Computational Physics 113 Homework 1

邱苡熏 110022127

1 Written Assignments

1.

(a) $m\ddot{x} + kx = 0$
Solution:
The characteristic equation for this ODE is:

$$mr^2 + k = 0 \quad \Rightarrow \quad r = \pm i\sqrt{\frac{k}{m}}$$

the general solution is:

$$x(t) = A\cos\left(\sqrt{\frac{k}{m}}t\right) + B\sin\left(\sqrt{\frac{k}{m}}t\right)$$

where A and B are constants determined by initial conditions.
—
(b) $m\ddot{x} + \lambda\dot{x} + kx = 0$
Solution:
The characteristic equation for this ODE is:

$$mr^2 + \lambda r + k = 0 \quad \Rightarrow \quad r = \frac{-\lambda \pm \sqrt{\lambda^2 - 4mk}}{2m}$$

1. Overdamped case ($\lambda^2 > 4mk$): The general solution is:

$$x(t) = C_1 e^{r_1 t} + C_2 e^{r_2 t}$$

where r_1 and r_2 are the two distinct real roots of characteristic equation.
2. Critically damped case ($\lambda^2 = 4mk$): The general solution is:

$$x(t) = (C_1 + C_2 t)e^{rt}$$

where $r = \frac{-\lambda}{2m}$.
3. Underdamped case ($\lambda^2 < 4mk$): The general solution is:

$$x(t) = e^{-\frac{\lambda}{2m}t}\left(C_1\cos(\omega t) + C_2\sin(\omega t)\right)$$

where $\omega = \sqrt{\frac{4mk-\lambda^2}{2m}}$.

— (c) $m\ddot{x} + \lambda\dot{x} + kx = F_0 \cos(\omega_f t)$

Solution:

The general solution consists of two parts:

1. Homogeneous solution: This is the same as the solution from part (b)

2. Particular solution: To find the particular solution, assume a solution of the form:

$$x_p(t) = C\cos(\omega_f t - \delta)$$

Substituting this into the ODE and solving for C and δ (the phase shift) gives:

$$C = \frac{F_0}{\sqrt{(k - m\omega_f^2)^2 + (\lambda\omega_f)^2}}$$

$$\tan(\delta) = \frac{\lambda\omega_f}{k - m\omega_f^2}$$

Thus, the general solution is:

$$x(t) = e^{-\frac{\lambda}{2m}t}\left(C_1\cos(\omega t) + C_2\sin(\omega t)\right) + \frac{F_0}{\sqrt{(k - m\omega_f^2)^2 + (\lambda\omega_f)^2}}\cos(\omega_f t - \delta)$$

2 Programming Assignments

5(a)

Using Kirchhoff's Voltage Law, the total voltage around the circuit must equal the external emf. Thus, we have:

$$\mathcal{E}(t) = V_R + V_L + V_C = iR + L\frac{di}{dt} + \frac{q}{C}$$

Since $i = \frac{dq}{dt}$, we can rewrite the equation as a second-order differential equation for the charge q:

$$\mathcal{E}_0\cos(\omega t) = R\frac{dq}{dt} + L\frac{d^2q}{dt^2} + \frac{q}{C}$$

This is the governing differential equation for the RLC circuit, analogous to the mass-spring system.

The answers to the other questions are written in the Jupyter Notebook.

project1_demo3

October 24, 2024

1 Demo 3: A general IVP solver

In this jupyter notebook, we will learn how to use the general IVP solver we wrote in `./project1/solver.py`. Let's start from reproducing the results in `demo1` and `demo2`.

Before we start using the IVP solver, let's import the related packages first.

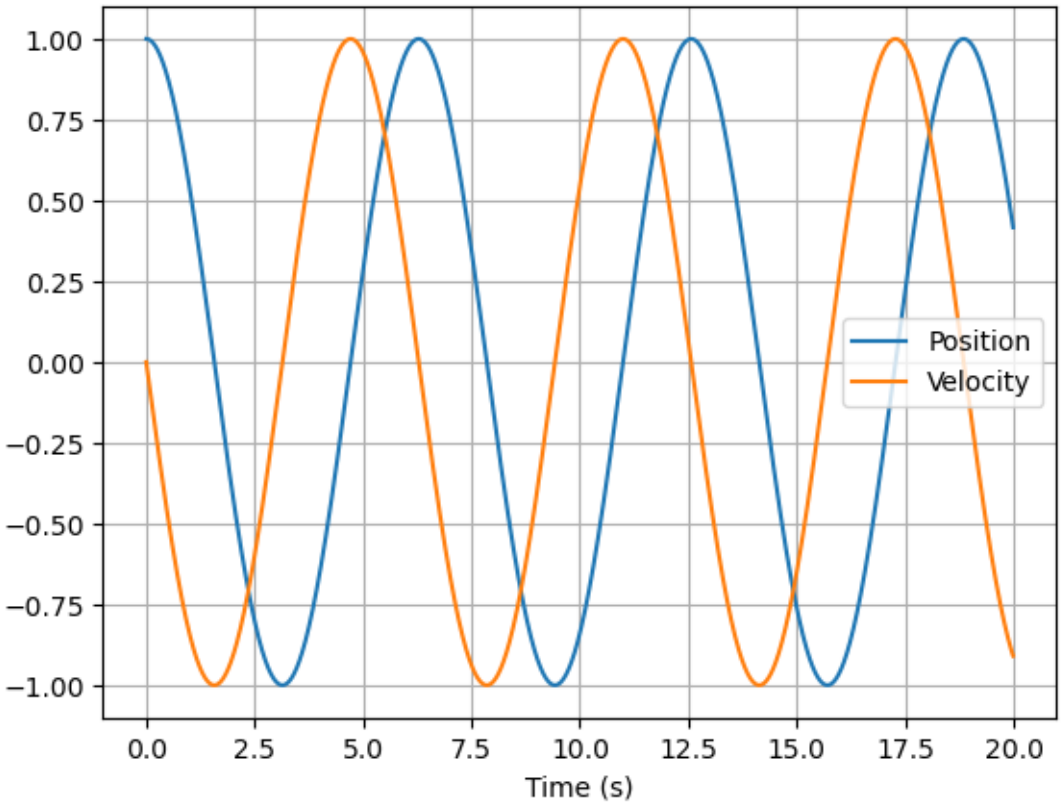
[2]:
```
import importlib
import solver
importlib.reload(solver)

import numpy as np
import matplotlib.pyplot as plt
import solver as mysolver              # your own solver
import solution.solver_sol as solver  # compare your results with solution
from scipy.integrate import solve_ivp as solver_scipy

# Note that if the path of your jupyter notebook is different from the path of␣
 ↪the solver.py file,
# you need to add the relative path of the solver.py file during the import,␣
 ↪for example:

#import project1.solver as mysolver
#import project1.solution.solver_sol as solver

# Or, you can add the path of the solver.py file to the system path, for␣
 ↪example:
# import sys
# sys.path.append('path_of_solver.py')
```

Now, let's start to reproduce `demo1` and `demo2`. Set * t0 = 0 sec and tmax = 20 sec * time step dt = 0.01 * The spring constant `K` = 1 * The mass `M` = 1 * Initial condition: `x=1`, `v=0`.

