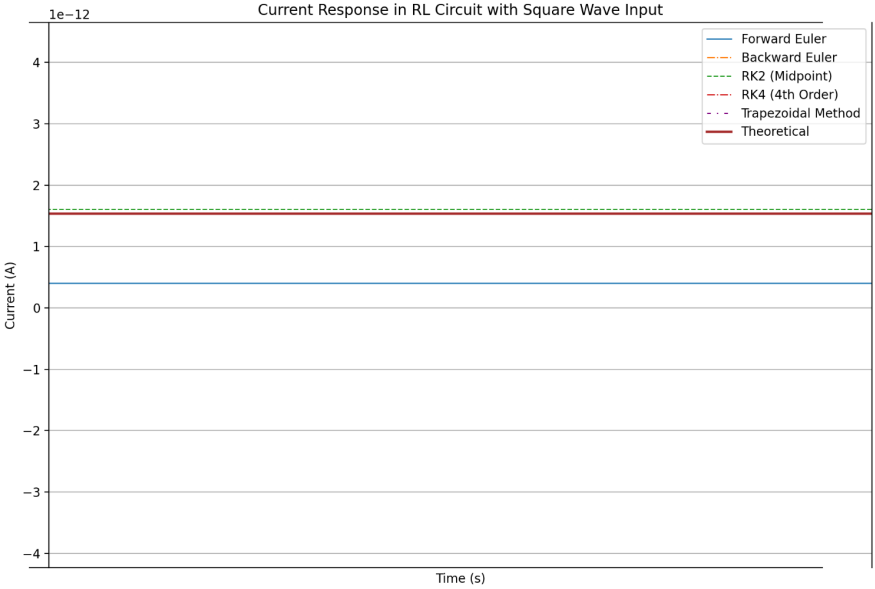

Current Response in RL Circuit with Square Wave Input
1e−9
1.5
1.0
0.5
0.0
−0.5
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
−4
−3
−2
−1
−0
1
2
3
4
Time (s)
1e−11+4.4205


Current Response in RL Circuit with Square Wave Input
1e−10
5.0
4.9
4.8
4.7
4.6
4.5
4.4
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
Time (s)
1e−12+4.4205

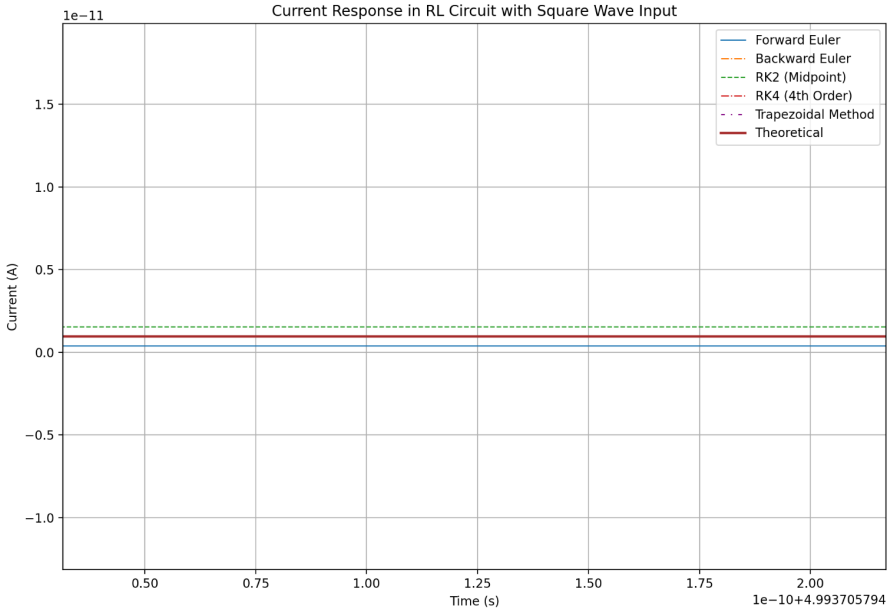
At t = 4.9901


Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
0.0015
0.0010
0.0005
0.0000
−0.0005
Current (A)
4.990
4.991
4.992
4.993
4.994
4.995
4.996
4.997
Time (s)


Current Response in RL Circuit with Square Wave Input
1e−11
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
2
1
0
−1
Current (A)
1.004
1.501
2.007
2.505
3.002
Time (s)
1e−13+4.990000764

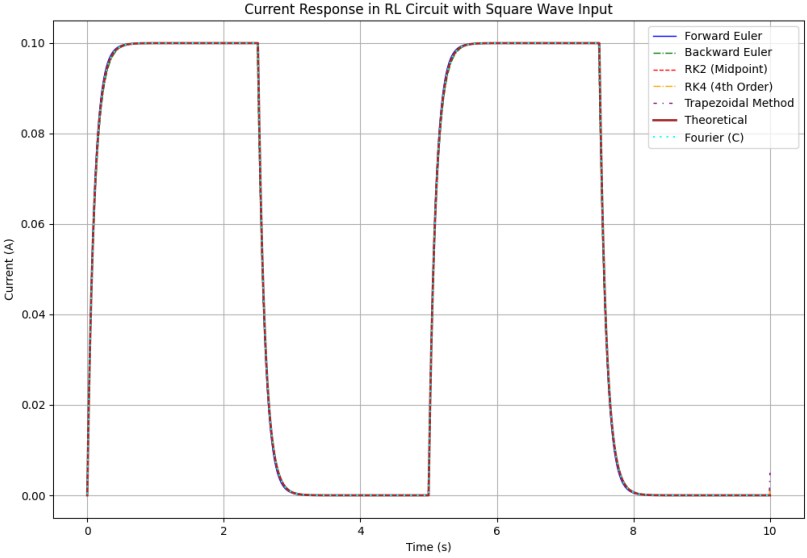

Current Response in RL Circuit with Square Wave Input
1e−12
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Current (A)
Time (s)

At t = 4.9937


Current Response in RL Circuit with Square Wave Input
1e−11
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Current (A)
Time (s)
1e−10+4.993705794

6.9 *COMPARISSION BETWEEN NUMERICAL AND FOURIER :*

The below plot consists of all the simulation methods , fourier series and theoretical method.This makes us easy to know which method is better to be used in view of accuracy.



