FDM Part 1: Vibration ODEs

Numerical Solutions of Differential Equations

Petar Bosnic

PhD Student,
Department of Process, Energy and Environmental Technology
Faculty of Technology, Natural Sciences and Maritime Sciences
University of South-Eastern Norway,
Campus Porsgrunn

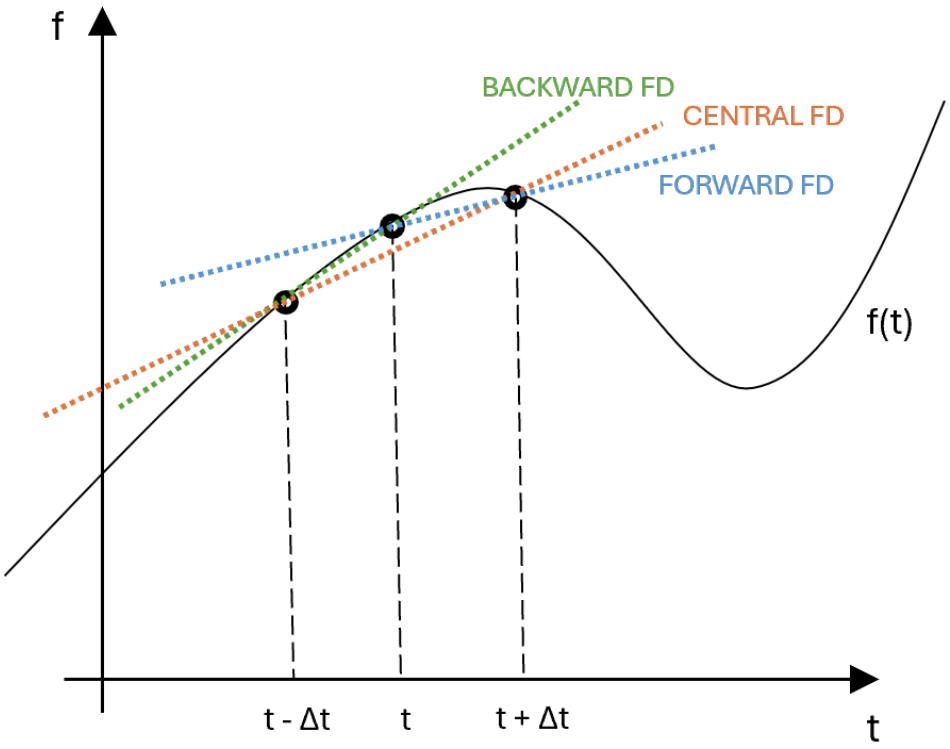
12/09/2024

Overview of Presentation

This presentation will cover the following key topics:

- **Introduction to FDM:** Basic principles of the Finite Difference Method (FDM).
- **Numerical Schemes:** Detailed discussion of numerical schemes for solving vibration ODEs.
- **Example Problem:** Step-by-step solution of a simple vibration problem using FDM.
- **Simulation Algorithm:** Construction of algorithms for simulation and analysis.
- **Verification and Validation:** Testing the accuracy and stability of the FDM implementation.
- **Generalization:** Extending FDM to solve more complex vibration problems.
- **Applications of Vibration Models:** Real-world applications of FDM in engineering and making code open-source (GitHub).
- **Open-source:** Making code open-source on GitHub.

Finite Difference Method: Overview and basic concept



First order derivative:

$$\frac{df(t)}{dt} = \lim_{\Delta t \to 0} \frac{f(t+\Delta t) - f(t)}{\Delta t}$$

$$\frac{df(t)}{dt} \approx \frac{f(t+\Delta t) - f(t)}{\Delta t}$$

Taylor Expansion:

$$f(t+\Delta t) = f(t) + \frac{df}{dt}\Delta t + \frac{d^2 f}{dt^2}\frac{\Delta t^2}{2!} + \frac{d^3 f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4) + \ldots$$

$$f(t-\Delta t) = f(t) - \frac{df}{dt}\Delta t + \frac{d^2 f}{dt^2}\frac{\Delta t^2}{2!} - \frac{d^3 f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4) - \ldots$$

Taylor Expansion:

$$f(t+\Delta t) = f(t) + \frac{df}{dt}\Delta t + \frac{d^2f}{dt^2}\frac{\Delta t^2}{2!} + \frac{d^3f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4) + ...$$

$$f(t-\Delta t) = f(t) - \frac{df}{dt}\Delta t + \frac{d^2f}{dt^2}\frac{\Delta t^2}{2!} - \frac{d^3f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4) - ...$$

Forward Difference Scheme:

$$\frac{df}{dt} \approx \frac{f(t+\Delta t) - f(t)}{\Delta t} = \frac{df}{dt} + \frac{d^2f}{dt^2}\frac{\Delta t}{2!} + \frac{d^3f}{dt^3}\frac{\Delta t^2}{3!} + ...$$

Backward Difference Scheme:

$$\frac{df}{dt} \approx \frac{f(t) - f(t-\Delta t)}{\Delta t} = \frac{df}{dt} - \frac{d^2f}{dt^2}\frac{\Delta t}{2!} + \frac{d^3f}{dt^3}\frac{\Delta t^2}{3!} - ...$$

Central Difference Scheme:

$$\frac{df}{dt} \approx \frac{f(t+\Delta t) - f(t-\Delta t)}{2\Delta t} = \frac{df}{dt} + \frac{d^3f}{dt^3}\frac{\Delta t^2}{3!} + \frac{d^4f}{dt^4}\frac{\Delta t^3}{4!} + ...$$

Second Order Derivative:

$$\frac{d^2 f(t)}{dt^2} = \lim_{\Delta t \to 0} \frac{f'(t + \Delta t) - f'(t)}{\Delta t}$$

Using the central difference scheme, we approximate the second derivative by combining forward and backward Taylor expansions.