[3]:
```
# define the y' function
def derive_func(t,y,K,M):
    f = np.zeros(len(y))
    f[0] = y[1]            # y'[0] = v
    f[1] = -K * y[0]/M         # y'[1] = a = F/M
    return f
```

```
# Prepare the input arguments
K = 1
M = 1
t0 = 0
tmax = 20
dt = 0.01
y = [1, 0]  # [x0, v0]
t_eval = np.arange(t0, tmax, dt)
t_span = t_eval

# use the IVP solver
sol = mysolver.solve_ivp(derive_func,t_span,y, "RK2", t_eval, args = (K, M))
```

[4]:
```
# visualize the results

plt.plot(t_eval, sol[0], label='Position')
plt.plot(t_eval, sol[1], label='Velocity')
plt.xlabel('Time (s)')
plt.legend()
plt.grid(True)
plt.show()
```



2 Damped Oscillation

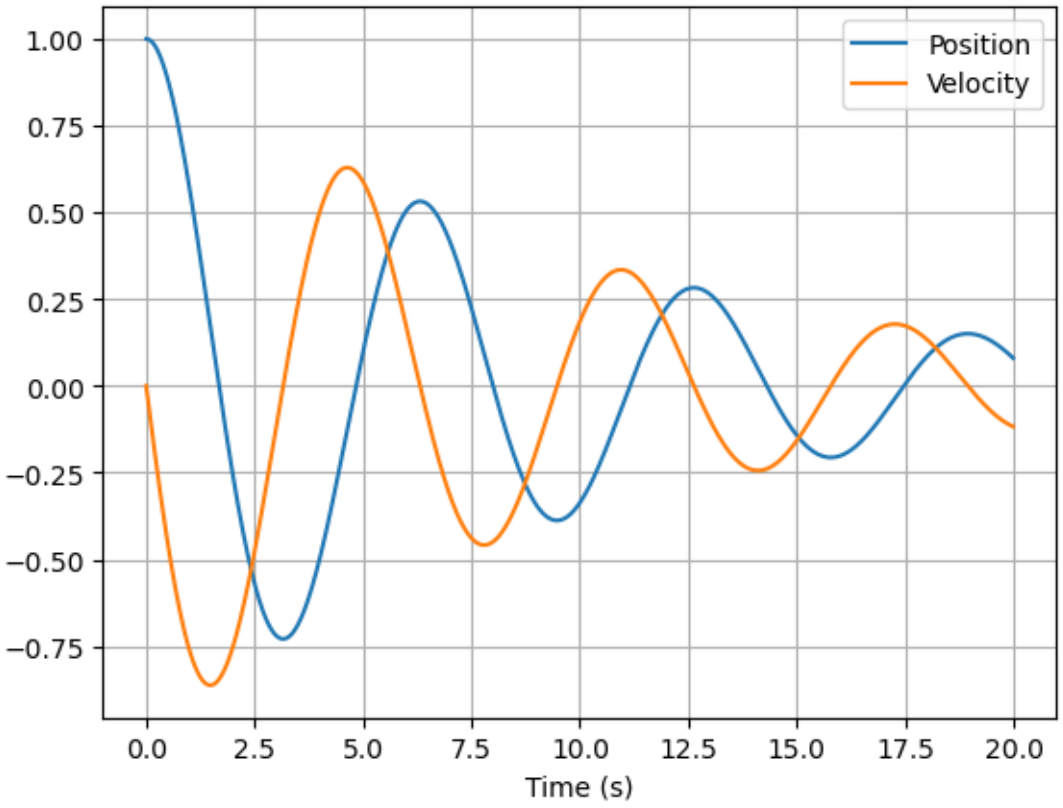
When there is a damping force ($F_{\text{damp}} = -\lambda\dot{x}$), the equation of motion becomes,

$$m\ddot{x} + \lambda\dot{x} + kx = 0$$

2.0.1 Exercise

- Use the IVP solver we developed. The only differences are the `func` and initial conditions.
- Modify the y' function (`func`) to simulate a damped oscillator (from t=0 to 20). IC: at t=0, K=M=1, A=1, $\phi = 0$,$\lambda = 0.2$.
- Make plots of position(t), velocity(t), and total energy (t). Comapred your results with analytical solutions.

[5]:
```
# TODO

# Define the damped oscillator function
def damped_oscillator(t, y, K, M, damping):
    f = np.zeros(len(y))
    f[0] = y[1]                              # dx/dt = velocity
    f[1] = -K/M * y[0] - damping/M * y[1] # dv/dt = -K/M * x - damping/M * v
    return f


# Prepare the input arguments
K = 1
M = 1
damping = 0.2  # Damping coefficient
t0 = 0
tmax = 20
dt = 0.01
y = [1, 0]  # [x0, v0] -0.5*L/M
t_eval = np.arange(t0, tmax, dt)
t_span = t_eval

# use the IVP solver
sol = mysolver.solve_ivp(damped_oscillator, t_span, y, "RK4", t_eval, args =␣
 ↪(K, M, damping))
# visualize the results
plt.plot(t_eval, sol[0], label='Position')
plt.plot(t_eval, sol[1], label='Velocity')
plt.xlabel('Time (s)')
plt.legend()
plt.grid(True)
plt.show()
```



