问题1：计算浮子、振子的垂荡位移和速度

In [651…

```
# TODO import

import re
import os
import sys
import hmz
import scipy
import pathlib
import mitosheet
import numpy as np
import pandas as pd

import time
import copy
import random

import sympy
from sympy import limit
from sympy import diff
from sympy import integrals

import sklearn
import graphviz
from sklearn import tree
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error as MSE
from sklearn.metrics import mean_absolute_error as MAE
from sklearn.metrics import classification_report, roc_auc_score

import sko

import plotly
import plotly.express as px
import plotly.graph_objects as go
import plotly.figure_factory as ff
plotly.offline.init_notebook_mode()

import chart_studio
```

```
import chart_studio.plotly as py
from chart_studio.plotly import plot, iplot
chart_studio.tools.set_credentials_file(username='zhiliao0824', api_key='qrGJtJRojwpsVJ3nosn0')

import cufflinks as cf
cf.set_config_file(
    offline=True,
    world_readable=True,
    theme='pearl',  # cf.getThemes()
)

import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']  # KaiTi
plt.rcParams['axes.unicode_minus'] = False

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'
# InteractiveShell.ast_node_interactivity = 'last'

import cv2 as cv

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as Data
from torch.utils.data import DataLoader
from torch.utils.data.dataset import Dataset

import pylatex
import latexify

import warnings
warnings.filterwarnings("ignore")
```

In [652…

```
# TODO DIR
ROOTDIR = pathlib.Path(os.path.abspath('.'))
IMG_HTML = ROOTDIR / 'img-html'
IMG_SVG = ROOTDIR / 'img-svg'
DATA_RAW = ROOTDIR / 'data-raw'
DATA_COOKED = ROOTDIR / 'data-processed'
```

In [653…

```
# TODO 4
浮子质量 = 4866  # kg
```

```
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m
```

In [654…

```
# TODO 3
question1234 = 1
if question1234 == 1:
    # 问题1: 参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2: 参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
elif question1234 == 3:
    # 问题3: 参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    纵摇附加转动惯量 = 7001.914
    垂荡兴波阻尼系数 = 683.4558
    纵摇兴波阻尼系数 = 654.3383
    垂荡激励力振幅 = 3640
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4: 参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
```

```
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140
```

In [655…

```
# TODO parameters func
def S浮子底面积_func(r=浮子底半径):
    return np.pi * r**2
S浮子底面积 = S浮子底面积_func()

def V排_func(h, pprint=True):
    """
    :param h: 圆柱壳体的入水深度
    :param pprint: 是否打印状态
    :return: V排 (m^3)
    """
    if h >= 0:
        print("圆锥壳体完全浸没")
        V排 = (1/3 * S浮子底面积 * 浮子圆锥部分高度) + (S浮子底面积 * h)
    else:
        print("圆锥壳体漂浮")
        depth = 浮子圆锥部分高度 + h
        r = 浮子底半径 * depth / 浮子圆锥部分高度
        V排 = 1/3 * S浮子底面积_func(r) * depth
    return V排
# print("浮子入水体积: ", V排_func(3))
# print("浮子入水体积: ", V排_func(2.4147))
# print("浮子入水体积: ", V排_func(0))
# print("浮子入水体积: ", V排_func(-0.001))
# print("浮子入水体积: ", V排_func(-0.8))

def F静水恢复力_func(h, pprint=False):
    """ 类似(就是)浮力 方向向上
    :param h: 圆柱壳体的入水深度
    :param pprint: 是否打印状态
    :return: F静水恢复力 (N)
    """
    F静水恢复力 = 海水的密度 * 重力加速度 * V排_func(h, pprint)
    return F静水恢复力
# F静水恢复力_func(2.4147)

def F兴波阻尼力_func(v, k=垂荡兴波阻尼系数):
    """ 方向同速度方向
    :param v: 速度
    :return:
```

```
    """
    F兴波阻尼力 = k * v
    return F兴波阻尼力

def F波浪激励力_func(t, omega=入射波浪频率, f=垂荡激励力振幅):
    """ 方向向上
    :param t: 时间
    :return: F波浪激励力 (N)
    """
    F波浪激励力 = f * np.cos(omega * t)
    return F波浪激励力
# F波浪激励力_func(0)

def F附加惯性力_func(m=垂荡附加质量, g=重力加速度):
    """ 方向向下 """
    F附加惯性力 = m * g
    return F附加惯性力
F附加惯性力 = F附加惯性力_func()

def F重力_func(m=浮子质量+振子质量, g=重力加速度):
    """ 方向向下 """
    F重力 = m * g
    return F重力
F重力 = F重力_func()

def c直线阻尼器的阻尼系数_func1():
    c直线阻尼器的阻尼系数 = 10000  # N·s/m
    return c直线阻尼器的阻尼系数

def c直线阻尼器的阻尼系数_func2(v浮子, v振子, k=10000, a=0.5):
    c直线阻尼器的阻尼系数 = k * abs(v浮子 - v振子)**a  # N·s/m
    return c直线阻尼器的阻尼系数
```

确定初始 $V_{排}$

In [656…

```
# # TODO 初始值（暂时注释）
# # 初始:              F重力 = F静水恢复力
# # 过程: F附加惯性力 + F重力 = F静水恢复力 + F兴波阻尼力 + F波浪激励力

# # 方法1
# V排 = (0 * F附加惯性力 + F重力) / (海水的密度 * 重力加速度)
# print(V排)
# h0 = V排 / np.pi - 0.8 / 3
# print(h0)
```

```
# print('-' * 100)

# # 方法2
# from scipy.optimize import minimize, root
# root_res = root(lambda x: F静水恢复力_func(x, False) - F重力, 2)
# print(root_res)
# h0 = root_res.x[0]
# print(h0)
```

In [657…

```
# TODO 暂时没用
# 时间间隔 = 0.2
# 波浪频率 = 入射波浪频率
# 波浪周期 = 1 / 波浪频率

# t01, t001, t0001 = [0.1, 0.001, 0.0001]
# for t in np.arange(0, 40 * 波浪周期):  # [10, 20, 40, 60, 100]  # s
#     ...
```

根据微分方程确定浮子和振子的位移方程

$X_1(t)$ 浮子位移，$X_2(t)$ 振子位移

$$m_2\frac{d^2X_2(t)}{dt^2}+c\frac{dX_2(t)}{dt}+kX_2(t)=c\frac{dX_1(t)}{dt}+kX_1(t)$$
$$m_1\frac{d^2X_1(t)}{dt^2}+m_2\frac{d^2X_2(t)}{dt^2}=F(t)$$

m_1 为定子与浮体总质量， kg；

m_2 为动子质量， kg；

$F(t)$ 为垂向波浪力， N；

k 为弹簧的劲度系数，N/m；

c 为动子与定子之间的阻尼系数

$F(t)=F_{波浪激励力}+F_{兴波阻尼力}+F_{静水恢复力}-F_{附加惯性力}+F_{浮力}-F_{重力}$

$F(t)=fcos(\omega t)+k_{垂荡兴波阻尼系数}v+\rho gSh-m_{垂荡附加质量}a$

$F(t)=fcos(\omega t)+k_{垂荡兴波阻尼系数}\frac{dX_1(t)}{dt}+\rho gSX_1(t)-m_{垂荡附加质量}\frac{d^2X_1(t)}{dt^2}$

$$F(t) = fcos(\omega t) + k_{垂荡兴波阻尼系数}\frac{dX_1(t)}{dt} + k_{系数}X_1(t) + k_{垂荡附加质量}\frac{d^2X_1(t)}{dt^2}$$

解析解（没前途）

In [658…
```
# X1, X2 = sympy.symbols("X1 X2")
m1, m2, F, k, c = sympy.symbols("m1 m2 F k c")
```

In [659…
```
# #（没前途）sympy 解微分方程的解析解：X1(t)、X2(t)

# m1, m2, k, c = sympy.symbols("m1 m2 k c")

# t = sympy.symbols('t')
# X1 = sympy.symbols('X1', cls=sympy.Function)
# X2 = sympy.symbols('X2', cls=sympy.Function)
# F = sympy.symbols('F', cls=sympy.Function)

# eq1 = sympy.Eq(
#     m2 * X2(t).diff(t, 2) + c * X2(t).diff(t, 1) + k * X2(t),
#     c * X1(t).diff(t, 1) + k * X1(t))
# eq2 = sympy.Eq(m1 * X1(t).diff(t, 2) + m2 * X2(t).diff(t, 2), F(t))
# eq = (eq1, eq2)

# t0 = time.time()
# res = sympy.dsolve(eq)
# print("用时：", time.time() - t0)

# print(sympy.latex(res))
```

In [660…
```
# #（没前途）sympy 解微分方程的解析解：X1(t)、X2(t)

# m1, m2, F, k, c = sympy.symbols("m1 m2 F k c")
# f, omega, k兴, k系, k重 = sympy.symbols("f omega k兴 k系 k重")

# t = sympy.symbols('t')
# X1 = sympy.symbols('X1', cls=sympy.Function)
# X2 = sympy.symbols('X2', cls=sympy.Function)

# eq1 = sympy.Eq(
#     m2 * X2(t).diff(t, 2) + c * X2(t).diff(t, 1) + k * X2(t).diff(t, 0),
#     c * X1(t).diff(t, 1) + k * X1(t).diff(t, 0)
# )
# eq2 = sympy.Eq(
#     m1 * X1(t).diff(t, 2) + m2 * X2(t).diff(t, 2),
```

```
#     f * sympy.cos(omega * t) + k兴 * X2(t).diff(t, 1) + k系 * X2(t).diff(t, 0) - k重
# )
# eq = (eq1, eq2)

# t0 = time.time()
# res = sympy.dsolve(eq)
# print("用时：", time.time() - t0)

# print(sympy.latex(res))
```

In [661…

```
# # （没前途）sympy 解微分方程的解析解：X1(t)、X2(t)

# TODO step1：求解析解
# m1, m2, F, k, c = sympy.symbols("m1 m2 F k c")
# f, omega, k兴, k系, k重 = sympy.symbols("f omega k兴 k系 k重")

# m1 = 浮子质量
# m2 = 振子质量
# k = 弹簧刚度
# c直线阻尼器的阻尼系数 = c直线阻尼器的阻尼系数_func1()
# c = c直线阻尼器的阻尼系数
# f = 垂荡激励力振幅
# omega = 入射波浪频率
# k兴 = 垂荡兴波阻尼系数
# k系 = 海水的密度 * 重力加速度 * S浮子底面积
# k重 = F附加惯性力 + F重力

# t = sympy.symbols('t')
# X1 = sympy.symbols('X1', cls=sympy.Function)
# X2 = sympy.symbols('X2', cls=sympy.Function)

# eq1 = sympy.Eq(
#     m2 * X2(t).diff(t, 2) + c * X2(t).diff(t, 1) + k * X2(t).diff(t, 0),
#     c * X1(t).diff(t, 1) + k * X1(t).diff(t, 0)
# )
# eq2 = sympy.Eq(
#     m1 * X1(t).diff(t, 2) + m2 * X2(t).diff(t, 2),
#     f * sympy.cos(omega * t) + k兴 * X2(t).diff(t, 1) + k系 * X2(t).diff(t, 0) - k重
# )
# eq = (eq1, eq2)


# t0 = time.time()
# res = sympy.dsolve(eq)
# print("用时：", time.time() - t0)
```

```
# print(sympy.latex(res))

# TODO step1: 带入具体数值求解
# # F = (F静水恢复力 + F兴波阻尼力 + F波浪激励力) - (F附加惯性力 + F重力)
# c直线阻尼器的阻尼系数 = c直线阻尼器的阻尼系数_func1()
# mapping = [
#     (m1, 浮子质量),
#     (m2, 振子质量),
#     (k, 弹簧刚度),
#     (c, c直线阻尼器的阻尼系数),
#     (f, 垂荡激励力振幅),
#     (omega, 入射波浪频率),
#     (k兴, 垂荡兴波阻尼系数),
#     (k系, 海水的密度 * 重力加速度 * S浮子底面积),
#     (k重, F附加惯性力 + F重力),
# ]
# res.subs(mapping)
```

数值解

旧的:

$$F(t) = F_{\text{波浪激励力}} + F_{\text{兴波阻尼力}} + F_{\text{静水恢复力}} - F_{\text{附加惯性力}}$$

$$F(t) = fcos(\omega t) + k_{\text{垂荡兴波阻尼系数}}\frac{dX_1(t)}{dt} + k_{\text{系数}}X_1(t) + k_{\text{垂荡附加质量}}\frac{d^2X_1(t)}{dt^2}$$

新的:

$$F(t) = F_{\text{波浪激励力}} - F_{\text{兴波阻尼力}} + F_{\text{静水恢复力}}$$

$$F(t) = fcos(\omega t) + k_{\text{垂荡兴波阻尼系数}}\frac{dX_1(t)}{dt} + k_{\text{系数}}X_1(t)$$

$$y' = \begin{bmatrix} y_1' \\ y_2' \\ y_3' \\ y_4' \end{bmatrix} = \begin{bmatrix} \dot{X}_1 \\ \ddot{X}_1 \\ \dot{X}_2 \\ \ddot{X}_2 \end{bmatrix} = \begin{bmatrix} \dot{X}_1 \\ (fcos(\omega t) + k_1\dot{X}_1 + k_2X_1 - m_2\ddot{X}_2)/m_1' \\ \dot{X}_2 \\ (c(\dot{X}_1 - \dot{X}_2) + k(X_1 - X_2))/m_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ (fcos(\omega t) + k_1y_2 + k_2y_1 - m_2\ddot{X}_2)/m_1' \\ y_4 \\ (c(y_2 - y_4) + k(y_1 - y_3))/m_2 \end{bmatrix}$$

其中 $m_1' = m_1 + m_{\text{垂荡附加质量}}$

(1) $c = 10000$

(2) $c = 10000|V|^{0.5} = 10000|V_1 - V_2|^{0.5} = 10000|\dot{X}_1 - \dot{X}_2|^{0.5}$

In [662…

```
import numpy as np
from scipy.integrate import odeint, ode, solve_ivp

import matlab.engine
# matlab_engine = matlab.engine.start_matlab()
```

(1) c 常数

$c = 10000$

常量，画图函数

In [663…

```
# TODO 数值解

m1 = 浮子质量
m2 = 振子质量
m1_ = 浮子质量 + 垂荡附加质量
m2_ = 振子质量 + 垂荡附加质量
k = 弹簧刚度
c直线阻尼器的阻尼系数 = c直线阻尼器的阻尼系数_func1()
c = c直线阻尼器的阻尼系数
f = 垂荡激励力振幅
omega = 入射波浪频率
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积
# k1 = k2 = k3 = 0  # 只有激励力

def plot_mat(t, y):
    plt.figure(dpi=100)
    plt.plot(t, y[:, 0], label="浮子位移" + "$X1$")
    plt.plot(t, y[:, 1], label="浮子速度" + "$V1$")
    plt.plot(t, y[:, 2], label="振子位移" + "$X2$")
    plt.plot(t, y[:, 3], label="振子速度" + "$V2$")
    plt.legend()
#     plt.grid()
    plt.show()
    return None

def plot_plotly(t, y, title, svg_name=None):
    trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子  X1$", yaxis='y1')
    trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子  V1$", yaxis='y2')
```

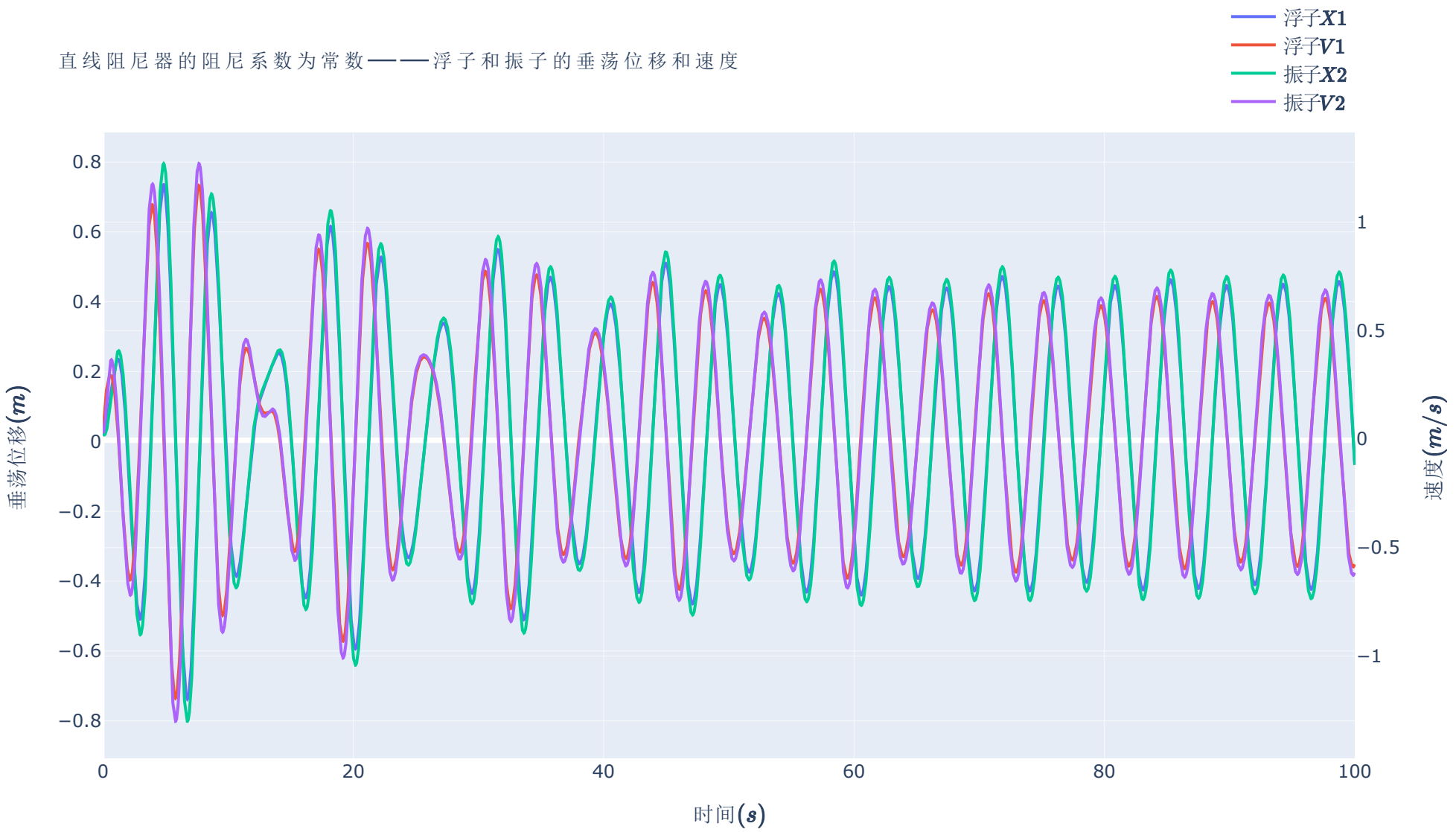
```
    trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子  X2$", yaxis='y1')
    trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子  V2$", yaxis='y2')
#     trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子位移X1$", line={"width": 1})
#     trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子速度V1$", line={"width": 1})
#     trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子位移X2$", line={"width": 1})
#     trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子速度V2$", line={"width": 1})
#     fig = go.Figure(data=[trace1, trace3])
    fig = go.Figure(data=[trace1, trace2, trace3, trace4])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$垂荡位移 (m)$'),
        yaxis2=dict(title='$速度 (m/s)$', anchor='x', overlaying='y', side='right'),
        legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
        title=title,
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name)
    fig.show()
    return None
```