Taylor Expansion:

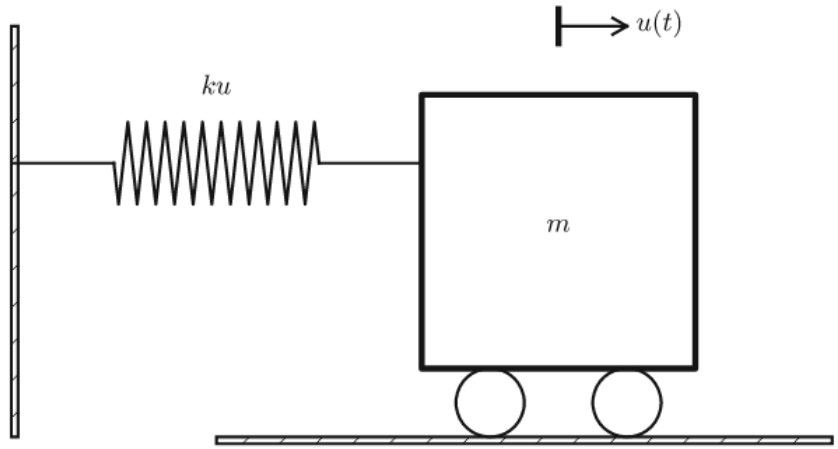
$$f(t + \Delta t) = f(t) + \frac{df}{dt}\Delta t + \frac{d^2 f}{dt^2}\frac{\Delta t^2}{2!} + \frac{d^3 f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4)$$

$$f(t - \Delta t) = f(t) - \frac{df}{dt}\Delta t + \frac{d^2 f}{dt^2}\frac{\Delta t^2}{2!} - \frac{d^3 f}{dt^3}\frac{\Delta t^3}{3!} + O(\Delta t^4)$$

Central Difference Approximation:

$$\frac{d^2 f(t)}{dt^2} \approx \frac{f(t + \Delta t) - 2f(t) + f(t - \Delta t)}{\Delta t^2} + O(\Delta t^2)$$

Vibration Problem Setup



We aim to solve the following second-order differential equation using the central difference method:

$$u''(t) + \omega^2 u(t) = 0$$

Initial Conditions:

- $u(0) = I$ (initial displacement)
- $u'(0) = 0$ (initial velocity)

Time Domain: $t \in [0, T]$

The goal is to approximate the displacement $u(t)$ at discrete time steps using the central difference method.

Central Difference Method

The central difference method approximates the second-order derivative of $u(t)$ using the formula:

$$u''(t_n) \approx \frac{u(t_{n+1}) - 2u(t_n) + u(t_{n-1})}{\Delta t^2}$$

Substituting this into the vibration equation $u''(t) + \omega^2 u(t) = 0$, we get:

$$\frac{u(t_{n+1}) - 2u(t_n) + u(t_{n-1})}{\Delta t^2} + \omega^2 u(t_n) = 0$$

Solving for $u(t_{n+1})$:

$$u(t_{n+1}) = 2u(t_n) - u(t_{n-1}) - \omega^2 \Delta t^2 u(t_n)$$

Discretization of Time

We discretize the time domain into N equally spaced intervals with time step Δt:

$$t_n = n\Delta t \quad \text{where} \quad n = 0, 1, 2, \ldots, N$$

Time Step Size:

$$\Delta t = \frac{T}{N}$$

At each time step t_n, we compute the displacement $u(t_n)$ using the central difference formula.

Applying Initial Conditions

To start the solution process, we need the values of u at the first two time steps.

- From the initial condition, $u(0) = I$.
- For $u'(0) = 0$, we use a central difference approximation for the first derivative:

$$u'(0) \approx \frac{u(1) - u(-1)}{2\Delta t} = 0 \quad \Rightarrow \quad u(-1) = u(1)$$

Recursive Solution

Using the central difference method, we compute $u(t_{n+1})$ recursively:

$$u(t_{n+1}) = 2u(t_n) - u(t_{n-1}) - \omega^2 \Delta t^2 u(t_n)$$

The algorithm proceeds as follows:

- Initialize $u(0) = I$ and $u(1) = I - 0.5\omega^2 \Delta t^2 u(0)$
- For each subsequent time step t_n, compute $u(t_{n+1})$ using the formula

Continue until $t = T$.

Python Implementation

```
def u_exact(t, I, w):
  return I * np.cos(w * t)

def solver(I, w, dt, T):
  dt = float(dt)              # Ensure dt is a float
  Nt = int(round(T / dt))     # Number of time steps
  u = np.zeros(Nt + 1)        # Solution array initialized to zero
  t = np.linspace(0, Nt * dt, Nt + 1)  # Time array

  u[0] = I      # Initial condition u(0) = I

  # Central Difference Method
  u[1] = I - 0.5 * w**2 * u[0] * dt**2  # u'(0) = 0

  for n in range(1, Nt):
    u[n+1] = 2 * u[n] - u[n-1] - w**2 * u[n] * dt**2

  return u, t
```