6.9.1 *Note :*

The above mentioned Python code was used to analyze the errors in different numerical methods and compare them with the Fourier approach. This comparison helps in understanding which numerical technique better approximates the theoretical method. By evaluating the error trends, we can assess the accuracy and reliability of numerical methods against the Fourier-based solution, providing insights into their ineffectiveness for this problem.

6.9.2 *Python Code :*

```
    import numpy as np
import matplotlib.pyplot as plt
import ctypes
from ForwardEuler import solve_forward_euler
from BackwardEuler import solve_backward_euler
from rk2 import solve_rk2
from rk4 import solve_rk4
from trapezoidal import trapezoidal
from theoretical import theoretical_i
rl_lib = ctypes.CDLL("./rl_fourier.so")

# Parameters
R = 100 # Resistance in ohms
L = 10 # Inductance in henries
V0 = 10 # Square wave amplitude
T = 5 # Period of the square wave
alpha = 0.5 # Duty ratio (0 < alpha <= 1)
N = 1000 # Number of harmonics
num_cycles = 2 # Number of cycles to display
num_points = 1000 # Number of points for Fourier

# Time parameters
t0 = 0 # Start time
t_end = 10 # End time
h = 0.01 # Step size

def square_wave(t, T, V0, alpha):
    return V0 if (t % T) < alpha * T else 0

# Differential equation for RL circuit
def rl_circuit(t, i):
    return (square_wave(t, T, V0, alpha) - R * i) / L

# Time array
t_values = np.arange(t0, t_end + h, h)

# Solve using different methods
_, i_forward = solve_forward_euler(rl_circuit, t0, t_end, 0, h)
_, i_backward = solve_backward_euler(rl_circuit, t0, t_end, 0, h)
_, i_rk2 = solve_rk2(rl_circuit, t0, t_end, 0, h)
_, i_rk4 = solve_rk4(rl_circuit, t0, t_end, 0, h)
_, i_trapezoidal = trapezoidal(R, L, V0, T, h, t_end)
i_theoretical = theoretical_i(R, L, T, V0, 0, alpha, t_values)

# Allocate arrays for Fourier
fourier_time = np.zeros(num_points, dtype=np.float64)
fourier_current = np.zeros(num_points, dtype=np.float64)

# Prepare ctypes pointers
fourier_time_ctypes = np.ctypeslib.as_ctypes(fourier_time)
fourier_current_ctypes = np.ctypeslib.as_ctypes(fourier_current)
```

```
# Set C function argument types
rl_lib.compute_current.argtypes = [
    ctypes.POINTER(ctypes.c_double),
    ctypes.POINTER(ctypes.c_double),
    ctypes.c_int, ctypes.c_int,
    ctypes.c_int, ctypes.c_double,
    ctypes.c_double, ctypes.c_double,
    ctypes.c_double
]

# Call C function for Fourier calculation
rl_lib.compute_current(fourier_time_ctypes, fourier_current_ctypes,
                       num_cycles, num_points, N, V0, T, R, L)

# Resample for error calculation
fine_t = np.arange(t0, t_end, 0.0001)
i_forward_interp = np.interp(fine_t, t_values, i_forward)
i_backward_interp = np.interp(fine_t, t_values, i_backward)
i_rk2_interp = np.interp(fine_t, t_values, i_rk2)
i_rk4_interp = np.interp(fine_t, t_values, i_rk4)
i_trapezoidal_interp = np.interp(fine_t, t_values, i_trapezoidal)
i_theoretical_interp = np.interp(fine_t, t_values, i_theoretical)
fourier_current_interp = np.interp(fine_t, fourier_time, fourier_current)

# Calculate errors including Fourier
def calculate_errors():
    errors = {
        'Forward Euler': np.abs(i_forward_interp - i_theoretical_interp),
        'Backward Euler': np.abs(i_backward_interp - i_theoretical_interp),
        'RK2 (Midpoint)': np.abs(i_rk2_interp - i_theoretical_interp),
        'RK4 (4th Order)': np.abs(i_rk4_interp - i_theoretical_interp),
        'Trapezoidal': np.abs(i_trapezoidal_interp - i_theoretical_interp),
        'Fourier (C)': np.abs(fourier_current_interp - i_theoretical_interp)
    }
    return errors

errors = calculate_errors()

# Track least error method
prev_least_error = None
for i, t in enumerate(fine_t):
    current_order = sorted(errors.items(), key=lambda x: x[1][i], reverse=
        True)
    least_error_method = current_order[-1][0]

    if least_error_method != prev_least_error:
        print(f"At t = {t:.4f}s, error order (desc):")
        for method, error in current_order:
            print(f"{method}: {error[i]:.6f}")
        prev_least_error = least_error_method

# Plot all results
plt.figure(figsize=(12, 8))
```

```
plt.plot(t_values, i_forward, label='Forward Euler', linestyle='-', linewidth
    =1, color='blue')
plt.plot(t_values, i_backward, label='Backward Euler', linestyle='dashdot',
    linewidth=1, color='green')
plt.plot(t_values, i_rk2, label='RK2 (Midpoint)', linestyle='--', linewidth
    =1, color='red')
plt.plot(t_values, i_rk4, label='RK4 (4th Order)', linestyle='-.', linewidth
    =1, color='orange')
plt.plot(t_values, i_trapezoidal, label='Trapezoidal Method', linestyle=(0,
    (3, 5, 1, 5)), linewidth=1, color='purple')
plt.plot(t_values, i_theoretical, label='Theoretical', linestyle='-',
    linewidth=2, color='brown')
plt.plot(fourier_time, fourier_current, label='Fourier (C)', linestyle=(0,
    (1,3)), linewidth=1.5, color='cyan')

# Customize plot
plt.title('Current Response in RL Circuit with Square Wave Input')
plt.xlabel('Time (s)')
plt.ylabel('Current (A)')
plt.legend()
plt.grid(True)
plt.show()
```

