Computational Physics Lab

Homework 2

108000204
Yuan-Yen Peng
Dept. of Physics, NTHU
Hsinchu, Taiwan

November 13, 2022

1 Written Assignments

The general second-order ordinary differential equations(x depend on time) is

$$\ddot{x} + b\dot{x} + cx = g(x)$$

where b, c is the constant, and $g(x)$ is the particular term. We can set the general solution as $x = x_c + x_p$, where x_c is the critical solution; x_p is the particular solution.

1.1 (a) Normal oscillations

$$\begin{aligned}
&solve \quad m\ddot{x} + kx = 0 \\
&\rightarrow \ddot{x} + \frac{k}{m}x = 0 \\
&set \quad x_c = e^{\alpha t - \phi}, \quad where\ \phi\ is\ a\ phase,\ depending\ on\ init.\ conds. \\
&\Rightarrow \alpha^2 e^{\alpha t - \phi} + \frac{k}{m}e^{\alpha t - \phi} = 0, \quad \alpha = \pm i\sqrt{\frac{k}{m}}, \quad set\ \omega = \sqrt{\frac{k}{m}} \\
&x = \mathbb{R}(x_c) = \cos(\omega t - \phi) \quad \blacksquare
\end{aligned}$$

1.2 (b) Damped oscillations

$$\begin{aligned}
&solve \quad m\ddot{x} + \lambda\dot{x} + kx = 0 \\
&\rightarrow \ddot{x} + \frac{\lambda}{m}\dot{x} + \frac{k}{m}x = 0 \\
&set \quad x_c = e^{\alpha t - \phi}, \quad where\ \phi\ is\ a\ phase,\ depending\ on\ init.\ conds. \\
&\Rightarrow \alpha^2 e^{\alpha t - \phi} + \alpha\frac{\lambda}{m}e^{\alpha t - \phi} + \frac{k}{m}e^{\alpha t - \phi} = 0, \\
&set \quad \omega_0 = \sqrt{\frac{k}{m}}\ and \quad 2\gamma = \frac{\lambda}{m} \\
&\Rightarrow \alpha^2 + 2\gamma\alpha + \omega_0^2 = 0, \quad \alpha_{1,\,2} = -\gamma \pm \sqrt{\gamma^2 - \omega_0^2} \\
&x = x_c = A_1 e^{\alpha_1 t} + A_2 e^{\alpha_2 t},\ where\ A_i\ depends\ on\ initi.\ conds. \quad \blacksquare
\end{aligned}$$

1.3 (c) Forced oscillations

The critical solution of (c), inhomogeneous ODE, is the same as the answer of (b) Hence, we only discuss the particular solution x_p, which means a stable oscillated solution. The ultimate answer will be $x = x_c + x_p$.

$$
\begin{aligned}
&solve \quad m\ddot{x} + \lambda\dot{x} + kx = F_0\cos(\omega_f t)\\
&\rightarrow \ddot{x} + \frac{\lambda}{m}\dot{x} + \frac{k}{m}x = \frac{F_0}{m}\cos(\omega_f t)\\
&set \quad x_p = Ae^{i(\alpha t)}\\
&\Rightarrow -\alpha^2 Ae^{i(\alpha t)} + i\alpha\frac{\lambda}{m}Ae^{i(\alpha t)} + \frac{k}{m}Ae^{i(\alpha t)} = \frac{F_0}{m}\cos(\omega_f t)\\
&set \quad \omega_0 = \sqrt{\frac{k}{m}}\ and \quad 2\gamma = \frac{\lambda}{m}\\
&at\ t = 0\ \wedge use\ the\ stable\ cond. \rightarrow \alpha = \omega_f\\
&\Rightarrow A(-\omega_f^2 + 2i\gamma\omega_f + \omega_0^2) = \frac{F_0}{m}, \quad A = \frac{F_0/m}{(\omega_0^2 + 2i\gamma\omega_f - \omega_f^2)}\\
&x_p = \mathbb{R}(Ae^{i(\alpha t)}) = \mathbb{R}(\frac{F_0}{m}\frac{(\omega_0^2 - \omega_f^2 - 2i\gamma\omega_f)e^{i\alpha t}}{(\omega_0^2 - \omega_f^2)^2 + 4\gamma^2\omega_f^2}) = \frac{F_0/m}{\sqrt{(\omega_0^2 - \omega_f)^2 + 4\gamma^2\omega_f^2}}\cos(\omega_f t - \phi),\\
&\quad where\ \phi = tan^{-1}(\frac{2\gamma\omega_f}{\omega_f^2 - \omega_0^2})\\
&\Rightarrow\ x = x_c + x_p \quad \blacksquare
\end{aligned}
$$

2 Programming Assignments

2.1 Damped oscillations

In this section, we reuse our IVP solver and perform simulations of a damped oscillator (the definition follows the lecture). Make a plot for x and v versus t and a phase diagram (in polar coordinates, u and w)

$$u = \sqrt{\omega_0^2 - \gamma^2}x$$
$$w = \gamma x + \dot{x}$$

Additionally, in the class, we have shown that RK4 is the most fitted for the theoretical solution, so we will utilize this as a theoretical reference.

2.1.1 (a) Under damping

The conditions are

$$A = 1[cm], \quad \omega_0 = 1[rads^{-1}], \quad \gamma = 0.2[s^{-1}], \quad \phi = -\pi/2[rad]$$

Our results, Figure 1 meet our expectations, under damping ($\sqrt{\gamma^2 - \omega_0^2} \in \mathbb{R}, \quad \gamma^2 - \omega_0^2 > 0$), with the exponential decay envelope and the trigonometric oscillating solutions of position and velocity. Meanwhile, the polar coordinate will show a spiral, not a circle, due to the dampness and it will evolve from the outer edge to the center where the amplitude is zero. In Figure 2, RK2 behaves more fitted with RK4 rather than the Euler.

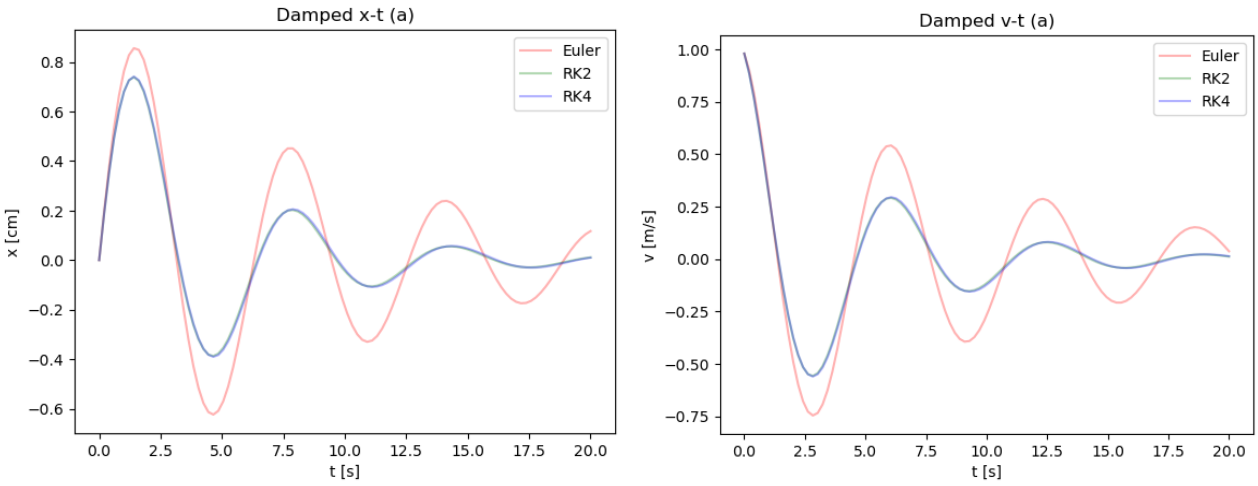

Figure 1: Left is the x-t figure, and right is the v-t.