2.0.2 Analytical Solutions

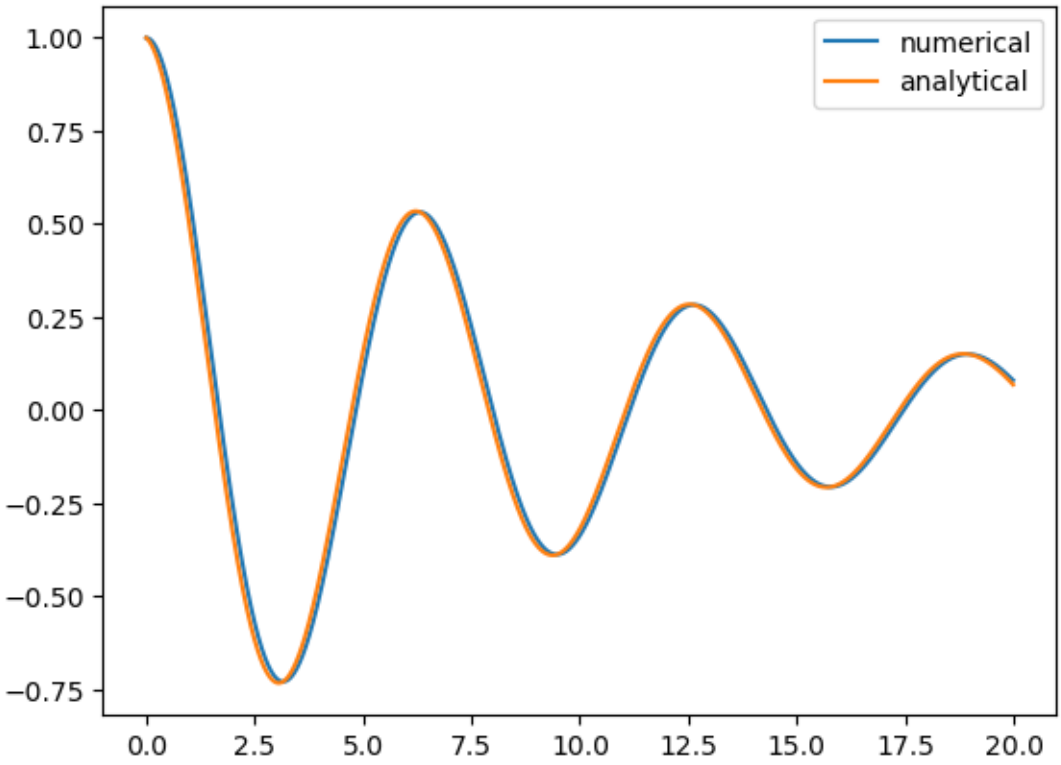
The analytical solution is

$$x(t) = Ae^{-\gamma t}\left[\cos(\omega t + \phi)\right],$$

where $\omega = \sqrt{\gamma^2 - \omega_0^2}$ or $\omega = \sqrt{\omega_0^2 - \gamma^2}$.

[6]:
```
# Compute the analytical solution


gamma = damping/(2*M)    # Damping coefficient (lambda/2M)
phi = 0        # Phase
omega_0 = np.sqrt(K/M)    # Natural frequency
omega = np.sqrt(omega_0**2 - gamma**2)  # Damped frequency

analytical_sol = np.exp(-gamma*t_eval) * (np.cos(omega*t_eval + phi))

plt.plot(t_eval, sol[0], label="numerical")
plt.plot(t_eval, analytical_sol, label="analytical")
plt.legend()
```

```
plt.show()
```



2.1 Part 2

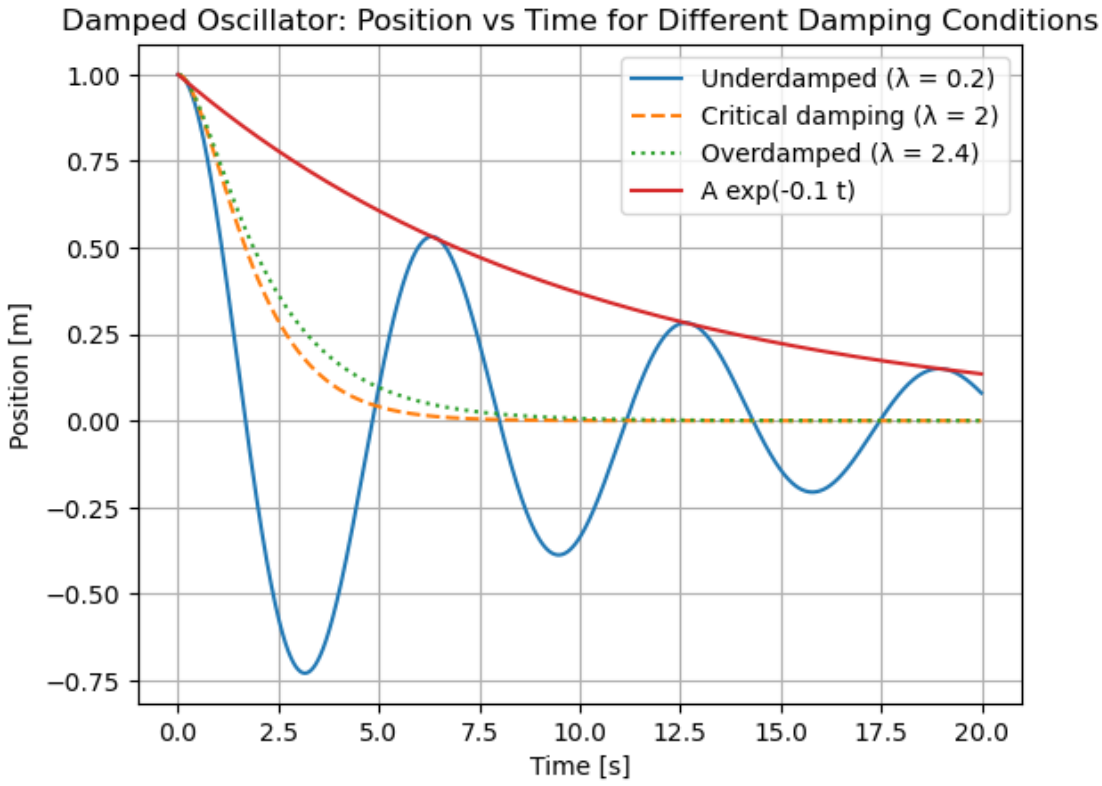
Now, let's explore the evolution of the three general cases:

- Underdamping: $\omega_0^2 > \gamma^2$
- Critical damping: $\omega_0^2 = \gamma^2$
- Overdamping: $\omega_0^2 < \gamma^2$

Vary λ, to the corresponding conditions: * $\lambda = 0.2$ ($\omega_0^2 > \gamma^2$) * $\lambda = 2$ ($\omega_0^2 = \gamma^2$) * $\lambda = 2.4$ ($\omega_0^2 < \gamma^2$)

[7]:
```
# TODO

# Helper function to run simulation for different damping values
def run_simulation(gamma, label, linestyle='-'):
    # Solve the IVP problem for the damped oscillator
    sol = mysolver.solve_ivp(damped_oscillator, t_eval, y, "RK4", t_eval,␣
 ↪args=(K, M, gamma))
    # Plot the position vs time
    plt.plot(t_eval, sol[0], label=label, linestyle=linestyle)

# Underdamping case (lambda = 0.2, omega_0^2 > gamma^2)
```