Visualization and Main Function

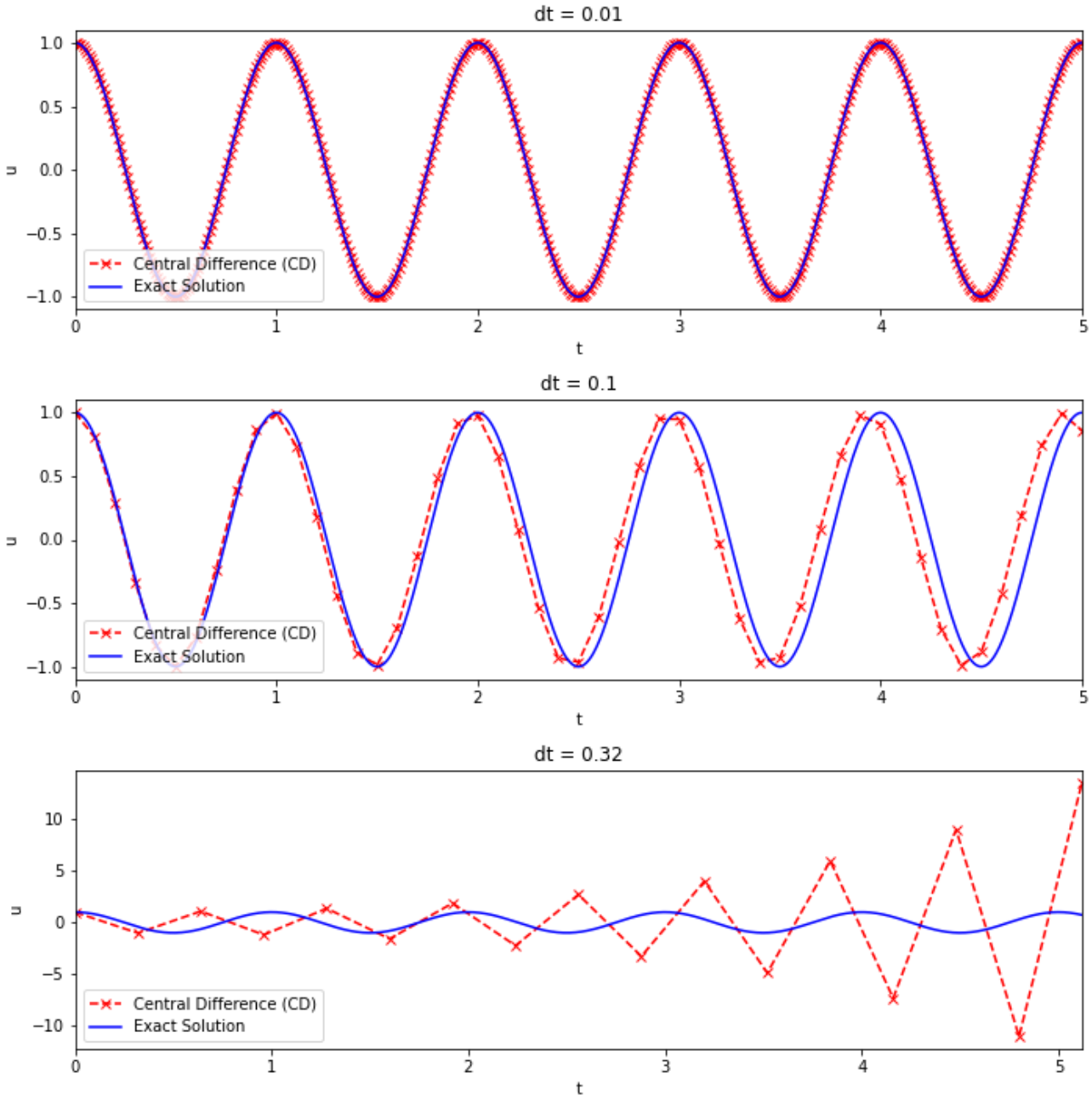
The following Python functions handle the visualization of the numerical solution compared to the exact solution:

```
def visualize(u, t, I, w, dt, ax):
 # Generate a finer time grid and compute the exact solution
 t_fine = np.linspace(0, t[-1], 1001)
 u_e = u_exact(t_fine, I, w)

 # Plot the numerical and exact solutions
 ax.plot(t, u, 'r--x', label='Central Difference (CD)')
 ax.plot(t_fine, u_e, 'b-', label='Exact Solution')

 ax.legend(loc='lower left')
 ax.set_xlabel('t')  # Label for the x-axis (time)
 ax.set_ylabel('u')  # Label for the y-axis (displacement)
 ax.set_title('dt = %g' % dt)  # Title indicating the time step
     size
 ax.set_xlim(t[0], t[-1])  # Set the limits for the x-axis (time)

 # Adjust layout to prevent overlap of plot elements
 plt.tight_layout()
```

Visualization and Main Function

```
# Main function to visualize the solutions for different dt
def main(I, w, dt_values, T):
 # Create a subplot for each time step value
 fig, axes = plt.subplots(len(dt_values), 1, figsize=(10, 10))
# Loop over each time step value
 for i, dt in enumerate(dt_values):
 # Solve the ODE using the central difference method
  u, t = solver(I, w, dt, T)
 # Visualize the numerical and exact solution
  visualize(u, t, I, w, dt, axes[i])
 # Display all the plots
 plt.show()

if __name__ == "__main__":
# User-defined parameters
I = 1.0              # Initial displacement
w = 2.0 * pi         # Angular frequency
num_periods = 5      # Number of oscillation periods to simulate
P = 2 * pi / w       # Calculate one period of the oscillation
T = P * num_periods  # Total time for the simulation
dt_values = [0.01, 0.1, 0.32]  # Different time steps to test

# Call the main function
main(I, w, dt_values, T)
```

Visualization of Central Difference Method



Verification: Test Three Steps

```
def test_three_steps():
    I = 1
    w = 2 * pi
    dt = 0.01
    T = 1
    # Expected result using exact solution (u_exact) for the first
        3 steps
    t_exact = np.array([0, dt, 2*dt])
    u_by_hand = u_exact(t_exact, I, w)

    # Solve using the solver function
    u, t = solver(I, w, dt, T)

    # Compute the difference for the first three time steps
    diff = np.abs(u_by_hand - u[:3]).max()

    # Set tolerance
    tol = 1E-3

    # Assert that the difference is smaller than tolerance
    assert diff < tol, "Test␣failed:␣max␣difference␣{}␣exceeds␣
        tolerance␣{}".format(diff, tol)

    print("Test␣passed:␣max␣difference␣is␣{}".format(diff))
```