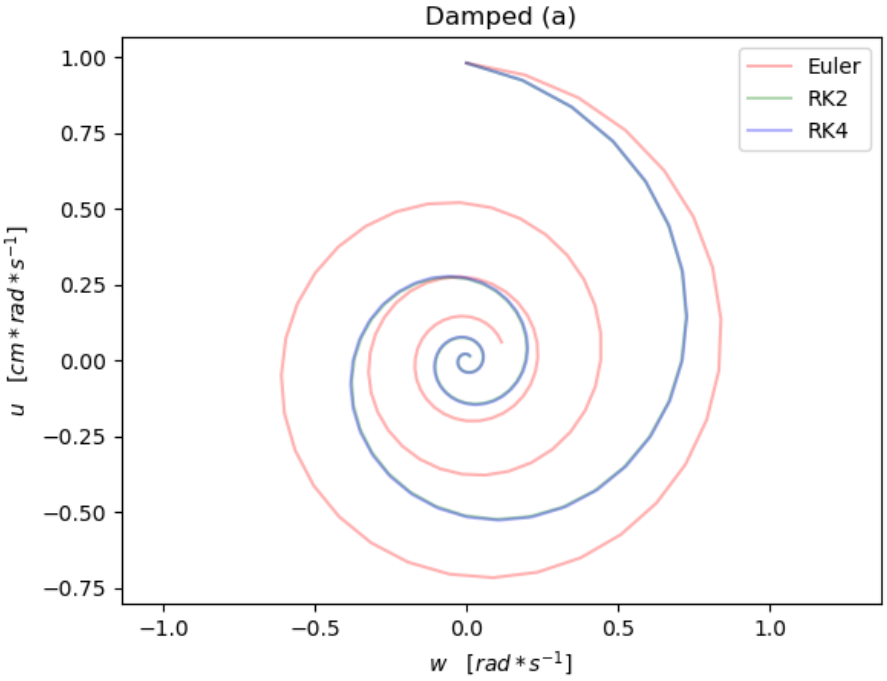

Figure 2: y-axis is u; x-axis is w.

2.1.2 (b) Critical damping

The conditions are

$$A = 1[cm], \quad \omega_0 = 1[rads^{-1}], \quad \gamma = 1.0[s^{-1}], \quad \phi = -\pi/2[rad]$$

Our results, this case, $\gamma^2 = \omega_0^2$, $\sqrt{\gamma^2 - \omega_0^2} = 0$, means "critical damped". So as to observe the critical damped figure, we set a very small amplitude instead of 0, and in Figure 3, they meet our expectations. Meanwhile, the polar coordinate will show a straight line, not a spiral, due to the critical condition and it will evolve from the top to the bottom where the amplitude is always zero (indicates that $w = 0$), see in Figure 2.

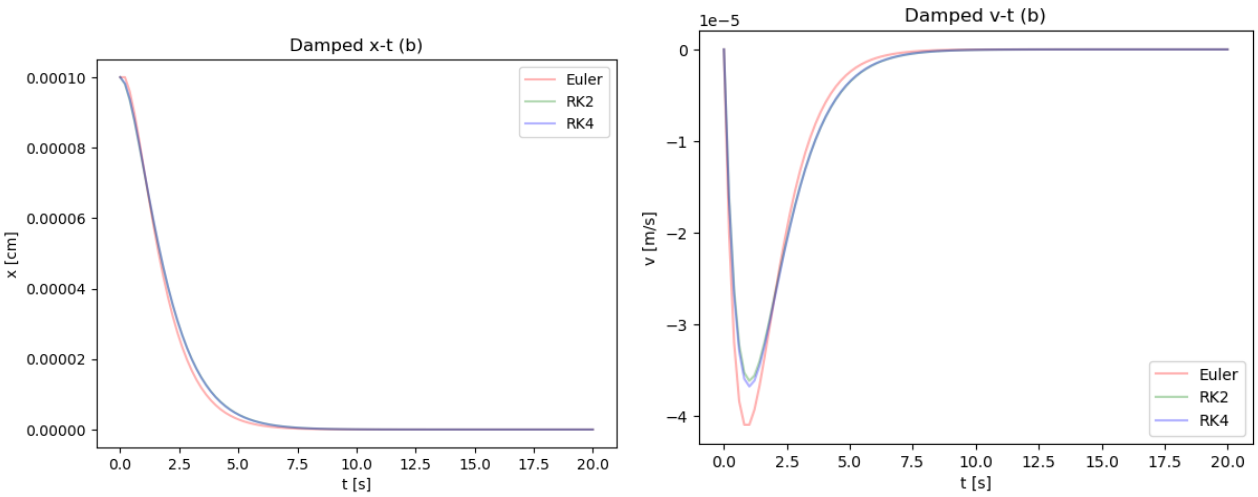

Figure 3: Left is the x-t figure, and right is the v-t.

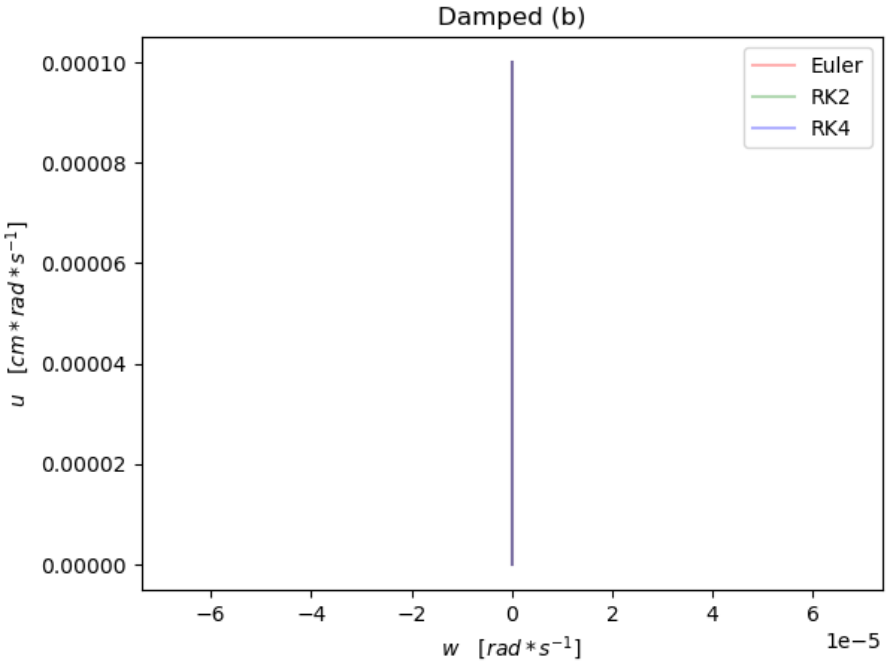

Figure 4: y-axis is u; x-axis is w.

2.1.3 (c) Over damping

The conditions are

$$A = 1[cm], \quad \omega_0 = 1[rads^{-1}], \quad \gamma = 1.2[s^{-1}], \quad \phi = -\pi/2[rad]$$

Our results, this case, $\gamma^2 - \omega_0^2 < 0$, means "overdamped".Theoretically, it cannot form a complete period; also with the exponential decay, and in Figure 5, they meet our expectations. Meanwhile, the polar coordinate will show a curve line, not a spiral, due to the overdamped condition and it will evolve from the top to the bottom where the amplitude is positive (indicates that $w > 0$), see in Figure 6.