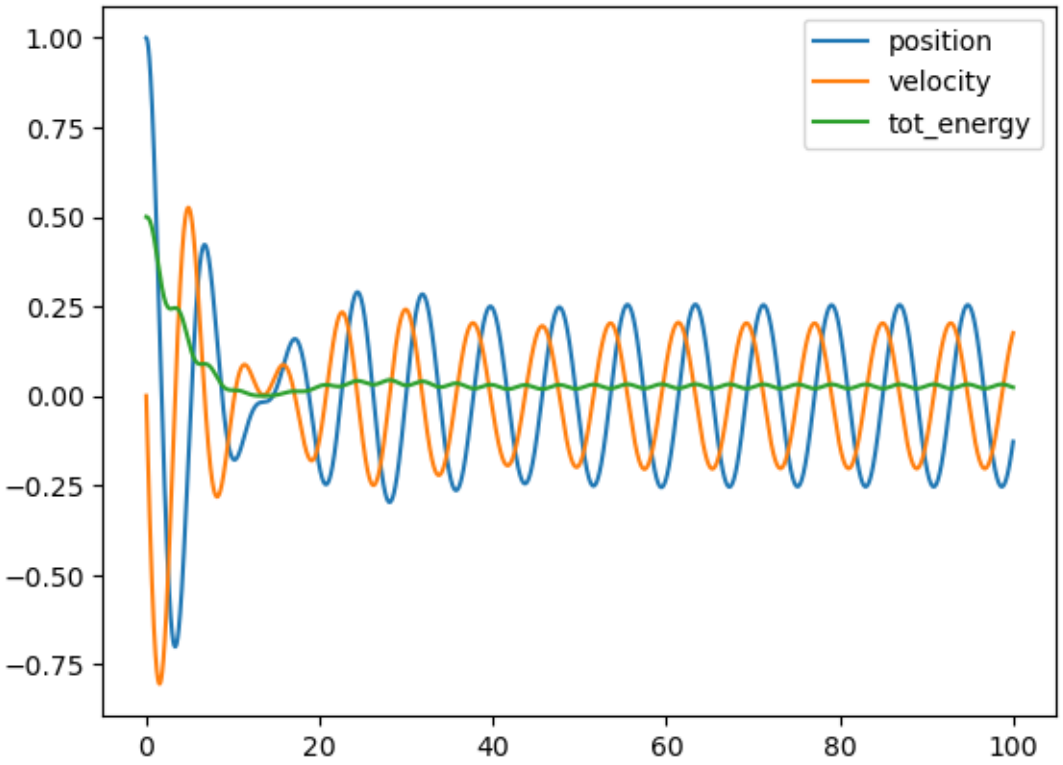
```
gamma_underdamped = 0.2
run_simulation(gamma_underdamped, 'Underdamped (  = 0.2)')

# Critical damping case (lambda = 2, omega_0^2 = gamma^2)
gamma_critical = 2
run_simulation(gamma_critical, 'Critical damping (  = 2)', linestyle='--')

# Overdamping case (lambda = 2.4, omega_0^2 < gamma^2)
gamma_overdamped = 2.4
run_simulation(gamma_overdamped, 'Overdamped (  = 2.4)', linestyle=':')

# Plot settings
plt.plot(t_eval, np.exp(-0.1*t_eval), label="A exp(-0.1 t)")
plt.title('Damped Oscillator: Position vs Time for Different Damping␣
 ↪Conditions')
plt.xlabel('Time [s]')
plt.ylabel('Position [m]')
plt.legend()
plt.grid(True)
plt.show()
```



3 Forced Oscillation

- Use the IVP solver we developed. The only differences are the `func` and initial conditions.
- Modify the y' function (`func`) to simulate a forced oscillator (from t=0 to 100).
- Set the inital conditions: A=1, K=M=1, $\lambda = 0.2$, $F_0 = 0.1$ and $\omega_f = 0.8$.
- Make plots of position(t), velocity(t), and total energy(t).

[8]:
```
# TODO

F0 = 0.1      # Amplitude of the external force
omega_f = 0.8 # Forcing frequency
# Modified function with forcing term
def func(t, y, K, M, damping, F0, omega_f):
    f = np.zeros(len(y))
    f[0] = y[1]  # dx/dt = v
    f[1] = -K/M * y[0] - damping/M * y[1] + F0/M * np.cos(omega_f * t)  # dv/dt␣
 ↪= -K/M * x - damping term + forcing term
    return f

t_eval = np.arange(0, 100, 0.005)
damping = 0.2
sol = mysolver.solve_ivp(func, t_eval, y, "RK4", t_eval, args=(K, M, damping,␣
 ↪F0, omega_f))

def total_energy(x, v, K, M):
    kinetic_energy = 0.5 * M * v**2
    potential_energy = 0.5 * K * x**2
    return kinetic_energy + potential_energy

energy = total_energy(sol[0], sol[1], K, M)
plt.plot(t_eval, sol[0], label="position")
plt.plot(t_eval, sol[1], label="velocity")
plt.plot(t_eval, energy, label="tot_energy")
plt.legend()
plt.show()
```

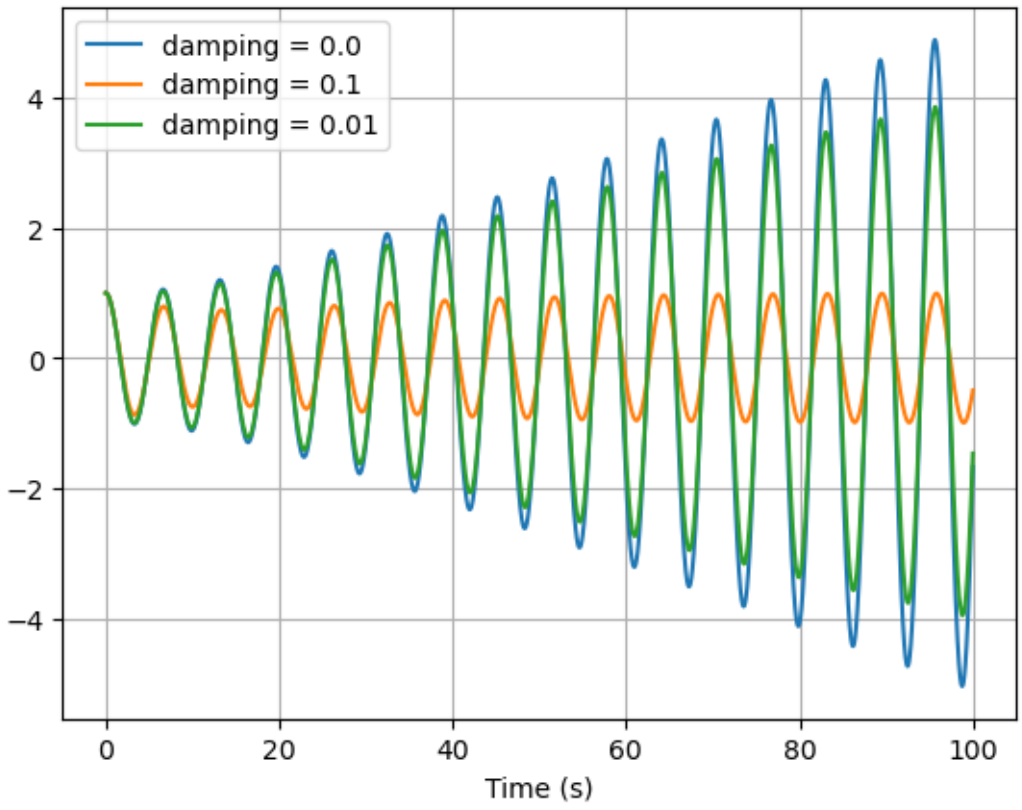


4 Resonance

- Resonance will happen when $\omega_0 = \omega_f$ without damping.
- Modify your `demo3` but set $\lambda = 0$ and $\omega_f = 1$.
- Re-run your simulation with $\lambda = 0.1$ and 0.01.