6.9.3 *Data Obtained :*

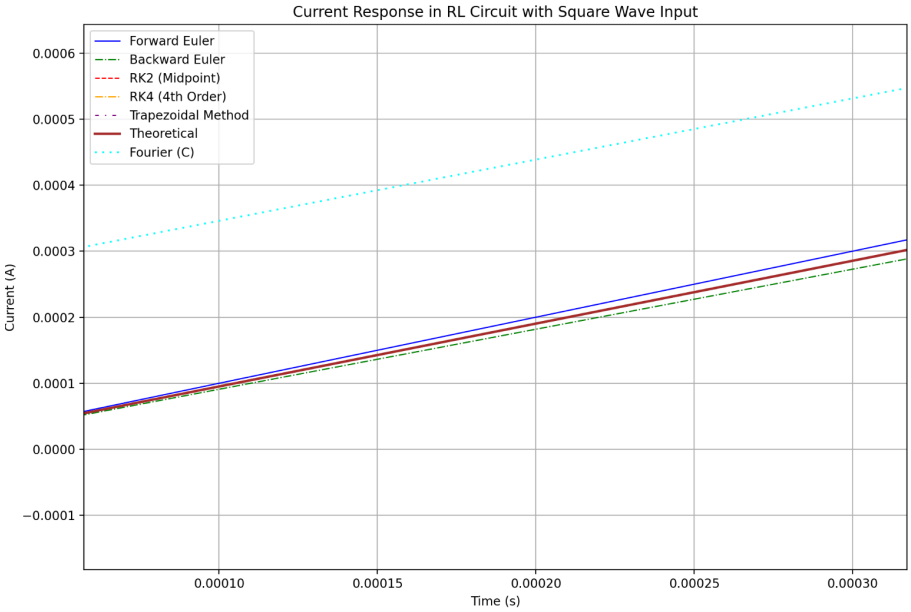
Time (s)	Highest Error		2nd Highest		3rd Highest		4th Highest		5th Highest		Least Error	
	Method	Error	Method	Error	Method	Error	Method	Error	Method	Error	Method	Error
0.0000	Four.	0.000253	FwdE	0.000000	BwdE	0.000000	RK2	0.000000	RK4	0.000000	Trap	0.000000
0.0001	Four.	0.000251	FwdE	0.000005	BwdE	0.000004	RK2	0.000000	Trap	0.000000	RK4	0.000000
0.0781	FwdE	0.001875	BwdE	0.001706	RK2	0.000064	Trap	0.000064	RK4	0.000000	Four.	0.000000
0.5400	BwdE	0.000130	FwdE	0.000114	RK2	0.000004	Trap	0.000004	Four.	0.000000	RK4	0.000000
2.4901	BwdE	0.000091	Trap	0.000050	RK4	0.000017	Four.	0.000006	FwdE	0.000000	RK2	0.000000
2.5095	BwdE	0.007902	Trap	0.004533	RK4	0.001516	FwdE	0.000460	RK2	0.000015	Four.	0.000015
3.8583	BwdE	0.000000	FwdE	0.000000	Trap	0.000000	RK2	0.000000	Four.	0.000000	RK4	0.000000
4.4205	Four.	0.000000	BwdE	0.000000	FwdE	0.000000	RK2	0.000000	RK4	0.000000	Trap	0.000000
4.9901	Trap	0.000050	RK4	0.000017	Four.	0.000006	BwdE	0.000000	FwdE	0.000000	RK2	0.000000
4.9937	Trap	0.001850	RK4	0.000617	Four.	0.000096	BwdE	0.000000	RK2	0.000000	FwdE	0.000000

TABLE 0: Error Order at Different Time Instances (Below 5s)