浮子和振子的垂荡位移和速度

In [664…

```
def diff_equation(ys, t):
    y1 = y2 = y3 = y4 = 0

    y1 = ys[2-1]
    y3 = ys[4-1]

    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2

    y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - (c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1]))
    y2 = y2 / m1_
    return [y1, y2, y3, y4]

时间间隔 = 0.005
_l, _r = 0, 100
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0, 0, 0, 0]
result1_1 = odeint(diff_equation, y0, t)

# plot_mat(t, result1_1)
plot_plotly(t, result1_1,
```

```
        '$直线阻尼器的阻尼系数为常数—浮子和振子的垂荡位移和速度$',
        "问题1-c常数：浮子振子位移速度图.svg")
```



In [ ]:

查看功率

In [665…

```
# TODO 查看功率
```

```
def get_power1(diff_t=0.01):
    c = 10000
    def diff_equation(ys, t):
        y1 = y2 = y3 = y4 = 0

        y1 = ys[2-1]
        y3 = ys[4-1]

        y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
        y4 = y4 / m2

        y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - (c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1]))
        y2 = y2 / m1_
        return [y1, y2, y3, y4]

    时间间隔 = diff_t
    _l, _r = 0, 100
    t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
    y0 = [0, 0, 0, 0]
    result1_1 = odeint(diff_equation, y0, t)

    delta = c * abs(result1_1[:, 1] - result1_1[:, 3])
#     power_i = delta[1:] * 时间间隔  # 矩形
    power_i = (delta[1:] + delta[:-1]) * 时间间隔 / 2  # 梯形
    power = power_i[int(60/时间间隔): int(100/时间间隔)]
    P = power.sum() / 40
    return P
get_power1()
```

Out[665]: 240.5413505711107

保存结果

In [565…

```
# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result1_1_df(t=t, result1_1=result1_1):
    columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result1_1 = pd.DataFrame(result1_1, columns=columns[1:])
    result1_1_df = pd.concat([shijian, result1_1], axis=1)
    return result1_1_df
def save_result1_1_df(result1_1_df=get_result1_1_df()):
    # 40 周期，间隔 0.2
```

```
    result1_1 = result1_1_df.iloc[:int(40 * 波浪周期 / 时间间隔) + 1, :]
    result1_1.to_csv('result1-1.csv', encoding='utf_8_sig')
    #  10 s、20 s、40 s、60 s、100 s
    idx = list(map(lambda x: x * 5, [10, 20, 40, 60, 100]))
    result1_1_paper = result1_1_df.iloc[idx, :]
    result1_1_paper.to_csv('result1-1-paper.csv', encoding='utf_8_sig')
    return None

# save_result1_1_df()
```

观察相对位移和相对速度

In [566…

```
# # TODO 准备问题2：观察相对位移
# result1_1_df = get_result1_1_df()
# result1_1_df.iloc[:, [1, 3]].iplot(kind='spread')

# # TODO 准备问题2：观察相对速度
# result1_1_df = get_result1_1_df()
# result1_1_df.iloc[:, [2, 4]].iplot(kind='spread')
```

仿真结果对比数值结果

In [567…

```
# TODO 仿真结果对比数值结果
def plot_fzj_szj(title, svg_name=None):
    # 仿真
    data_fangzhenjie = pd.read_excel(DATA_COOKED / 'data500.xlsx')
    data_fangzhenjie

    cond1 = abs(data_fangzhenjie.iloc[:, 0] - 70) < 0.1
    idx_min = data_fangzhenjie[cond1].index[0]
    cond2 = abs(data_fangzhenjie.iloc[:, 0] - 140) < 0.1
    idx_max = data_fangzhenjie[cond2].index[0]

    plot_data_fangzhenjie_time = data_fangzhenjie.iloc[idx_min: idx_max, 0]
    plot_data_fangzhenjie = data_fangzhenjie.iloc[idx_min: idx_max, 1]

    # 时间间隔 = 0.2
    _l, _r = 0, 200
    t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
    y0 = [0, 0, 0, 0]
    data_shuzhijie = odeint(diff_equation, y0, t)

#     columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
#     shijian = pd.DataFrame(t, columns=columns[:1])
```

```
#     data_shuzhijie = pd.DataFrame(data_shuzhijie, columns=columns[1:])
#     data_shuzhijie = pd.concat([shijian, data_shuzhijie], axis=1)

    plot_data_shuzhijie_time = t[70*5:140*5+1]
    plot_data_shuzhijie = data_shuzhijie[70*5:140*5+1, 0] - data_shuzhijie[70*5:140*5+1, 2]

    trace1 = go.Scatter(
        x=plot_data_fangzhenjie_time, y=plot_data_fangzhenjie,
        name='$仿真解$',
    )
    trace2 = go.Scatter(
        x=plot_data_shuzhijie_time, y=plot_data_shuzhijie,
        name='$数值解$',
    )

    fig = go.Figure(data=[trace1, trace2])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$相对位移 (m)$'),
        legend=dict(y=1.13, yanchor="top", x=1, xanchor="right"),
        title=title,
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name)
    fig.show()
    del fig
    return None
```

In [568…
```
# plot_fzj_szj(
#     "$直线阻尼器的阻尼系数为常数—浮子和振子的相对垂荡位移$",
#     "问题1-c常数：浮子振子相对位移图.svg")
```

In [ ]:
```
```

（2）c 非常数

$$c = 10000|V|^{0.5} = 10000|V_1 - V_2|^{0.5} = 10000|\dot{X}_1 - \dot{X}_2|^{0.5}$$

画图函数

In [569…
```
# TODO plot
```

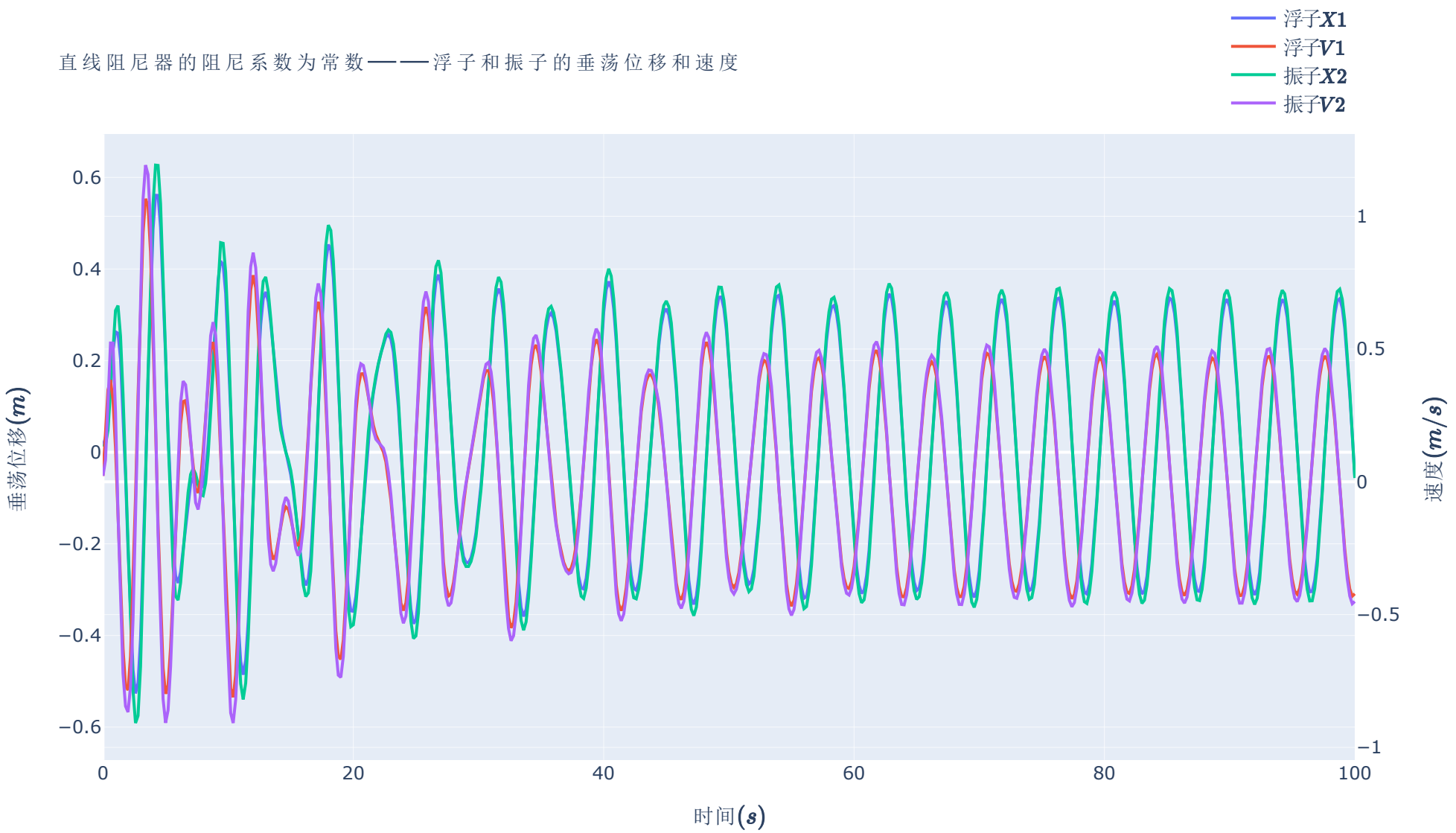
```
def plot_mat(t, y):
    plt.figure(dpi=100)
    plt.plot(t, y[:, 0], label="浮子位移" + "$X1$")
    plt.plot(t, y[:, 1], label="浮子速度" + "$V1$")
    plt.plot(t, y[:, 2], label="振子位移" + "$X2$")
    plt.plot(t, y[:, 3], label="振子速度" + "$V2$")
    plt.legend()
#     plt.grid()
    plt.show()
    return None
def plot_plotly(t, y, title, svg_name=None):
    trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子X1$", yaxis='y1')
    trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子V1$", yaxis='y2')
    trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子X2$", yaxis='y1')
    trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子V2$", yaxis='y2')
#     trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子位移X1$", line={"width": 1})
#     trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子速度V1$", line={"width": 1})
#     trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子位移X2$", line={"width": 1})
#     trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子速度V2$", line={"width": 1})
#     fig = go.Figure(data=[trace1, trace3])
    fig = go.Figure(data=[trace1, trace2, trace3, trace4])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$垂荡位移 (m)$'),
        yaxis2=dict(title='$速度 (m/s)$', anchor='x', overlaying='y', side='right'),
        legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
        title=dict(text=title),
#         titlefont=dict(size=50),
#         font=dict(
#             family="Courier New, monospace", # 所有标题文字的字体
#             size=24,                         # 所有标题文字的大小
#             color="RebeccaPurple"            # 所有标题的颜色
#         ),
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name, )
    fig.show()
    return None
```

浮子和振子的垂荡位移和速度

In [570…

```
# TODO 数值解
def diff_equation(ys, t):
```

```
    y1 = y2 = y3 = y4 = 0

    y1 = ys[2-1]
    y3 = ys[4-1]

    Ft = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1]
    y2 = Ft - m2 * y4
    y2 = y2 / m1_

    c = c直线阻尼器的阻尼系数_func2(ys[2-1], ys[4-1], k=10000, a=0.5)
    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2
    return np.array([y1, y2, y3, y4])

时间间隔 = 0.2
_l, _r = 0, 100
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0, 0, 0, 0]
result1_2 = odeint(diff_equation, y0, t)
# plot_mat(t, result1_2)
plot_plotly(t, result1_2,
            '$直线阻尼器的阻尼系数为常数—浮子和振子的垂荡位移和速度$',
            "问题1-c非常数：浮子振子位移速度图.svg")
```



查看功率

In [572…

```python
# TODO 查看功率
def get_power2(diff_t=0.01):

    def diff_equation(ys, t):
        y1 = y2 = y3 = y4 = 0
```

```
        y1 = ys[2-1]
        y3 = ys[4-1]

        Ft = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1]
        y2 = Ft - m2 * y4
        y2 = y2 / m1_

        c = c直线阻尼器的阻尼系数_func2(ys[2-1], ys[4-1], k=10000, a=0.5)
        y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
        y4 = y4 / m2
        return np.array([y1, y2, y3, y4])

    时间间隔 = diff_t
    _l, _r = 0, 100
    t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
    y0 = [0, 0, 0, 0]
    result1_2 = odeint(diff_equation, y0, t)

    c = 10000 * abs(result1_2[:, 1] - result1_2[:, 3])**0.5
    delta = c * abs(result1_2[:, 1] - result1_2[:, 3])

#     power_i = delta[1:] * 时间间隔  # 矩形
    power_i = (delta[1:] + delta[:-1]) * 时间间隔 / 2  # 梯形
    power = power_i[int(60/时间间隔): int(100/时间间隔)]
    P = power.sum() / 40
    return P
get_power2()
```

Out[572]: 27.180567841500487

保存结果

In [ ]:
```
# TODO 保存结果
# 时间间隔 = 0.2
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result1_2_df(t=t, result1_2=result1_2):
    columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result1_2 = pd.DataFrame(result1_2, columns=columns[1:])
    result1_2_df = pd.concat([shijian, result1_2], axis=1)
    return result1_2_df
def save_result1_2_df(result1_2_df=get_result1_2_df()):
    # 40 周期，间隔 0.2
```

```
    result1_2 = result1_2_df.iloc[:int(40 * 波浪周期 / 时间间隔) + 1, :]
    result1_2.to_csv('result1-2.csv', encoding='utf_8_sig')
    #  10 s、20 s、40 s、60 s、100 s
    idx = list(map(lambda x: x * 5, [10, 20, 40, 60, 100]))
    result1_2_paper = result1_2_df.iloc[idx, :]
    result1_2_paper.to_csv('result1-2-paper.csv', encoding='utf_8_sig')
    return None

save_result1_2_df()
```

观察相对速度和相对速度

In [574…

```
# # TODO 准备问题2：观察相对速度
# result1_2_df = get_result1_2_df()
# result1_2_df.iloc[:, [1, 3]].iplot(kind='spread')

# # TODO 准备问题2：观察相对速度
# result1_2_df = get_result1_2_df()
# result1_2_df.iloc[:, [2, 4]].iplot(kind='spread')
```

