ZJU Computational Physics: Homework #1

Due on Monday, October 18, 2024

管一沣

Question 1

The cooling coffee.

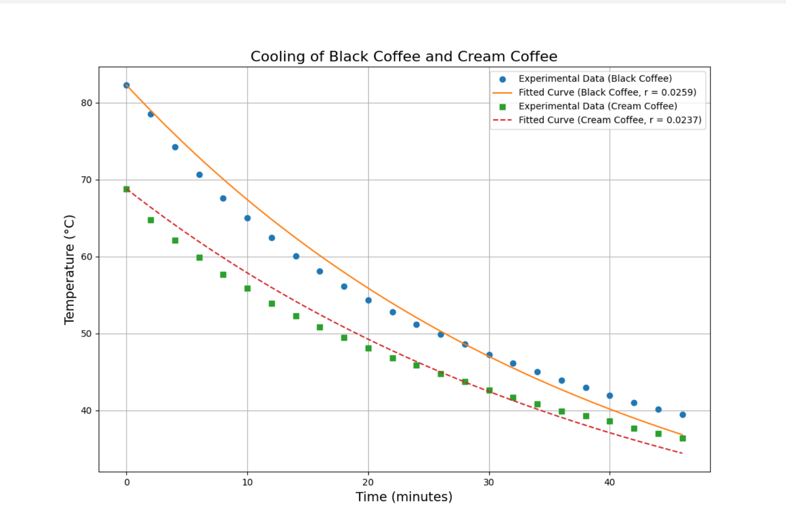
a. 计算方法采用两种：第一种是简单地对每一个 Δt 求 r 值，之后对所有 r 值求均值；第二种是将所有数据在牛顿冷却函数下做最小二乘法拟合。由于时间间隔 2min 并不是一个足够小的值，所以第一种方法的误差会偏大，而采用最小二乘法拟合会更加精准。
通过代码计算（参见/1/a.py）可得，由第一种方法计算得到的黑咖啡 r 值为 0.0232min^{-1}，奶咖啡为 0.0214min^{-1}；由第二种方法得到的黑咖啡 r 值为 0.0259min^{-1}，奶咖啡为 0.0237min^{-1}。之后的计算都采用第二种方法得到的结果。

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def newton_cooling_function(t, r, env_T, initial_T):
    return env_T+(initial_T-env_T)*np.exp(-r*t)

def calculate_r_seperatedly(t, T, env_T):
    r_values = []
    for i in range(len(t) - 1):
        delta_t = t[i + 1] - t[i]
        T_diff = T[i] - env_T
        next_T_diff = T[i + 1] - env_T
        r = -np.log(next_T_diff / T_diff) / delta_t
        r_values.append(r)
    return np.mean(r_values), np.std(r_values)

env_T = 17

t_black = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
    38, 40, 42, 44, 46])
T_black = np.array([82.3, 78.5, 74.3, 70.7, 67.6, 65.0, 62.5, 60.1, 58.1, 56.1, 54.3, 52.8,
                    51.2, 49.9, 48.6, 47.2, 46.1, 45.0, 43.9, 43.0, 41.9, 41.0, 40.1, 39.5])

t_cream = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
    38, 40, 42, 44, 46])
T_cream = np.array([68.8, 64.8, 62.1, 59.9, 57.7, 55.9, 53.9, 52.3, 50.8, 49.5, 48.1, 46.8,
                    45.9, 44.8, 43.7, 42.6, 41.7, 40.8, 39.9, 39.3, 38.6, 37.7, 37.0, 36.4])

initial_T_black = T_black[0]
initial_T_cream = T_cream[0]


```

Question 1 continued on next page...

```python
params_black, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_black),
                            t_black, T_black)
r_black = params_black[0]

params_cream, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_cream),
                            t_cream, T_cream)
r_cream = params_cream[0]

print("==========Least␣Squares␣Methods==========")
print(f"black␣coffee's␣r:␣{r_black}␣min^-1")
print(f"cream␣coffee's␣r:␣{r_cream}␣min^-1")

r_black_mean, r_black_std = calculate_r_seperatedly(t_black, T_black, env_T)
r_cream_mean, r_cream_std = calculate_r_seperatedly(t_cream, T_cream, env_T)

print("==========Calculate␣partially==========")
print("black␣coffee's␣r:␣mean␣=␣{:.4f}␣min^-1,␣std␣=␣{:.4f}␣min^-1".format(r_black_mean,
    r_black_std))
print("cream␣coffee's␣r:␣mean␣=␣{:.4f}␣min^-1,␣std␣=␣{:.4f}␣min^-1".format(r_cream_mean,
    r_cream_std))
```

b. 参见/1/b.py，图像中可见仍然存在一定的与实际值的误差。




```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def newton_cooling_function(t, r, env_T, initial_T):
    return env_T+(initial_T-env_T)*np.exp(-r*t)
```

```python

t_black = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46])
T_black = np.array([82.3, 78.5, 74.3, 70.7, 67.6, 65.0, 62.5, 60.1, 58.1, 56.1, 54.3, 52.8,
                    51.2, 49.9, 48.6, 47.2, 46.1, 45.0, 43.9, 43.0, 41.9, 41.0, 40.1, 39.5])
t_cream = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46])
T_cream = np.array([68.8, 64.8, 62.1, 59.9, 57.7, 55.9, 53.9, 52.3, 50.8, 49.5, 48.1, 46.8,
                    45.9, 44.8, 43.7, 42.6, 41.7, 40.8, 39.9, 39.3, 38.6, 37.7, 37.0, 36.4])
env_T = 17
initial_T_black = T_black[0]
initial_T_cream = T_cream[0]

params_black, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T, initial_T_black),
                            t_black, T_black)
r_black = params_black[0]

params_cream, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T, initial_T_cream),
                            t_cream, T_cream)
r_cream = params_cream[0]

time_fit = np.linspace(0, 46, 200)
temp_fit_black = newton_cooling_function(time_fit, r_black, env_T, initial_T_black)
temp_fit_cream = newton_cooling_function(time_fit, r_cream, env_T, initial_T_cream)

plt.figure(figsize=(12, 8))

plt.plot(t_black, T_black, 'o', label='Experimental Data (Black Coffee)')
plt.plot(time_fit, temp_fit_black, '-', label=f'Fitted Curve (Black Coffee, r = {r_black:.4f})')

plt.plot(t_cream, T_cream, 's', label='Experimental Data (Cream Coffee)')
plt.plot(time_fit, temp_fit_cream, '--', label=f'Fitted Curve (Cream Coffee, r = {r_cream:.4f})')

plt.xlabel('Time (minutes)', fontsize=14)
plt.ylabel('Temperature (°C)', fontsize=14)
plt.title('Cooling of Black Coffee and Cream Coffee', fontsize=16)
plt.legend()
plt.grid()
plt.show()
```

Question 1 continued on next page...

c. 设置两个新的步长 4min 和 1min，其中 4min 步长通过直接取数据中的每个 4min 时间间隔即可，1min 步长通过差值实现。