Convergence Rates Function

```
# Function to compute convergence rates
def convergence_rates(I, w, dt, T, m, solver_function,
    num_periods=8):
  dt_values = []
  E_values = []
  for i in range(m):
    u, t = solver_function(I, w, dt, T)
    u_e = u_exact(t, I, w)
    # Error estimation using L2 norm
    E = np.sqrt(dt * np.sum((u_e - u)**2))
    dt_values.append(dt)
    E_values.append(E)
    dt = dt / 2  # Halve the time step

  # Convergence rate calculation
  r = [np.log(E_values[i-1] / E_values[i]) /
  np.log(dt_values[i-1] / dt_values[i])
  for i in range(1, m)]
  return r, E_values, dt_values
```

Convergence Rates Test Function

```
def test_convergence_rates():
    I = 1.0
    w = 2.0 * pi
    num_periods = 8
    P = 2 * pi / w
    T = P * num_periods
    dt = P / 30
    m = 5
    # Run convergence rate test for the solver
    r, E, dt_values = convergence_rates(I, w, dt, T, m, solver)
    # Set tolerance for the convergence rate (one decimal place)
    tol = 0.1
    # Check if the last calculated convergence rate is
        approximately 2.0
    assert abs(r[-1] - 2.0) < tol, "Test failed: Convergence rate
        {} deviates from expected 2.0".format(r[-1])
    print("Solver convergence rates: {}".format(r))
    print("Test passed: Convergence rate {} is within tolerance {}"
        .format(r[-1], tol))
```

Test Output from Pytest

```
=============== test session starts ================


CD_vib.py::test_three_steps Test passed: max difference is
    2.59210806663e-06
PASSED
CD_vib.py::test_convergence_rates Solver convergence rates:
    [2.003636668736194 6, 2.000949732812305, 2.000240105979095 5,
    2.000060197334432 3]
Test passed: Convergence rate 2.00006019733 is within tolerance
    0.1
PASSED

============== 2 passed in 0.35 seconds =============
```

Rewriting Second-Order ODE into First-Order System

To solve a second-order ODE numerically, we convert it into a system of first-order ODEs. Consider the second-order ODE:

$$u''(t) + \omega^2 u(t) = 0$$

Step 1: Define new variables: Let:

$$v(t) = u'(t)$$

Here, $u(t)$ is the displacement and $v(t)$ is the velocity.

Step 2: Express the second-order ODE as a system of first-order ODEs:

$$u'(t) = v(t)$$

$$v'(t) = -\omega^2 u(t)$$

Conclusion: The system is now a pair of first-order ODEs that can be solved numerically using methods such as Euler's method, Crank-Nicolson, Euler-Cromer or Runge-Kutta methods.

Forward Euler Method

The Forward Euler method solves the system using the following update scheme:

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

Algorithm:

1. Initialize $u_0 = I$, $v_0 = 0$.
2. For each time step n from 0 to $N - 1$:

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

Backward Euler Method

The Backward Euler method is implicit and requires solving for v_{n+1} at each step:

$$v_{n+1} = \frac{v_n - \Delta t \cdot \omega^2 \cdot u_n}{1 + \Delta t^2 \cdot \omega^2}$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Algorithm:

1. Initialize $u_0 = I$, $v_0 = 0$.
2. For each time step n from 0 to $N - 1$:

$$v_{n+1} = \frac{v_n - \Delta t \cdot \omega^2 \cdot u_n}{1 + \Delta t^2 \cdot \omega^2}$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Crank-Nicolson Method

The Crank-Nicolson method is semi-implicit and combines the forward and backward Euler methods:

$$u_{n+1} = u_n + \Delta t \cdot v_n^*$$

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n^*$$

where u_n^* and v_n^* are intermediate values:

$$u_n^* = u_n + 0.5 \cdot \Delta t \cdot v_n$$

$$v_n^* = v_n - 0.5 \cdot \Delta t \cdot \omega^2 \cdot u_n$$

Algorithm:

1. Initialize $u_0 = I$, $v_0 = 0$.
2. For each time step n from 0 to $N - 1$:

$$u_n^* = u_n + 0.5 \cdot \Delta t \cdot v_n$$

$$v_n^* = v_n - 0.5 \cdot \Delta t \cdot \omega^2 \cdot u_n$$

$$u_{n+1} = u_n + \Delta t \cdot v_n^*$$

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n^*$$

Runge-Kutta 2 Method (Heun's)

The RK2 method (Heun's method) uses two intermediate steps to improve accuracy:

$$u_{n+1} = u_n + \frac{\Delta t}{2}(k_1^u + k_2^u)$$

$$v_{n+1} = v_n + \frac{\Delta t}{2}(k_1^v + k_2^v)$$

where:

$$k_1^u = v_n, \quad k_1^v = -\omega^2 u_n$$

$$u_{tilde} = u_n + \Delta t \cdot k_1^u, \quad v_{tilde} = v_n + \Delta t \cdot k_1^v$$

$$k_2^u = v_{tilde}, \quad k_2^v = -\omega^2 u_{tilde}$$

Algorithm:

1. Initialize $u_0 = I$, $v_0 = 0$.
2. For each time step n from 0 to $N-1$:

$$k_1^u = v_n, \quad k_1^v = -\omega^2 u_n$$

$$u_{tilde} = u_n + \Delta t \cdot k_1^u, \quad v_{tilde} = v_n + \Delta t \cdot k_1^v$$

$$k_2^u = v_{tilde}, \quad k_2^v = -\omega^2 u_{tilde}$$

$$u_{n+1} = u_n + \frac{\Delta t}{2}(k_1^u + k_2^u)$$

Euler-Cromer Method

The Euler-Cromer method updates velocity first and then uses the updated velocity to compute displacement:

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Algorithm:

1. Initialize $u_0 = I$, $v_0 = 0$.
2. For each time step n from 0 to $N - 1$:

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Implicit vs. Explicit Schemes

Explicit Schemes:

- Directly calculate the next time step based on known values from the current or previous steps.