仿真结果对比数值结果

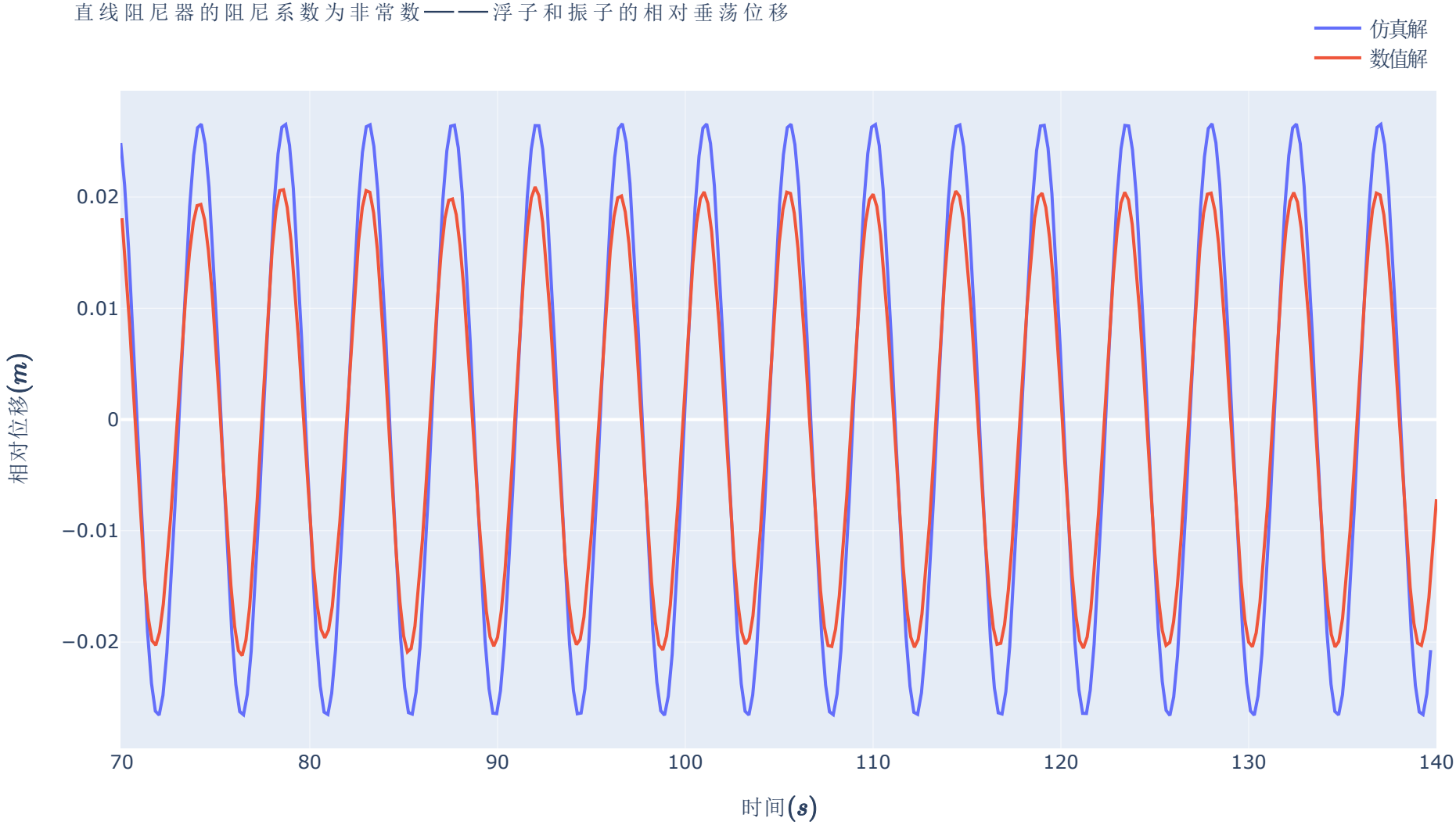
In [575…

```
# TODO 仿真结果对比数值结果
def plot_fzj_szj(title, svg_name=None):
    # 仿真
    data_fangzhenjie = pd.read_excel(DATA_COOKED / 'data500-diff_c.xlsx')
    data_fangzhenjie

    cond1 = abs(data_fangzhenjie.iloc[:, 0] - 70) < 0.1
    idx_min = data_fangzhenjie[cond1].index[0]
    cond2 = abs(data_fangzhenjie.iloc[:, 0] - 140) < 0.1
    idx_max = data_fangzhenjie[cond2].index[0]

    plot_data_fangzhenjie_time = data_fangzhenjie.iloc[idx_min: idx_max, 0]
    plot_data_fangzhenjie = data_fangzhenjie.iloc[idx_min: idx_max, 1]

    # 时间间隔 = 0.2
    _l, _r = 0, 200
    t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
    y0 = [0, 0, 0, 0]
    data_shuzhijie = odeint(diff_equation, y0, t)

#     columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
#     shijian = pd.DataFrame(t, columns=columns[:1])
```

```
#     data_shuzhijie = pd.DataFrame(data_shuzhijie, columns=columns[1:])
#     data_shuzhijie = pd.concat([shijian, data_shuzhijie], axis=1)

    plot_data_shuzhijie_time = t[70*5:140*5+1]
    plot_data_shuzhijie = data_shuzhijie[70*5:140*5+1, 0] - data_shuzhijie[70*5:140*5+1, 2]

    trace1 = go.Scatter(
        x=plot_data_fangzhenjie_time, y=plot_data_fangzhenjie,
        name='$仿真解$',
    )
    trace2 = go.Scatter(
        x=plot_data_shuzhijie_time, y=plot_data_shuzhijie,
        name='$数值解$',
    )

    fig = go.Figure(data=[trace1, trace2])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$相对位移 (m)$'),
        legend=dict(y=1.13, yanchor="top", x=1, xanchor="right"),
        title=title,
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name)
    fig.show()
    del fig
    return None
```

In [576…

```
plot_fzj_szj(
    "$直线阻尼器的阻尼系数为非常数—浮子和振子的相对垂荡位移$",
    "问题1-c非常数：浮子振子相对位移图.svg")
```



In [ ]:

问题2

In [112…

```
# TODO 3
question1234 = 2
```

```
if question1234 == 1:
    # 问题1：参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2：参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
elif question1234 == 3:
    # 问题3：参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    纵摇附加转动惯量 = 7001.914
    垂荡兴波阻尼系数 = 683.4558
    纵摇兴波阻尼系数 = 654.3383
    垂荡激励力振幅 = 3640
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4：参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140
```

In [112…

```
from sko.GA import GA
```

平均输出功率：$P = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} F_G \Delta v dt = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} c \Delta v^2 dt$

In [112…

```
# 时间间隔 = 0.01
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率
```

```
m1 = 浮子质量
m2 = 振子质量
m1_ = 浮子质量 + 垂荡附加质量
m2_ = 振子质量 + 垂荡附加质量
k = 弹簧刚度
c直线阻尼器的阻尼系数 = c直线阻尼器的阻尼系数_func1()
c = c直线阻尼器的阻尼系数
f = 垂荡激励力振幅
omega = 入射波浪频率
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积
# k1 = k2 = k3 = 0  # 只有激励力
```

(1) c 常量

$c = 10000$

In [ ]:

In [108…

```
# TODO 数值解：在 ode 里面计算 power_i
def diff_equation(ys, t, c, k=弹簧刚度):
    y1 = ys[2-1]
    y3 = ys[4-1]

    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2

    y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - m2 * y4
    y2 = y2 / m1_
    return [y1, y2, y3, y4]
def GA_func(c, period=2, pprint=True):
    """ sko 遗传目标函数
    :param zunixishu: 阻尼系数/比例系数
    :param power_i: list
    :param period: 计算平均功率的时间区间类型
    """
#     if period == 1:  # 40T: 0~28.5 s
#         begin = 0
#         end = int(40 * 波浪周期 / 时间间隔)
#     elif period == 2:  # 稳定: 60~100 s
#         begin = int(60 / 时间间隔)
#         end = int(100 / 时间间隔)
#     elif period == 3:  # 所有: 0~100 s
```

```
#         begin = 0
#         end = -2

    时间间隔 = 0.5
    _l, _r = 0, 100

    res = odeint(
        diff_equation,
        [0, 0, 0, 0],
        np.linspace(_l, _r, num=int(_r / 时间间隔) + 1),
        args=(c, )
    )  # zunixishu
    delta = abs(res[:, 1] - res[:, 3])

    f_zunili = c * delta
    power_i = f_zunili * delta

    begin = int(60 / 时间间隔)
    end = int(100 / 时间间隔)

#     power_ = power_i[1:] * 时间间隔  # 矩形
    power_ = (power_i[1:] + power_i[:-1]) * 时间间隔 / 2  # 梯形
    power = power_[begin: end]
#     print(power.sum())
    P = power.sum() / 40

    if pprint:
        print("平均输出功率：", P)

    target = -P
    return target
```

最大功率——遗传

In [902…

```
# TOD 遗传
lb, ub = 0, 1e5
ga = GA(
    func=GA_func,
    n_dim=1,
    size_pop=50,
    max_iter=200,
    prob_mut=0.001,
    lb=[lb],
    ub=[ub],
    constraint_eq=tuple(),
```

```
    constraint_ueq=tuple(),
    precision=1e-07,
    early_stop=True,
)
t0 = time()
xbest, ybest = ga.run()
t1 = time()
print()
print("遗传:")
print("直线阻尼器的阻尼系数：", xbest, "最大输出功率", -ybest)
print("总用时：", t1 - t0)
```

平均输出功率： 176.4529404579941
平均输出功率： 169.55315891144443
平均输出功率： 210.47790972033062
平均输出功率： 228.05186558107147
平均输出功率： 189.48446752750283
平均输出功率： 193.97091467666846
平均输出功率： 215.39144126354932
平均输出功率： 153.55905685833713
平均输出功率： 180.9272555562817
平均输出功率： 219.6945337732419
平均输出功率： 227.48284918228038
平均输出功率： 11.388678759977953
平均输出功率： 23.915192427015228
平均输出功率： 226.83141528768328
平均输出功率： 227.50474204276844
平均输出功率： 208.79341972329638
平均输出功率： 187.45462009040807
平均输出功率： 225.1921761182452
平均输出功率： 132.65209467744938
平均输出功率： 220.66183038594664
平均输出功率： 184.80949360452374
平均输出功率： 58.7048009768299
平均输出功率： 213.1145318228361
平均输出功率： 186.0182866616787
平均输出功率： 208.22955446379154
平均输出功率： 223.72578537307646
平均输出功率： 227.99869323084027
平均输出功率： 86.14550829049979
平均输出功率： 176.23943059232445
平均输出功率： 219.9343428301654
平均输出功率： 160.574227682032
平均输出功率： 121.88745253315042
平均输出功率： 170.3833298630404
平均输出功率： 170.24676695822603
平均输出功率： 214.1965253420878
平均输出功率： 78.5764994364029
平均输出功率： 112.07844639447599
平均输出功率： 223.3180389645019
平均输出功率： 217.61081479213243
平均输出功率： 226.59119137551784
平均输出功率： 167.62255463350436
平均输出功率： 226.86443262608617
平均输出功率： 194.5662035437965
平均输出功率： 187.49826015148827
平均输出功率： 213.3820632818371
平均输出功率： 178.84711666251025

```
平均输出功率： 174.07603760960183
平均输出功率： 219.78000126224416
平均输出功率： 202.76738503078798
平均输出功率： 222.8791952857272
平均输出功率： 228.05186558107147

遗传：
直线阻尼器的阻尼系数： [36561.71309525] 最大输出功率 [228.05186558]
总用时： 2.7345292568206787
```

直线阻尼器的阻尼系数： [36933.65769162] 最大输出功率 [228.04098758] 总用时： 2.785310745239258

直线阻尼器的阻尼系数： [36917.85232344] 最大输出功率 [228.04191625] 总用时： 2.4696505069732666

直线阻尼器的阻尼系数： [36803.02231918] 最大输出功率 [228.04742113] 总用时： 2.6217758655548096

直线阻尼器的阻尼系数： [36572.41264494] 最大输出功率 [228.05186568] 总用时： 2.569674491882324

规划

In [903…

```python
# TODO 规划
from scipy.optimize import minimize

minimize_fun = lambda x: GA_func(x)

t0 = time()
opt = minimize(
    minimize_fun,
    x0=[35000],
    bounds=((0, 100000), ),
    method='trust-constr',  #  SLSQP  trust-constr
#     options={'xtol': 1e-30,  'gtol': 1e-30, 'disp': True},
)
t1 = time()
print("总用时：", t1 - t0)

xopt = opt.x
power = minimize_fun(xopt)
print()
print("规划:")
print("(直线阻尼器的阻尼系数，幂指数)：", xopt, "最大输出功率", -power)
```

平均输出功率： 227.83599355347468
平均输出功率： 227.83599932570155
平均输出功率： 227.83601193801314
平均输出功率： 227.83600878546946
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599432965875
平均输出功率： 227.83599886770358
平均输出功率： 227.8360024453948
平均输出功率： 227.83599627046215
平均输出功率： 227.83600920443186
平均输出功率： 227.83599489098668
平均输出功率： 227.83598884641464
平均输出功率： 227.83600015129605
平均输出功率： 227.83599508891183
平均输出功率： 227.83599469310474
平均输出功率： 227.8359897706602
平均输出功率： 227.83598689368964
平均输出功率： 227.83599815775406
平均输出功率： 227.8359980055514
平均输出功率： 227.83600144008898
平均输出功率： 227.8360023299652
平均输出功率： 227.83600869367265
平均输出功率： 227.83601652956176
平均输出功率： 227.83600091317447
平均输出功率： 227.83600890363613
平均输出功率： 227.83599236711024
平均输出功率： 227.8360058553123
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.8359975964134
平均输出功率： 227.8360129450213
平均输出功率： 227.8359972816751
平均输出功率： 227.83599845083842
平均输出功率： 227.8360112025604
平均输出功率： 227.83600593777163
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688

平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183

平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
平均输出功率： 227.83601424901562
平均输出功率： 227.83600498213622
平均输出功率： 227.83599594727661
平均输出功率： 227.8359871982074
平均输出功率： 227.83599977691497
平均输出功率： 227.83600782669183
平均输出功率： 227.83599664371232
平均输出功率： 227.83601625091688
平均输出功率： 227.83599191195427
平均输出功率： 227.83600904833924
平均输出功率： 227.8360072156282
平均输出功率： 227.83600443995041
总用时： 13.688641786575317
平均输出功率： 227.83601424901562

规划：
(直线阻尼器的阻尼系数，幂指数)： [35000.00715896] 最大输出功率 227.83601424901562

最大功率——变步长

In [717…
```
from time import time
from numba import jit, prange
```

In [789…
```
# TODO 遍历
@jit
def run_travel_get_xy(lb, ub, num):
    ps = []
    t00 = time()
    t0 = time()
    print(f"[", lb, ',', ub, "]: ")
    for i, c in enumerate(np.linspace(lb, ub, num)):
        p = -GA_func(c, pprint=False)
        ps.append(p)
        _n = num // 100
        if i % (num // _n) == 0:
            t1 = time()
            _stars = '[' + '*' * (i // _n) + '.' * (num // _n - i // _n) + ']'
            print("%6d/%6d" % (i, num), _stars, round(t1 - t0, 2), "s/iter")
            t0 = time()
    t11 = time()
    print('总用时: ', t11 - t00)
    return np.linspace(lb, ub, num), np.array(ps)
```

In [793…
```
x, y = run_travel_get_xy(0, 100000, 10000)
print("x:", x[np.argmax(y)], "max:", max(y))
```

```
[ 0 , 100000 ]:
    0/ 10000 [..........................................................................................] 0.09 s/iter
  100/ 10000 [*.........................................................................................] 6.24 s/iter
  200/ 10000 [**........................................................................................] 4.73 s/iter
  300/ 10000 [***.......................................................................................] 3.97 s/iter
  400/ 10000 [****......................................................................................] 3.47 s/iter
  500/ 10000 [*****.....................................................................................] 3.62 s/iter
  600/ 10000 [******....................................................................................] 4.03 s/iter
  700/ 10000 [*******...................................................................................] 3.59 s/iter
  800/ 10000 [********..................................................................................] 3.44 s/iter
  900/ 10000 [*********.................................................................................] 3.38 s/iter
 1000/ 10000 [**********................................................................................] 4.0 s/iter
 1100/ 10000 [***********...............................................................................] 3.79 s/iter
 1200/ 10000 [************..............................................................................] 3.21 s/iter
 1300/ 10000 [*************.............................................................................] 3.17 s/iter
 1400/ 10000 [**************............................................................................] 3.15 s/iter
 1500/ 10000 [***************...........................................................................] 3.22 s/iter
 1600/ 10000 [****************..........................................................................] 3.32 s/iter
 1700/ 10000 [*****************.........................................................................] 3.33 s/iter
 1800/ 10000 [******************........................................................................] 3.24 s/iter
 1900/ 10000 [*******************.......................................................................] 3.59 s/iter
 2000/ 10000 [********************......................................................................] 3.11 s/iter
 2100/ 10000 [*********************.....................................................................] 3.22 s/iter
 2200/ 10000 [**********************....................................................................] 3.04 s/iter
 2300/ 10000 [**********************....................................................................] 3.03 s/iter
 2400/ 10000 [***********************...................................................................] 2.88 s/iter
 2500/ 10000 [************************..................................................................] 3.03 s/iter
 2600/ 10000 [*************************.................................................................] 3.11 s/iter
 2700/ 10000 [*************************.................................................................] 3.36 s/iter
 2800/ 10000 [**************************................................................................] 3.49 s/iter
 2900/ 10000 [***************************...............................................................] 3.47 s/iter
 3000/ 10000 [***************************...............................................................] 3.83 s/iter
 3100/ 10000 [****************************..............................................................] 3.77 s/iter
 3200/ 10000 [*****************************.............................................................] 3.85 s/iter
 3300/ 10000 [*****************************.............................................................] 4.06 s/iter
 3400/ 10000 [******************************............................................................] 4.16 s/iter
 3500/ 10000 [*******************************...........................................................] 4.41 s/iter
 3600/ 10000 [*******************************...........................................................] 4.46 s/iter
 3700/ 10000 [********************************..........................................................] 4.59 s/iter
 3800/ 10000 [*********************************.........................................................] 4.46 s/iter
 3900/ 10000 [*********************************.........................................................] 4.55 s/iter
 4000/ 10000 [**********************************........................................................] 4.7 s/iter
 4100/ 10000 [**********************************........................................................] 5.11 s/iter
 4200/ 10000 [***********************************.......................................................] 5.15 s/iter
 4300/ 10000 [************************************......................................................] 5.08 s/iter
 4400/ 10000 [************************************......................................................] 5.23 s/iter
```