```python
import numpy as np
from scipy.optimize import curve_fit

def newton_cooling_function(t, r, env_T, initial_T):
    return env_T + (initial_T - env_T) * np.exp(-r * t)

t_black = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
    38, 40, 42, 44, 46])
T_black = np.array([82.3, 78.5, 74.3, 70.7, 67.6, 65.0, 62.5, 60.1, 58.1, 56.1, 54.3, 52.8,
                    51.2, 49.9, 48.6, 47.2, 46.1, 45.0, 43.9, 43.0, 41.9, 41.0, 40.1, 39.5])

t_cream = np.array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,
    38, 40, 42, 44, 46])
T_cream = np.array([68.8, 64.8, 62.1, 59.9, 57.7, 55.9, 53.9, 52.3, 50.8, 49.5, 48.1, 46.8,
                    45.9, 44.8, 43.7, 42.6, 41.7, 40.8, 39.9, 39.3, 38.6, 37.7, 37.0, 36.4])

env_T = 17
initial_T_black = T_black[0]
initial_T_cream = T_cream[0]

params_black, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_black), t_black, T_black)
r_black_original = params_black[0]

params_cream, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_cream), t_cream, T_cream)
r_cream_original = params_cream[0]

t_black_large_step = t_black[::2]
T_black_large_step = T_black[::2]
params_black_large, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_black), t_black_large_step, T_black_large_step)
r_black_large_step = params_black_large[0]

t_cream_large_step = t_cream[::2]
T_cream_large_step = T_cream[::2]
params_cream_large, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_cream), t_cream_large_step, T_cream_large_step)
r_cream_large_step = params_cream_large[0]

t_black_small_step = np.linspace(0, 46, 47)
T_black_small_step = np.interp(t_black_small_step, t_black, T_black)
params_black_small, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
```

```
    initial_T_black), t_black_small_step, T_black_small_step)
r_black_small_step = params_black_small[0]

t_cream_small_step = np.linspace(0, 46, 47)
T_cream_small_step = np.interp(t_cream_small_step, t_cream, T_cream)
params_cream_small, _ = curve_fit(lambda t, r: newton_cooling_function(t, r, env_T,
    initial_T_cream), t_cream_small_step, T_cream_small_step)
r_cream_small_step = params_cream_small[0]

relative_error_black_large = abs(r_black_large_step - r_black_original) / r_black_original *
    100
relative_error_black_small = abs(r_black_small_step - r_black_original) / r_black_original *
    100

relative_error_cream_large = abs(r_cream_large_step - r_cream_original) / r_cream_original *
    100
relative_error_cream_small = abs(r_cream_small_step - r_cream_original) / r_cream_original *
    100

print("Black␣coffee␣cooling␣constant␣r:")
print(f"␣␣Original␣step:␣r␣=␣{r_black_original:.6f}")
print(f"␣␣Larger␣step:␣r␣=␣{r_black_large_step:.6f}␣(Relative␣error:␣{
    relative_error_black_large:.2f}%)")
print(f"␣␣Smaller␣step:␣r␣=␣{r_black_small_step:.6f}␣(Relative␣error:␣{
    relative_error_black_small:.2f}%)")

print("\nCream␣coffee␣cooling␣constant␣r:")
print(f"␣␣Original␣step:␣r␣=␣{r_cream_original:.6f}")
print(f"␣␣Larger␣step:␣r␣=␣{r_cream_large_step:.6f}␣(Relative␣error:␣{
    relative_error_cream_large:.2f}%)")
print(f"␣␣Smaller␣step:␣r␣=␣{r_cream_small_step:.6f}␣(Relative␣error:␣{
    relative_error_cream_small:.2f}%)")
```

d. 降温到 49 度需要 27.54min，降温到 33 度需要 54.30 分钟，降温到 25 度需要 81.06 分钟。这说明物体与周围环境温差越小降温越慢。

```
import numpy as np

def time_to_temperature(r, T_env, T_initial, T_target):
    return -np.log((T_target - T_env) / (T_initial - T_env)) / r

r = 0.0259
T_env = 17
T_initial_black = 82.3
T_targets = [49, 33, 25]
```

Question 1 continued on next page...

```python

times = [time_to_temperature(r, T_env, T_initial_black, T_target) for T_target in T_targets]

for T_target, time in zip(T_targets, times):
    print(f"Cool␣down␣to␣{T_target}°C␣needed:␣{time:.2f}␣minutes")
```

e. 牛顿冷却定律并不完全适合这个问题，还存在液体的蒸发所带走的热量，以及液体表面积和与环境接触材质的影响，需要考虑物体与环境的热传导系数、物体的传热方式来进行优化。

Question 2

Accuracy and stability of the Eular method.

a. 牛顿冷却定律：$\frac{dT}{dt} = -r \times (T - T_{env})$，将 T 项移到左边后对时间从 0 到 t 积分得：$T(t) - T_{env} = e^{-rt} \times (T_0 - T_{env})$，其中 $T_0 = T(\inf) = T_s$，可得 $T(t) = T_s - (T_s - T_0)e^{-rt}$

b. 参见/2/b.py

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

r = 0.0259
T_s = 17
T_0 = 82.3
t_target = 1.60

def analytical_solution(t, T_s, T_0, r):
    return T_s + (T_0 - T_s) * np.exp(-r * t)

def euler_method(T_s, T_0, r, delta_t, t_target):
    time_points = np.arange(0, t_target + delta_t, delta_t)
    T = np.zeros(len(time_points))
    T[0] = T_0
    for n in range(1, len(time_points)):
        T[n] = T[n - 1] + delta_t * (-r * (T[n - 1] - T_s))
    return time_points, T

delta_ts = [0.1, 0.05, 0.025, 0.01, 0.005]
errors = []

```

Question 2 continued on next page...