[9]:
```
# TODO
# parameters
t_span = np.arange(0,100, 0.005)
t_eval = t_span
y = [1,0] # initial condition
K = 1
M = 1
dampomg = 0.0
F0 =0.1
omega_f = 1.0
args = (K,M,damping,F0,omega_f)

# use IVP solver to solve the problem
sol1 = mysolver.solve_ivp(func,t_span,y, "RK4", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
```

```
sol2 = mysolver.solve_ivp(func,t_span,y, "RK4", t_eval, args=(K,M,0.
↪1,F0,omega_f))
sol3 = mysolver.solve_ivp(func,t_span,y, "RK4", t_eval, args=(K,M,0.
↪01,F0,omega_f))

# visualize the results
plt.plot(t_eval, sol1[0], label='damping = 0.0')
plt.plot(t_eval, sol2[0], label='damping = 0.1')
plt.plot(t_eval, sol3[0], label='damping = 0.01')
plt.xlabel('Time (s)')
plt.legend()
plt.grid(True)
plt.show()
```

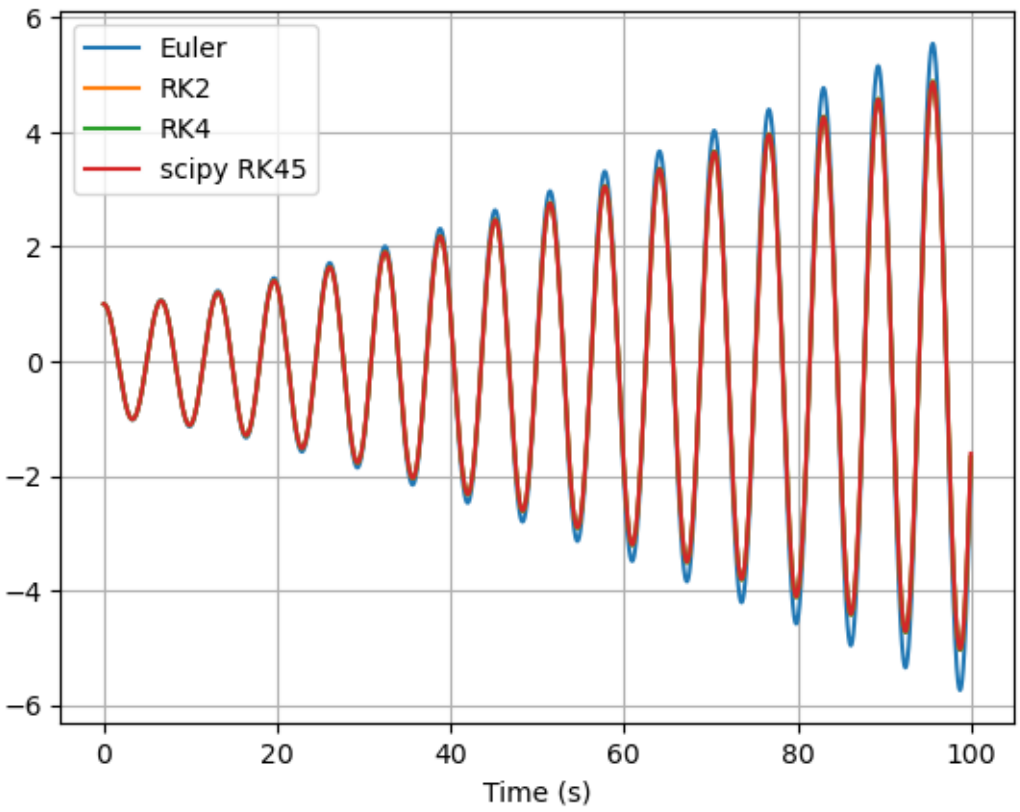


5 Using `scipy`

Compare the results of our solver (`mysolver`) with the `solve_ivp` in `scipy`.

[10]:
```
# TODO
```

```
sol_euler = mysolver.solve_ivp(func,t_span,y, "Euler", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
sol_rk2 = mysolver.solve_ivp(func,t_span,y, "RK2", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
sol_rk4 = mysolver.solve_ivp(func,t_span,y, "RK4", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
sol_scipy = solver_scipy(func, [0,100], y, args=(K,M,0.0,F0,omega_f),␣
 ↪t_eval=t_eval, method='RK45')

# visualize the results
plt.plot(t_eval, sol_euler[0], label="Euler")
plt.plot(t_eval, sol_rk2[0], label="RK2")
plt.plot(t_eval, sol_rk4[0], label="RK4")
plt.plot(sol_scipy.t, sol_scipy.y[0], label="scipy RK45")
plt.xlabel('Time (s)')
plt.grid(True)
plt.legend()
plt.show()
```



6 Performance

We could mesure the performance of our solver and compare it with scipy.

[11]:
```
#TODO

%timeit mysolver.solve_ivp(func,t_span,y, "RK4", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
%timeit mysolver.solve_ivp(func,t_span,y, "RK2", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
%timeit mysolver.solve_ivp(func,t_span,y, "Euler", t_eval, args=(K,M,0.
 ↪0,F0,omega_f))
%timeit solver_scipy(func, [0,100], y, args=(K,M,0.0,F0,omega_f),␣
 ↪t_eval=t_eval, method='RK45')
```

```
265 ms ± 12.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
123 ms ± 3.05 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
58.6 ms ± 521  s per loop (mean ± std. dev. of 7 runs, 10 loops each)
6.5 ms ± 248  s per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

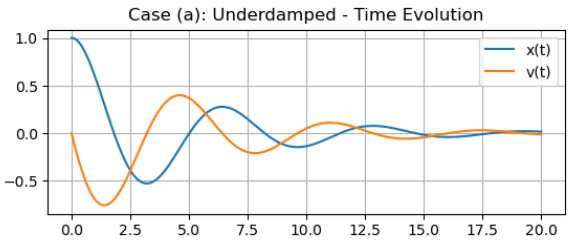
Which one is faster?

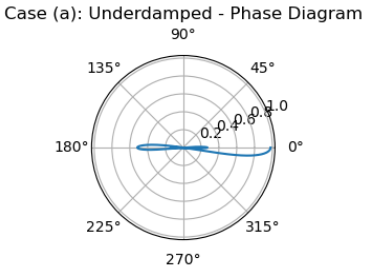
7 Qestion2, 3

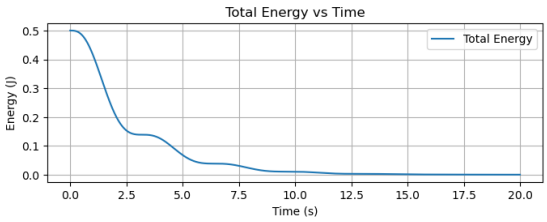
[12]:
```
# Define the damped oscillator differential equation
def damped_oscillator(t, y, omega0, gamma):
    f = np.zeros(2)
    f[0] = y[1]
    f[1] = -2 * gamma * y[1] - omega0**2 * y[0]
    return f