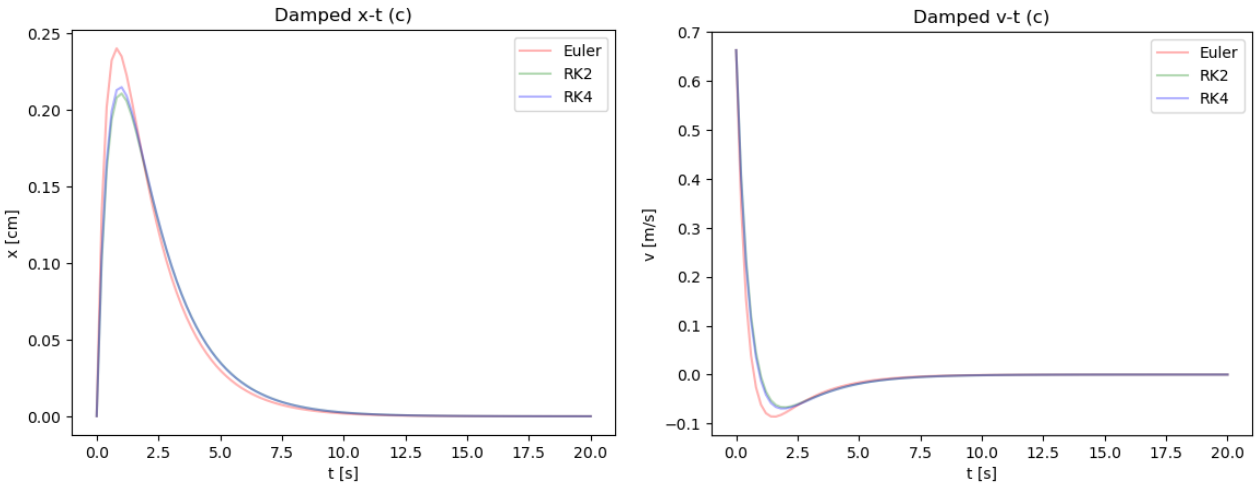

Figure 5: Left is the x-t figure, and right is the v-t.

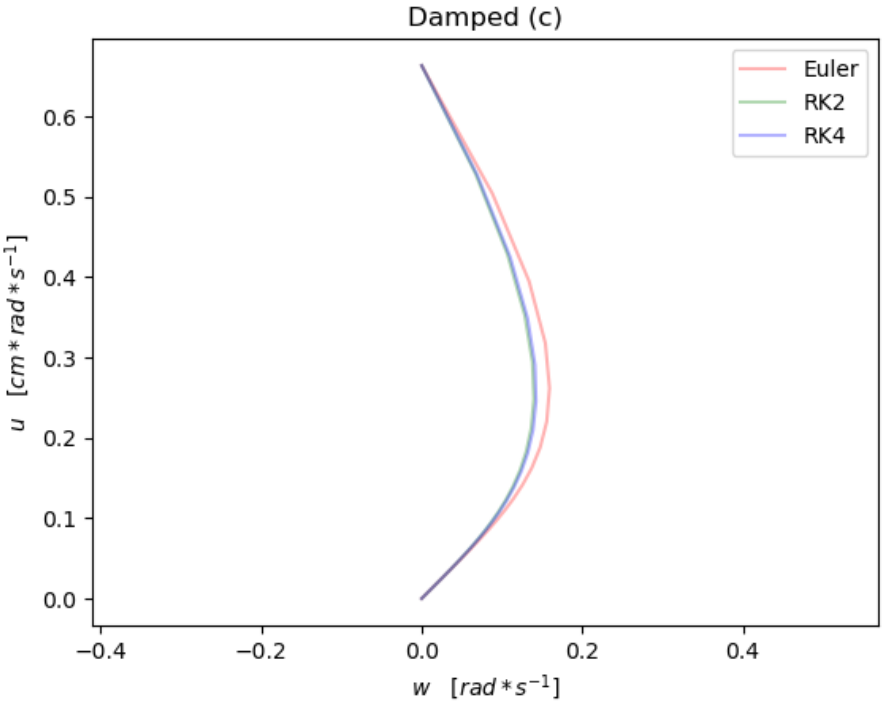

Figure 6: y-axis is u; x-axis is w.

2.2 Energy of sec. 2.1.1

In this section, we make plots of the total energy and energy loss rate versus time for the underdamped oscillator (a). In Figure 7, we can see that the energy is not conserved owing to the dampness; likewise, the loss rate is quick in the beginning and slow in the end since it bases on the velocity, which is correlated to the dampness. On the other hand, we plot the energy difference rate in Figure 8 as well, showing that the loss rate is not merely oscillating, but also decays along with time.

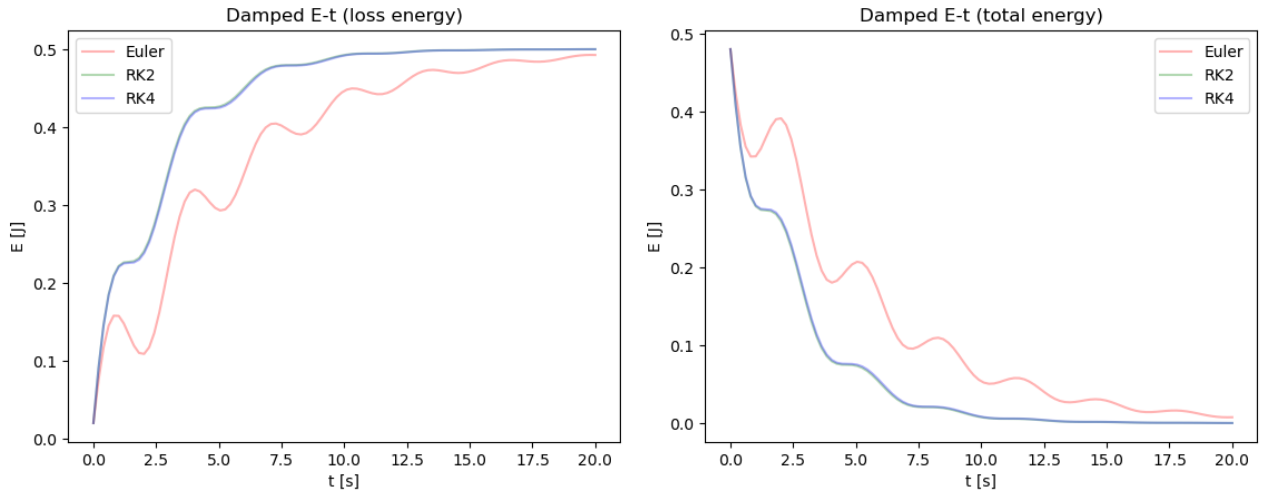

Figure 7: Left is the loss energy figure compared with the no-damped model, and right is the total energy depiction.

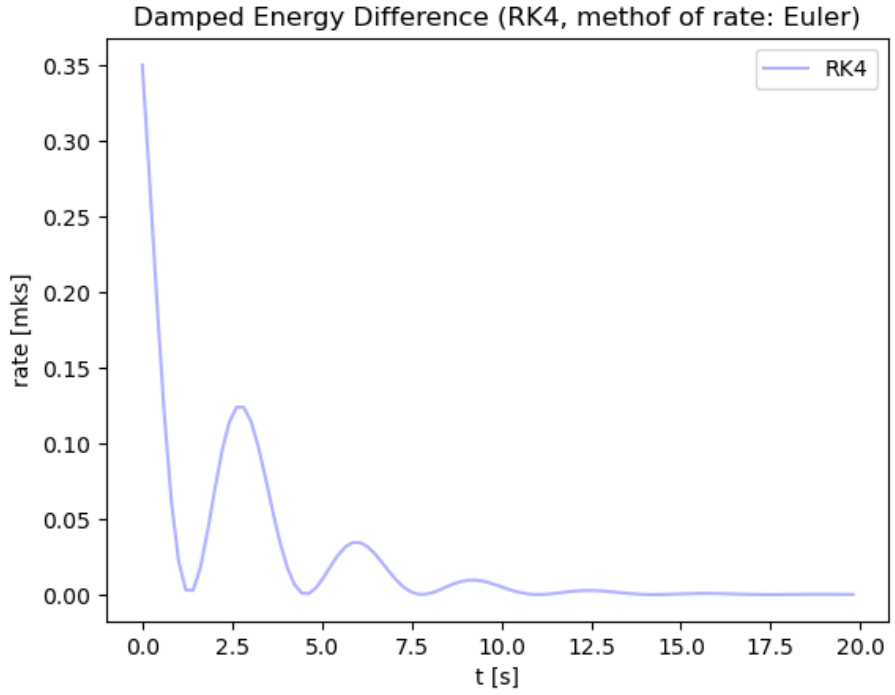

Figure 8: This is the energy loss rate, utilizing RK4 sample and calculating the rate with Euler method.