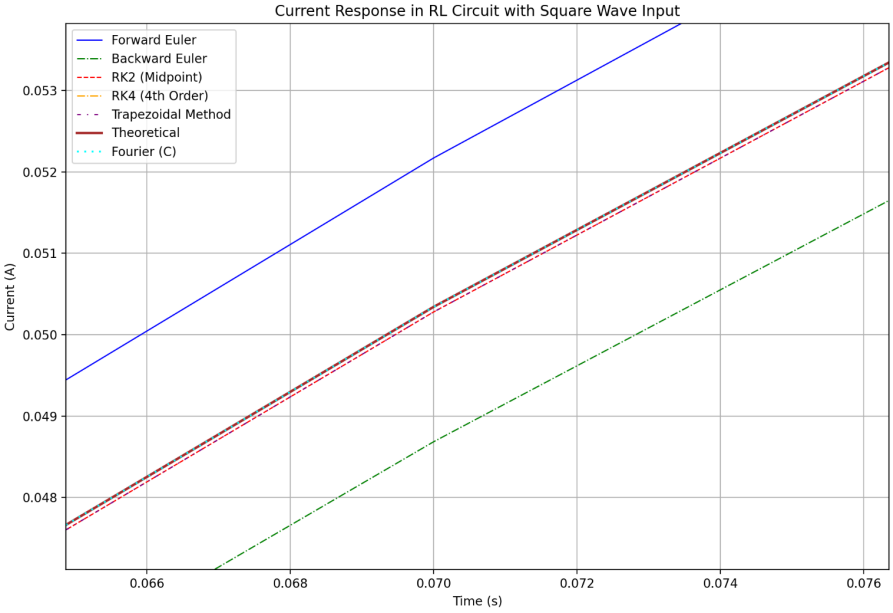
Table 0: The table shows the error rankings of different numerical methods at specific time points in the RL circuit simulation. The following abbreviations are used in the table:

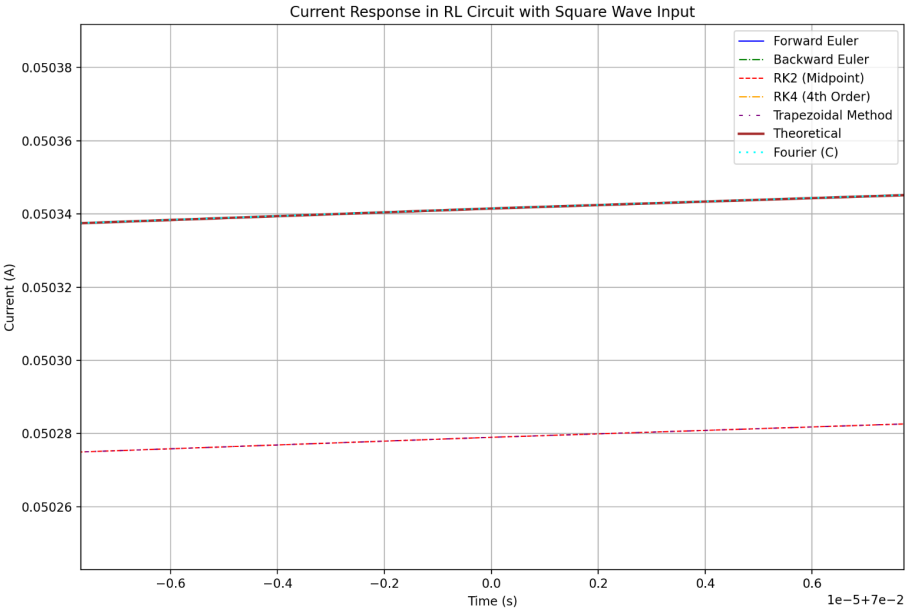
- **Four.** - Fourier (C)
- **FwdE** - Forward Euler
- **BwdE** - Backward Euler
- **RK2** - Runge-Kutta 2nd Order (Midpoint)
- **RK4** - Runge-Kutta 4th Order
- **Trap** - Trapezoidal Method