Implicit Schemes:

- Involve solving an equation to calculate the next time step.

Table of Explicit and Implicit Schemes:

Scheme	**Type**	**Description**
Forward Euler	Explicit	Simple, first-order accurate
Central Difference	Explicit	Second-order accurate
Runge-Kutta (RK2, RK4)	Explicit	Higher-order accurate, multiple steps
Backward Euler	Implicit	First-order accurate
Crank-Nicolson	Implicit	Second-order accurate
Euler-Cromer	Semi-Implicit	First-order accurate, common in oscillatory systems

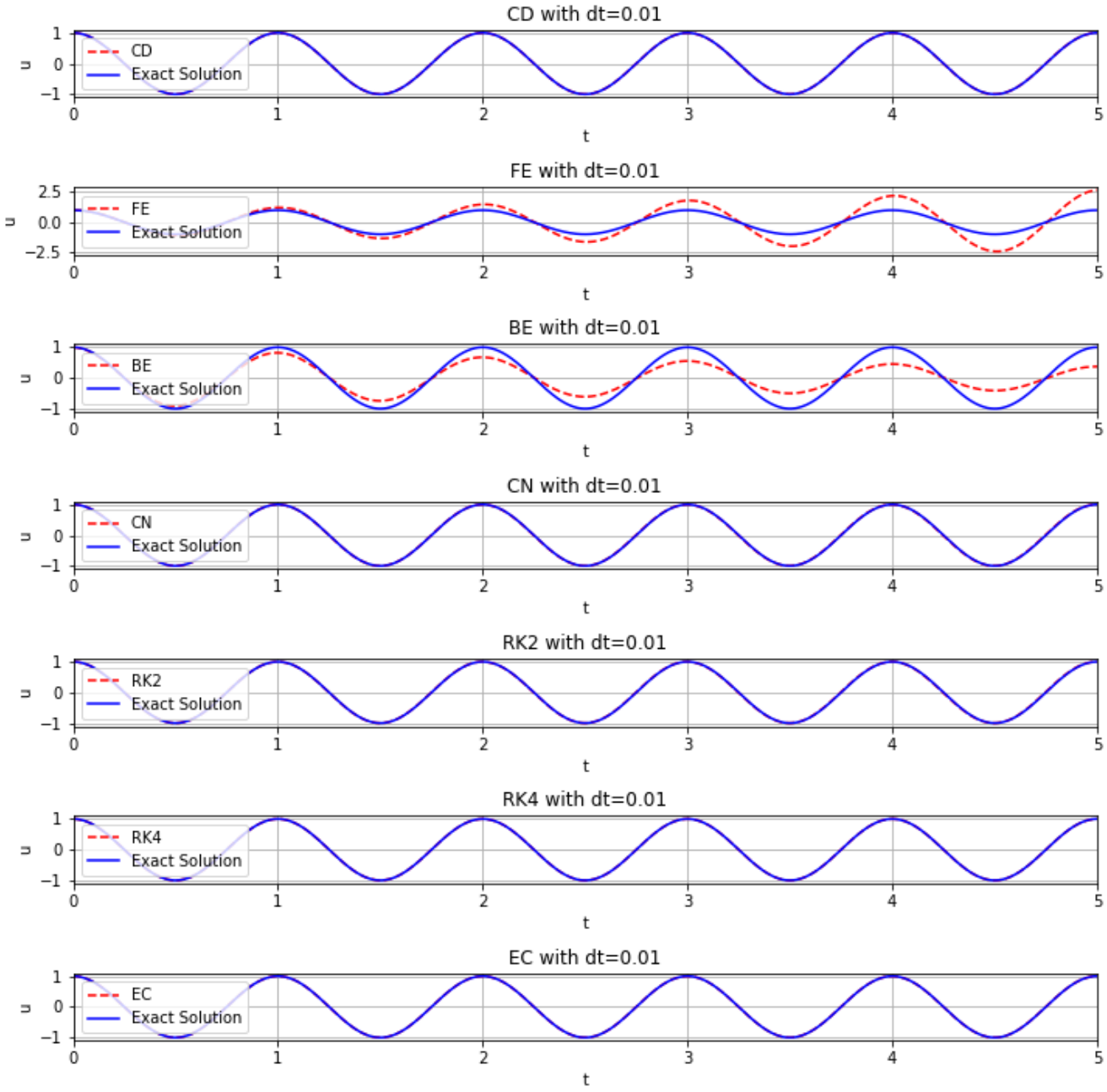
Numerical Solvers for ODEs for simple vibration system with multiple schemes

```
def solver(I, w, dt, T, method):
        dt = float(dt)
        Nt = int(round(T/dt))
        u = np.zeros(Nt + 1)
        t = np.linspace(0, Nt*dt, Nt+1)
        u[0] = I  # Initial condition u(0) = I

        if method == 'CD':  # Central Difference Method
        u[1] = I - 0.5 * w**2 * u[0] * dt**2  # u'(0) = 0
        for n in range(1, Nt):
                u[n+1] = 2 * u[n] - u[n-1] - w**2 * u[n] * dt**2

        elif method == 'FE':  # Forward Euler Method
        v = np.zeros(Nt + 1)
        v[0] = 0  # Initial velocity u'(0) = 0
        for n in range(Nt):
                u[n+1] = u[n] + dt * v[n]
                v[n+1] = v[n] - dt * w**2 * u[n]
```