2.3 Resonance of forced oscillating

In this section, we add a sinusoidal driving force ($F = F_0 \cos(\omega_f t)$) in the damped oscillator with $F_0 = 0.5$. Vary ω_f from 0.5 to 1.5 with an interval of 0.01 (the question is 0.05, but I think it is too wide). Here, we rerun your simulation up to $t_{max} = 50$, measuring the average amplitude of oscillator (define $D =< |x(t)| >$ between $40 < t < 50$). Appling $\lambda = 0.01,\ 0.1,\ 0.3$, we plot these figures in Figure 9. The resonance appears on ω_f 1, and the smaller λ the more obvious maximum peak it has. Theoretically, according to sec. 1.3, the amplitude of forced oscillation is

$$\frac{F_0/m}{\sqrt{(\omega_0^2 - \omega_f^2)^2 + 4\gamma^2\omega_f^2}} = \frac{F_0/m}{\sqrt{(\omega_f^2 - \omega_0^2 + 2\gamma^2)^2 + 4\gamma^2(\omega_0^2 - \gamma^2)}}$$

Thence, when $\omega_f = \sqrt{\omega_0^2 - 2\gamma^2} (= \sqrt{\omega_0^2 - \lambda^2},\ m = 1)$, the maximum amplitude is

$$\frac{F_0/m}{2\gamma\sqrt{\omega_0^2 - \gamma^2}}$$

On account of $2\gamma = \lambda/m = \lambda$, and $\omega_0 = 1$, the maximum (peak) will appear around 1; as λ small, it will be more manifest. As a result, compared to the theoretical outcomes, the consequences of the simulation make sense.

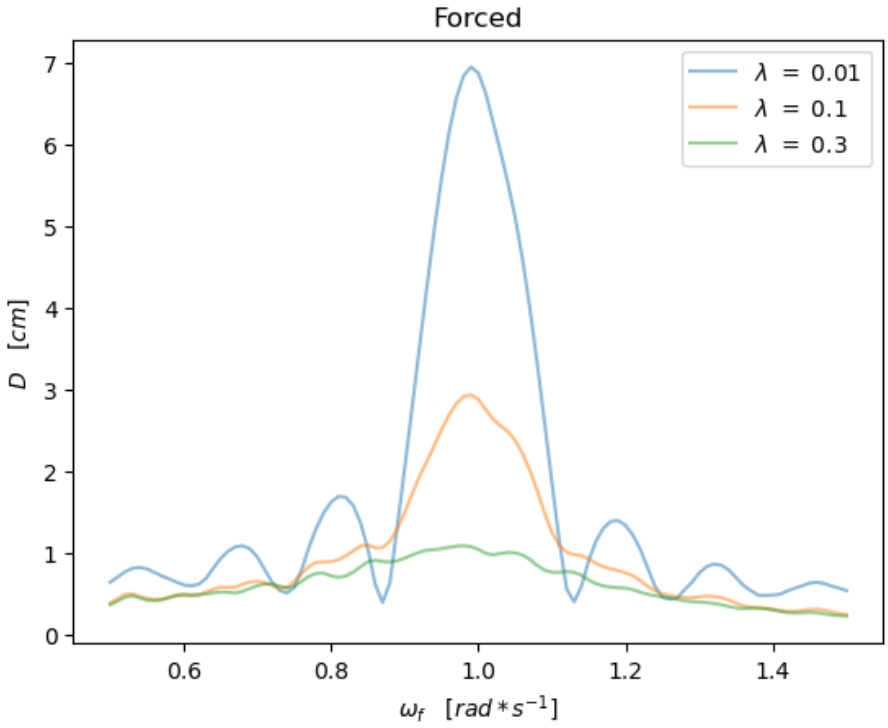

Figure 9: This is the $D =< |x(t)| >$ vs ω_f between $40 < t < 50$.
$\lambda = 0.01$, the resonance frequency is on $\omega_f = 0.99$, and the maximum average amplitude is $D = 6.94$.
$\lambda = 0.1$, the resonance frequency is on $\omega_f = 0.99$, and the maximum average amplitude is $D = 2.94$.
$\lambda = 0.3$, the resonance frequency is on $\omega_f = 0.98$, and the maximum average amplitude is $D = 1.09$.

2.4 RLC circuit systems.

2.4.1 (a) RLC ODE

We can apply Kirchoff's rule, which means the current will be conserved or "the potential(voltage) conservation".

$$V_L = L\ddot{q}, \quad V_R = R\dot{q}, \quad V_C = q/C$$

and also need to consider the source term, combining that together. We, afterward, can get

$$L\ddot{q} + R\dot{q} + \frac{q}{C} = E_0 \sin(\omega t) \quad \blacksquare$$

Additionally, we can analog RLC system to the forced oscillating system.

$$x \leftrightarrow q, \quad \dot{x} \leftrightarrow I, \quad m \leftrightarrow L,$$
$$\lambda \leftrightarrow R, \quad 1/k \leftrightarrow C, \quad F \leftrightarrow E_0$$

2.4.2 (b) V-t and I-t

Here, we use the conditions

$$L = C = E_0 = 1, \quad R = 0.8, \quad \omega = 0.7$$

Implementing `myslover.py` with RK4, we can get the outcomes, as below figures.

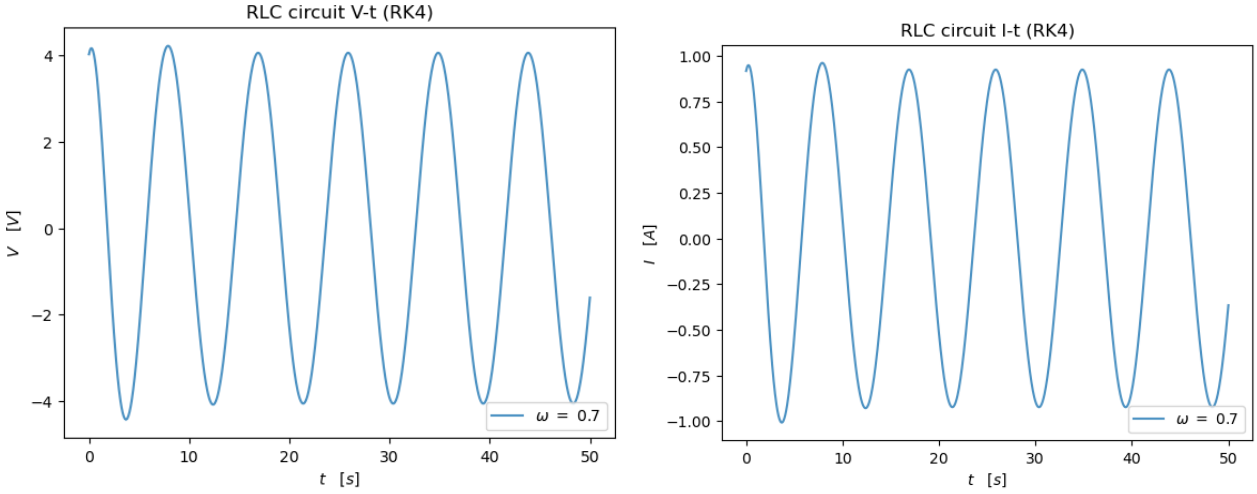

Figure 10: Left is the V-t figure, and right is the I-t.