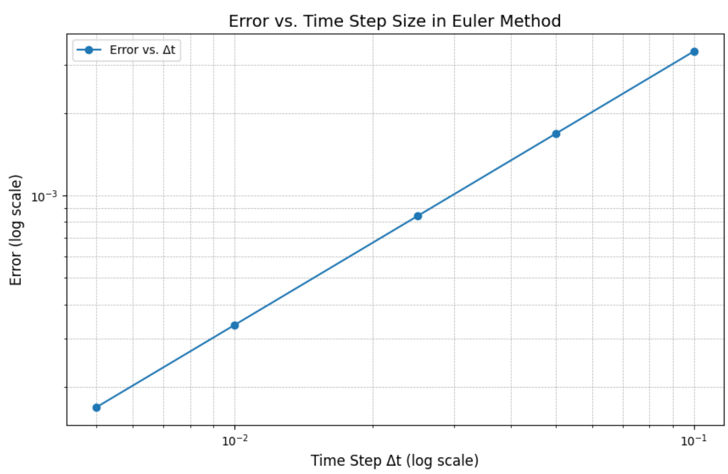
```python
for delta_t in delta_ts:
    _, T_numeric = euler_method(T_s, T_0, r, delta_t, t_target)
    T_exact = analytical_solution(t_target, T_s, T_0, r)
    error = abs(T_numeric[-1] - T_exact)
    errors.append(error)

plt.figure(figsize=(10, 6))
plt.loglog(delta_ts, errors, 'o-', label='Error vs. Δt')
plt.xlabel('Time Step Δt (log scale)', fontsize=12)
plt.ylabel('Error (log scale)', fontsize=12)
plt.title('Error vs. Time Step Size in Euler Method', fontsize=14)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.legend()
plt.show()

print("b：")
for i, delta_t in enumerate(delta_ts):
    print(f"Δt = {delta_t:.3f}, Error = {errors[i]:.6f}")

def find_delta_t(T_s, T_0, r, t_target, tolerance):
    delta_t = 0.1
    while True:
        _, T_numeric = euler_method(T_s, T_0, r, delta_t, t_target)
        T_exact = analytical_solution(t_target, T_s, T_0, r)
        error = abs(T_numeric[-1] - T_exact)
        if error/T_exact <= tolerance:
            return delta_t, error
        delta_t /= 2

tolerance = 0.001
delta_t_1_60, error_1_60 = find_delta_t(T_s, T_0, r, 1.60, tolerance)
delta_t_5_5, error_5_5 = find_delta_t(T_s, T_0, r, 5.5, tolerance)

print("c：")
print(f"t = 1.60， Δt = {delta_t_1_60:.6f}, respective_error = {error_1_60*100:.6f}%")
print(f"t = 5.50， Δt = {delta_t_5_5:.6f}, respective_error = {error_5_5*100:.6f}%")
```

运行结果为:

```
b:
Δt = 0.100, Error = 0.003368
Δt = 0.050, Error = 0.001682
Δt = 0.025, Error = 0.000841
Δt = 0.010, Error = 0.000336
Δt = 0.005, Error = 0.000168
c:
t = 1.60, Δt = 0.100000, respective_error = 0.336779%
t = 5.50, Δt = 0.100000, respective_error = 1.046383%
```

整理如下

Δt	Error
0.100	0.003368
0.050	0.001682
0.025	0.000841
0.010	0.000336
0.005	0.000168



由图可见，Eular method 是 1 阶方法

c. 由运行结果，$\Delta t = 0.1$ 和 $\Delta t = 0.1$

Question 3 continued on next page...

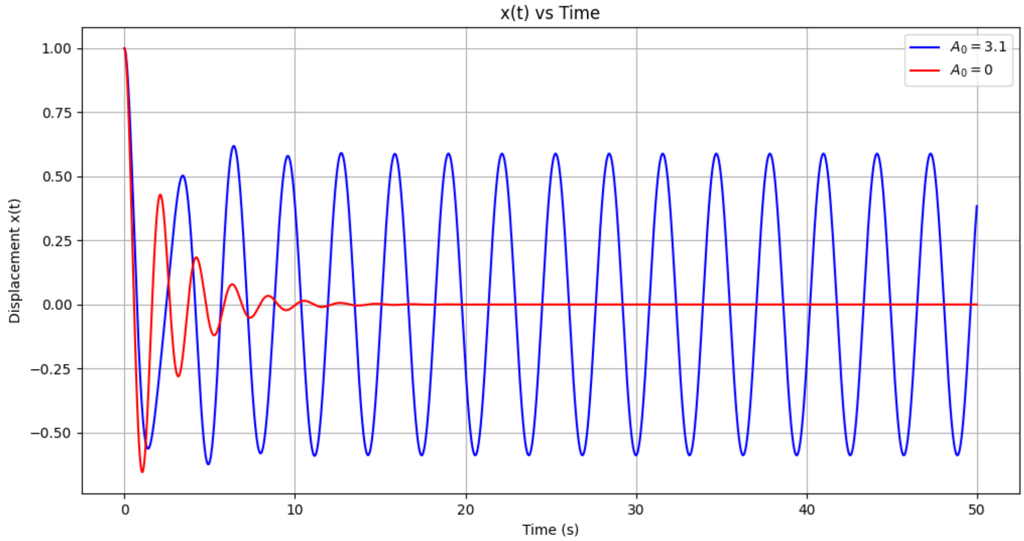
Question 3

Motion of a linear oscillator.

a. 参见/5/a.py

```python
import numpy as np
import matplotlib.pyplot as plt
from utils import driven_oscillator
gamma = 0.4
omega0 = 3.0
omega = 2.0
A0 = 3.1

x0, v0 = 1.0, 0.0
t_end = 50
dt = 0.01


t, x, _ = driven_oscillator(A0, omega, omega0, gamma, x0, v0, t_end, dt)
t1, x1, _ = driven_oscillator(0, omega, omega0, gamma, x0, v0, t_end, dt)
plt.figure(figsize=(12, 6))
plt.plot(t, x, label=r"$A_0=3.1$", color="blue")
plt.plot(t, x1, label=r"$A_0=0$", color="red")
plt.xlabel("Time (s)")
plt.ylabel("Displacement x(t)")
plt.title("x(t) vs Time")
plt.legend()
plt.grid()
plt.show()

x0_new, v0_new = 0.5, 1.0

t, x_new, _ = driven_oscillator(A0, omega, omega0, gamma, x0_new, v0_new, t_end, dt)
t1, x_new1, _ = driven_oscillator(0, omega, omega0, gamma, x0_new, v0_new, t_end, dt)
plt.figure(figsize=(12, 6))
plt.plot(t, x, label=r"Initial: $x(0)=1.0, \dot{x}(0)=0.0$", color="blue")
plt.plot(t, x_new, label=r"Initial: $x(0)=0.5, \dot{x}(0)=1.0$", color="orange")
#plt.plot(t1,x_new1, label=r"Initial: $x(0)=0.5, \dot{x}(0)=1.0, A_0=0$")
plt.xlabel("Time (s)")
plt.ylabel("Displacement x(t)")
plt.title("Driven Oscillator with Different Initial Conditions")
plt.legend()
plt.grid()
plt.show()
```

Question 3 continued on next page...