t = 0.0001

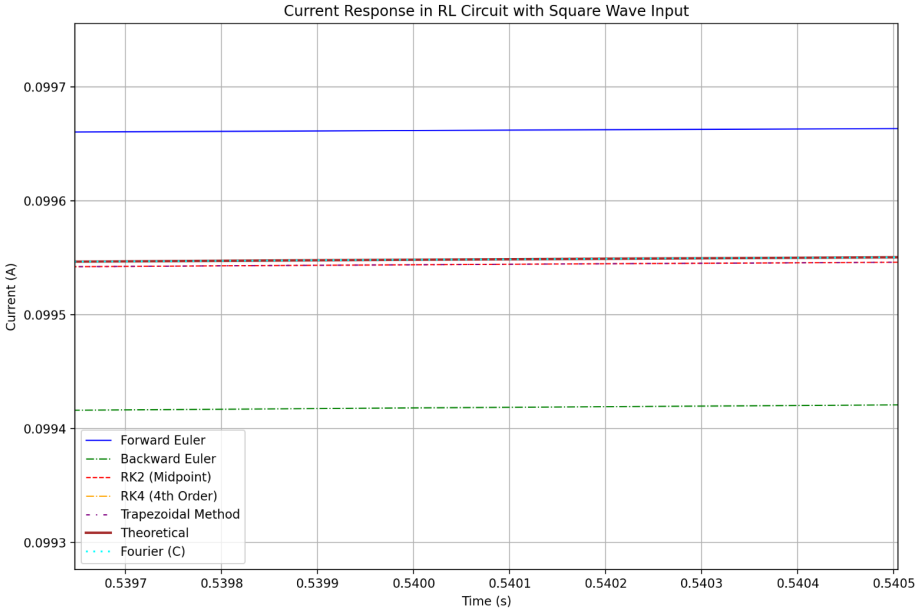


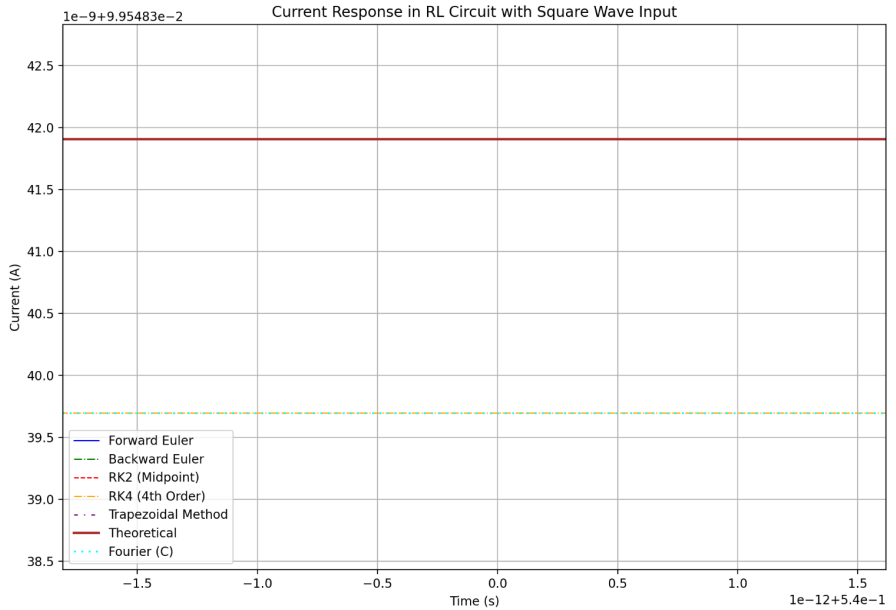
t = 0.0781

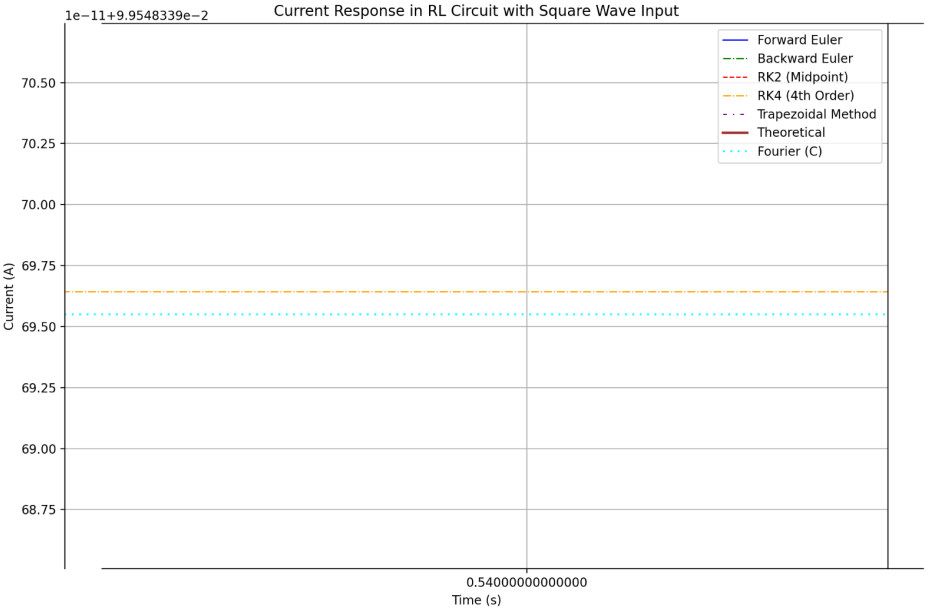



Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
Current (A)
0.05038
0.05036
0.05034
0.05032
0.05030
0.05028
0.05026
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
Time (s)
1e−5+7e−2

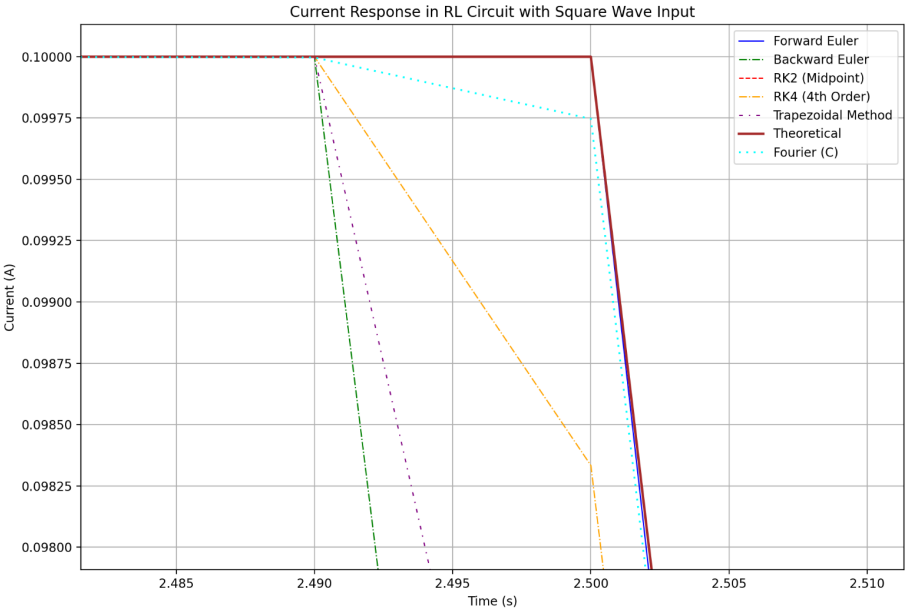
t = 0.54


Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
Current (A)
0.0997
0.0996
0.0995
0.0994
0.0993
0.5397
0.5398
0.5399
0.5400
0.5401
0.5402
0.5403
0.5404
0.5405
Time (s)