2.4.3 (c) Different ω

Before utilizing the numerical method, we can calculate theoretical ωs. First is resonance frequency ω_R, and another is maximum amplitude frequency ω_M, the third is the damped frequency, ω_d. (The Right approximation corresponds to our varying ω)

$$\begin{aligned} \omega_R &= \sqrt{\omega_0^2 - 2\gamma^2} \approx 0.8 \\ \omega_M &= \frac{1}{LC - R^2C^2/2} \approx 1.2 \qquad [1] \\ \omega_d &= \sqrt{|\omega_0^2 - \gamma^2|} \approx 0.9 \end{aligned}$$

Referencing the Figure 11 and 12, from the left V-t figure, we can find that $\omega \approx \omega_{MAX}$, there is a maximum peak, and when $\omega = \omega_R$, it shows that the oscillating frequency is approximately equal to the driving frequency; when $\omega < \omega_d$ the response of results are greatly "distort", compared to the particular solution, and if $\omega > \omega_d$, in high frequencies level, it is more likely follow the steady-state solution. All in all, not only the maximum voltage but also the resonance frequency meet our expectations. Besides, we find that the "transient" phenomenon still needs to be considered in the low frequencies regime. (Figure 12, is the small number of group comparisons)

[1] Classical Dynamics, 5th edition, page 125, Thornton, Marion

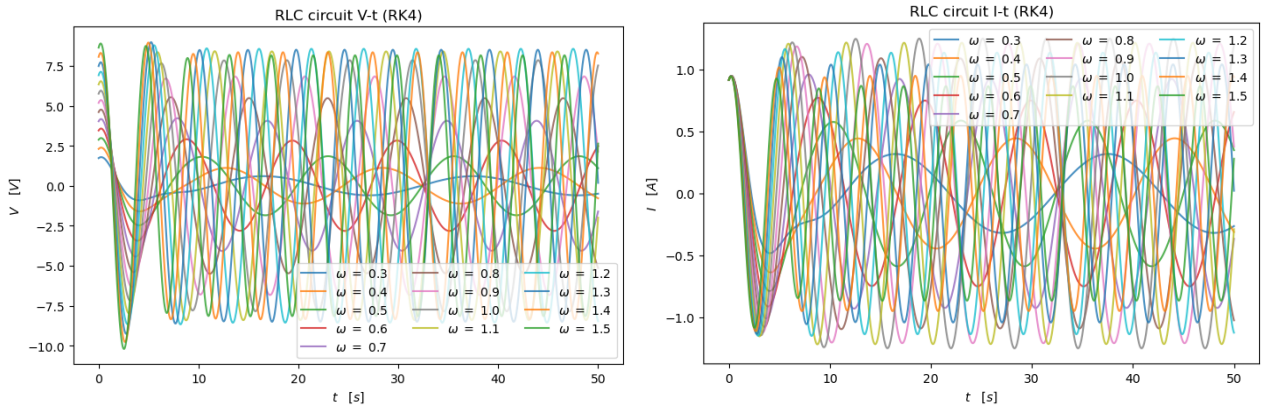

Figure 11: Left is the V-t figure, and right is the I-t. The plot consists of 13 different frequencies.

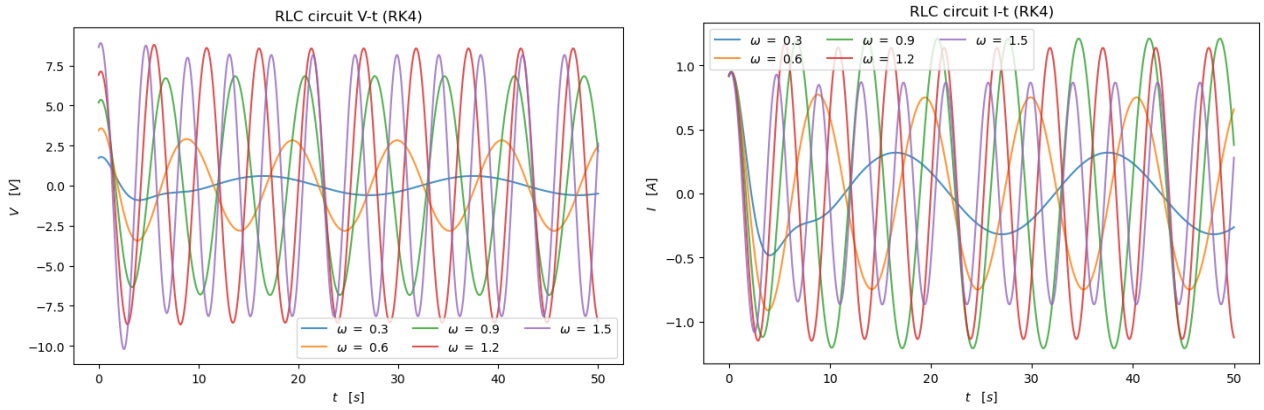

Figure 12: Left is the V-t figure, and right is the I-t. (more concise edition of Figure 11)

3 Codes

All the codes are transferred from jupyterlab or python codes; hence, if you want to re-run them, please see the source code in the attached files or my GitHub repository: <https://github.com/gary20000915/Comphyslab-HW2.git>.

3.1 `myslover.py`

```
"""

This program solves Initial Value Problems (IVP).
We support three numerical meothds: Euler, Rk2, and Rk4

Author: Yuan-Yen Peng (edited from Prof. Kuo-Chuan Pan, NTHU 2022.10.06)
For the course, computational physics lab

"""

import numpy as np

def solve_ivp(derive_func, y0, t, dt, N, method, args):
    """
    Solve Initial Value Problems.