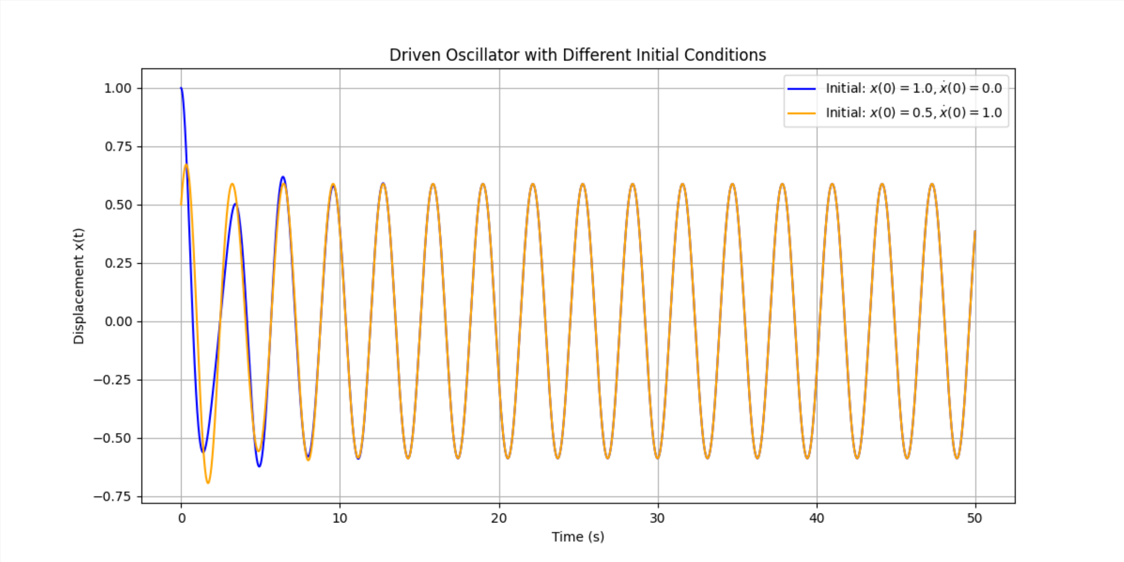
Q1: 当 $A_0 = 3.1$ 时，与未受驱动力影响的情况 ($A_0 = 0$) 相比，位移 $x(t)$ 的行为发生了显著变化。首先，在 $A_0 = 0$ 的情况下，系统的运动表现为单调衰减至零的阻尼行为，即在初始位移之后，系统因阻尼逐渐失去能量，位移 $x(t)$ 以指数形式衰减，不会出现持续的周期性振荡。

然而，当驱动力 $A_0 = 3.1$ 被引入后，系统的运动从单纯的阻尼衰减变为受驱动的强迫振荡。在初始阶段，系统的运动仍然包含一些瞬态行为，与 $A_0 = 0$ 的情况相似。然而，随着时间推移，这种瞬态行为逐渐消失，系统过渡到稳态振荡。在稳态阶段，系统以驱动力的频率 ω 振荡，而非系统的自然频率 ω_0。驱动力的振幅 A_0 决定了稳态振荡的振幅 $A(\omega)$，且其大小依赖于驱动力的频率 ω 和系统的自然频率 ω_0 之间的关系。

此外，当驱动力的频率 ω 接近系统的自然频率 ω_0 时，共振效应显现，导致系统振幅显著增大。

Q2: 计数估计 $T = 3.33s, \omega = 1.88rad \cdot s^{-1}$，接近外驱动力频率。

b. 同为 a.py 的输出结果，见下图



Q3: 是的，在初始时刻，两振子由于不同初始条件有着明显不同的运动状态。在足够长时间后，在外驱动力的影响下，两种初始条件的振子都会达到与外驱动力相同的运动状态。

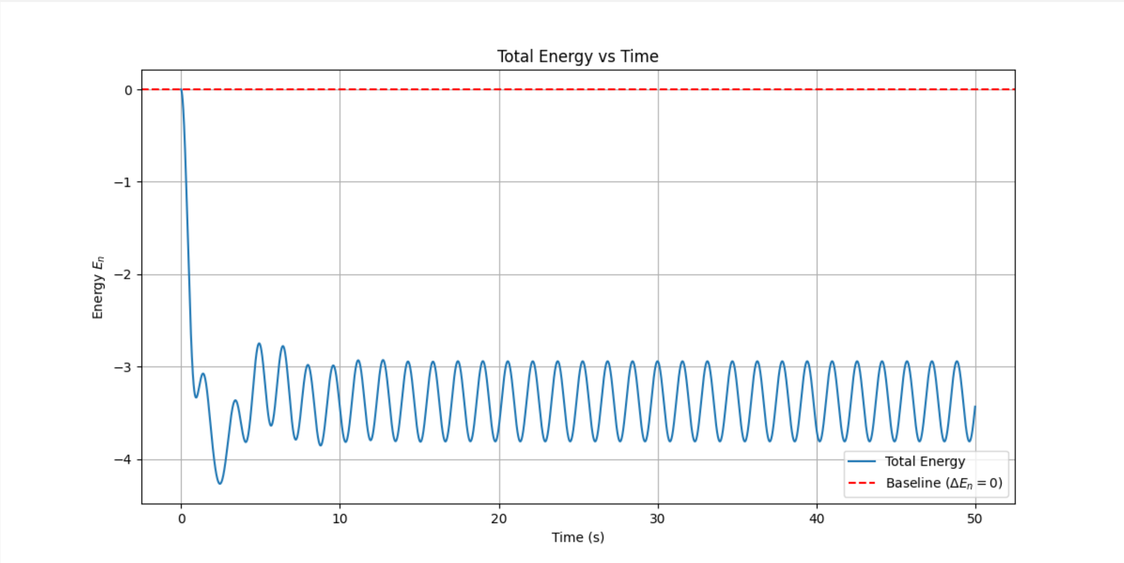
c. 参见/5/c.py

```python
import numpy as np
import matplotlib.pyplot as plt
from utils import driven_oscillator

gamma = 0.4
omega0 = 3.0
omega = 2.0
A0 = 3.1

x0, v0 = 1.0, 0.0
t_end = 50
dt = 0.01

def total_energy(x, v, omega0):
    return 0.5 * v**2 + 0.5 * omega0**2 * x**2

t, x, v = driven_oscillator(A0, omega, omega0, gamma, x0, v0, t_end, dt)
E = total_energy(x, v, omega0)
E_0 = E[0]
plt.figure(figsize=(12, 6))
plt.plot(t, E-E_0, label="Total␣Energy")
plt.axhline(0, color="red", linestyle="--", label="Baseline␣($\Delta␣E_n␣=␣0$)")
plt.xlabel("Time␣(s)")
plt.ylabel("Energy␣$E_n$")
```