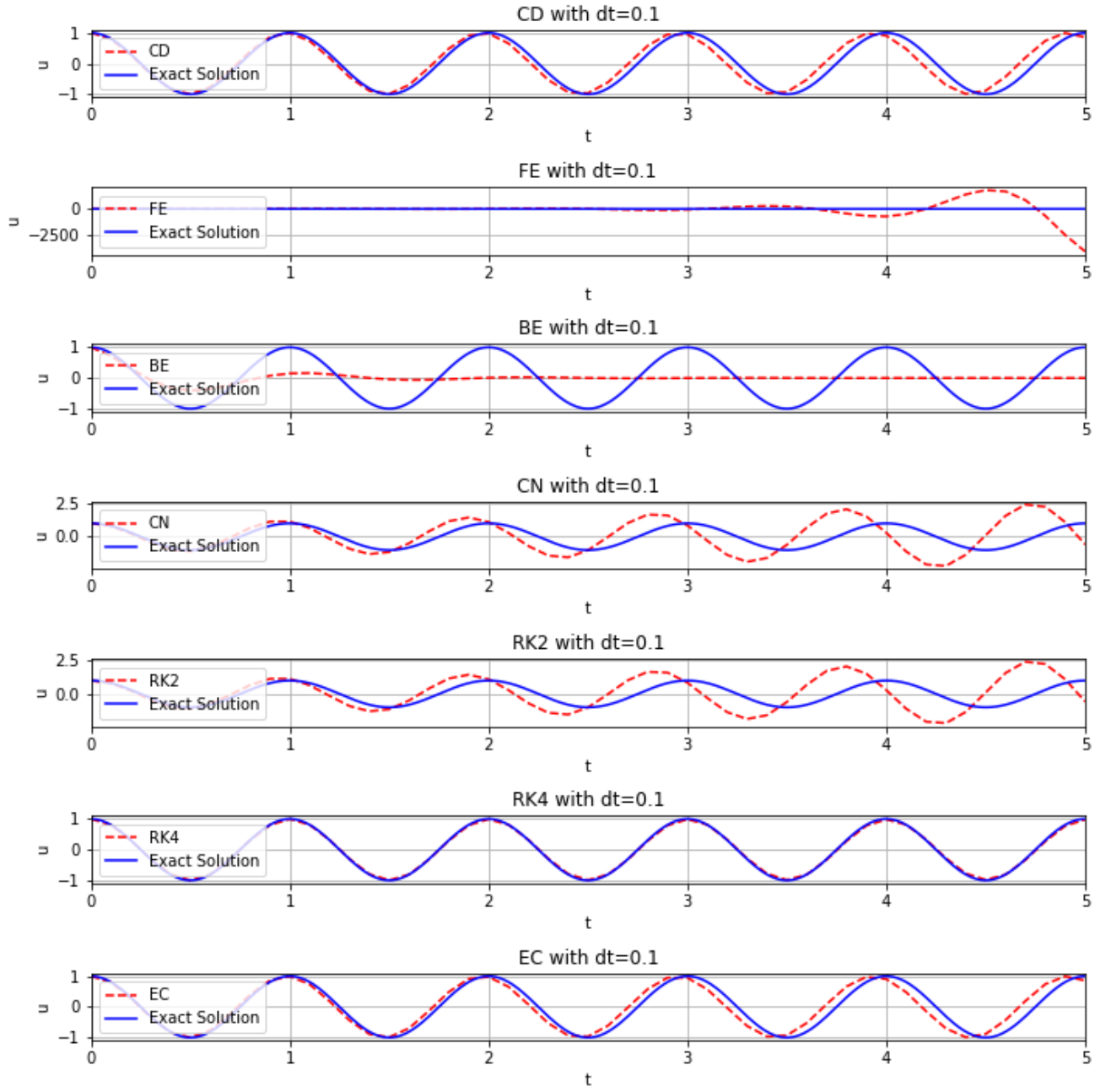
```
    elif method == 'BE':  # Backward Euler Method
    v = np.zeros(Nt + 1)
    v[0] = 0  # Initial velocity u'(0) = 0
    for n in range(Nt):
            v[n+1] = (v[n] - dt * w**2 * u[n]) / (1 + dt**2 *
                w**2)
            u[n+1] = u[n] + dt * v[n+1]

    elif method == 'CN':  # Crank-Nicolson Method
    v = np.zeros(Nt + 1)
    v[0] = 0  # Initial velocity u'(0) = 0
    for n in range(Nt):
            u_star = u[n] + 0.5 * dt * v[n]
            v_star = v[n] - 0.5 * dt * w**2 * u[n]
            u[n+1] = u[n] + dt * v_star
            v[n+1] = v[n] - dt * w**2 * u_star

    elif method == 'EC':  # Euler-Cromer Method
            v = np.zeros(Nt + 1)
            v[0] = 0  # Initial velocity u'(0) = 0
            for n in range(Nt):
                    if n == 0:
                            v[1] = v[0] - 0.5*dt*w**2*u[n]
                    else:
                            v[n+1] = v[n] - dt * w**2 * u[n]
                            u[n+1] = u[n] + dt * v[n+1]
```

```
    elif method == 'RK2':  # Runge-Kutta 2 Method
        v = np.zeros(Nt + 1)
        v[0] = 0  # Initial velocity u'(0) = 0
        for n in range(Nt):
            k1_u = v[n]
            k1_v = -w**2 * u[n]
            u_tilde = u[n] + dt * k1_u
            v_tilde = v[n] + dt * k1_v
            k2_u = v_tilde
            k2_v = -w**2 * u_tilde
            u[n+1] = u[n] + 0.5 * dt * (k1_u + k2_u)
            v[n+1] = v[n] + 0.5 * dt * (k1_v + k2_v)

    elif method == 'RK4':  # Runge-Kutta 4 Method
        v = np.zeros(Nt + 1)
        v[0] = 0  # Initial velocity u'(0) = 0
        for n in range(Nt):
            k1_u = v[n]
            k1_v = -w**2 * u[n]
            k2_u = v[n] + 0.5 * dt * k1_v
            k2_v = -w**2 * (u[n] + 0.5 * dt * k1_u)
            k3_u = v[n] + 0.5 * dt * k2_v
            k3_v = -w**2 * (u[n] + 0.5 * dt * k2_u)
            k4_u = v[n] + dt * k3_v
            k4_v = -w**2 * (u[n] + dt * k3_u)
            u[n+1] = u[n] + (dt/6.0) * (k1_u + 2*k2_u
                + 2*k3_u + k4_u)
            v[n+1] = v[n] + (dt/6.0) * (k1_v + 2*k2_v
                + 2*k3_v + k4_v)

    return u, v, t
```