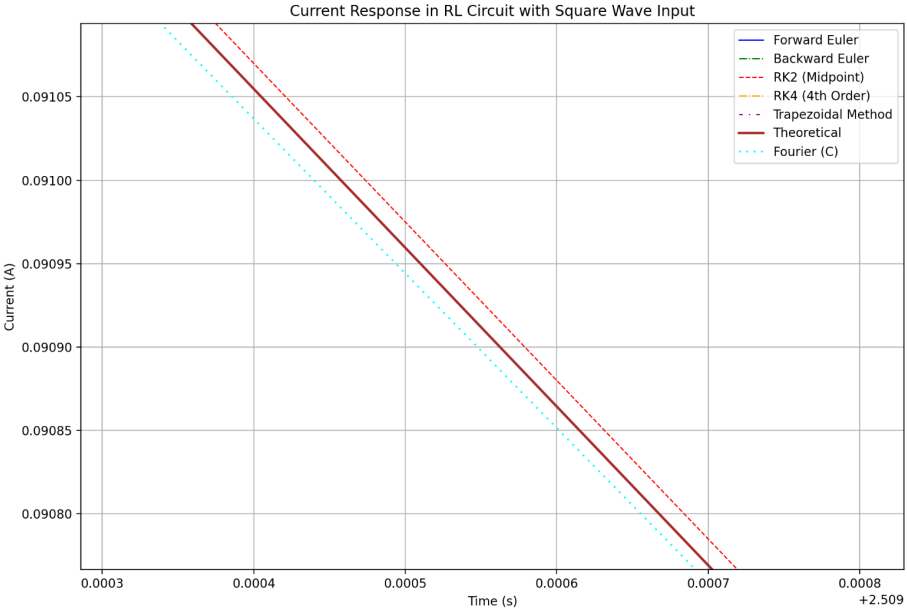

Current Response in RL Circuit with Square Wave Input
1e−9+9.95483e−2
42.5
42.0
41.5
41.0
40.5
40.0
39.5
39.0
38.5
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
Time (s)
1e−12+5.4e−1


Current Response in RL Circuit with Square Wave Input
1e−11+9.9548339e−2
70.50
70.25
70.00
69.75
69.50
69.25
69.00
68.75
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
0.54000000000000
Time (s)