# Function to calculate total energy and energy loss rate
def calculate_energy(x, v, M, K, gamma):
    E = 0.5 * M * v**2 + 0.5 * K * x**2
    dE_dt = -gamma * v**2
    return E, dE_dt

# Function to simulate and plot results
def simulate_and_plot(A, omega0, gamma, phi, label):
    # Initial conditions
    x0 = A * np.cos(phi)
    v0 = -A * omega0 * np.sin(phi)
    y0 = [x0, v0]

    # Time parameters
    t0, tmax, dt = 0, 20, 0.01
    t_eval = np.arange(t0, tmax, dt)
```

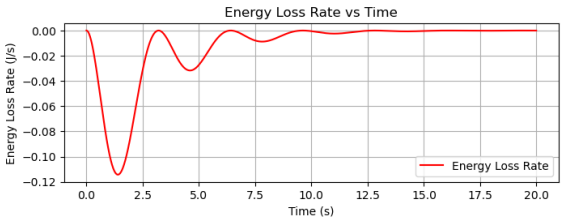
```
    # Solve using the IVP solver
    sol = mysolver.solve_ivp(damped_oscillator, t_eval, y0, "RK4", t_eval,␣
↪args=(omega0, gamma))
    x = sol[0, :] # []
    v = sol[1, :]

    # Calculate energy and energy loss rate
    E, dE_dt = calculate_energy(x, v, M, K, gamma)

    # Calculate u and w
    omega1 = np.sqrt(np.abs(omega0**2 - gamma**2))
    u = omega1 * x
    w = gamma * v * x

    # Plot x(t) and v(t)
    plt.figure(figsize=(14, 5))
    plt.subplot(2, 2, 1)
    plt.plot(t_eval, x, label='x(t)')
    plt.plot(t_eval, v, label='v(t)')
    plt.title(f'{label} - Time Evolution')
    plt.legend()
    plt.grid()

    # Plot phase diagram in polar coordinates
    plt.subplot(2, 2, 2, polar=True)
    r = np.sqrt(u**2 + w**2)
    theta = np.arctan2(w, u)
    plt.plot(theta, r)
    plt.title(f'{label} - Phase Diagram')

    # Plot total energy versus time
    plt.figure(figsize=(14, 5))
    plt.subplot(2, 2, 3)
    plt.plot(t_eval, E, label='Total Energy')
    plt.title('Total Energy vs Time')
    plt.xlabel('Time (s)')
    plt.ylabel('Energy (J)')
    plt.legend()
    plt.grid()

    # Plot energy loss rate versus time
    plt.subplot(2, 2, 4)
    plt.plot(t_eval, dE_dt, label='Energy Loss Rate', color='red')
    plt.title('Energy Loss Rate vs Time')
    plt.xlabel('Time (s)')
    plt.ylabel('Energy Loss Rate (J/s)')
    plt.legend()
```

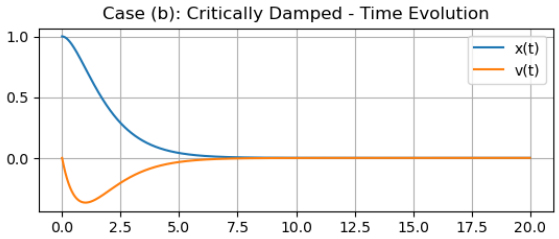
```
    plt.grid()


    plt.tight_layout()
    plt.show()