Visualization of Different Methods: dt = 0.01



Visualization of Different Methods: dt = 0.1



Error and Convergence Test Results

3-Step Error Test (dt = 0.01):

- **Method CD:** PASSED. Max difference is 2.59e-06, expected [1.00, 0.9980, 0.9921], got [1.00, 0.9980, 0.9921].
- **Method FE:** FAILED (Difference: 0.00394, Tolerance: 0.001).
- **Method BE:** FAILED (Difference: 0.00388, Tolerance: 0.001).
- **Method CN:** PASSED. Max difference is 6.49e-06, expected [1.00, 0.9980, 0.9921], got [1.00, 0.9980, 0.9921].
- **Method RK2:** PASSED. Max difference is 6.49e-06, expected [1.00, 0.9980, 0.9921], got [1.00, 0.9980, 0.9921].
- **Method RK4:** PASSED. Max difference is 1.20e-09, expected [1.00, 0.9980, 0.9921], got [1.00, 0.9980, 0.9921].
- **Method EC:** PASSED. Max difference is 2.59e-06, expected [1.00, 0.9980, 0.9921], got [1.00, 0.9980, 0.9921].

Error Convergence Test (dt = 0.01):

- **Method CD:** PASSED. Convergence rate 2.0000 within tolerance 0.1.
- **Method FE:** FAILED. Convergence rate 1.0204 deviates from 2.0.
- **Method BE:** FAILED. Convergence rate 0.9808 deviates from 2.0.
- **Method CN:** PASSED. Convergence rate 2.0003 within tolerance 0.1.
- **Method RK2:** PASSED. Convergence rate 2.0003 within tolerance 0.1.
- **Method RK4:** FAILED. Convergence rate 4.0005 deviates from 2.0.
- **Method EC:** PASSED. Convergence rate 2.0000 within tolerance 0.1.

Error Calculation (dt = 0.01):

- **Method CD:** Global Error = 0.822483, Local Errors: [6.49e-07, 2.59e-06, 5.81e-06].
- **Method FE:** Global Error = 224.324891, Local Errors: [0.00197, 0.00394, 0.00587].
- **Method BE:** Global Error = 116.320050, Local Errors: [0.00196, 0.00388, 0.00574].
- **Method CN:** Global Error = 3.298773, Local Errors: [6.49e-07, 6.49e-06, 1.75e-05].
- **Method RK2:** Global Error = 3.298773, Local Errors: [6.49e-07, 6.49e-06, 1.75e-05].
- **Method RK4:** Global Error = 0.000651, Local Errors: [8.55e-11, 1.20e-09, 3.32e-09].
- **Method EC:** Global Error = 0.822483, Local Errors: [6.49e-07, 2.59e-06, 5.81e-06].

Observations from Analysis

- **Central Difference Method:** Performs well for a simple method. Stable over time.
- **Forward Euler:** Exhibits numerical amplification of amplitude, leading to instability over time in vibration problems.
- **Backward Euler:** Causes numerical damping of amplitude. Not stable for vibration problems.
- **Crank-Nicolson:** For larger dt, it can numerically amplify the amplitude, but it performs better than Forward and Backward Euler.
- **Runge-Kutta 2nd Order:** Similar behavior to Crank-Nicolson. For larger dt, it amplifies the amplitude and becomes unstable.
- **Runge-Kutta 4th Order:** The most accurate method presented. It shows great performance and stability for larger dt, though it is numerically the most 'expensive' method.
- **Euler-Cromer:** (Forward-backward scheme) Performs similarly to the Central Difference Method. Shows good performance for vibration problems.

Generalization of Vibration Equation

The general vibration equation:

$$m \cdot u''(t) + f(u'(t)) + s(u(t)) = F(t)$$

can be reduced to a system of two first-order differential equations for easier numerical solving.

Step 1: Introduce Variables

- Let $u(t)$ be the displacement.
- Let $v(t) = u'(t)$ be the velocity.

Step 2: Rewrite the System

$$u'(t) = v(t)$$

$$v'(t) = \frac{1}{m}\left(F(t) - f(v(t)) - s(u(t))\right)$$

Step 3: Initial Conditions

- The system requires two initial conditions to start the numerical integration:
  - $u(0)$: The initial displacement of the system.
  - $v(0)$: The initial velocity of the system.
- These initial conditions define the starting point for the solver

Forward Euler Method

Deriving the Velocity and Displacement Updates:

1. Acceleration: The equation of motion is:

$$m \cdot u''(t) + f(u'(t)) + s(u(t)) = F(t)$$

Solving for the acceleration $a(t)$:

$$u''(t) = \frac{F(t) - f(u'(t)) - s(u(t))}{m} = a(t)$$

At each time step, the acceleration at time t_n is:

$$a_n = \frac{F(t_n) - f(v_n) - s(u_n)}{m}$$

2. Velocity Update: Using the Forward Euler approximation for the first derivative $v'(t) = u''(t)$:

$$v'(t) \approx \frac{v_{n+1} - v_n}{dt}$$

Rearranging, we get:

$$v_{n+1} = v_n + dt \cdot a_n$$

This updates the velocity at the next time step using the acceleration at the current time step.

Forward Euler Method

3. Displacement Update: Similarly, for displacement u, we approximate the first derivative $u'(t) = v(t)$:

$$u'(t) \approx \frac{u_{n+1} - u_n}{dt}$$

Rearranging, we get:

$$u_{n+1} = u_n + dt \cdot v_n$$

This updates the displacement at the next time step using the current velocity.