```

```
17          :param derive_func: a function to describe the derivative of the desired
                function
18          :param y0: an array. The initial state
19          :param: t: the instant time of the motion.
20          :param dt: the step time
21          :param N: the number of steps.
22          :param method: string. Numerical method to compute.
23                      We support "Euler", "RK2" and "RK4".
24          :param *args: extra arguments for the derive func.
25
26          :return: array_like. solutions.
27          """
28          sol_pos, sol_vel = np.array([]), np.array([])
29          t = 0
30          for _ in range(N):
31              t += dt
32              sol_pos = np.append(sol_pos, y0[0])
33              sol_vel = np.append(sol_vel, y0[1])
34              y0 = _update(derive_func, y0, t, dt, method, *args)
35
36          return [sol_pos, sol_vel]
37
38      def _update(derive_func, y0, t, dt, method, *args):
39          """
40          Update the IVP with different numerical method
41
42          :param derive_func: the derivative of the function y'
43          :param y0: the initial conditions at time t
44          :param: t: the instant time of the motion
45          :param dt: the time step dt
46          :param method: the numerical method
47          :param *args: extral parameters for the derive_func
48
49          :return: the next step condition y0
50
51          """
52
53          if method=="Euler":
54              ynext = _update_euler(derive_func,y0, t, dt,*args)
55          elif method=="RK2":
56              ynext = _update_rk2(derive_func,y0, t, dt,*args)
57          elif method=="RK4":
58              ynext = _update_rk4(derive_func,y0,t, dt,*args)
59          else:
60              print("Error: mysolve doesn't supput the method",method)
61              quit()
62          return ynext
63
64      def _update_euler(derive_func,y0, t, dt,*args):
65          """
66          Update the IVP with the Euler's method
67
68          :return: the next step solution y
```

```
69
70          """
71          y0 = np.add(y0, derive_func(y0, t, *args) * dt)
72
73          return y0 # <- change here. just a placeholder
74
75      def _update_rk2(derive_func, y0, t, dt,*args):
76          """
77          Update the IVP with the RK2 method
78
79          :return: the next step solution y
80          """
81
82          k1 = derive_func(y0, t, *args)
83          y_temp = y0 + dt * k1
84          k2 = derive_func(y_temp, t, *args)
85
86          y0 = np.add(y0, (dt / 2) * (k1 + k2))
87          # note: if use: y0 += (dt / 2) * (k1 + k2) ==> error
88          # Yet use y0 = y0 + (dt / 2) * (k1 + k2) ==> it can work, and np.add() can
                work too.
89
90          return y0 # <- change here. just a placeholder
91
92      def _update_rk4(derive_func,y0, t, dt,*args):
93          """
94          Update the IVP with the RK4 method
95
96          :return: the next step solution y
97          """
98
99          k1 = derive_func(y0, t, *args)
100         y_temp = y0 + (dt/2) * k1 # temp for virtual step y*
101         k2 = derive_func(y_temp, t, *args)
102         y_temp = y0 + (dt/2) * k2
103         k3 = derive_func(y_temp, t, *args)
104         y_temp = y0 + dt * k3
105         k4 = derive_func(y_temp, t, *args)
106
107         y0 = np.add(y0, (1/6) * dt * (k1 + 2*k2 + 2*k3 + k4))
108
109         return y0 # <- change here. just a placeholder
110
111
112     if __name__=='__main__':
113
114
115         """
116
117         Testing mysolver.solve_ivp()
118
119         Kuo-Chuan Pan 2022.10.07
120
```

```
    """


    def oscillator(y,t,K,M):
        '''
        This is the function (osci) defined in the [position, velocity]
        and [derivative(position), derivative(velocity)]
        :param y: [position, velocity]
        :param k: spring constants
        :param m: mass constant
        '''

        yder = np.zeros(2)
        yder[0] = y[1]
        yder[1] = -y[0] * K/M # the difinition of the acceleration, which is
            depend on the position.
        print(t) # check to update time
        return yder

    K, M = 1, 1
    N, t = 100, 20
    dt = t/N
    y0 = np.array([1, 0]) # [pos_0, vel_0]

    sol = solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(K,M))

    # print("sol=",sol)
    print("Done!")
```

3.2 Forced oscillations and RLC circuit

```
# %% [markdown]
#
# ## Programming Assignment 3 and 4
# ### 111 Computational Physics Lab
#   >Author: Yuan-Yen Peng 108000204
#   >Email: garyphys0915@gapp.nthu.edu.com
#   >Date: Nov. 11, 2022
#   >LINCENCE: MIT

# %% [markdown]
# #### 3. Forced oscillator

# %%
import numpy as np
import matplotlib.pyplot as plt
import mysolver as solver

# %%
def oscillator(y,t, lam, wf, F0, K, M):
        '''
        This is the function (osci) defined in the [position, velocity]
```

```
        and [derivative(position), derivative(velocity)]
        :param y: [position, velocity]
        :param t: time (time varying)
        :param lam: \lambda ==> damping constant
        :param wf: \omega_f ==> forceing frequency
        :param F0: initial forceing force
        :param K: spring constants
        :param M: mass constant
        '''
        yder = np.zeros(2)
        yder[0] = y[1]
        yder[1] = -y[0] * K/M - y[1] * lam / M + F0 * np.cos(wf * t) / M # the
            difinition of the acceleration, which is depend on the position.

        return yder

# %%
def plot(u1, u2, u3, wf, lam):
  '''
  This is the plotting function
  :param ui: u is outcomes. (i = 1, 2, 3) ==> (Euler, RK2, RK4) -> Array
  :param wf: wf is the specified \omega_f. -> Array
  :param lam: the value of lambda.
  '''
  # plt.plot(wf, u1, "r", alpha = 0.3, label = "Euler")
  # plt.plot(wf, u2, "g", alpha = 0.3, label = "RK2")
  plt.plot(wf, u3, alpha = 0.5, label = f"$\lambda\ =\ {lam}$")
  plt.title(f"Forced")
  plt.ylabel("$D\quad [cm]$")
  plt.xlabel("$\omega_{f}\quad [rad*s^{-1}]$")
  plt.legend(loc = "best")

# %%
def CIR_V(t, V, wf):
  '''
  This is the plotting function
  :param t: time. -> Array
  :param V: input voltage. -> Array
  :param wf: wf is the specified \omega_f. -> Array
  '''
  plt.plot(t, V, alpha = 0.8, label = f"$\omega\ =\ {np.round(float(wf), 2)}$")