# Simulate and plot for the three cases
simulate_and_plot(A=1, omega0=1, gamma=0.2, phi=0, label='Case (a): Underdamped')
simulate_and_plot(A=1, omega0=1, gamma=1.0, phi=0, label='Case (b): Critically Damped')
simulate_and_plot(A=1, omega0=1, gamma=1.2, phi=0, label='Case (c): Overdamped')
```

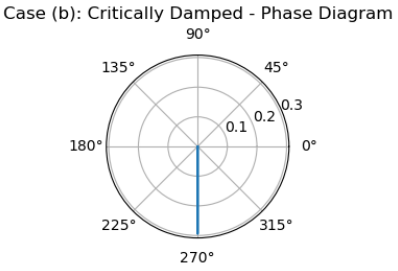


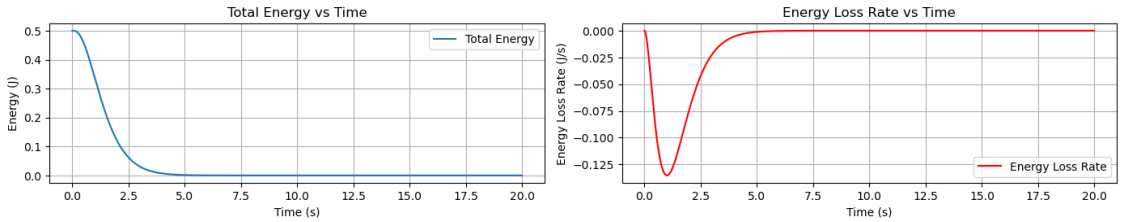


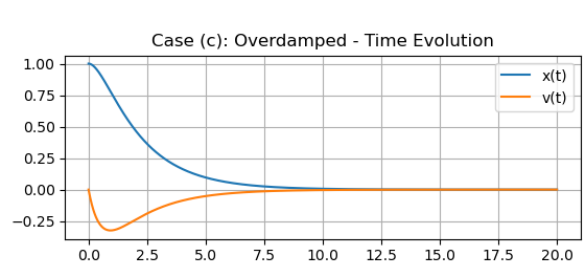


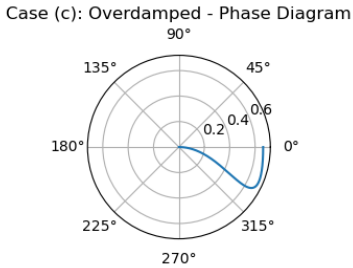


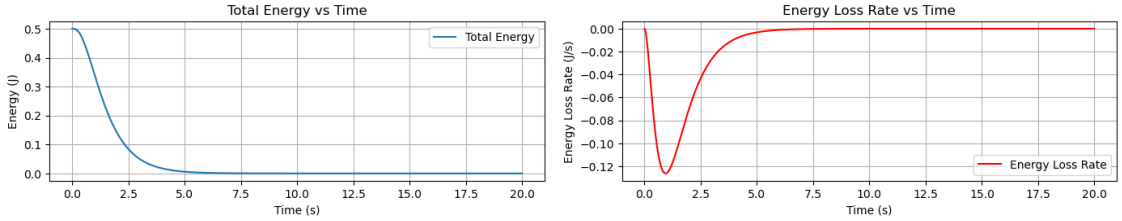












8 Qestion 4.

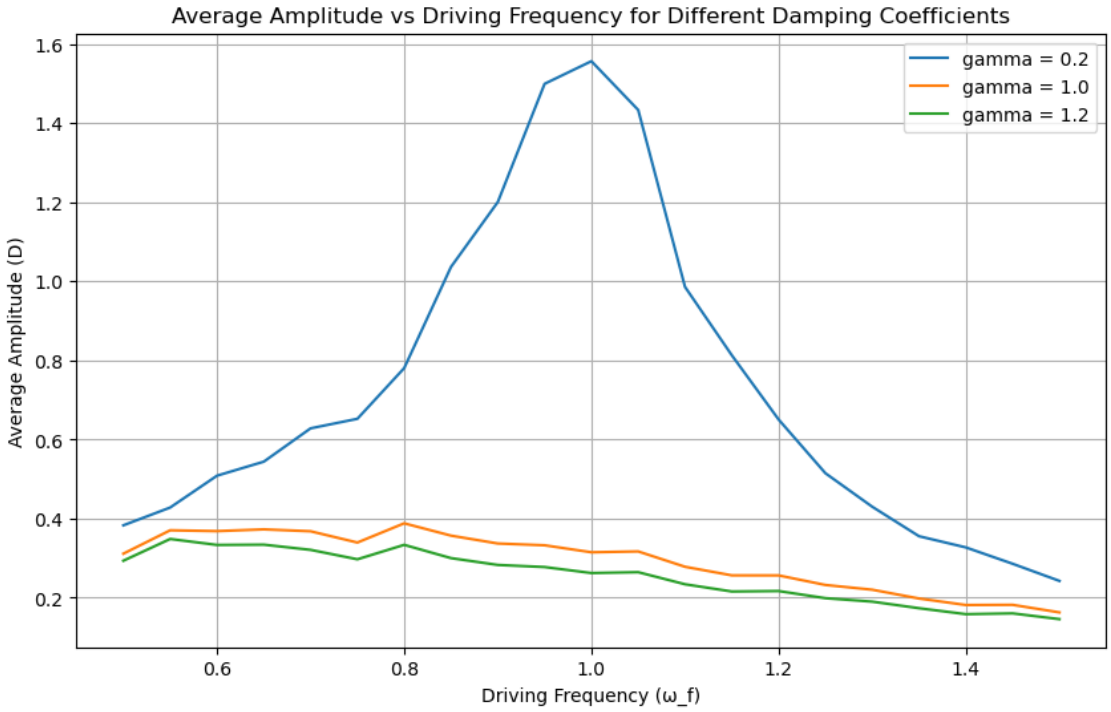
8.1 Resonance:

We can observe a peak in the average amplitude at a specific frequency (around the natural frequency of the system) in this case for an undamped system. ## Analytical Comparison: The peak resonance frequency should match the theoretical value $\sqrt{(K/M)}$ which is 1.0 in this case for an undamped system.

[13]:
```
# Constants
M = 1.0
K = 1.0
F0 = 0.5
t_eval = np.arange(0, 50, 0.005)
gamma_values = [0.2, 1.0, 1.2]
omega_f_range = np.arange(0.5, 1.55, 0.05)
```

```
# Modified function with forcing term
def func(t, y, K, M, damping, F0, omega_f):
    f = np.zeros(len(y))
    f[0] = y[1]  # dx/dt = v
    f[1] = -K/M * y[0] - damping/M * y[1] + F0/M * np.cos(omega_f * t)  # a =␣
 ↪-K/M * x - damping term + forcing term
    return f

# Total energy function
def total_energy(x, v, K, M):
    return 0.5 * M * v**2 + 0.5 * K * x**2

# Function to calculate average amplitude between t=40 and t=50
def average_amplitude(x, t):
    x_in_range = x[(t > 40) & (t < 50)]
    return np.mean(np.abs(x_in_range))

# Initialize an empty dictionary to store results for each gamma
results = {}

# Loop over each gamma value
for gamma in gamma_values:
    avg_amplitudes = []

    for omega_f in omega_f_range:
        y0 = [0,0]  # Initial conditions: x=0, v=0

        # Solve the system with the current damping and driving frequency
        sol = mysolver.solve_ivp(func, t_eval, y0, "RK4", t_eval, args=(K, M,␣
 ↪gamma, F0, omega_f))
        position = sol[0]

        # Calculate average amplitude between t=40 and t=50
        avg_amp = average_amplitude(position, t_eval)
        avg_amplitudes.append(avg_amp)

    # Store the average amplitude result for this gamma value
    results[gamma] = avg_amplitudes


plt.figure(figsize=(10, 6))
# Plot the average amplitude vs omega_f for each damping value
for gamma, avg_amplitudes in results.items():
    plt.plot(omega_f_range, avg_amplitudes, label=f"gamma = {gamma}")

plt.xlabel("Driving Frequency ( _f)")
plt.ylabel("Average Amplitude (D)")
```

```
plt.title("Average Amplitude vs Driving Frequency for Different Damping␣
 ↪Coefficients")
plt.legend()
plt.grid(True)
plt.show()
```

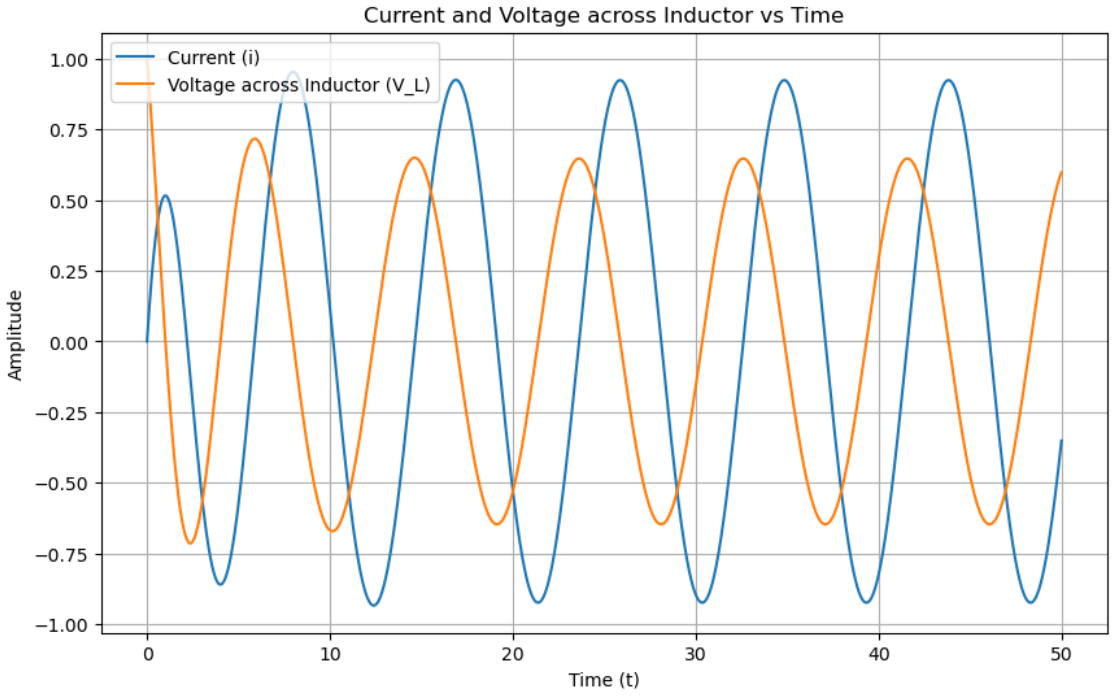


9 Question 5

(b) Use the same IVP solver we developed in the class to numerically solve the system with initial conditions: $L, C, E_0 = 1$ and $R = 0.8$. Make plots of the current and the voltage across the inductor as functions of time.

[14]:
```
# Constants
R = 0.8
L = 1.0
C = 1.0
E0 = 1.0
omega = 0.7
t_eval = np.arange(0, 50, 0.01)