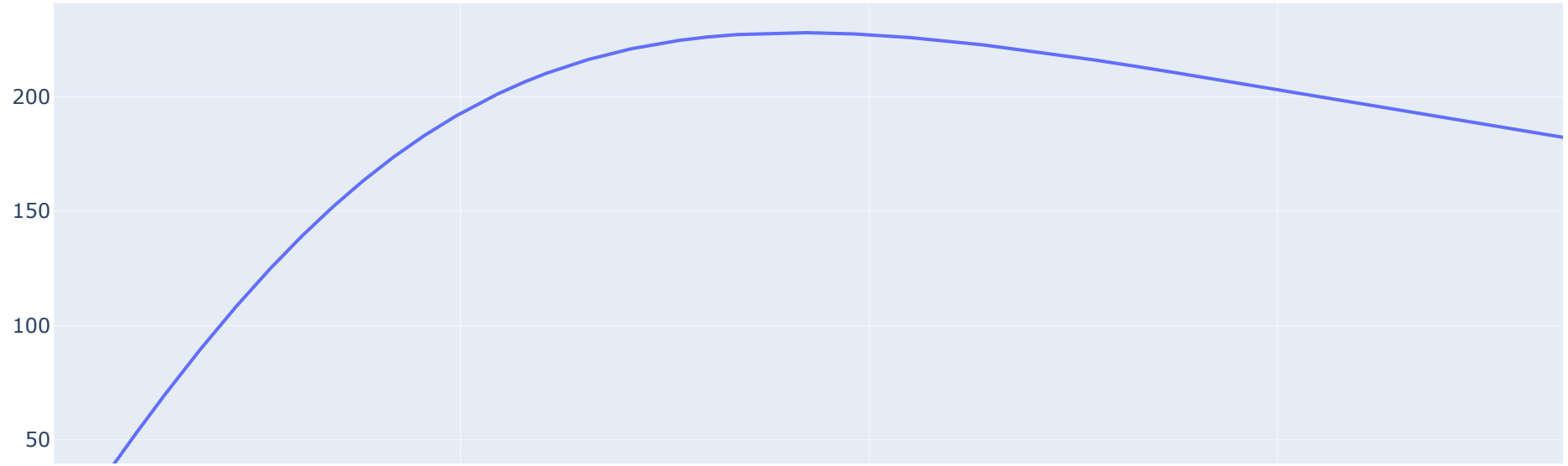
```
4500/ 10000 [*****************************************.....................................] 5.41 s/iter
4600/ 10000 [******************************************....................................] 5.44 s/iter
4700/ 10000 [******************************************....................................] 5.45 s/iter
4800/ 10000 [*******************************************...................................] 5.65 s/iter
4900/ 10000 [*******************************************...................................] 5.39 s/iter
5000/ 10000 [********************************************..................................] 5.44 s/iter
5100/ 10000 [********************************************..................................] 5.69 s/iter
5200/ 10000 [*********************************************.................................] 5.63 s/iter
5300/ 10000 [*********************************************.................................] 5.65 s/iter
5400/ 10000 [**********************************************................................] 5.76 s/iter
5500/ 10000 [**********************************************................................] 6.03 s/iter
5600/ 10000 [***********************************************...............................] 7.0 s/iter
5700/ 10000 [***********************************************...............................] 6.41 s/iter
5800/ 10000 [************************************************..............................] 6.55 s/iter
5900/ 10000 [************************************************..............................] 6.76 s/iter
6000/ 10000 [*************************************************.............................] 6.6 s/iter
6100/ 10000 [*************************************************.............................] 6.79 s/iter
6200/ 10000 [**************************************************............................] 6.6 s/iter
6300/ 10000 [**************************************************............................] 7.01 s/iter
6400/ 10000 [***************************************************...........................] 7.02 s/iter
6500/ 10000 [***************************************************...........................] 7.11 s/iter
6600/ 10000 [****************************************************..........................] 7.2 s/iter
6700/ 10000 [****************************************************..........................] 7.84 s/iter
6800/ 10000 [*****************************************************.........................] 7.55 s/iter
6900/ 10000 [*****************************************************.........................] 7.49 s/iter
7000/ 10000 [******************************************************........................] 7.35 s/iter
7100/ 10000 [******************************************************........................] 7.12 s/iter
7200/ 10000 [*******************************************************.......................] 6.64 s/iter
7300/ 10000 [*******************************************************.......................] 5.69 s/iter
7400/ 10000 [********************************************************......................] 7.01 s/iter
7500/ 10000 [********************************************************......................] 6.61 s/iter
7600/ 10000 [*********************************************************.....................] 6.85 s/iter
7700/ 10000 [*********************************************************.....................] 6.49 s/iter
7800/ 10000 [**********************************************************....................] 6.53 s/iter
7900/ 10000 [**********************************************************....................] 6.4 s/iter
8000/ 10000 [***********************************************************...................] 6.67 s/iter
8100/ 10000 [***********************************************************...................] 6.57 s/iter
8200/ 10000 [************************************************************..................] 6.21 s/iter
8300/ 10000 [************************************************************..................] 6.07 s/iter
8400/ 10000 [*************************************************************.................] 6.37 s/iter
8500/ 10000 [*************************************************************.................] 6.41 s/iter
8600/ 10000 [**************************************************************................] 5.75 s/iter
8700/ 10000 [**************************************************************................] 5.63 s/iter
8800/ 10000 [***************************************************************..............] 5.59 s/iter
8900/ 10000 [***************************************************************..............] 5.63 s/iter
9000/ 10000 [****************************************************************.............] 5.62 s/iter
```

```
  9100/ 10000 [**************************************************************************************.....] 5.48 s/iter
  9200/ 10000 [**************************************************************************************.....] 5.8 s/iter
  9300/ 10000 [**************************************************************************************.....] 5.46 s/iter
  9400/ 10000 [**************************************************************************************.....] 5.52 s/iter
  9500/ 10000 [**************************************************************************************.....] 5.47 s/iter
  9600/ 10000 [**************************************************************************************.....] 5.61 s/iter
  9700/ 10000 [**************************************************************************************.....] 5.57 s/iter
  9800/ 10000 [**************************************************************************************.....] 5.53 s/iter
  9900/ 10000 [**************************************************************************************.....] 5.69 s/iter
总用时：  518.0268075466156
max: 228.05189673677995
```

In [ ]:

In [794…

```
# TODO 阻尼系数和平均输出功率的关系
trace = go.Scatter(x=x, y=y)
fig = go.Figure(data=trace)
fig.show()
```

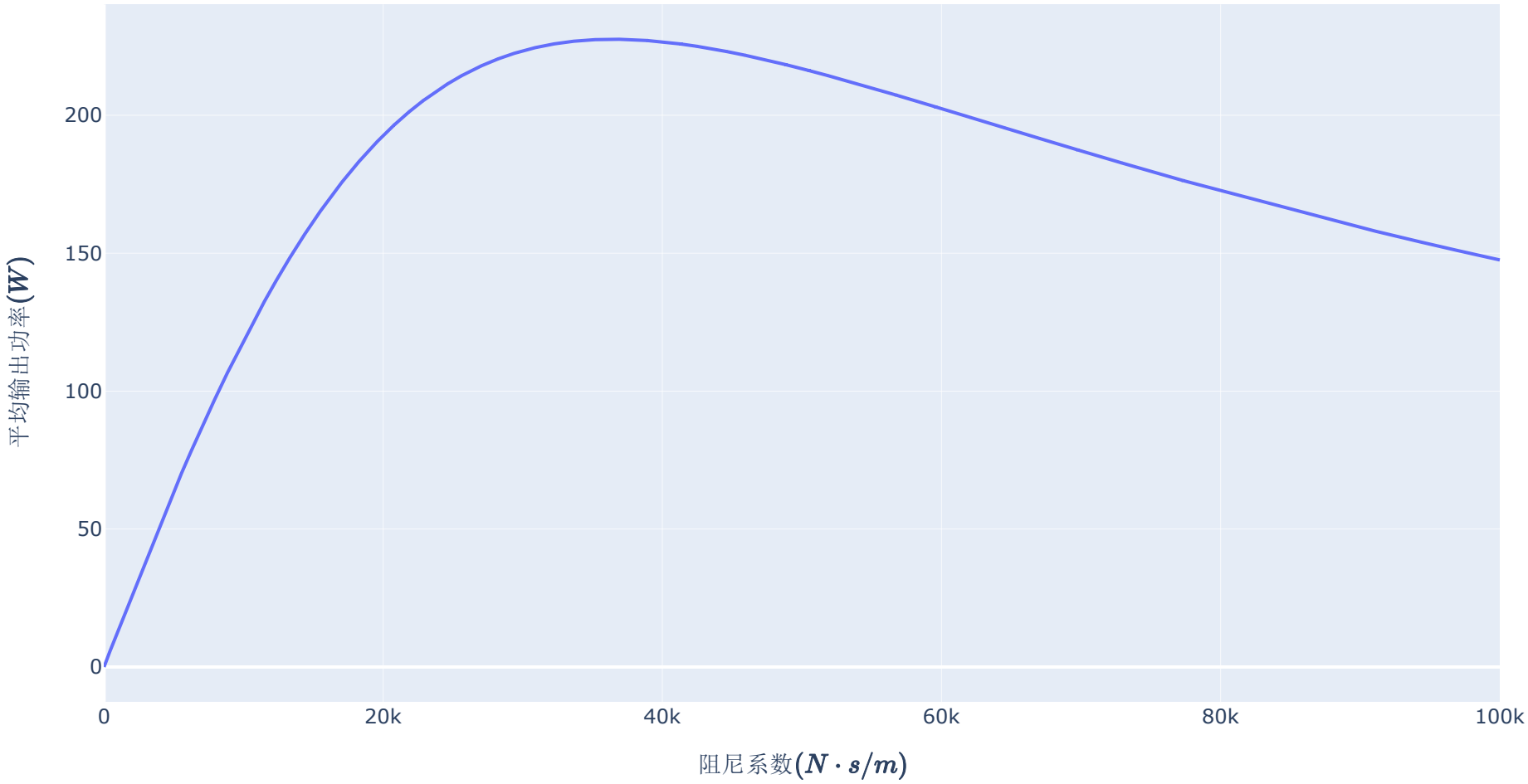


保存验证图

In [712…

```
# # TODO 阻尼系数和平均输出功率的关系
# y = np.load('y.npy')
# trace = go.Scatter(x=np.linspace(0, 100000, 100001), y=y)
# fig = go.Figure(data=trace)
# fig.update_layout(
#     width=1000,
#     height=600,
#     xaxis=dict(title='$阻尼系数 (N·s/m)$'),
#     yaxis=dict(title='$平均输出功率 (W)$'),
#     title=dict(text='阻尼系数和平均输出功率的关系'),
# )
```

```
# fig.write_image(IMG_SVG / "问题2-阻尼系数和平均输出功率的关系.svg")
# fig.show()
```



In [ ]:

最大功率：GA-变步长

In [109…

```
# TOD 遗传
```

```
lb, ub = 0, 1e5
ga = GA(
    func=GA_func,
    n_dim=1,
    size_pop=50,
    max_iter=200,
    prob_mut=0.001,
    lb=[lb],
    ub=[ub],
    constraint_eq=tuple(),
    constraint_ueq=tuple(),
    precision=1e-02,
    early_stop=True,
)

# TODO 变步长
@jit
def run_travel_get_xy(lb, ub, num=100):
    ps = []
    t00 = time()
    t0 = time()
    print(f"[", lb, ',', ub, "]: ")
    for i, c in enumerate(np.linspace(lb, ub, num)):
        p = -GA_func(c, pprint=False)
        ps.append(p)
        _n = 10
#         _n = num // 100
        if i % (num // _n) == 0:
            t1 = time()
            _stars = '[' + '*' * (i // _n) + '.' * (num // _n - i // _n) + ']'
            print("%6d/%6d" % (i, num), _stars, round(t1 - t0, 2), "s/iter")
            t0 = time()
    t11 = time()
    print('总用时：', t11 - t00)
    return np.linspace(lb, ub, num), np.array(ps)
```

In [110…

```
t0 = time()
xbest, ybest = ga.run()
t1 = time()
print()
print("遗传:")
print("直线阻尼器的阻尼系数：", xbest, "最大输出功率", -ybest)
print("遗传总用时：", t1 - t0)

mi = xbest * (1 - 0.05)
mx = xbest * (1 + 0.05)
```

```
x, y = run_travel_get_xy(mi, mx, 200)
print("x:", x[np.argmax(y)], "max:", max(y))

t1 = time()
print("遗传+变步长总用时：", t1 - t0)
```

平均输出功率： 223.05413846435266
平均输出功率： 225.7710032348512
平均输出功率： 172.59883194698523
平均输出功率： 188.2845583766262
平均输出功率： 199.6364822849565
平均输出功率： 224.93815553214077
平均输出功率： 217.87698362280116
平均输出功率： 173.36278201818203
平均输出功率： 210.60611922880815
平均输出功率： 168.62758685957226
平均输出功率： 225.24961277064463
平均输出功率： 180.99085892395715
平均输出功率： 223.44891191373694
平均输出功率： 227.74157545970155
平均输出功率： 216.85221355611003
平均输出功率： 224.94097847999257
平均输出功率： 224.0165214720333
平均输出功率： 179.77990675589928
平均输出功率： 223.4672349990961
平均输出功率： 174.6089609613383
平均输出功率： 226.56327313925854
平均输出功率： 201.75253282503786
平均输出功率： 224.0774812411013
平均输出功率： 219.48469520633307
平均输出功率： 220.64920395774416
平均输出功率： 221.69444987576497
平均输出功率： 223.95220236363326
平均输出功率： 217.38491740492208
平均输出功率： 181.35898566765587
平均输出功率： 38.94067177409817
平均输出功率： 227.9890843251898
平均输出功率： 227.60109129408528
平均输出功率： 227.43610342218068
平均输出功率： 220.98243571798312
平均输出功率： 224.93932710153254
平均输出功率： 228.02124373377583
平均输出功率： 201.74701116895616
平均输出功率： 217.12754436492082
平均输出功率： 224.9926519915271
平均输出功率： 223.29599559067668
平均输出功率： 201.74936832191085
平均输出功率： 227.9331630789796
平均输出功率： 166.33441291806355
平均输出功率： 224.90370605023662
平均输出功率： 227.3540669766631
平均输出功率： 224.0633643782948

```
平均输出功率： 227.9891334320715
平均输出功率： 223.12234312416686
平均输出功率： 224.99618177975967
平均输出功率： 204.35229772564495
平均输出功率： 228.02124373377583

遗传：
直线阻尼器的阻尼系数： [37181.03392011] 最大输出功率 [228.02124373]
遗传总用时： 5.871513843536377
[ [35321.98222411] , [39040.08561612] ]：
     0/   200 [..................] 0.2 s/iter
    20/   200 [**................] 3.93 s/iter
    40/   200 [****..............] 3.87 s/iter
    60/   200 [******............] 3.97 s/iter
    80/   200 [********..........] 4.03 s/iter
   100/   200 [**********........] 3.99 s/iter
   120/   200 [************......] 4.2 s/iter
   140/   200 [**************....] 4.06 s/iter
   160/   200 [****************..] 4.22 s/iter
   180/   200 [******************] 4.24 s/iter
总用时： 40.70883846282959
x: [36573.8059792] max: 228.05187380968127
遗传+变步长总用时： 46.58035230636597
```

x: [36550.48839975] max: 228.0518457848093 遗传+变步长总用时： 29.480774879455566

x: [36566.41502498] max: 228.05188102735542 遗传+变步长总用时： 28.962661027908325

In [ ]:

In [ ]:

In [ ]:

（2）c 非常量

$$c = 10000|V|^{0.5} = 10000|V_1 - V_2|^{0.5} = 10000|\dot{X}_1 - \dot{X}_2|^{0.5}$$

In [112…

```python
# TODO 数值解：在 ode 里面计算 power_i
def diff_equation(ys, t, xishu, mici, k=弹簧刚度):
    y1 = ys[2-1]
    y3 = ys[4-1]

    c = c直线阻尼器的阻尼系数_func2(ys[2-1], ys[4-1], k=xishu, a=mici)
```

```
    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2

    y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - m2 * y4
    y2 = y2 / m1_
    return [y1, y2, y3, y4]
def GA_func(xishu, mici, period=2, pprint=True):
    """ sko 遗传目标函数
    :param zunixishu: 阻尼系数/比例系数
    :param power_i: list
    :param period: 计算平均功率的时间区间类型
    """
#     if period == 1:  # 40T: 0~28.5 s
#         begin = 0
#         end = int(40 * 波浪周期 / 时间间隔)
#     elif period == 2:  # 稳定: 60~100 s
#         begin = int(60 / 时间间隔)
#         end = int(100 / 时间间隔)
#     elif period == 3:  # 所有: 0~100 s
#         begin = 0
#         end = -2

    时间间隔 = 0.5
    _l, _r = 0, 100

    res = odeint(
        diff_equation,
        [0, 0, 0, 0],
        np.linspace(_l, _r, num=int(_r / 时间间隔) + 1),
        args=(xishu, mici),
    )  # zunixishu
    delta = abs(res[:, 1] - res[:, 3])

    c = xishu * delta**mici
    f_zunili = c * delta
    power_i = f_zunili * delta

    begin = int(60 / 时间间隔)
    end = int(100 / 时间间隔)

#     power_ = power_i[1:] * 时间间隔  # 矩形
    power_ = (power_i[1:] + power_i[:-1]) * 时间间隔 / 2  # 梯形
    power = power_[begin: end]
#     print(power.sum())
    P = power.sum() / 40
```

```
    if pprint:
        print("阻尼系数", xishu, "\t幂次", mici, "\t平均输出功率：", P)

    target = -P
    return target
```

最大功率——遗传

In [ ]:

In [112…

```
lb, ub = 0, 1e5
ga = GA(
    func=GA_func,
    n_dim=2,
    size_pop=50,
    max_iter=200,
    prob_mut=0.001,
    lb=[lb, 0],
    ub=[ub, 1],
    constraint_eq=tuple(),
    constraint_ueq=tuple(),
    precision=1e-02,
    early_stop=True,
)
t0 = time()
xbest, ybest = ga.run()
t1 = time()
print("(直线阻尼器的阻尼系数, 幂指数)：", xbest, "最大输出功率", -ybest)
print("总用时：", t1 - t0)
```

阻尼系数 83090.96593206919 幂次 0.031496062992125984 平均输出功率： 178.94245217495404
阻尼系数 47140.25539995762 幂次 0.6141732283464567 平均输出功率： 145.2663633992895
阻尼系数 56631.94397878313 幂次 0.23622047244094488 平均输出功率： 226.56411200231196
阻尼系数 93537.0441399243 幂次 0.031496062992125984 平均输出功率： 166.07183452337762
阻尼系数 93425.41655453542 幂次 0.07874015748031496 平均输出功率： 180.81497272428777
阻尼系数 29292.507725507483 幂次 0.07874015748031496 平均输出功率： 211.45132180775764
阻尼系数 25651.94521259935 幂次 0.05511811023622047 平均输出功率： 204.89040651521435
阻尼系数 57119.903392785985 幂次 0.12598425196850394 平均输出功率： 225.91712869035982
阻尼系数 19719.619734264597 幂次 0.6771653543307087 平均输出功率： 63.53827867338523
阻尼系数 34393.78347359797 幂次 0.3779527559055118 平均输出功率： 164.47259204867368
阻尼系数 31324.668605605875 幂次 0.5039370078740157 平均输出功率： 127.4476460703494
阻尼系数 50481.06017595888 幂次 0.31496062992125984 平均输出功率： 212.69396233654317
阻尼系数 61381.141029664344 幂次 0.84251968503937 平均输出功率： 124.13624415675056
阻尼系数 97082.03059923831 幂次 0.5354330708661418 平均输出功率： 221.85186456528362
阻尼系数 95228.11145950027 幂次 0.7952755905511811 平均输出功率： 177.45527209856408
阻尼系数 98366.41540327163 幂次 0.07874015748031496 平均输出功率： 175.48192234262214
阻尼系数 58702.98497098595 幂次 0.015748031496062992 平均输出功率： 208.60086694975385
阻尼系数 50106.98736351653 幂次 0.03937007874015748 平均输出功率： 222.86084488137584
阻尼系数 25113.566226575746 幂次 0.1732283464566929 平均输出功率： 177.5041042596536
阻尼系数 10215.771807180154 幂次 0.905511811023622 平均输出功率： 22.56379023481545
阻尼系数 48411.66427204992 幂次 0.007874015748031496 平均输出功率： 220.6290302555551
阻尼系数 8359.760544285806 幂次 0.2204724409448819 平均输出功率： 67.78868788904913
阻尼系数 70765.69025312009 幂次 0.07874015748031496 平均输出功率： 206.9231025515315
阻尼系数 72103.88017319918 幂次 0.5669291338582677 平均输出功率： 197.19120511230466
阻尼系数 69914.34514012009 幂次 0.25984251968503935 平均输出功率： 227.98241427745296
阻尼系数 16778.952883419566 幂次 0.3779527559055118 平均输出功率： 94.67321923058952
阻尼系数 93387.16825170328 幂次 0.29133858267716534 平均输出功率： 223.29237457636083
阻尼系数 28517.96916234309 幂次 0.7086614173228346 平均输出功率： 83.16220763369867
阻尼系数 18570.221577299926 幂次 0.18110236220472442 平均输出功率： 143.17559109287555
阻尼系数 62143.29374690615 幂次 0.1968503937007874 平均输出功率： 227.66389311294407
阻尼系数 91908.7762778268 幂次 0.49606299212598426 平均输出功率： 223.24700741951696
阻尼系数 10693.157356569609 幂次 0.952755905511811 平均输出功率： 21.57091487897093
阻尼系数 57417.42595538055 幂次 0.5354330708661418 平均输出功率： 182.41053572606626
阻尼系数 34903.36745401427 幂次 0.952755905511811 平均输出功率： 64.05241657009935
阻尼系数 42300.66193942201 幂次 0.11023622047244094 平均输出功率： 226.5893547315623
阻尼系数 97832.95380073511 幂次 0.4094488188976378 平均输出功率： 228.28060831250423
阻尼系数 16714.246077194573 幂次 0.06299212598425197 平均输出功率： 158.8036043680901
阻尼系数 91744.34493448406 幂次 0.6692913385826772 平均输出功率： 198.75750295524335
阻尼系数 76459.85939859506 幂次 0.9606299212598425 平均输出功率： 121.64026922592038
阻尼系数 51919.60644242803 幂次 0.2755905511811024 平均输出功率： 219.8165525145338
阻尼系数 57940.61767701017 幂次 0.7322834645669292 平均输出功率： 141.00946887724194
阻尼系数 16869.55790934312 幂次 0.1968503937007874 平均输出功率： 129.71840970542175
阻尼系数 83413.8741143867 幂次 0.6614173228346457 平均输出功率： 191.982462185914
阻尼系数 47775.545583697894 幂次 0.8031496062992126 平均输出功率： 108.77770625352005
阻尼系数 54107.24008722544 幂次 0.952755905511811 平均输出功率： 93.45901756764047
阻尼系数 48218.64057890419 幂次 0.49606299212598426 平均输出功率： 173.38965930419837

```
阻尼系数 84863.36975475369      幂次 0.8503937007874016         平均输出功率： 154.36720562275565
阻尼系数 65366.34954013524      幂次 0.18110236220472442        平均输出功率： 225.95243307347647
阻尼系数 50753.03618628002      幂次 0.10236220472440945        平均输出功率： 227.2788336564105
阻尼系数 64402.23839296332      幂次 0.25984251968503935        平均输出功率： 227.7744983468163
阻尼系数 97832.95380073511      幂次 0.4094488188976378         平均输出功率： 228.28060831250423
(直线阻尼器的阻尼系数, 幂指数)： [9.78329538e+04 4.09448819e-01] 最大输出功率 [228.28060831]
总用时： 16.35539484024048
```

In [ ]:

In [ ]:

规划

In [906…

```python
# TODO
minimize_fun = lambda x: GA_func(x[0], x[1])

t0 = time()
opt = minimize(
    minimize_fun,
    x0=[80000, 0.4],
    bounds=(
        (0, 100000), (0, 1)
    ),
    method='SLSQP',  #  SLSQP  trust-constr
    options={'xtol': 1e-5,  'gtol': 1e-5, 'disp': True}
)
t1 = time()
print("总用时: ", t1 - t0)

xopt = opt.x
power = minimize_fun(xopt)
# kkk, aaa = xopt
print("(直线阻尼器的阻尼系数, 幂指数): ", xopt, "最大输出功率", -power)
```

阻尼系数 80000.0 幂次 0.4 平均输出功率： 225.85486326551927
阻尼系数 80000.0000000149 幂次 0.4 平均输出功率： 225.85485392348863
阻尼系数 80000.0 幂次 0.40000000149011612 平均输出功率： 225.85488369652785
阻尼系数 79373.06693649292 幂次 0.999999160243442 平均输出功率： 117.55055373423582
阻尼系数 79686.61964043377 幂次 0.6999171100096269 平均输出功率： 179.95647293096226
阻尼系数 79843.34634074115 幂次 0.5499236034523086 平均输出功率： 208.34023788532005
阻尼系数 79921.68710721185 幂次 0.4749484636306695 平均输出功率： 218.96603437453797
阻尼系数 79960.84903543357 幂次 0.4374689854945225 平均输出功率： 222.93929212522136
阻尼系数 79980.42683785014 幂次 0.418732272290006 平均输出功率： 224.53598283718193
阻尼系数 79990.2144684666 幂次 0.40936513169321254 平均输出功率： 225.23081515009545
阻尼系数 79995.10773084252 幂次 0.4046820905724007 平均输出功率： 225.55178235921966
阻尼系数 79997.55410660972 幂次 0.4023408144595204 平均输出功率： 225.70558325950668
阻尼系数 79998.77717210042 幂次 0.40117029353781264 平均输出功率： 225.78077787250623
阻尼系数 79999.3886450067 幂次 0.4005850903452844 平均输出功率： 225.8179635104334
阻尼系数 79999.3886450216 幂次 0.4005850903452844 平均输出功率： 225.81794075501017
阻尼系数 79999.3886450067 幂次 0.4005851052464456 平均输出功率： 225.81794525674158
阻尼系数 72187.42391830786 幂次 0.0 平均输出功率： 184.43661446518738
阻尼系数 76093.41983013414 幂次 0.2002932399169247 平均输出功率： 222.2263169957017
阻尼系数 78046.40541349823 幂次 0.300439225430807 平均输出功率： 227.96282189373852
阻尼系数 79022.89632701132 幂次 0.35051212187824 平均输出功率： 227.7894826839604
阻尼系数 79511.14184051991 幂次 0.3755485730121106 平均输出功率： 227.0172441831471
阻尼系数 79755.26485011034 幂次 0.38806681154408035 平均输出功率： 226.46963678736148
阻尼系数 79877.32653419607 幂次 0.39432594000379584 平均输出功率： 226.1566239187443
阻尼系数 79938.35747872155 幂次 0.39745550948879926 平均输出功率： 225.9904751801019
阻尼系数 79968.87300538257 幂次 0.399020297020758 平均输出功率： 225.90500902636555
阻尼系数 79984.1307966953 幂次 0.39980269222162135 平均输出功率： 225.86170779407476
阻尼系数 79991.75970652881 幂次 0.40019389054903376 平均输出功率： 225.83987638268732
阻尼系数 79991.75970654371 幂次 0.40019389054903376 平均输出功率： 225.8398908510566
阻尼系数 79991.75970652881 幂次 0.40019390545019495 平均输出功率： 225.839892363612
阻尼系数 79994.17346853144 幂次 1.0 平均输出功率： 118.258277979902
阻尼系数 79992.92462989714 幂次 0.6896707124558241 平均输出功率： 182.44010937541043
阻尼系数 79992.32514485015 幂次 0.540702092800827 平均输出功率： 209.9366348832394
阻尼系数 79992.03614271374 幂次 0.468886707990824 平均输出功率： 219.7165446991428
阻尼系数 79991.89549397044 幂次 0.4339362963824444 平均输出功率： 223.27455066432248
阻尼系数 79991.82657930466 幂次 0.41681139445419196 平均输出功率： 224.6880782717391
阻尼系数 79991.79268391605 幂次 0.40838858344633105 平均输出功率： 225.29944434348218
阻尼系数 79991.77597971242 幂次 0.40423768398794657 平均输出功率： 225.57993612750087
阻尼系数 79991.76773942905 幂次 0.402190020503024 平均输出功率： 225.71322218281762
阻尼系数 79991.76367244756 幂次 0.40117939876731157 平均输出功率： 225.77774311326712
阻尼系数 79991.76166469682 幂次 0.4006804841454243 平均输出功率： 225.8093089231301
阻尼系数 79991.76166471172 幂次 0.4006804841454243 平均输出功率： 225.80930483717287
阻尼系数 79991.76166469682 幂次 0.4006804990465855 平均输出功率： 225.8093031235716
阻尼系数 79992.23145681589 幂次 1.1586681614161876e-10 平均输出功率： 173.1637542395613
阻尼系数 79991.84154153822 幂次 0.33255441269182906 平均输出功率： 228.12671633773508
阻尼系数 79991.84154155312 幂次 0.33255441269182906 平均输出功率： 228.12670705437867
阻尼系数 79991.84154153822 幂次 0.33255442759299025 平均输出功率： 228.12670633320164

```
阻尼系数 79991.80815861095      幂次 0.359557848692565  平均输出功率： 227.69504256364135
阻尼系数 79991.82717497465      幂次 0.34417551999634943        平均输出功率： 228.02237713433706
阻尼系数 79991.83495674632      幂次 0.3378808481249124         平均输出功率： 228.09439442562172
阻尼系数 79991.83843968665      幂次 0.3350634991381805         平均输出功率： 228.11479815401373
阻尼系数 79991.84006176716      幂次 0.33375139888861854        平均输出功率： 228.12175810351678
阻尼系数 79991.84083142945      幂次 0.3331288193768032         平均输出功率： 228.12451357825293
阻尼系数 79991.84119975218      幂次 0.3328308832776911         平均输出功率： 228.12569031445702
阻尼系数 79991.84137683292      幂次 0.33268764274109675        平均输出功率： 228.1262490943963
阻尼系数 79991.84146200903      幂次 0.33261874381776196        平均输出功率： 228.12650079633096
阻尼系数 79991.8415030781       幂次 0.33258552306747247        平均输出功率： 228.12662002267197
阻尼系数 79991.84152289252      幂次 0.3325694951890493         平均输出功率： 228.12666494266122
阻尼系数 79991.84152290742      幂次 0.3325694951890493         平均输出功率： 228.12667037932653
阻尼系数 79991.84152289252      幂次 0.3325695100902105         平均输出功率： 228.12665283816915
阻尼系数 79991.84112086087      幂次 0.32961617469608634        平均输出功率： 228.13319365098496
阻尼系数 79991.84132129233      幂次 0.3310885422348085         平均输出功率： 228.1309605717898
阻尼系数 79991.84142170759      幂次 0.3318261917168583         平均输出功率： 228.12907558786182
阻尼系数 79991.841472084        幂次 0.33219625626838567        平均输出功率： 228.12795744768545
阻尼系数 79991.84149737237      幂次 0.33238202448858506        平均输出功率： 228.1273444538016
阻尼系数 79991.8415100715       幂次 0.33247531221938437        平均输出功率： 228.1269866229778
阻尼系数 79991.84151645318      幂次 0.3325221918597186         平均输出功率： 228.12682898755037
阻尼系数 79991.84151965813      幂次 0.3325457354841761         平均输出功率： 228.1267394384476
阻尼系数 79991.84152126865      幂次 0.3325575662950591         平均输出功率： 228.1266913010066
阻尼系数 79991.84152207822      幂次 0.332563513411134  平均输出功率： 228.12668009320797
阻尼系数 79991.84152248401      幂次 0.3325664943341949         平均输出功率： 228.12666185532817
阻尼系数 79991.84152249891      幂次 0.3325664943341949         平均输出功率： 228.12668770906265
阻尼系数 79991.84152248401      幂次 0.3325665092353561         平均输出功率： 228.12666477384505
阻尼系数 79991.84154461786      幂次 0.33253769925823545        平均输出功率： 228.1267660348029
阻尼系数 79991.84153358624      幂次 0.33255205086870504        平均输出功率： 228.12672001521088
阻尼系数 79991.84152805484      幂次 0.3325592469524279         平均输出功率： 228.12670455565404
阻尼系数 79991.84152528392      幂次 0.33256285178106104        平均输出功率： 228.1266943698003
阻尼系数 79991.84152389504      幂次 0.33256465865622287        平均输出功率： 228.12665978094947
阻尼系数 79991.84152318882      幂次 0.3325655774058861         平均输出功率： 228.12665677233198
阻尼系数 79991.84152283471      幂次 0.3325660380929231         平均输出功率： 228.12667176383758
阻尼系数 79991.84152266277      幂次 0.3325662617745537         平均输出功率： 228.12668863438867
Optimization terminated successfully    (Exit mode 0)
            Current function value: -228.12668863438867
            Iterations: 7
            Function evaluations: 80
            Gradient evaluations: 7
总用时： 12.517472982406616
阻尼系数 79991.84152266277      幂次 0.3325662617745537         平均输出功率： 228.12668863438867
(直线阻尼器的阻尼系数, 幂指数)： [7.99918415e+04 3.32566262e-01] 最大输出功率 228.12668863438867
```

总用时： 10.424811124801636 阻尼系数 79991.84152266277 幂次 0.3325662617745537 平均输出功率： 228.12668863438867

最大功率——变步长

In [115…

```python
def run_travel_get_xy(num=100):
    pss = []
    t00 = time()
    t0 = time()
    n_ = 10
    n = num // n_
    xishu_s = np.linspace(99990, 100000, num+1)
    mici_s = np.linspace(0.4241, 0.4243, num+1)
    for i, xishu in enumerate(xishu_s):
        if i % n == 0:
            t1 = time()
            _stars = '[' + '*' * (i // n_) + '.' * (num // n_ - i // n_) + ']'
            print("%6d/%6d" % (i, num), _stars, round(t1 - t0, 2), "s/iter")
            t0 = time()

        ps = []
        for j, mici in enumerate(mici_s):
            p = -GA_func(xishu, mici, pprint=False)
            ps.append(p)
        pss.append(ps)
    t11 = time()
    print('总用时：', t11 - t00)
    return xishu_s, mici_s, pss
```

In [908…

```python
x1, x2, y = run_travel_get_xy(100)
```

```
     0/   100 [..........] 0.0 s/iter
    10/   100 [*.........] 159.66 s/iter
    20/   100 [**........] 120.59 s/iter
    30/   100 [***.......] 115.41 s/iter
    40/   100 [****......] 120.17 s/iter
    50/   100 [*****.....] 140.97 s/iter
    60/   100 [******....] 131.82 s/iter
    70/   100 [*******...] 133.05 s/iter
    80/   100 [********..] 134.99 s/iter
    90/   100 [*********.] 144.27 s/iter
总用时： 1355.369698524475
```

In [909…

```python
y_np = np.array(y)
np.save("question2-y.npy", y_np)
index = np.unravel_index(y_np.argmax(), y_np.shape)
print(y_np.max())
print(index[0], "\tx1:", x1[index[1]])
print(index[1], "\tx2:", x2[index[0]])
print(index, "\ty:", y_np[index])
```

```
228.30561856731447
99      x1: 42424.242424242424
42      x2: 1.0
(99, 42)        y: 228.30561856731447
```