  plt.title(f"RLC circuit V-t (RK4)")
  plt.ylabel("$V\quad [V]$")
  plt.xlabel("$t\quad [s]$")
  plt.legend(loc = "best", ncol = 3)

def CIR_I(t, I, wf):
  '''
  This is the plotting function
  :param t: time. -> Array
  :param V: input current. -> Array
  :param wf: wf is the specified \omega_f. -> Array
  '''
```

```
74          plt.plot(t, I, alpha = 0.8, label = f"$\omega\ =\ {np.round(float(wf), 2)}$")
75          plt.title(f"RLC circuit I-t (RK4)")
76          plt.ylabel("$I\quad [A]$")
77          plt.xlabel("$t\quad [s]$")
78          plt.legend(loc = "best", ncol = 3)
79
80        # %%
81        def para(lam):
82          N, t = int(1e3), 50 # divided into 100
83          dt = t/N
84          T = np.linspace(0, 50, N)
85
86          M, K = 1, 1
87          A = 1 # initial amplitude
88
89          phi = - np.pi/2
90          r = lam/(2 * M)
91          F0 = 0.5
92          space = 0.01
93          wf = np.arange(0.5, 1.5 + space, space) # [start, stop)
94          w0 = np.sqrt(K/M)
95          w = np.sqrt(abs(np.square(w0) - np.square(r)))
96
97          y0  = np.zeros(2)
98          y0[0]= 0 # initial position
99          y0[1] = -A * r * np.cos(phi) - A * w * np.sin(phi)
100
101         # x = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
                parts.
102
103         ind = np.where(T >= 40)
104         RANGE = np.array([ind])
105
106         D1 = np.zeros(len(wf))
107         D2 = np.zeros(len(wf))
108         D3 = np.zeros(len(wf))
109
110         for i in range(len(wf)):
111           sol1 = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler",
                  args=(lam, wf[i], F0, K, M))[0]
112           D1[i] = np.average(np.abs(sol1[RANGE]))
113           sol2 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam,
                  wf[i], F0, K, M))[0]
114           D2[i] = np.average(np.abs(sol2[RANGE]))
115           sol3 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam,
                  wf[i], F0, K, M))[0]
116           D3[i] = np.average(np.abs(sol3[RANGE]))
117
118         D3_MAX_ind = np.where(D3 == np.max(D3))
119         wf_MAX = float(wf[D3_MAX_ind])
120         print(f"When \lambda = {lam}, the resonance frequency is on \omega_f =
              {wf_MAX}, and the average amplitude is D = {float(np.max(D3))}.")
121
```

```
    return D1, D2, D3, wf, lam

# %%
out = para(0.01)
plot(out[0], out[1], out[2], out[3], out[4])
out = para(0.1)
plot(out[0], out[1], out[2], out[3], out[4])
out = para(0.3)
plot(out[0], out[1], out[2], out[3], out[4])

# %% [markdown]
# #### 4. RLC circuit

# %%
def RLC(y, t, L, R, C, wf, E0):
        '''
        This is the function (osci) defined in the [position, velocity]
        and [derivative(position), derivative(velocity)]
        :param y: [position, velocity]
        :param t: time (time varying)
        :param lam: \lambda ==> damping constant
        :param wf: \omega_f ==> forceing frequency
        :param F0: initial forceing force
        :param K: spring constants
        :param M: mass constant
        '''
        yder = np.zeros(2)
        yder[0] = y[1]
        yder[1] = -y[0] / (C * L) - y[1] * R / L + E0 * np.cos(wf * t) / L #
            the difinition of the acceleration, which is depend on the position.

        return yder

# %%

N, t = int(1e3), 50 # divided into 100
dt = t/N
T = np.linspace(0, 50, N)

L, C = 1, 1
A = 1 # initial amplitude
phi = - np.pi/2

R = 0.8
r = R / (2 * L)
E0 = 1
# space = 0.1
# wf = np.arange(0.3, 1.5 + space, space) # [start, stop)
wf = np.array([0.7])
w0 = np.sqrt(1 / (C * L))
w = np.sqrt(abs(np.square(w0) - np.square(r)))

y0  = np.zeros(2)
```

```
y0[0]= 0 # initial position
y0[1] = -A * r * np.cos(phi) - A * w * np.sin(phi)

# x = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
    parts.

for i in range(len(wf)):
  X_L = 2 * np.pi * wf[i] * L
  I1 = solver.solve_ivp(RLC, y0, t, dt, N, method="Euler", args=(L, R, C,
      wf[i], E0))[1]
  V1 = I1 * X_L
  I2 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK2", args=(L, R, C, wf[i],
      E0))[1]
  V2 = I2 * X_L
  I3 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK4", args=(L, R, C, wf[i],
      E0))[1]
  V3 = I3 * X_L
  CIR_V(T, V3, wf[i])
  plt.show()
  CIR_I(T, I3, wf[i])
  plt.show()

# %%
N, t = int(1e3), 50 # divided into 100
dt = t/N
T = np.linspace(0, t, N)

L, C = 1, 1
A = 1 # initial amplitude
phi = - np.pi/2

R = 0.8
r = R / (2 * L)
E0 = 1
space = 0.1
wf = np.arange(0.3, 1.5 + space, space) # [start, stop)
# wf = np.array([0.7])
w0 = np.sqrt(1 / (C * L))
w = np.sqrt(abs(np.square(w0) - np.square(r)))

y0 = np.zeros(2)
y0[0]= 0 # initial position
y0[1] = -A * r * np.cos(phi) - A * w * np.sin(phi)

print(np.sqrt(w0 ** 2 - 2 * r ** 2))
print(1/np.sqrt(L*C - 0.5 * (R*C) ** 2))
print(w)
# x = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
    parts.

plt.figure(figsize = (8.1,5))
for i in range(len(wf)):
  X_L = 2 * np.pi * wf[i] * L
```

```
    I1 = solver.solve_ivp(RLC, y0, t, dt, N, method="Euler", args=(L, R, C,
        wf[i], E0))[1]
    V1 = I1 * X_L
    I2 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK2", args=(L, R, C, wf[i],
        E0))[1]
    V2 = I2 * X_L
    I3 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK4", args=(L, R, C, wf[i],
        E0))[1]
    V3 = I3 * X_L
    CIR_I(T, I3, wf[i])

plt.figure(figsize = (8.1,5))
for i in range(len(wf)):
    X_L = 2 * np.pi * wf[i] * L
    I1 = solver.solve_ivp(RLC, y0, t, dt, N, method="Euler", args=(L, R, C,
        wf[i], E0))[1]
    V1 = I1 * X_L
    I2 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK2", args=(L, R, C, wf[i],
        E0))[1]
    V2 = I2 * X_L
    I3 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK4", args=(L, R, C, wf[i],
        E0))[1]
    V3 = I3 * X_L
    CIR_V(T, V3, wf[i])

# %%
plt.figure(figsize = (8.1,5))
for i in range(0, len(wf), 3):
    X_L = 2 * np.pi * wf[i] * L
    I3 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK4", args=(L, R, C, wf[i],
        E0))[1]
    V3 = I3 * X_L
    CIR_V(T, V3, wf[i])

# %%
plt.figure(figsize = (8.1,5))
for i in range(0, len(wf), 3):
    X_L = 2 * np.pi * wf[i] * L
    I3 = solver.solve_ivp(RLC, y0, t, dt, N, method="RK4", args=(L, R, C, wf[i],
        E0))[1]
    V3 = I3 * X_L
    CIR_I(T, I3, wf[i])

# %%
```