# describing the RLC circuit
def rlc_system(t, y, R, L, C, E0, omega):
    f = np.zeros(2)
```

```
    f[0] = y[1]  # dq/dt = i
    f[1] = (E0 * np.cos(omega * t) - R * y[1] - y[0] / C) / L  # di/dt
    return f

# initial conditions
y0 = [0, 0]

# using the IVP solver
sol = mysolver.solve_ivp(rlc_system, t_eval, y0, "RK4", t_eval, args=(R, L, C,␣
 ↪E0, omega))
charge = sol[0]
current = sol[1]

# plot the results: current and inductor voltage
voltage_L = L * np.gradient(current, t_eval)  # V_L = L * di/dt
plt.figure(figsize=(10, 6))
plt.plot(t_eval, current, label="Current (i)")
plt.plot(t_eval, voltage_L, label="Voltage across Inductor (V_L)")
plt.xlabel("Time (t)")
plt.ylabel("Amplitude")
plt.title("Current and Voltage across Inductor vs Time")
plt.legend()
plt.grid(True)
plt.show()
```



(c) Redo the problem by varying from 0.3 to 1.5 with an interval 0.1. Do you see any special ? What are the meaning of these frequencies?

9.0.1 Result Analysis

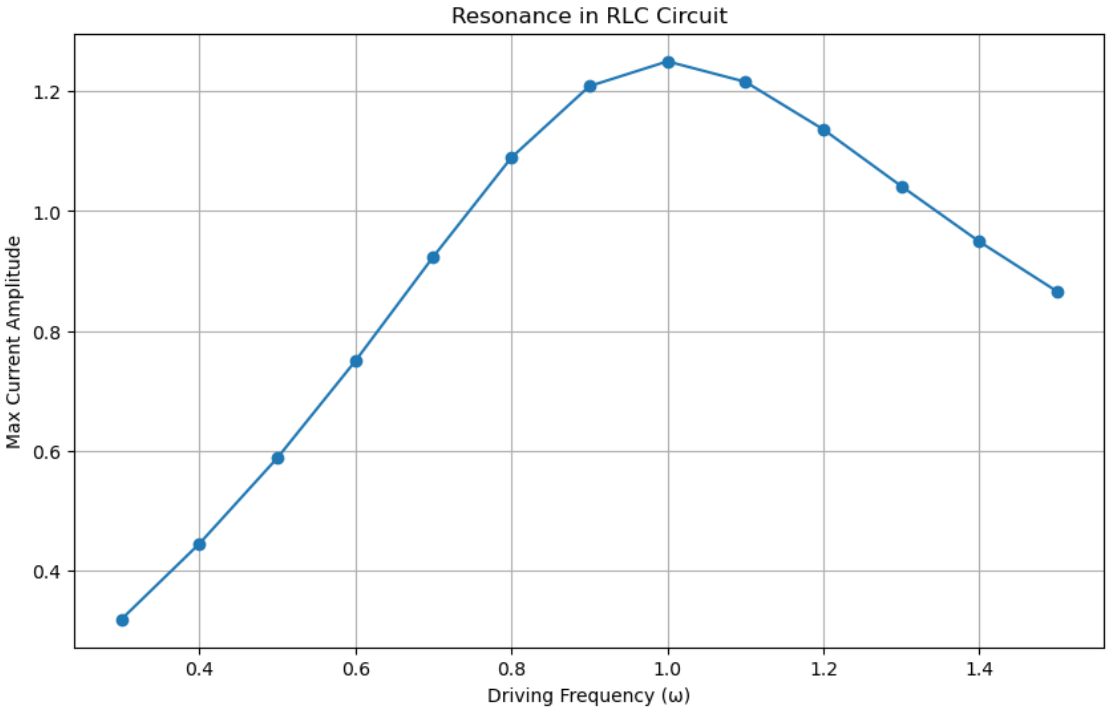
The maximum amplitude of the current varies with different driving frequencies $ $. If the amplitude reaches its peak at a particular frequency, that frequency is the resonance frequency of the system. In an RLC circuit, the theoretical resonance frequency $ _r $ is given by:

$\omega_r = \frac{1}{\sqrt{LC}}$

At this frequency, the amplitude of the current reaches its maximum value, which is known as the resonance phenomenon.

Through this simulation, we can observe a significant increase in current amplitude as the driving frequency approaches the system's resonance frequency.

[15]:
```
# drive frequency range
omega_range = np.arange(0.3, 1.55, 0.1)
amplitudes = []

# loop over each drive frequency
for omega in omega_range:
    sol = mysolver.solve_ivp(rlc_system, t_eval, y0, "RK4", t_eval, args=(R, L,␣
 ↪C, E0, omega))
    current = sol[1]

    # compute the max current amplitude in the steady-state part (t > 40)
    steady_state_mask = (t_eval > 40)
    max_amplitude = np.max(np.abs(current[steady_state_mask]))
    amplitudes.append(max_amplitude)

# plot current amplitude vs driving frequency
plt.figure(figsize=(10, 6))
plt.plot(omega_range, amplitudes, marker='o')
plt.xlabel("Driving Frequency ( )")
plt.ylabel("Max Current Amplitude")
plt.title("Resonance in RLC Circuit")
plt.grid(True)
plt.show()
```


Resonance in RLC Circuit
Max Current Amplitude
Driving Frequency (ω)
0.4
0.6
0.8
1.0
1.2
1.4