In [ ]:

最大功率——GA、变步长

In [ ]:

In [115…

```python
t0 = time()
xbest, ybest = ga.run()
t1 = time()
print("(直线阻尼器的阻尼系数, 幂指数): ", xbest, "最大输出功率", -ybest)
print("总用时: ", t1 - t0)

x1, x2, y = run_travel_get_xy(10)
y_np = np.array(y)
# np.save("question2-y.npy", y_np)
index = np.unravel_index(y_np.argmax(), y_np.shape)
print(y_np.max())
print(index[0], "\tx1:", x1[index[0]])
print(index[1], "\tx2:", x2[index[1]])
print(index, "\ty:", y_np[index[::-1]])

t1 = time()
print("总用时: ", t1 - t0)
```

```
 0/    10 [.] 0.0 s/iter
 1/    10 [.] 3.34 s/iter
 2/    10 [.] 3.29 s/iter
 3/    10 [.] 3.25 s/iter
 4/    10 [.] 3.46 s/iter
 5/    10 [.] 3.3 s/iter
 6/    10 [.] 3.27 s/iter
 7/    10 [.] 3.63 s/iter
 8/    10 [.] 3.36 s/iter
 9/    10 [.] 3.32 s/iter
10/    10 [*] 3.36 s/iter
总用时：  36.885353803634644
228.30579950102674
10      x1: 100000.0
0       x2: 0.4241
(10, 0)         y: 228.30536604109298
总用时：  36.88644480705261
```

In [ ]:

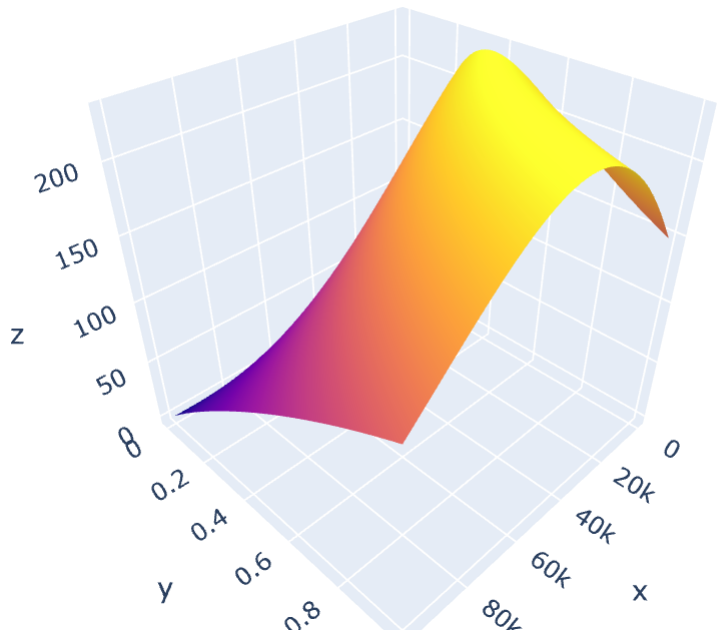
In [ ]:

In [ ]:

绘制：3D曲线图、二维等高线图

In [116…

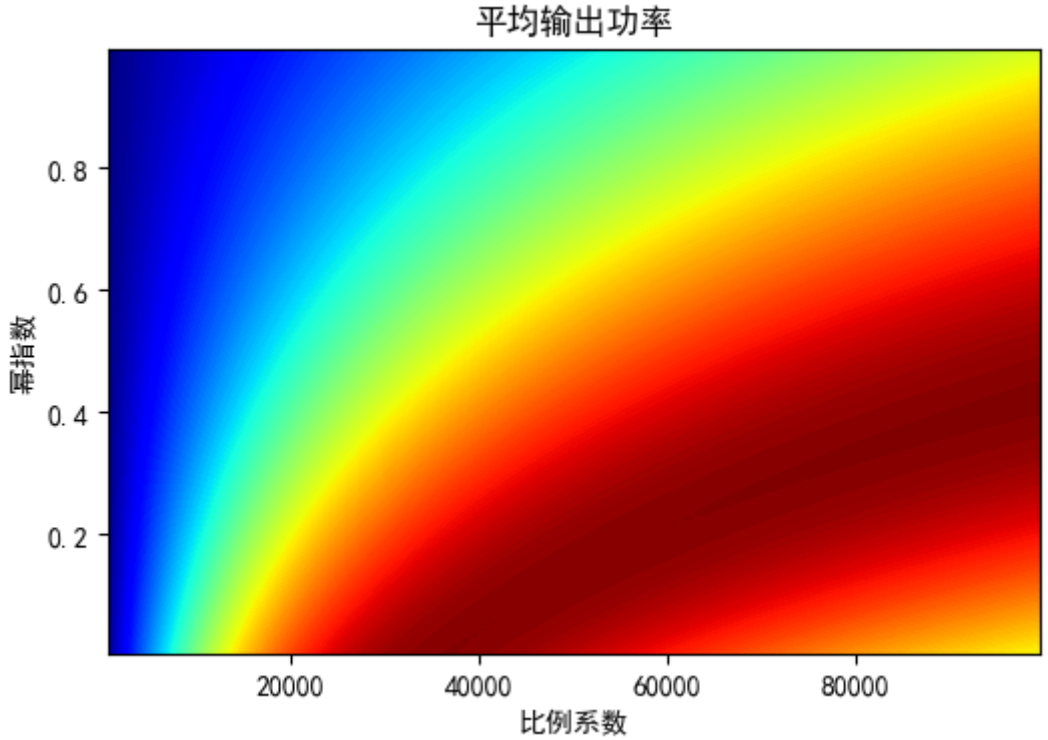
```
y = np.load('question2-y.npy')

fig1 = go.Figure(data=[go.Surface(
    x=np.linspace(0, 100000, 100),
    y=np.linspace(0, 1, 100),
    z=y,
)])
fig1.write_image(IMG_SVG / '问题2-3D曲线图.svg')
fig1.show()
```



In [ ]:

In [116…]:

```
y = np.load('question2-y.npy')
plt.figure(dpi=100)
plt.contourf(y.T, np.arange(0, 230, 1.5), origin='lower', extent=[0, 100000, 0, 1], cmap=plt.cm.jet)
plt.xlabel('比例系数')
plt.ylabel('幂指数')
plt.title("平均输出功率")
plt.savefig(IMG_SVG / '问题2-二维等高线图.svg')
plt.show()
```



In [ ]:

In [985…

```
# fig2 = go.Figure(data=go.Contour(
# #     x=x1, y=x2,
# #     z=y,
# #     x=xnew, y=ynew,
#     z=y,
# #     fillcolor=True,
#     autocolorscale=False,
# #     colorscale='jet',
#     autocontour=False,
#     colorscale=[[0.0, "rgb(165,0,38)"],
#                 [0.1111111111111111, "rgb(215,48,39)"],
#                 [0.2222222222222222, "rgb(244,109,67)"],
#                 [0.3333333333333333, "rgb(253,174,97)"],
#                 [0.4444444444444444, "rgb(254,224,144)"],
#                 [0.5555555555555556, "rgb(224,243,248)"],
#                 [0.6666666666666666, "rgb(171,217,233)"],
#                 [0.7777777777777778, "rgb(116,173,209)"],
#                 [0.8888888888888888, "rgb(69,117,180)"],
#                 [1.0, "rgb(49,54,149)"]],
# ))
```

```
# # fig2.write_image('temp.png')
# # fig2.show()
```

In [ ]:

In [ ]:

问题3

$X_1(t)$ 浮子位移，$X_2(t)$ 振子位移

$$m_2\frac{d^2X_2(t)}{dt^2}+c\frac{dX_2(t)}{dt}+kX_2(t)=c\frac{dX_1(t)}{dt}+kX_1(t)$$

$$m_1\frac{d^2X_1(t)}{dt^2}+m_2\frac{d^2X_2(t)}{dt^2}=F(t)$$

m_1 为定子与浮体总质量， kg；

m_2 为动子质量， kg；

$F(t)$ 为垂向波浪力， N；

k 为弹簧的劲度系数，N/m；

c 为动子与定子之间的阻尼系数

参数

In [156…

```
# TODO 4
浮子质量 = 4866  # kg
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
```

```
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m
```

In [156…

```
# TODO 3
question1234 = 3
if question1234 == 1:
    # 问题1：参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2：参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
elif question1234 == 3:
    # 问题3：参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    垂荡兴波阻尼系数 = 683.4558
    垂荡激励力振幅 = 3640

    纵摇附加转动惯量 = 7001.914
    纵摇兴波阻尼系数 = 654.3383
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4：参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140
```

$$m_2\frac{d^2X_2(t)}{dt^2}cos(\theta_2(t))+c\frac{dX_2(t)}{dt}cos(\theta_2(t))+kX_2(t)cos(\theta_2(t))=c\frac{dX_1(t)}{dt}+kX_1(t)$$

$$m_1'\frac{d^2X_1(t)}{dt^2}+m_2\frac{d^2X_2(t)}{dt^2}cos(\theta_2(t))=F(t)$$

$$j_2\frac{d^2\theta_2(t)}{dt^2}+C\frac{d\theta_2(t)}{dt}+K\theta_2(t)=C\frac{d\theta_1(t)}{dt}+K\theta_1(t)$$

$$j_1'\frac{d^2\theta_1(t)}{dt^2}+j_2\frac{d^2\theta_2(t)}{dt^2}=M(t)$$

m 质量

j 转动惯量

c 力的阻尼系数

C 力矩的阻尼系数

k 刚度系数

K 扭转弹簧的刚度系数

$F(t)=F_{波浪激励力}-F_{兴波阻尼力}-F_{静水恢复力}$

$=fcos(\omega t)+k_{垂荡兴波阻尼系数}v+\rho gSh$

$=fcos(\omega t)+k_{垂荡兴波阻尼系数}\frac{dX1(t)}{dt}+k_{系数}X_1(t)$

$=fcos(\omega t)+k_1y_2+k_2y_1$

$M(t)=M_{波浪激励力矩}-M_{兴波阻尼力矩}-M_{静水恢复力矩}$

$=Lcos(\omega t)+K_{纵摇兴波阻尼力矩系数}\frac{d\theta1(t)}{dt}+K_{静水恢复力矩系数}\theta_1(t)$

$=Lcos(\omega t)+K_1y_6+K_2y_5$

$$y' = \begin{bmatrix} y_1' \\ y_2' \\ y_3' \\ y_4' \\ y_5' \\ y_6' \\ y_7' \\ y_8' \end{bmatrix} = \begin{bmatrix} \dot{X}_1 \\ \ddot{X}_1 \\ \dot{X}_2 \\ \ddot{X}_2 \\ \dot{\theta}_1 \\ \ddot{\theta}_1 \\ \dot{\theta}_2 \\ \ddot{\theta}_2 \end{bmatrix} = \begin{bmatrix} \dot{X}_1 \\ (fcos(\omega t) + k_1\dot{X}_1 + k_2X_1 - m_2\ddot{X}_2cos(\theta_2))/m_1' \\ \dot{X}_2 \\ (c(\dot{X}_1 - \dot{X}_2cos(\theta_2)) + k(X_1 - X_2cos(\theta_2)))/m_2cos(\theta_2) \\ \dot{\theta}_1 \\ (Lcos(\omega t) + K_1\dot{\theta}_1 + K_2\theta_1 - j_2\ddot{\theta}_2)/j_1' \\ \dot{\theta}_2 \\ (C(\dot{\theta}_1 - \dot{\theta}_2) + K(\theta_1 - \theta_2))/j_2 \end{bmatrix}$$

$$= \begin{bmatrix} y_2 \\ (fcos(\omega t) + k_1y_2 + k_2y_1 - m_2\ddot{X}_2cos(y_7))/m_1' \\ y_4 \\ (c(y_2 - y_4cos(y_7)) + k(y_1 - y_3cos(y_7)))/m_2cos(y_7) \\ y_6 \\ (Lcos(\omega t) + K_1y_6 + K_2y_5 - j_2\ddot{\theta}_2)/j_1' \\ y_8 \\ (C(y_6 - y_8) + K(y_5 - y_7))/j_2 \end{bmatrix}$$

$j_1 = 33000$

$j_2 = \left((0.469588 + 0.5 + X_1cos(\theta_2) - X_2(t))^3 - (0.469558 + X_1cos(\theta_2) - X_2(t))^3\right) m_2/1.5$

$j_2 = ((0.969588 + y_3)^3 - (0.469558 + y_3)^3)m_2/1.5$

浮子与振子的垂荡位移与速度
和纵摇角位移与角速度

In [156…

```
# TODO 问题3: 参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量
f = 垂荡激励力振幅

c = 10000
k = 弹簧刚度
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积
```

```
C = 1000
L = 纵摇激励力矩振幅
K = 扭转弹簧刚度
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数
```

In [156…
```
def ode_func(y, t):
    dy1 = y[2-1]
    dy3 = y[4-1]
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])

    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]
```

In [157…
```
时间间隔 = 0.01
_l, _r = 0, 100
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res3 = odeint(ode_func, y0, t)
```

In [ ]:
```
```

保存结果

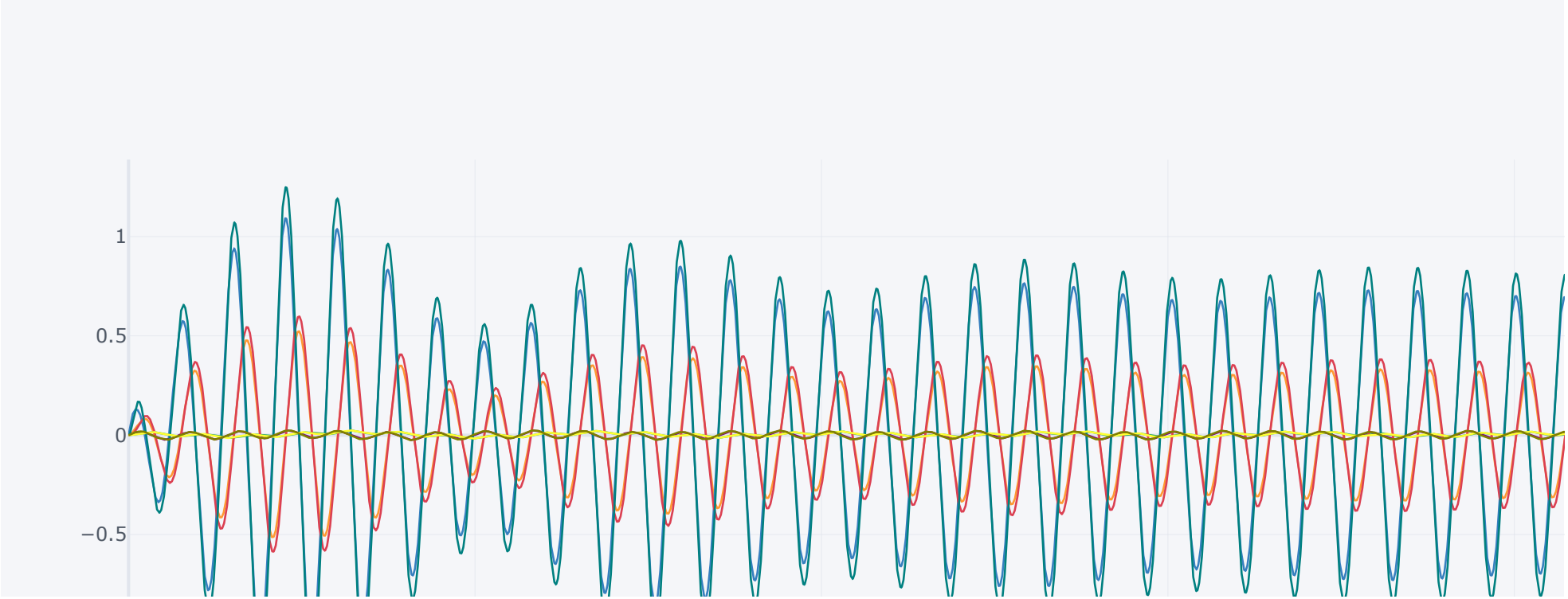
In [157…
```
# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result3_df(t=t, result3=res3):
    columns = ['时间 (s)', '浮子垂荡位移 (m)', '浮子垂荡速度 (m/s)', '振子垂荡位移 (m)', '振子垂荡速度 (m/s)',
               '浮子纵摇角位移 (rad)', '浮子纵摇角速度 (rad/s)', '振子纵摇角位移 (rad)', '振子纵摇角速度 (rad/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result3 = pd.DataFrame(result3, columns=columns[1:])
```

```
    result3_df = pd.concat([shijian, result3], axis=1)
    result3_df = result3_df.iloc[:, [0, 1, 2, 5, 6, 3, 4, 7, 8]]
    return result3_df
def save_result3_df(result3_df=get_result3_df()):
    print()
    result3 = result3_df.iloc[:int(40 * 波浪周期 / 时间间隔) + 1:int(0.2 / 时间间隔), :]
#     print(result3)
    result3.to_csv('result3.csv', encoding='utf_8_sig')
    #  10 s、20 s、40 s、60 s、100 s
    idx = list(map(lambda x: int(x / 时间间隔), [10, 20, 40, 60, 100]))
    result3_paper = result3_df.iloc[idx, :]
#     print(result3_paper)
    result3_paper.to_csv('result3-paper.csv', encoding='utf_8_sig')
    return result3_df

result3_df = save_result3_df()
```

绘图

In [157…

```
result3_df.iplot(x='时间 (s)')
```



In [ ]:

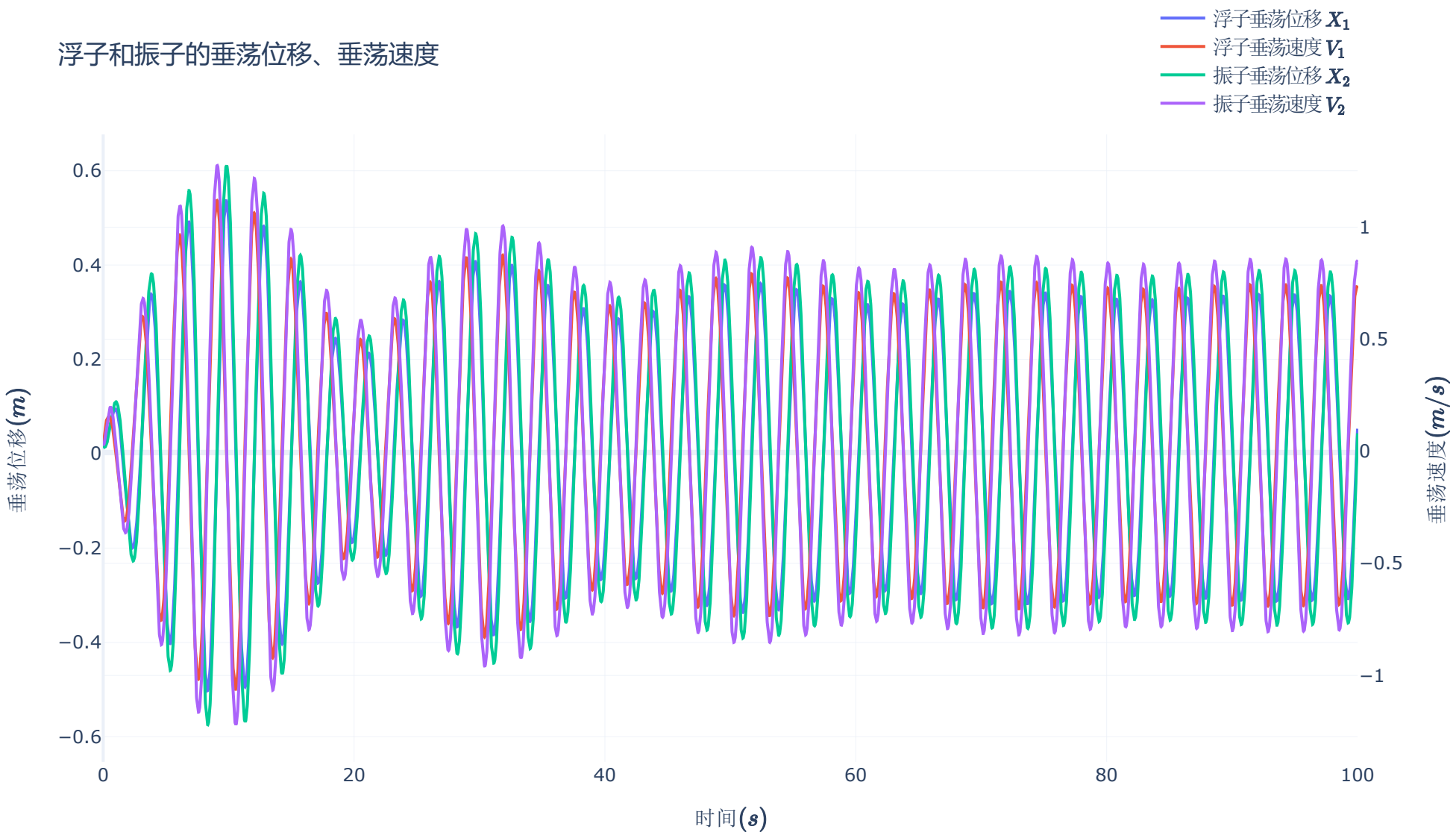
In [157…

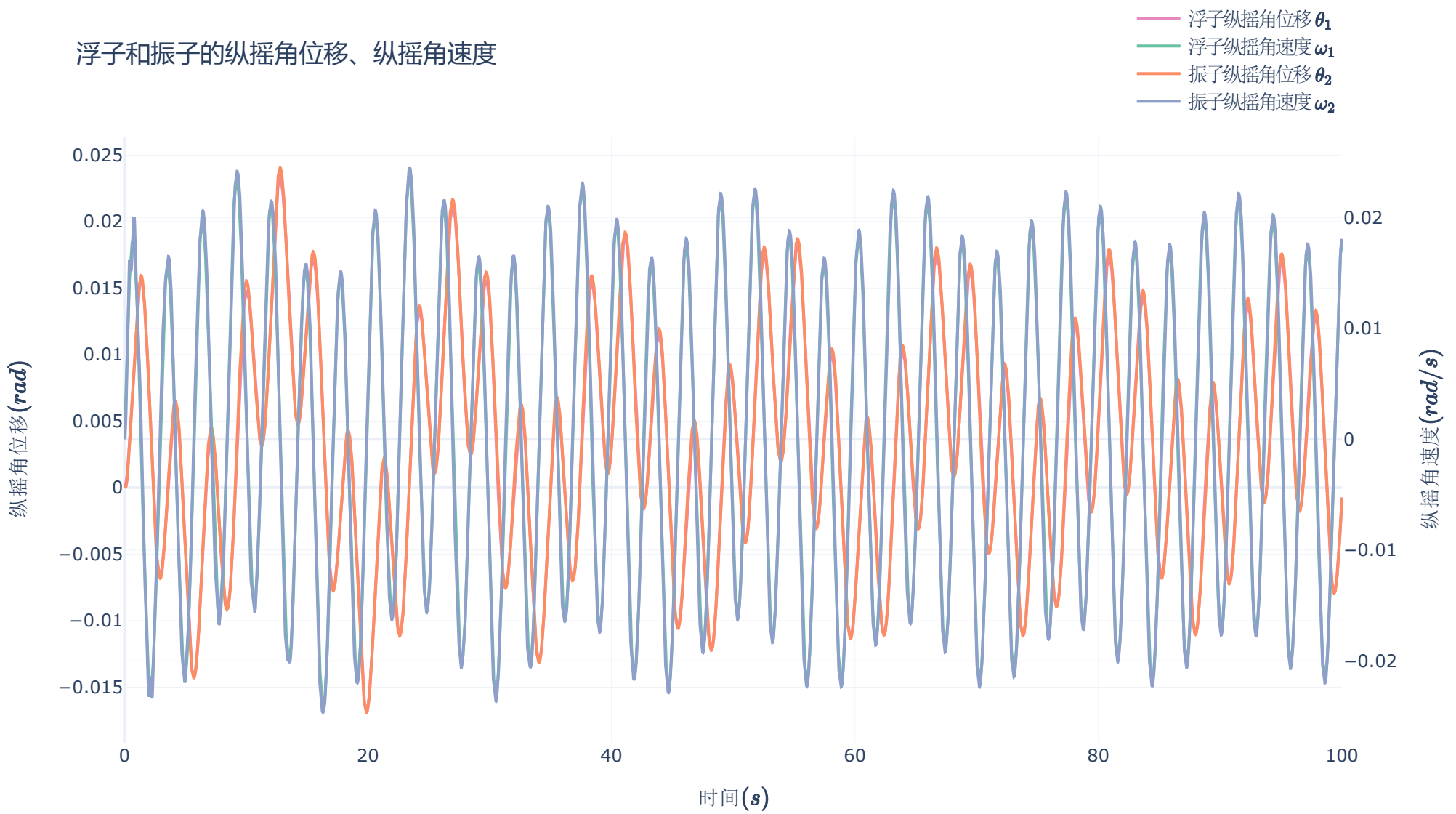
```python
# 位移 速度
def plot_plotly_xv(title, t=t, y=res3, svg_name=None):
    trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子垂荡位移 ~ X_1$", yaxis='y1')
    trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子垂荡速度 ~ V_1$", yaxis='y2')
    trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子垂荡位移 ~ X_2$", yaxis='y1')
    trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子垂荡速度 ~ V_2$", yaxis='y2')
    fig = go.Figure(data=[trace1, trace2, trace3, trace4])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$垂荡位移 (m)$'),
```

```
        yaxis2=dict(title='$垂荡速度 (m/s)$', anchor='x', overlaying='y', side='right'),
        legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
        title=title,
        template='plotly_white',
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name)
    fig.show()
    return None
def plot_plotly_rw(title, t=t, y=res3, svg_name=None):
    trace1 = go.Scatter(x=t, y=y[:, 4], name="$浮子纵摇角位移~\\theta_1$", yaxis='y1', line=dict(color='rgb(232,137,189)'))
    trace2 = go.Scatter(x=t, y=y[:, 5], name="$浮子纵摇角速度~ \omega_1$", yaxis='y2', line=dict(color='rgb(103,194,163)'))
    trace3 = go.Scatter(x=t, y=y[:, 6], name="$振子纵摇角位移~\\theta_2$", yaxis='y1', line=dict(color='rgb(252,140,99)'))
    trace4 = go.Scatter(x=t, y=y[:, 7], name="$振子纵摇角速度~ \omega_2$", yaxis='y2', line=dict(color='rgb(142,160,201)'))
    fig = go.Figure(data=[trace1, trace2, trace3, trace4])
    fig.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$纵摇角位移 (rad)$'),
        yaxis2=dict(title='$纵摇角速度 (rad/s)$', anchor='x', overlaying='y', side='right'),
        legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
        title=title,
        template='plotly_white',
    )
    if svg_name is not None:
        fig.write_image(IMG_SVG / svg_name)
    fig.show()
    return None
```

In [157…

```
plot_plotly_xv('浮子和振子的垂荡位移、垂荡速度', svg_name='问题3-浮子和振子的垂荡位移、垂荡速度.svg')
plot_plotly_rw('浮子和振子的纵摇角位移、纵摇角速度', svg_name='问题3-浮子和振子的纵摇角位移、纵摇角速度.svg')
```


浮子和振子的垂荡位移、垂荡速度
浮子垂荡位移 X_1
浮子垂荡速度 V_1
振子垂荡位移 X_2
振子垂荡速度 V_2
垂荡位移(m)
垂荡速度(m/s)
时间(s)
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
1
0.5
0
−0.5
−1
0
20
40
60
80
100



In [106…

```python
# 相对位移 相对速度
def plot_plotly_diff_xvrw(t=t, y=res3, save=False, show=False):
    trace1 = go.Scatter(x=t, y=y[:, 0]-y[:, 2], name="$浮子和振子的相对垂荡位移 ~ X$", line=dict(color='#8ECFC9'))
    trace2 = go.Scatter(x=t, y=y[:, 1]-y[:, 3], name="$浮子和振子的相对垂荡速度 ~ X$", line=dict(color='#FA7F6F'))
    trace3 = go.Scatter(x=t, y=y[:, 4]-y[:, 6], name="$浮子和振子的相对纵摇角位移~\\theta$", line=dict(color='#82B0D2'))
    trace4 = go.Scatter(x=t, y=y[:, 5]-y[:, 7], name="$浮子和振子的相对纵摇角速度~ \omega$", line=dict(color='#BEB8DC'))
    fig1 = go.Figure(data=[trace1])
    fig2 = go.Figure(data=[trace2])
```

```
    fig3 = go.Figure(data=[trace3])
    fig4 = go.Figure(data=[trace4])
    fig1.update_layout(
        width=1000, height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='相对垂荡位移 (m)'),
        title='$浮子和振子的相对垂荡位移$',
    )
    fig2.update_layout(
        width=1000, height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='相对垂荡速度 (m/s)'),
        title='$浮子和振子的相对垂荡速度$',
    )
    fig3.update_layout(
        width=1000, height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='相对纵摇角位移 (rad)'),
        title='$浮子和振子的相对纵摇角位移$',
    )
    fig4.update_layout(
        width=1000, height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='相对纵摇角速度 (rad/s)'),
        title='$浮子和振子的相对纵摇角速度$',
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         template='plotly_white',
    )

    if save:
        fig1.write_image(IMG_SVG / svg_name)
        fig2.write_image(IMG_SVG / svg_name)
        fig3.write_image(IMG_SVG / svg_name)
        fig4.write_image(IMG_SVG / svg_name)
    if show:
        fig1.show()
        fig2.show()
        fig3.show()
        fig4.show()
    return None
```

In [107…

```
# plot_plotly_diff_xvrw(save=False, show=True)
```

问题4

参数

In [147…

```
# TODO 4
浮子质量 = 4866  # kg
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m
```

In [147…

```
# TODO 3
question1234 = 4
if question1234 == 1:
    # 问题1：参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2：参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
elif question1234 == 3:
    # 问题3：参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    垂荡兴波阻尼系数 = 683.4558
    垂荡激励力振幅 = 3640
```

```
    纵摇附加转动惯量 = 7001.914
    纵摇兴波阻尼系数 = 654.3383
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4: 参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140
```

In [147…

```
# TODO 问题4: 参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量
f = 垂荡激励力振幅
omega = 入射波浪频率

# c = 10000  # 直线阻尼器
k = 弹簧刚度
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积

# C = 1000  # 旋转阻尼器
L = 纵摇激励力矩振幅
K = 扭转弹簧刚度
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数
```

查看何时稳定

In [147…

```
c = 10000  # 直线阻尼器
C = 1000  # 旋转阻尼器

def ode_func(y, t):
    dy1 = y[2-1]
    dy3 = y[4-1]
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])
```

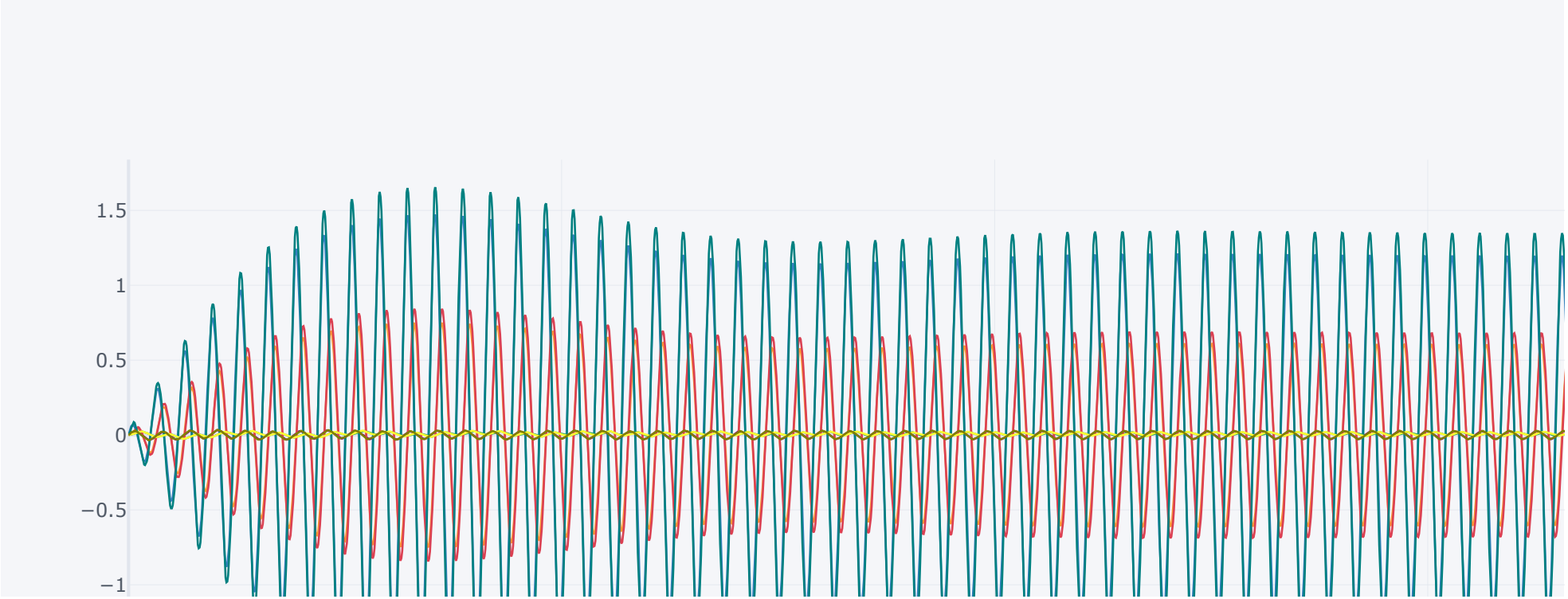
```
    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]
```

In [120…

```
时间间隔 = 0.1
_l, _r = 0, 200
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res4 = odeint(ode_func, y0, t)
```

In [120…

```
# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result4_df(t=t, result4=res4):
    columns = ['时间 (s)', '浮子垂荡位移 (m)', '浮子垂荡速度 (m/s)', '振子垂荡位移 (m)', '振子垂荡速度 (m/s)',
               '浮子纵摇角位移 (rad)', '浮子纵摇角速度 (rad/s)', '振子纵摇角位移 (rad)', '振子纵摇角速度 (rad/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result4 = pd.DataFrame(result4, columns=columns[1:])
    result4_df = pd.concat([shijian, result4], axis=1)
    result4_df = result4_df.iloc[:, [0, 1, 2, 5, 6, 3, 4, 7, 8]]
    return result4_df
# def save_result3_df(result3_df=get_result3_df()):
#     # 40 周期，间隔 0.2
#     result3 = result3_df.iloc[:int(40 * 波浪周期 / 时间间隔) + 1, :]
#     result3.to_csv('result3.csv', encoding='utf_8_sig')
#     #  10 s、20 s、40 s、60 s、100 s
#     idx = list(map(lambda x: x * 5, [10, 20, 40, 60, 100]))
#     result3_paper = result3_df.iloc[idx, :]
#     result3_paper.to_csv('result3-paper.csv', encoding='utf_8_sig')
#     return result3_df

result4_df = get_result4_df()
result4_df.iplot(x='时间 (s)', )
```