Key Equations:

- Acceleration:

$$a_n = \frac{F(t_n) - f(v_n) - s(u_n)}{m}$$

- Velocity Update:

$$v_{n+1} = v_n + dt \cdot a_n$$

- Displacement Update:

$$u_{n+1} = u_n + dt \cdot v_n$$

Euler-Cromer Method

The Euler-Cromer method is a variant of the Euler method where the velocity is updated first, and then the updated velocity is used to compute the displacement. This makes the method more stable for oscillatory problems.

1. Acceleration: The equation of motion is:

$$m \cdot u''(t) + f(u'(t)) + s(u(t)) = F(t)$$

Solving for the acceleration $a(t)$:

$$u''(t) = \frac{F(t) - f(u'(t)) - s(u(t))}{m} = a(t)$$

At each time step, the acceleration at time t_n is:

$$a_n = \frac{F(t_n) - f(v_n) - s(u_n)}{m}$$

2. Velocity Update: The velocity is updated first using the current acceleration:

$$v_{n+1} = v_n + dt \cdot a_n$$

This ensures that the updated velocity is used to compute the displacement.

Euler-Cromer Method

3. Displacement Update: The displacement is updated using the newly computed velocity v_{n+1}:

$$u_{n+1} = u_n + dt \cdot v_{n+1}$$

This update uses the velocity at the next time step to compute the new displacement, making the method more stable for oscillatory systems.

Key Equations:

- Acceleration:

$$a_n = \frac{F(t_n) - f(v_n) - s(u_n)}{m}$$

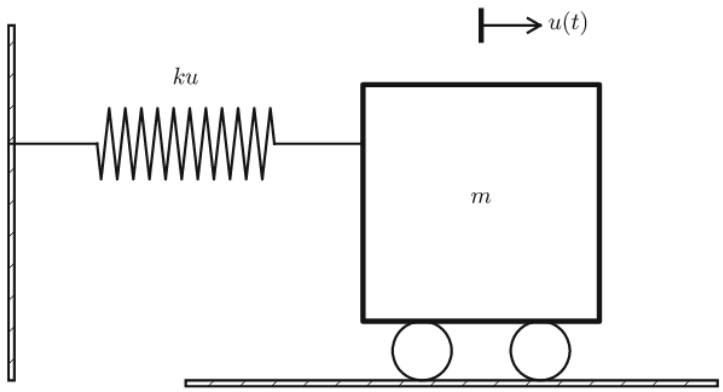
- Velocity Update:

$$v_{n+1} = v_n + dt \cdot a_n$$

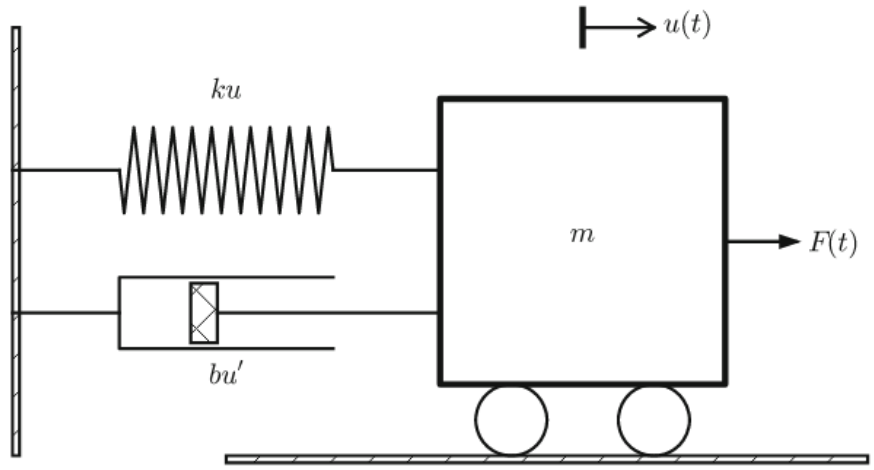
- Displacement Update:

$$u_{n+1} = u_n + dt \cdot v_{n+1}$$

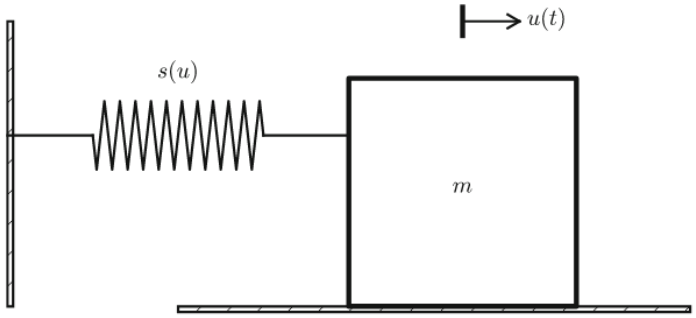
Applications in Engineering



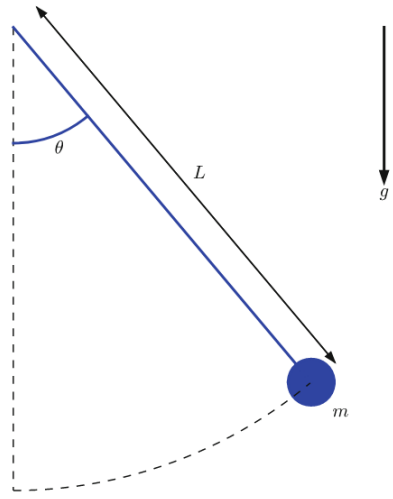
Example 1: Vibrating System



Example 2: Damped Vibrating



Example 3: Vibrating Sliding System



Example 4: Simple Pendulum System

Open Sourcing of Code

GitHub Repository:

- Explore the code and examples at: `GitHub Repository Link`