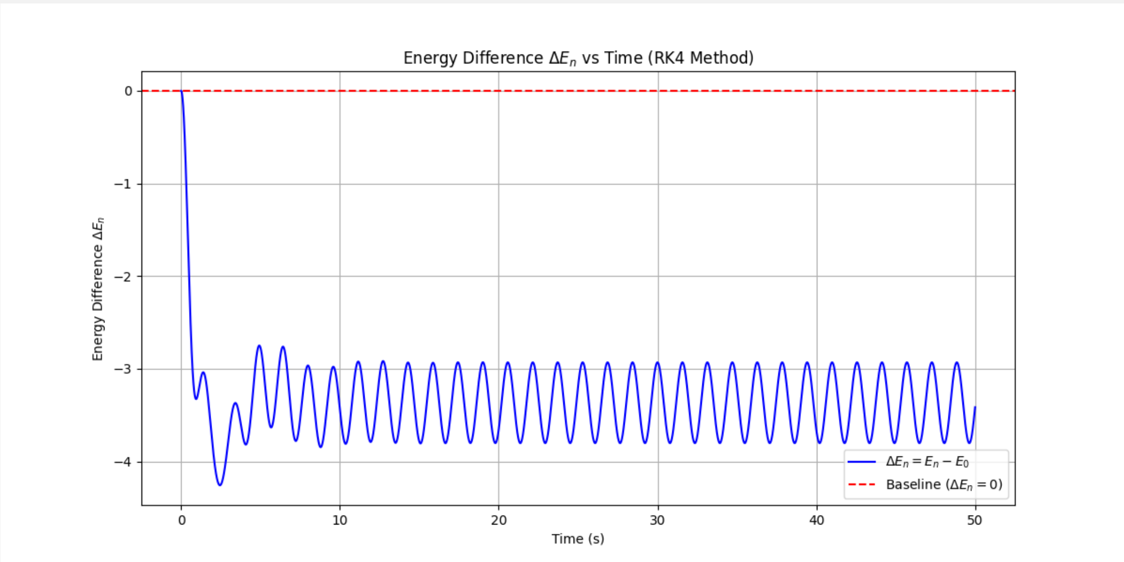
Question 3 continued on next page...

```
plt.title("Total␣Energy␣vs␣Time")
plt.legend()
plt.grid()
plt.show()
```

Q4: 能量不守恒，由于存在阻尼和外力，所以只有在外力提供被阻尼损耗掉的能量的情况下才能保持简谐振动，而振子本身能量是不守恒且随着周期性外力做周期性变化的。



Q5:



d. 参见/5/d.py

```
import numpy as np
```

Question 3 continued on next page...

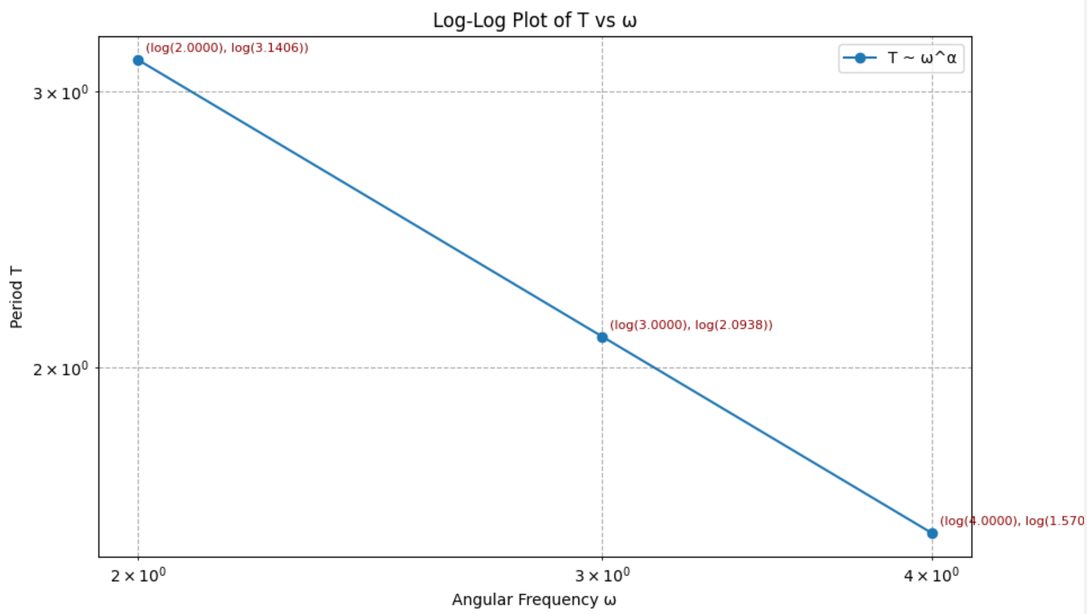
```python
import matplotlib.pyplot as plt
from scipy.stats import linregress

def compute_oscillator(t, omega0, gamma, omega, A0):
    dt = t[1] - t[0]
    x = np.zeros(len(t))
    v = np.zeros(len(t))

    for i in range(1, len(t)):
        a = A0 * np.cos(omega * t[i - 1]) - 2 * gamma * v[i - 1] - omega0 ** 2 * x[i - 1]
        v[i] = v[i - 1] + a * dt
        x[i] = x[i - 1] + v[i] * dt

    return x, v


t = np.linspace(0, 50, 5000)
gamma = 0.4
A0 = 3.1

params = [
    {"omega0": 3.0, "omega": 2.0},
    {"omega0": 4.0, "omega": 3.0},
    {"omega0": 5.0, "omega": 4.0}
]

results = []

for p in params:
    x, _ = compute_oscillator(t, p["omega0"], gamma, p["omega"], A0)
    results.append({"params": p, "x": x})

#Q6
periods = []
for i, result in enumerate(results):
    p = result["params"]
    x = result["x"]

    steady_state = x[int(len(x) * 0.8):]
    time_steady = t[int(len(x) * 0.8):]

    peaks = np.where((steady_state[1:-1] > steady_state[:-2]) & (steady_state[1:-1] >
        steady_state[2:]))[0] + 1
    peak_times = time_steady[peaks]

    if len(peak_times) > 1:
        period = np.mean(np.diff(peak_times))
```

Question 3 continued on next page...

```
    else:
        period = np.nan

    periods.append(period)
    angular_frequency = 2 * np.pi / period if period else np.nan

    print(f"Case {i + 1} ( ={p['omega0']},  ={p['omega']}):")
    print(f"  Period (T): {period:.4f}")
    print(f"  Angular Frequency ( ): {angular_frequency:.4f}")
    print()

#loglog
frequencies = [result["params"]["omega"] for result in results]
valid_periods = [p for p in periods if not np.isnan(p)]

plt.figure(figsize=(10, 6))
plt.loglog(frequencies[:len(valid_periods)], valid_periods, 'o-', label="T ~ ^ ")
for freq, period in zip(frequencies[:len(valid_periods)], valid_periods):
    plt.annotate(f"({freq:.4f}, {period:.4f})", xy=(freq, period),
                 xytext=(5, 5), textcoords="offset points", fontsize=8, color="darkred")
plt.xlabel("Angular Frequency  ")
plt.ylabel("Period T")
plt.title("Log-Log Plot of T vs  ")
plt.grid(True, which="both", linestyle="--")
plt.legend()
plt.show()
log_frequencies = np.log(frequencies)
log_periods = np.log(valid_periods)

slope, intercept, r_value, p_value, std_err = linregress(log_frequencies, log_periods)



plt.xlabel("Angular Frequency  ")
plt.ylabel("Period T")
plt.title("Log-Log Plot of T vs   with Fitted Line")
plt.grid(True, which="both", linestyle="--")
plt.legend()
plt.show()

# Print results
print(f"Slope ( ): {slope:.4f}")
print(f"Intercept: {intercept:.4f}")
print(f"R-squared: {r_value**2:.4f}")
```