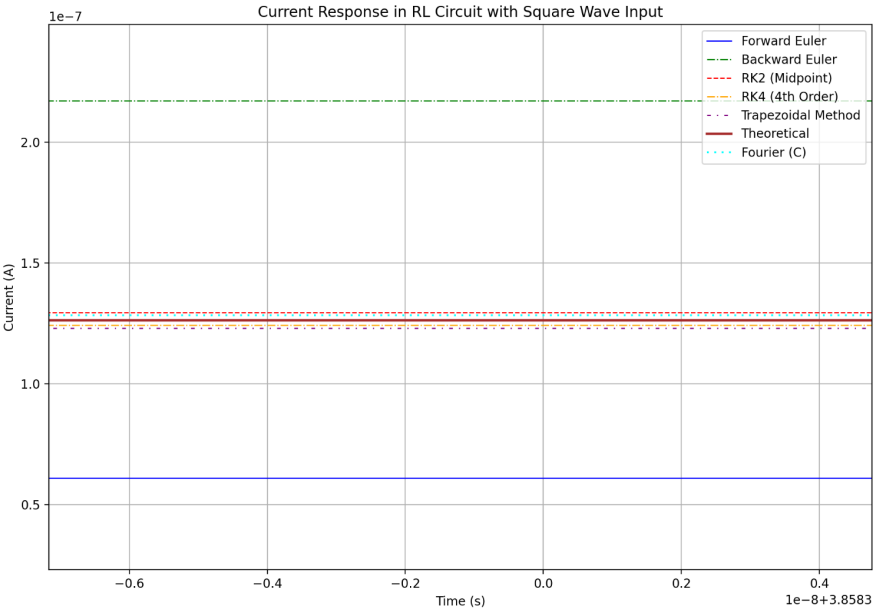
t = 2.4901



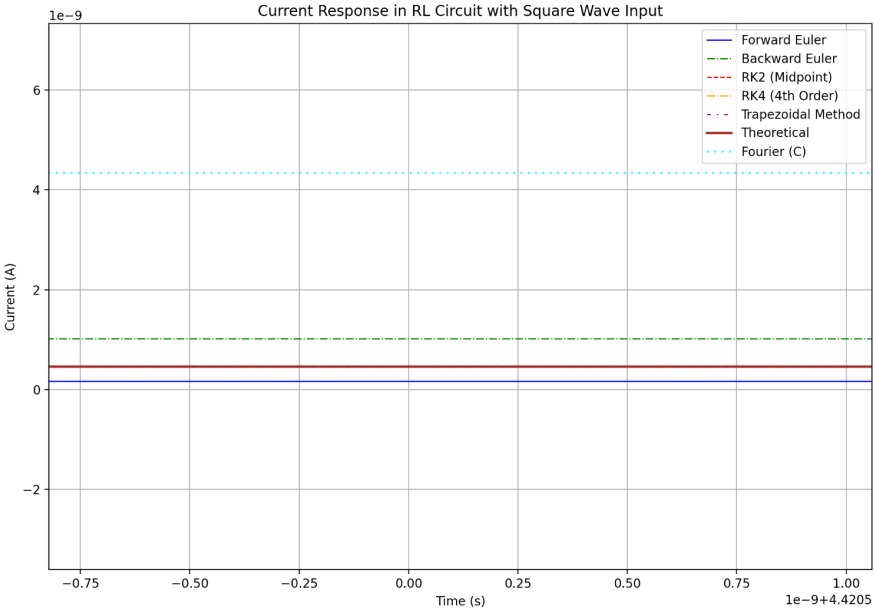
t = 2.5095

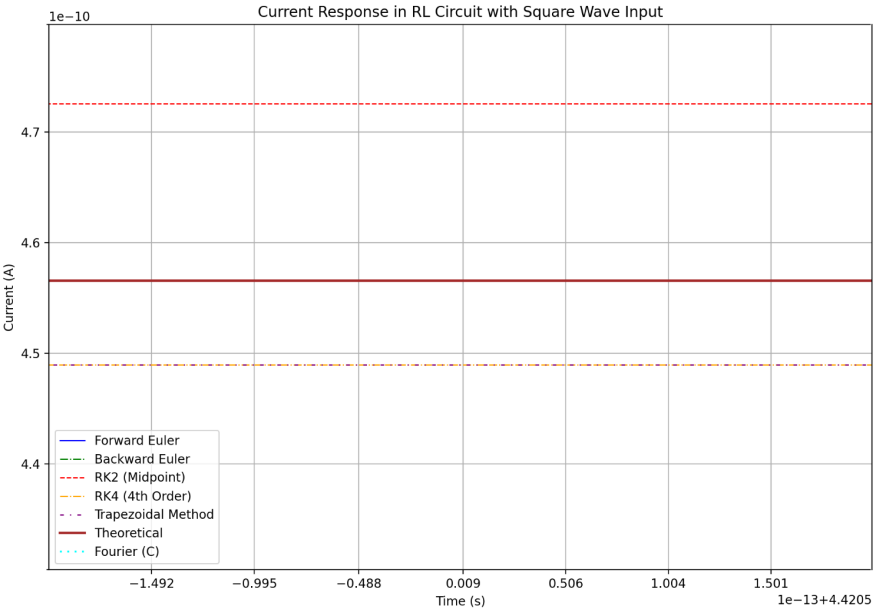


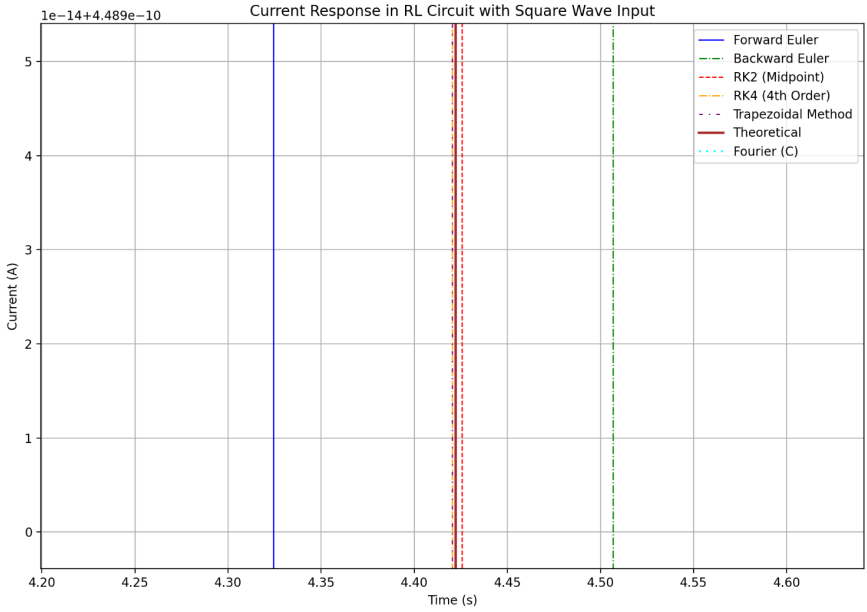
t = 3.8583



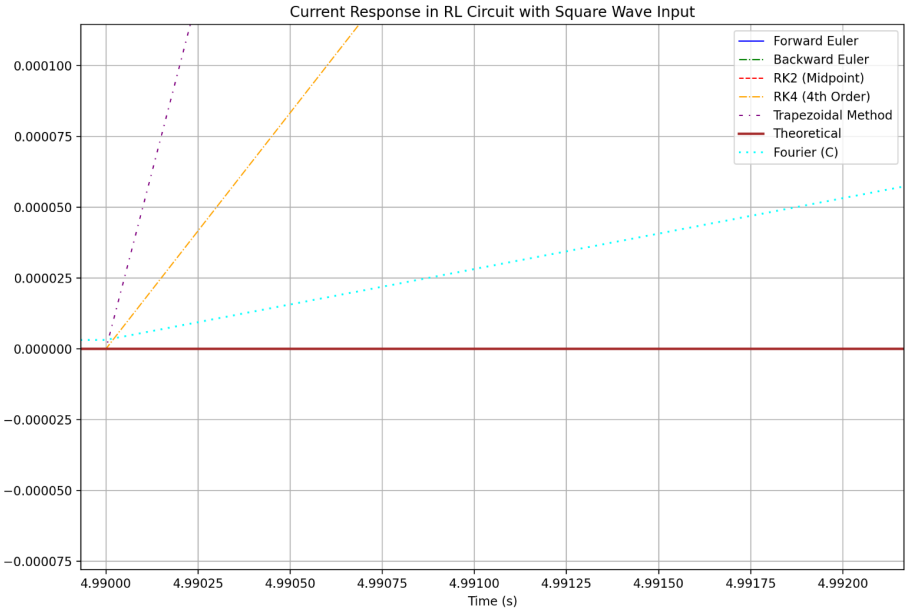
t = 4.4205

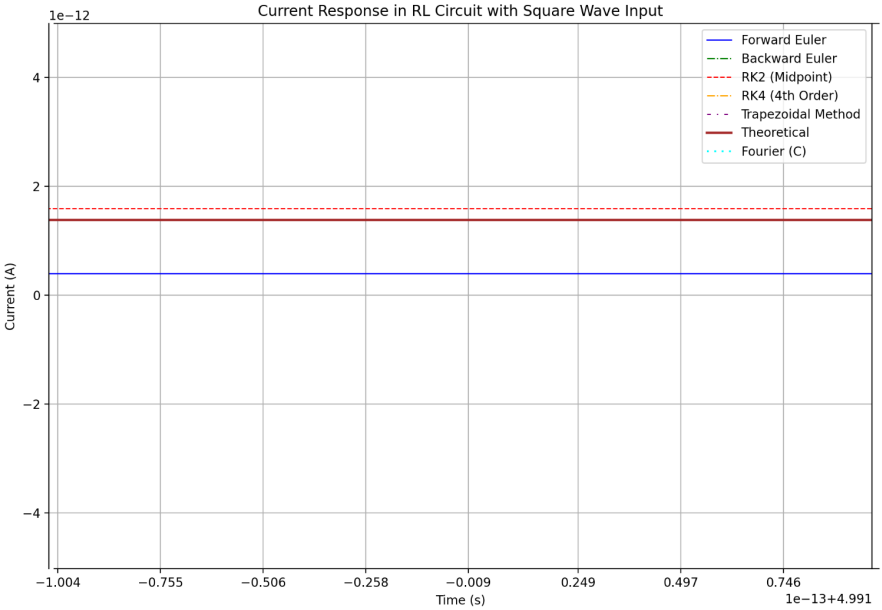



Current Response in RL Circuit with Square Wave Input
1e−10
4.7
4.6
4.5
4.4
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
−1.492
−0.995
−0.488
0.009
0.506
1.004
1.501
Time (s)
1e−13+4.4205

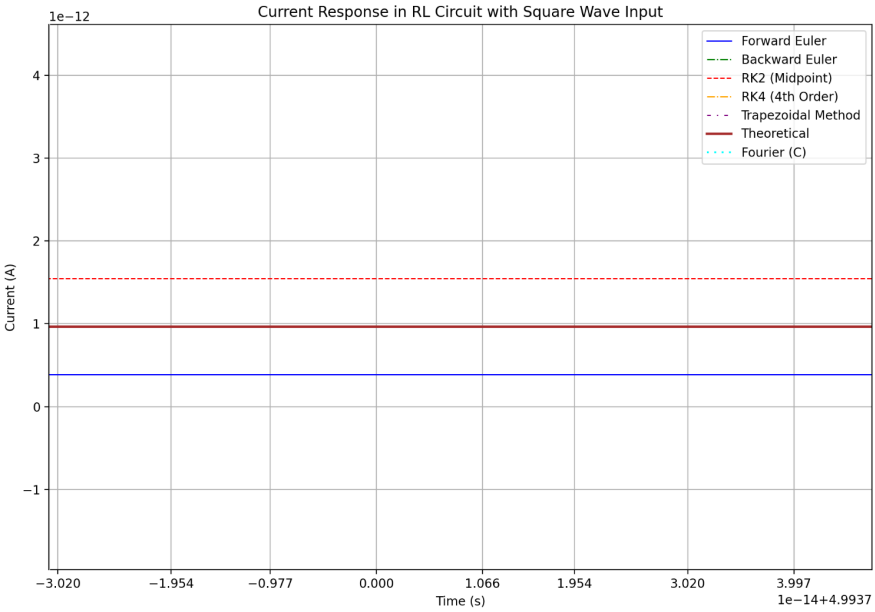

Current Response in RL Circuit with Square Wave Input
1e−14+4.489e−10
5
4
3
2
1
0
Current (A)
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
4.20
4.25
4.30
4.35
4.40
4.45
4.50
4.55
4.60
Time (s)

t = 4.9901


Current Response in RL Circuit with Square Wave Input
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
0.000100
0.000075
0.000050
0.000025
0.000000
−0.000025
−0.000050
−0.000075
4.99000
4.99025
4.99050
4.99075
4.99100
4.99125
4.99150
4.99175
4.99200
Time (s)


Current Response in RL Circuit with Square Wave Input
1e−12
Forward Euler
Backward Euler
RK2 (Midpoint)
RK4 (4th Order)
Trapezoidal Method
Theoretical
Fourier (C)
4
2
0
−2
−4
Current (A)
−1.004
−0.755
−0.506
−0.258
−0.009
0.249
0.497
0.746
Time (s)
1e−13+4.991

t = 4.9937



6.10 Conclusion

The analysis of the data and plots shows that no single numerical method consistently minimizes the error compared to the theoretical solution. Each method has strengths and weaknesses based on different factors.

Explicit methods, which rely on the next time step $(n + 1)$, struggle in regions with sudden changes or steep transitions. These methods anticipate a smooth progression, leading to errors when a drop occurs. In contrast, conventional methods like forward Euler can sometimes reduce these errors but have inherent limitations.

The Fourier-based approach maintains accuracy within the cycle but loses precision near its endpoints. Its accuracy depends on the number of terms in the Fourier series. While more terms improve accuracy, simulating infinitely many terms is impractical, creating a trade-off between computational feasibility and precision.

$$\text{Error} \propto \frac{1}{N}, \tag{0.100}$$

where N is the number of Fourier terms. This relationship shows that while error decreases with more terms, perfect accuracy is unattainable due to finite computational resources.

Each method has conditions where it performs best, but none completely dominate in minimizing error.

7 Conclusion