3.3 Damped Oscillator

```
# %% [markdown]
#
# ## Programming Assignment 1 and 2
# ### 111 Computational Physics Lab
#   >Author: Yuan-Yen Peng 108000204
```

```
#   >Email: garyphys0915@gapp.nthu.edu.com
#   >Date: Nov. 11, 2022
#   >LINCENCE: MIT

# %% [markdown]
# #### 1. Damped oscillator

# %%
import numpy as np
import matplotlib.pyplot as plt
import mysolver as solver

# %%
def oscillator(y,t, lam, K, M):
        '''
        This is the function (osci) defined in the [position, velocity]
        and [derivative(position), derivative(velocity)]
        :param y: [position, velocity]
        :param t: time (time varying)
        :param lam: \lambda ==> damping constant
        :param K: spring constants
        :param M: mass constant
        '''
        yder = np.zeros(2)
        yder[0] = y[1]
        yder[1] = -y[0] * K/M - y[1] * lam / M # the difinition of the
            acceleration, omghich is depend on the position.

        return yder

# %%
def plot(u1, u2, u3, w1, w2, w3, num):
  '''
  This is the plotting function
  :param ui: u is the specified polared unit for x-axis. (i = 1, 2, 3) ==>
      (Euler, RK2, RK4)
  :param wi: w is the specified polared unit for y-axis. (i = 1, 2, 3) ==>
      (Euler, RK2, RK4)
  '''
  plt.plot(u1, w1, "r", alpha = 0.3, label = "Euler")
  plt.plot(u2, w2, "g", alpha = 0.3, label = "RK2")
  plt.plot(u3, w3, "b", alpha = 0.3, label = "RK4")
  plt.axis("equal")
  plt.title(f"Damped ({num})")
  plt.xlabel("$w\quad [rad*s^{-1}]$")
  plt.ylabel("$u\quad [cm*rad*s^{-1}]$")
  plt.legend(loc = "best")
  plt.show()

# %%
def xt(t_eval, sol1, sol2, sol3, sub):
  plt.plot(t_eval, sol1, "r", label = "Euler", alpha = 0.3)
  plt.plot(t_eval, sol2, "g", label = "RK2", alpha = 0.3)
```

```
    plt.plot(t_eval, sol3, "b", label = "RK4", alpha = 0.3)
    plt.title(f"Damped x-t ({sub})")
    plt.xlabel("t [s]")
    plt.ylabel("x [cm]")
    plt.legend()
    plt.show()

def vt(t_eval, sol1, sol2, sol3, sub):
    plt.plot(t_eval, sol1, "r", label = "Euler", alpha = 0.3)
    plt.plot(t_eval, sol2, "g", label = "RK2", alpha = 0.3)
    plt.plot(t_eval, sol3, "b", label = "RK4", alpha = 0.3)
    plt.title(f"Damped v-t ({sub})")
    plt.xlabel("t [s]")
    plt.ylabel("v [m/s]")
    plt.legend()
    plt.show()

# %%
def et(t_eval, sol1, sol2, sol3, sub):
    plt.plot(t_eval, sol1, "r", label = "Euler", alpha = 0.3)
    plt.plot(t_eval, sol2, "g", label = "RK2", alpha = 0.3)
    plt.plot(t_eval, sol3, "b", label = "RK4", alpha = 0.3)
    plt.title(f"Damped E-t ({sub})")
    plt.xlabel("t [s]")
    plt.ylabel("E [J]")
    plt.legend()
    plt.show()

# %%
def rate(t_eval, sol1, sol2, sol3, sub):
    # plt.plot(t_eval, sol1, "r", label = "Euler", alpha = 0.3)
    # plt.plot(t_eval, sol2, "g", label = "RK2", alpha = 0.3)
    plt.plot(t_eval, sol3, "b", label = "RK4", alpha = 0.3)
    plt.title(f"Damped Energy Difference ({sub})")
    plt.xlabel("t [s]")
    plt.ylabel("rate [mks]")
    plt.legend()
    plt.show()

# %% [markdown]
# (a) $A = 1 [cm],\quad \omega_0 = 1 [rads^{-1}],\quad \gamma = 0.2 [s^{-1}],
    \quad \phi = -\pi / 2 [rad]$

# %%
# Setting parameters
N, t = 100, 20
dt = t/N

M, K = 1, 1
A = 1 # initial amplitude
r  = 0.2 # \gamma
lam = 2 * M * r # \lamda
phi = - np.pi/2
```

```
omg0 = np.sqrt(K/M)
omg1 = np.sqrt(abs(np.square(omg0) - np.square(r)))

y0  = np.zeros(2)
y0[0]= 0 # initial position
y0[1] = - A * r * np.cos(phi) - A * omg1 * np.sin(phi) # initial velocity

t_eval = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
    parts.

# %%
# position
sol1 = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler", args=(lam,
    K, M))[0]
sol2 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam, K,
    M))[0]
sol3 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam, K,
    M))[0]

# velocity
sol1_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler",
    args=(lam, K, M))[1]
sol2_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam,
    K, M))[1]
sol3_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam,
    K, M))[1]

# %%
# Visualize
xt(t_eval, sol1, sol2, sol3, "a")
vt(t_eval, sol1_v, sol2_v, sol3_v, "a")

# %%
# polar coordinates
u1 = omg1 * sol1
u2 = omg1 * sol2
u3 = omg1 * sol3

w1 = r * sol1 + sol1_v
w2 = r * sol2 + sol2_v
w3 = r * sol3 + sol3_v

# plot it!
plot(u1, u2, u3, w1, w2, w3, "a")

# %% [markdown]
# #### 2. Total energy and the energy loss

# %%
# kenetical energy
K1 = 0.5 * M * np.square(sol1)
K2 = 0.5 * M * np.square(sol2)
K3 = 0.5 * M * np.square(sol3)
```

```
154
155     # potential energy
156     U1 = 0.5 * K * np.square(sol1_v)
157     U2 = 0.5 * K * np.square(sol2_v)
158     U3 = 0.5 * K * np.square(sol3_v)
159
160     # total energy
161     tot1 = K1 + U1
162     tot2 = K2 + U2
163     tot3 = K3 + U3
164
165     # energy loss
166     no_loss = 0.5 * K * np.square(A)
167     loss1 = no_loss - tot1
168     loss2 = no_loss - tot2
169     loss3 = no_loss - tot3
170
171     # rate of energy loss
172     rate3 = np.array([])
173     for i in range (len(loss3) - 1):
174         rate3 = np.append(rate3, (loss3[i+1] - loss3[i]) / dt)
175
176     rate2 = (loss2 / no_loss) * 100
177     rate1 = (loss1 / no_loss) * 100
178
179     # plot them!
180     et(t_eval, tot1, tot2, tot3, "total energy")
181     et(t_eval, loss1, loss2, loss3, "loss energy")
182     t_eval_prime = t_eval[0:-1]
183     rate(t_eval_prime, rate1, rate2, rate3, "RK4, methof of rate: Euler")
184
185
186     # %% [markdown]
187     # ### 1. (conti)
188     # (b) $A = 1 [cm],\quad \omega_0 = 1 [rads^{-1}],\quad \gamma = 1.0 [s^{-1}],
            \quad \phi = -\pi / 2 [rad]$
189
190     # %%
191     # update the parameter \gamma
192     # Setting parameters
193     N, t = 100, 20
194     dt = t/N
195
196     M, K = 1, 1
197     A = 1 # initial amplitude
198     r  = 1.0 # \gamma
199     lam = 2 * M * r # \lamda
200     phi = - np.pi/2
201     omg0 = np.sqrt(K/M)
202     omg1 = np.sqrt(abs(np.square(omg0) - np.square(r)))
203
204     y0  = np.zeros(2)
205     y0[0]= 10e-5 # initial position (use tolerence = 10e-5)
```

```
206     y0[1] = - A * r * np.cos(phi) - A * omg1 * np.sin(phi) # initial velocity
207 
208     t_eval = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
            parts.
209 
210     # %%
211     # position
212     sol1 = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler", args=(lam,
            K, M))[0]
213     sol2 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam, K,
            M))[0]
214     sol3 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam, K,
            M))[0]
215 
216     # velocity
217     sol1_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler",
            args=(lam, K, M))[1]
218     sol2_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam,
            K, M))[1]
219     sol3_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam,
            K, M))[1]
220 
221     # %%
222     # Visualize
223     xt(t_eval, sol1, sol2, sol3, "b")
224     vt(t_eval, sol1_v, sol2_v, sol3_v, "b")
225 
226     # %%
227     # polar coordinates
228     u1 = omg1 * sol1
229     u2 = omg1 * sol2
230     u3 = omg1 * sol3
231 
232     w1 = r * sol1 + sol1_v
233     w2 = r * sol2 + sol2_v
234     w3 = r * sol3 + sol3_v
235 
236     # plot it!
237     plot(u1, u2, u3, w1, w2, w3, "b")
238 
239     # %% [markdown]
240     # (c) $A = 1 [cm],\quad \omega_0 = 1 [rads^{-1}],\quad \gamma = 1.2 [s^{-1}],
            \quad \phi = -\pi / 2 [rad]$
241 
242     # %%
243     # update the parameter \gamma
244     # Setting parameters
245     N, t = 100, 20
246     dt = t/N
247 
248     M, K = 1, 1
249     A = 1 # initial amplitude
250     r  = 1.2 # \gamma
```

```
    lam = 2 * M * r # \lamda
    phi = - np.pi/2
    omg0 = np.sqrt(K/M)
    omg1 = np.sqrt(abs(np.square(omg0) - np.square(r)))

    y0  = np.zeros(2)
    y0[0]= 0 # initial position (use tolerence = 10e-5)
    y0[1] = - A * r * np.cos(phi) - A * omg1 * np.sin(phi) # initial velocity

    t_eval = np.linspace(0, t, N) # from 0 to t divided by N+1, i.e., N+1 equal
        parts.

    # %%
    # position
    sol1 = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler", args=(lam,
        K, M))[0]
    sol2 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam, K,
        M))[0]
    sol3 = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam, K,
        M))[0]

    # velocity
    sol1_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="Euler",
        args=(lam, K, M))[1]
    sol2_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK2", args=(lam,
        K, M))[1]
    sol3_v = solver.solve_ivp(oscillator, y0, t, dt, N, method="RK4", args=(lam,
        K, M))[1]

    # %%
    # Visualize
    xt(t_eval, sol1, sol2, sol3, "c")
    vt(t_eval, sol1_v, sol2_v, sol3_v, "c")

    # %%
    # polar coordinates
    u1 = omg1 * sol1
    u2 = omg1 * sol2
    u3 = omg1 * sol3

    w1 = r * sol1 + sol1_v
    w2 = r * sol2 + sol2_v
    w3 = r * sol3 + sol3_v

    # plot it!
    plot(u1, u2, u3, w1, w2, w3, "c")
```