运行结果为:

```
Case 1 (ω₀=3.0, ω=2.0):
  Period (T): 3.1406
  Angular Frequency (ω): 2.0006

Case 2 (ω₀=4.0, ω=3.0):
  Period (T): 2.0938
  Angular Frequency (ω): 3.0009

Case 3 (ω₀=5.0, ω=4.0):
  Period (T): 1.5703
  Angular Frequency (ω): 4.0012

Slope (α): -1.0000
Intercept: 1.8376
R-squared: 1.0000
```



直观上看 log-log 图像为直线，通过图中数据点线性拟合可得 $\alpha = -1$，且由输出结果可以看到拟合的 $R^2 = 1$，这说明拟合非常吻合。

Q6: 整理输出结果得

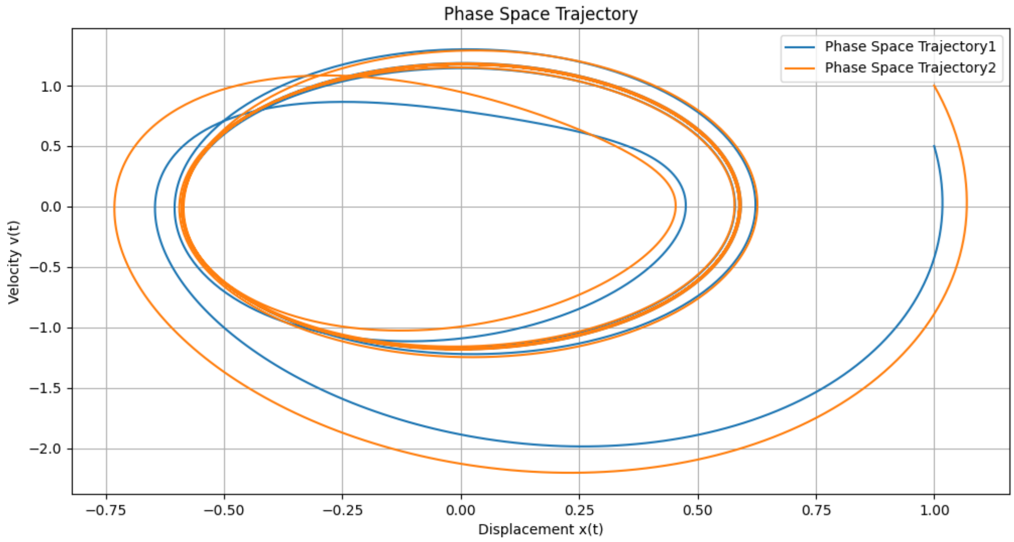
Q7: 影响稳态时频率的因素是外驱动力频率，一定时间后振子会与外驱动力达到同频。由输出结果可得 $\alpha = -1$ 和 log-log 图像

Question 3 continued on next page...

Case	ω_0	ω	Period (T)	Angular Frequency (ω)
Case 1	3.0	2.0	3.1406	2.0006
Case 2	4.0	3.0	2.0938	3.0009
Case 3	5.0	4.0	1.5703	4.0012

e. 参见/5/e.py

```
import numpy as np
import matplotlib.pyplot as plt
from utils import driven_oscillator

gamma = 0.4
omega0 = 3.0
omega = 2.8
A0 = 3.1

x0, v0 = 1.0, 0.0
t_end = 50
dt = 0.01

t, xp, vp = driven_oscillator(A0, omega, omega0, gamma, x0, 0.5, t_end, dt)
t2, x2, v2 = driven_oscillator(A0, omega, omega0, gamma, 1,1, t_end, dt)
#PRINT(T,XP,VP)
plt.figure(figsize=(12, 6))
#PLT.PLOT(X, V, LABEL="PHASE SPACE TRAJECTORY")
plt.plot(xp, vp, label=r"Phase Space Trajectory1")
plt.plot(x2, v2, label=r"Phase Space Trajectory2")
plt.xlabel("Displacement x(t)")
plt.ylabel("Velocity v(t)")
plt.title("Phase Space Trajectory")
plt.legend()
plt.grid()
plt.show()
```



图中 trajctort1 是初始条件 x=1.0，v=0.5；trajctory2 是 x=1.0，v=1.0。可见两条图线在起始点和起始之后的一段距离上相空间图像不一样，在振子与外驱动力同频后便都在相同的位置进行周期性的运动，相空间图像表现为两个重合的椭圆。由图像可以看出两种振子由初始逐渐达到与外驱动力共振德过程。

f. 参见/5/f.py

```
import numpy as np
import matplotlib.pyplot as plt
from utils import driven_oscillator

omega0 = 3
gamma = 0.5
A0 = 3.1
gamma_values = [0.5, 2.0]
omega_values = [0.5, 1.0, 2.0, 2.8, 3.0]
x0=1.0
v0=0.0
t_end = 500
dt=0.1
A_omega = []
delta_omega = []

for omega in omega_values:
    t, x, _ = driven_oscillator(A0, omega, omega0, gamma, x0, v0, t_end, dt)
    steady_state = x[int(len(x) * 0.8):]
    time_steady = t[int(len(x) * 0.8):]

```

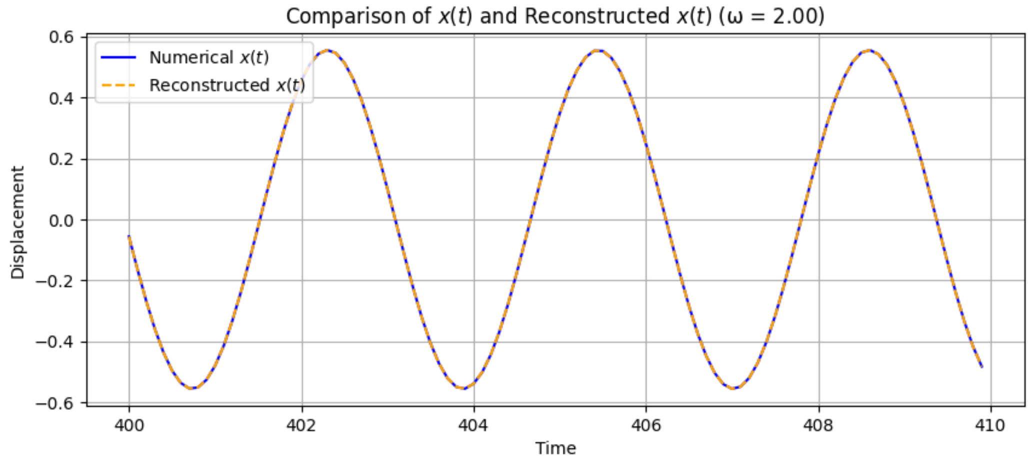
```
    amplitude = (np.max(steady_state) - np.min(steady_state)) / 2
    A_omega.append(amplitude)

    applied_force = A0 * np.cos(omega * time_steady)
    correlation = np.dot(steady_state, applied_force) / (np.linalg.norm(steady_state) * np.linalg.norm(applied_force))
    delta = -np.arccos(np.clip(correlation, -1, 1))
    delta_omega.append(delta)

    reconstructed_x = amplitude * np.cos(omega * time_steady + delta)

    plt.figure(figsize=(10, 4))
    plt.plot(time_steady[:100], steady_state[:100], label="Numerical $x(t)$", color="blue")
    plt.plot(time_steady[:100], reconstructed_x[:100], '--', label="Reconstructed $x(t)$", color="orange")
    plt.xlabel("Time")
    plt.ylabel("Displacement")
    plt.title(f"Comparison of $x(t)$ and Reconstructed $x(t)$ ( = {omega:.2f})")
    plt.legend()
    plt.grid()
    plt.show()




omega_values = [0.0, 1.0, 2.0, 2.3, 2.6, 2.9, 3.2, 3.5, 3.8]

results = {}
for gamma in gamma_values:
    A_omega = []
    delta_omega = []

    for omega in omega_values:
        _, x, _ = driven_oscillator(A0, omega, omega0, gamma, x0, v0, t_end, dt)
        steady_state = x[int(len(x) * 0.8):]
        amplitude = np.max(np.abs(steady_state))

        applied_force = A0 * np.cos(omega * t[int(len(x) * 0.8):])
        delta = -np.arccos(np.correlate(steady_state, applied_force) /
                           (np.linalg.norm(steady_state) * np.linalg.norm(applied_force)))

        A_omega.append(amplitude)
        delta_omega.append(delta[0] if isinstance(delta, np.ndarray) else delta)

    results[gamma] = {"A_omega": A_omega, "delta_omega": delta_omega}

for gamma in gamma_values:
```

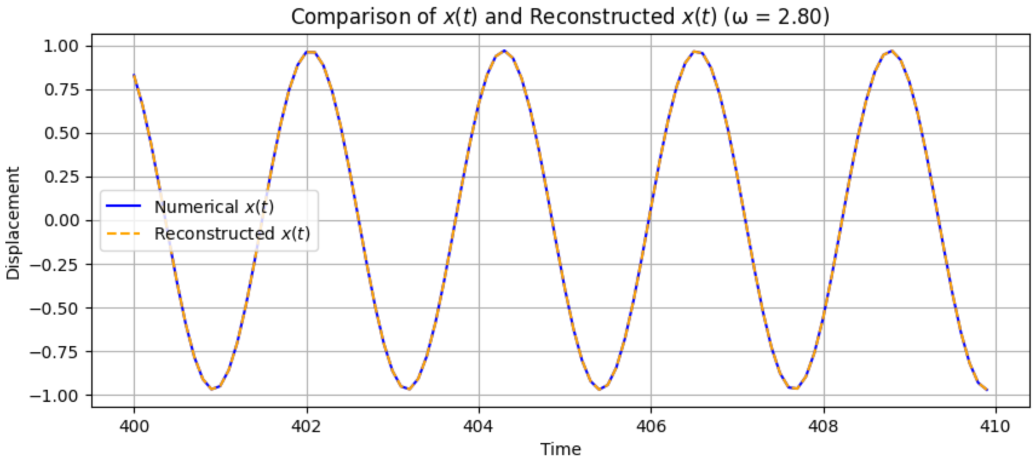
Question 3 continued on next page...

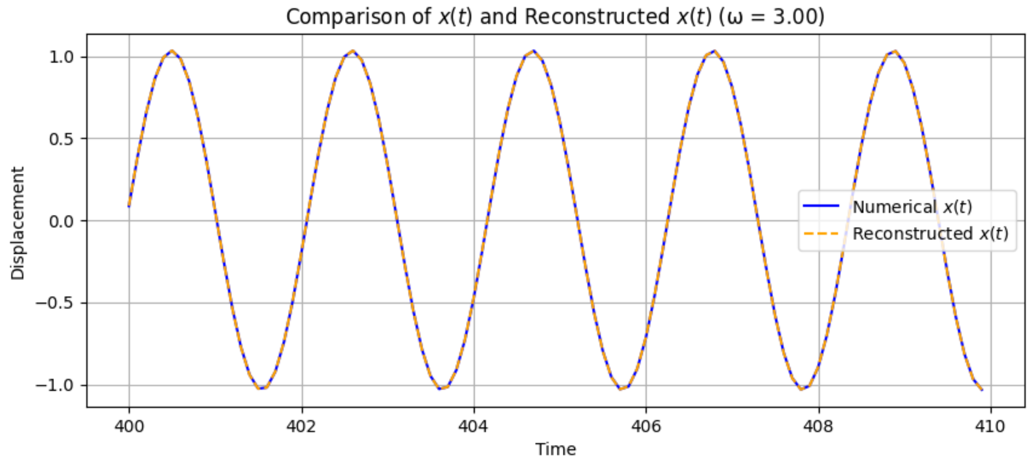
```
    plt.figure(figsize=(10, 6))

    plt.subplot(2, 1, 1)
    plt.plot(omega_values, results[gamma]["A_omega"], 'o-', label=f" = {gamma}")
    plt.xlabel("Angular Frequency  ")
    plt.ylabel("Amplitude A( )")
    plt.title(f"Amplitude vs Angular Frequency (  = {gamma})")
    plt.grid()
    plt.legend()

    plt.subplot(2, 1, 2)
    plt.plot(omega_values, results[gamma]["delta_omega"], 'o-', label=f" = {gamma}")
    plt.xlabel("Angular Frequency  ")
    plt.ylabel("Phase Difference  ( )")
    plt.title(f"Phase Difference vs Angular Frequency (  = {gamma})")
    plt.grid()
    plt.legend()

    plt.tight_layout()
    plt.show()

for gamma in gamma_values:
    A_omega = results[gamma]["A_omega"]
    max_index = np.argmax(A_omega)
    omega_m = omega_values[max_index]
    print(f"  = {gamma}: Maximum A( ) occurs at  _m = {omega_m:.2f} (natural   = {omega0})")
```

首先验证稳态的振子坐标的数值解可以表示成解析解：$x(t) = A(\omega)\cos(\omega t + \delta(\omega))$。在三组 ω 下进行对稳态时数值解和解析解的绘图，可以看到两种图线十分重合，这说明稳态时振子的数值解可以表示成上述形式。
代码输出结果如下

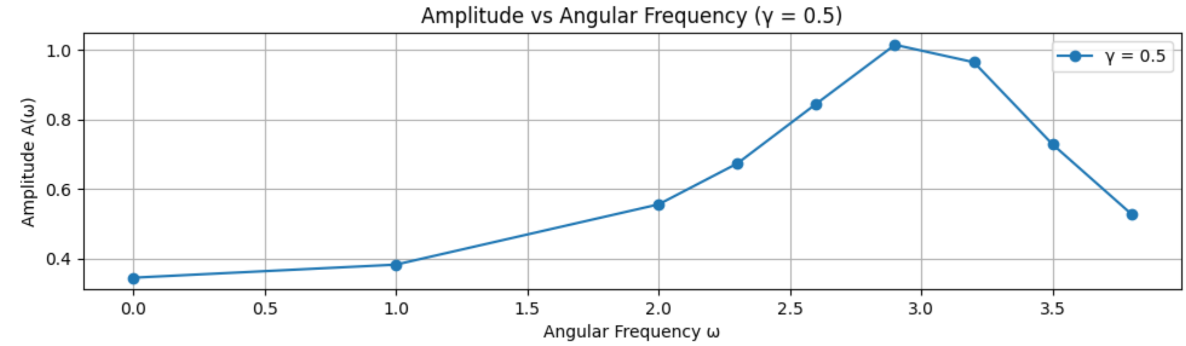
Question 3 continued on next page...

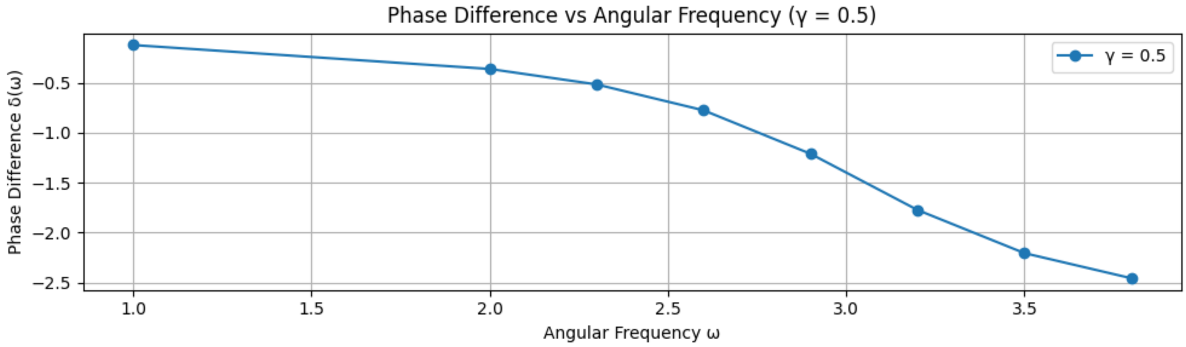


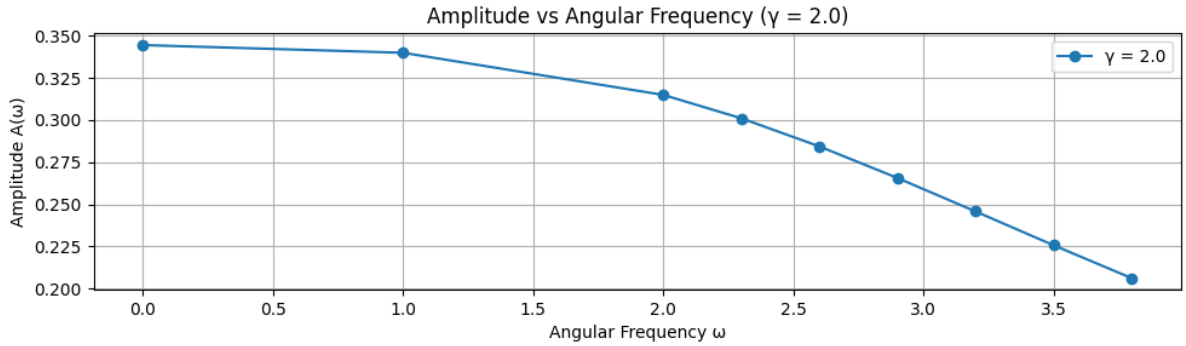


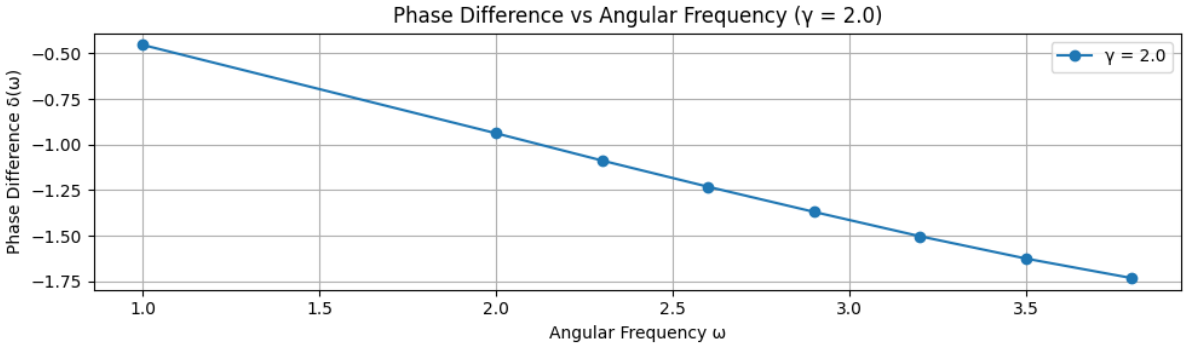


Question 3 continued on next page...










```
γ = 0.5: Maximum A(ω) occurs at ω_m = 2.90 (natural ω₀ = 3)
γ = 2.0: Maximum A(ω) occurs at ω_m = 0.00 (natural ω₀ = 3)
```

Q9: 见输出结果中的 Amplitude vs Angular Frequency 图和 Phase Difference vs Angular Frequency 图。

Question 3 continued on next page…

Q10: 见输出结果中的最后一张图，当阻尼系数为 0.5 时，存在最大振幅出现在 $\omega = 2.90$ 处，接近外驱动力频率。当阻尼系数为 2.0 时，相对来说阻尼系数较大，外驱动力不能抵抗阻尼力，所以表现出图像所示结果，即外驱动力角频率为 0 时振幅最大，而其他任意频率的外驱动力都不能抵消阻尼力，因此不能达到共振，导致振幅减小。
与无外力的阻尼振子对比，无外力的阻尼振子在最开始时的频率是与自然频率 ω_0 接近的，因此也与 ω_m 很接近，而随着阻尼耗散，频率会指数形降低最后静止。