平均输出功率公式

平均输出功率：$P = \frac{1}{t_2 - t_1} \int_{t1}^{t2} F_G \Delta v + M_G \Delta \omega dt = \frac{1}{t_2 - t_1} \int_{t1}^{t2} c \Delta v^2 + C \Delta \omega^2 dt$

In [147…

```
# TODO 数值解：在 ode 里面计算 power_i

时间间隔 = 0.1
_l, _r = 0, 150

def ode_func(y, t, c, C, k=弹簧刚度, K=扭转弹簧刚度):
    dy1 = y[2-1]
```

```
    dy3 = y[4-1]
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])

    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]
def Power_func(res, c, C):
    """
    :param res: [X1, X1', X2, X2', theta1, theta1', theta2, theta2']
                [X1, V1, X2, V2, theta1, omega1, theta2, omega2]
    :return: P
    """
    begin = int(100 / 时间间隔)
    end = -1  # int(150 / 时间间隔)

    delta_V = (res[:, 1] - res[:, 3])**2
    delta_omega = (res[:, 5] - res[:, 7])**2
    power_i = c * delta_V + C * delta_omega
    power_ = power_i[begin:end]

#     power = power_ * 时间间隔  # 矩形
#     P = power.sum() / 50

    power = (power_[1:] + power_[:-1]) * 时间间隔 / 2  # 梯形
    P = power.sum() / (50 - 时间间隔)

    return P
def optimize_func(c, C, pprint=True):
    """ sko 遗传目标函数
    :param zunixishu: 阻尼系数/比例系数
    :param power_i: list
    :param period: 计算平均功率的时间区间类型
    """
    res = odeint(
        ode_func,
        [0 for _ in range(8)],
        np.linspace(_l, _r, num=int(_r / 时间间隔) + 1),
```

```
        args=(c, C)
    )  # zunixishu
    P = Power_func(res, c, C)
    if pprint:
        print("平均输出功率：", P)
    target = -P
    return target
```

遗传

In [121…

```
lb, ub = 0, 1e5
ga = GA(
    func=optimize_func,
    n_dim=2,
    size_pop=50,
    max_iter=200,
    prob_mut=0.001,
    lb=[lb, lb],
    ub=[ub, ub],
    constraint_eq=tuple(),
    constraint_ueq=tuple(),
    precision=1e-06,
    early_stop=True,
)
t0 = time()
xbest, ybest = ga.run()
t1 = time()
print("(直线阻尼器的阻尼系数，旋转阻尼器的阻尼系数)：", xbest, "最大输出功率", -ybest)
print("总用时：", t1 - t0)
```

平均输出功率： 313.47911625416623
平均输出功率： 292.77133807052735
平均输出功率： 312.95096791634245
平均输出功率： 315.4628724768086
平均输出功率： 251.62656581861697
平均输出功率： 298.39416238502633
平均输出功率： 308.89864065588296
平均输出功率： 314.1918498069244
平均输出功率： 149.54946406927814
平均输出功率： 11.405983464639142
平均输出功率： 214.33407667319528
平均输出功率： 302.0911224036704
平均输出功率： 297.4423103740071
平均输出功率： 286.27902648767656
平均输出功率： 307.2723907448717
平均输出功率： 315.2844860499195
平均输出功率： 313.1194273901596
平均输出功率： 158.7013216160529
平均输出功率： 315.35068868908934
平均输出功率： 314.22200735108794
平均输出功率： 315.44039440304954
平均输出功率： 302.11262882228743
平均输出功率： 297.772845033763
平均输出功率： 315.05223418865904
平均输出功率： 314.58333437860045
平均输出功率： 299.193574446875
平均输出功率： 299.9452965874831
平均输出功率： 291.61860187574706
平均输出功率： 272.1051881572125
平均输出功率： 193.04430035354676
平均输出功率： 291.4038108467862
平均输出功率： 284.3350679867566
平均输出功率： 305.0983146217583
平均输出功率： 66.28816672762645
平均输出功率： 303.26656602198636
平均输出功率： 292.4723091900931
平均输出功率： 292.8644555713518
平均输出功率： 290.40625693873
平均输出功率： 49.510054584229174
平均输出功率： 315.62203529358277
平均输出功率： 302.44328094162967
平均输出功率： 263.12739345105973
平均输出功率： 31.512688778142778
平均输出功率： 285.9156215031698
平均输出功率： 291.3705695833081
平均输出功率： 127.15235921201942

```
平均输出功率： 308.9292795942929
平均输出功率： 295.9898783741033
平均输出功率： 293.70660591643565
平均输出功率： 288.73504087865376
平均输出功率： 315.62203529358277
(直线阻尼器的阻尼系数，旋转阻尼器的阻尼系数)： [60804.32100979   403.04781942] 最大输出功率 [315.62203529]
总用时： 27.506685495376587
```

In [149…
```
# -optimize_func(60804.32100979, 50648.04781942)
-optimize_func(58000, 45000)
-optimize_func(58000, 49608)
-optimize_func(58000, 49608.1)
-optimize_func(58010, 49608.2)
None
```

```
平均输出功率： 315.82624549004527
平均输出功率： 315.8259735259724
平均输出功率： 315.826100738722
平均输出功率： 315.8265439696769
```

In [152…
```
optimize_func(58467.74816206559, 47662.22129472)
```

```
平均输出功率： 315.833869128142
```

Out[1525]: -315.833869128142

In [152…
```
optimize_func(58466.54816206559, 47662.22129472)
```

```
平均输出功率： 315.83410819043434
```

Out[1526]: -315.83410819043434

In [152…
```
optimize_func(58467.64816206559, 47662.22129472)
```

```
平均输出功率： 315.83428685207804
```

Out[1522]: -315.83428685207804

变步长

In [151…
```
num = 10

@jit
def run_travel_get_xy(num=num):
    pss = []
    t00 = time()
```

```
    t0 = time()
    n_ = 10
    n = num // n_
    cs0 = 58467.62477501568
    CS0 = 47662.22129472001
    cs = np.linspace(cs0 * (1-0.000001), cs0 * (1+0.000001), num+1)
    CS = np.linspace(CS0 * (1-0.000001), CS0 * (1+0.000001), num+1)
    for i, c in enumerate(cs):
        if i % n == 0:
            t1 = time()
            _stars = '[' + '*' * (i // n_) + '.' * (num // n_ - i // n_) + ']'
            print("%6d/%6d" % (i, num), _stars, round(t1 - t0, 2), "s/iter")
            t0 = time()

        ps = []
        for j, C in enumerate(CS):
            p = -optimize_func(c, C, pprint=False)
            ps.append(p)
        pss.append(ps)
    t11 = time()
    print('总用时：', t11 - t00)
    return cs, CS, pss
```

In [152…

```
x41, x42, y4 = run_travel_get_xy()
```

```
     0/    10 [.] 0.67 s/iter
     1/    10 [.] 5.56 s/iter
     2/    10 [.] 5.39 s/iter
     3/    10 [.] 5.56 s/iter
     4/    10 [.] 5.4 s/iter
     5/    10 [.] 5.48 s/iter
     6/    10 [.] 5.38 s/iter
     7/    10 [.] 5.35 s/iter
     8/    10 [.] 5.47 s/iter
     9/    10 [.] 5.3 s/iter
    10/    10 [*] 5.49 s/iter
总用时： 60.5766441822052
```

In [152…

```
y4_np = np.array(y4)
# np.save("question2-y.npy", y_np)
index = np.unravel_index(y4_np.argmax(), y4_np.shape)
print(y4_np.max())
print(index[0], "\tx1:", x41[index[0]])
print(index[1], "\tx2:", x42[index[1]])
print(index, "\ty:", y4_np[index[::]])
```

```
315.83428685207804
7       x1: 58467.64816206559
5       x2: 47662.22129472
(7, 5)  y: 315.83428685207804
```

总用时： 71.84193420410156
315.8341932877408
9 x1: 58464.0
5 x2: 47700.0
(9, 5) y: 315.8341932877408

总用时： 58.40457725524902
315.8342183096599
5 x1: 58464.0
1 x2: 47661.840000000004
(5, 1) y: 315.8342183096599

总用时： 60.35779786109924
315.8342190462612
8 x1: 58467.50784
5 x2: 47661.840000000004
(8, 5) y: 315.8342190462612

总用时： 60.237072467803955
315.8342773709658
6 x1: 58467.62477501568
9 x2: 47662.22129472001
(6, 9) y: 315.8342773709658

总用时： 59.65555119514465
315.83422585714607
5 x1: 58467.62477501569
7 x2: 47662.41194360519
(5, 7) y: 315.83422585714607

总用时： 59.65555119514465
315.83422585714607
5 x1: 58467.62477501569

7 x2: 47662.41194360519
(5, 7) y: 315.83422585714607

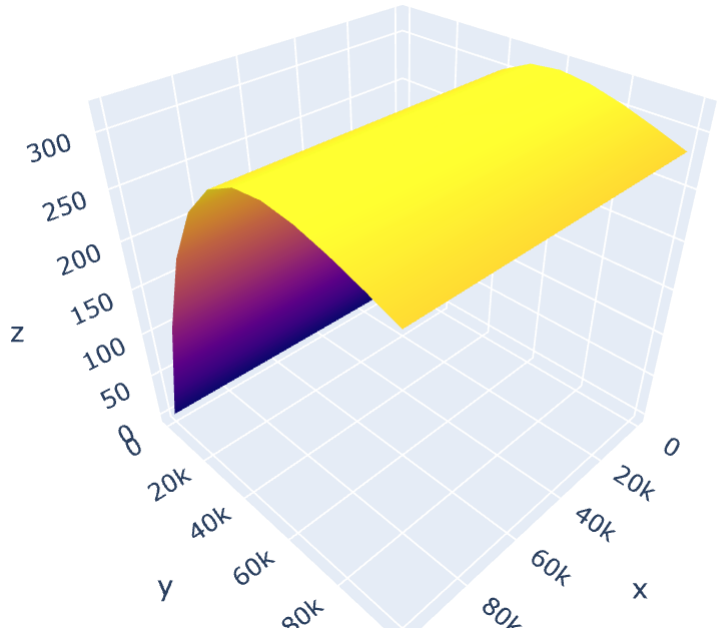
总用时： 60.5766441822052
315.83428685207804
7 x1: 58467.64816206559
5 x2: 47662.22129472
(7, 5) y: 315.83428685207804

GA-LMS：直线：58467.64816206559，旋转：47662.22129472，功率：315.83428685207804，用时：60.5766441822052

In [121…
```
np.save("question4-y.npy", y4)
```

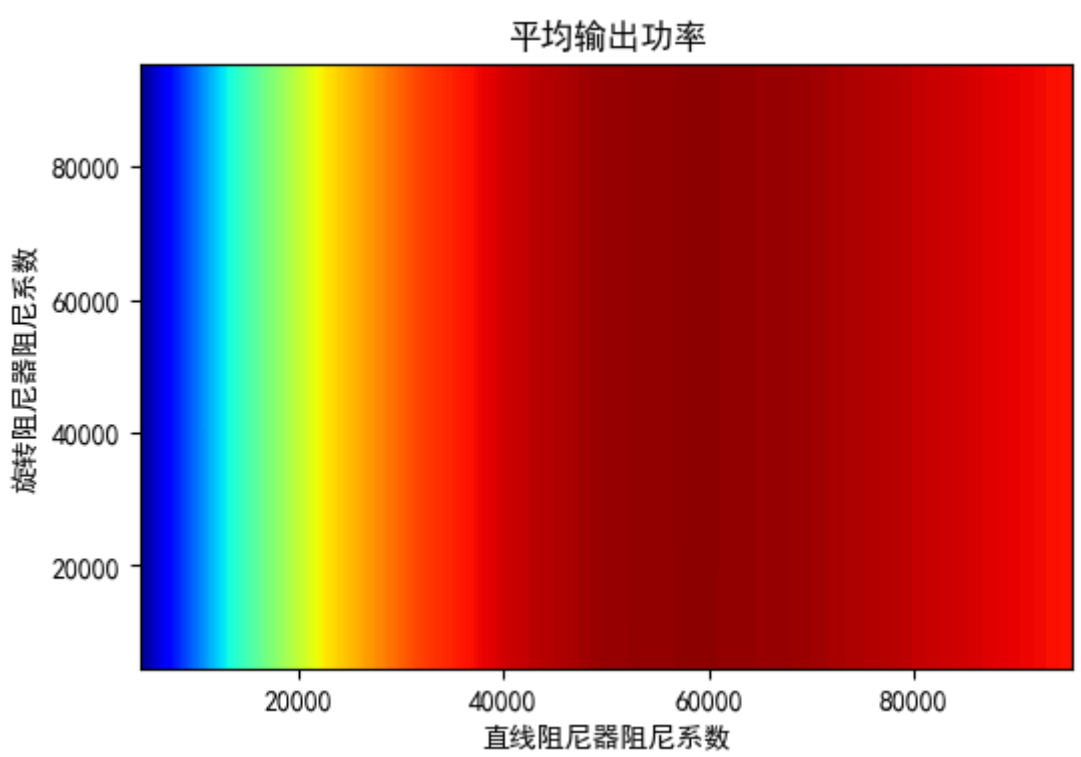
绘图

In [121…
```
y4 = np.load('question4-y.npy')

fig1 = go.Figure(data=[go.Surface(
    x=np.linspace(0, 100000, num+1),
    y=np.linspace(0, 100000, num+1),
    z=y4,
)])
# fig1.write_image(IMG_SVG / '问题4-3D曲线图.svg')
fig1.show()
```



In [121…

```
y = np.load('question4-y.npy')
plt.figure(dpi=100)
plt.contourf(y.T, np.arange(0, 320, 1.5), origin='lower', extent=[0, 100000, 0, 100000], cmap=plt.cm.jet)
plt.xlabel('直线阻尼器阻尼系数')
plt.ylabel('旋转阻尼器阻尼系数')
plt.title("平均输出功率")
# plt.savefig(IMG_SVG / '问题4-二维等高线图.svg')
plt.show()
```



In [ ]:

In [ ]:

检验与分析

检验

吻合度

$$wenhedu = 1 - \frac{y1 - y2}{y1}$$

问题1参数

In [128…

```python
def wenhedu_func(y1, y2):
    """
    :param y1: 问题1模型的结果
```

```
    :param y2: 退化的问题3模型的结果
    :return: 吻合度
    """
    return 1 - (y1 - y2) / y1
def wenhedu_avg(res1, res3):
    avg = 1 - (res1[:, :4] - res3[:, :4]) / res1[:, :4]
    return avg.mean()
```

In [122…

```
# TODO 参数

# TODO 4
浮子质量 = 4866  # kg
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m


# TODO 3
question1234 = 1  # """ 问题 1 的参数 """
if question1234 == 1:
    # 问题1：参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2：参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
```

```
elif question1234 == 3:
    # 问题3: 参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    垂荡兴波阻尼系数 = 683.4558
    垂荡激励力振幅 = 3640

    纵摇附加转动惯量 = 7001.914
    纵摇兴波阻尼系数 = 654.3383
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4: 参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140


# TODO 检验: 参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量
f = 垂荡激励力振幅
omega = 入射波浪频率

# c = 10000  # 直线阻尼器
k = 弹簧刚度
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积

# C = 1000  # 旋转阻尼器
L = 纵摇激励力矩振幅
K = 扭转弹簧刚度
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数
```

In [132…

```
时间间隔 = 0.2
_l, _r = 0, 400
```

问题1模型

In [132…

```
def diff_equation(ys, t):
```

```
    y1 = y2 = y3 = y4 = 0

    y1 = ys[2-1]
    y3 = ys[4-1]

    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2

    y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - (c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1]))
    y2 = y2 / m1_
    return [y1, y2, y3, y4]

def get_result1_1_df(t, result1_1):
    columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result1_1 = pd.DataFrame(result1_1, columns=columns[1:])
    result1_1_df = pd.concat([shijian, result1_1], axis=1)
    return result1_1_df

# 时间间隔 = 0.2
# _l, _r = 0, 200
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0, 0, 0, 0]
result1 = odeint(diff_equation, y0, t)
result1_df = get_result1_1_df(t, result1)
result1_df
```

Out[1322]:

	时间 (s)	浮子位移 (m)	浮子速度 (m/s)	振子位移 (m)	振子速度 (m/s)
0	0.0	0.000000	0.000000	0.000000	0.000000
1	0.2	0.017848	0.163187	0.005633	0.081593
2	0.4	0.059669	0.243720	0.037968	0.241117
3	0.6	0.111698	0.268485	0.097504	0.336066
4	0.8	0.163602	0.240825	0.164661	0.317472
...	...	...	...	...	...
1996	399.2	0.417679	0.112626	0.442369	0.126207
1997	399.4	0.423633	-0.053477	0.450041	-0.049983
1998	399.6	0.396567	-0.215411	0.422636	-0.222278
1999	399.8	0.338590	-0.360555	0.362288	-0.377248
2000	400.0	0.254223	-0.477595	0.273702	-0.502813

2001 rows × 5 columns

In [ ]:

In [ ]:

问题3模型

In [132…

```
# TODO 检验：参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量
f = 垂荡激励力振幅
omega = 入射波浪频率

# c = 10000  # 直线阻尼器
k = 弹簧刚度
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积

# C = 1000  # 旋转阻尼器
L = 纵摇激励力矩振幅
K = 扭转弹簧刚度
```

```
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数
```

In [132…]
```
def ode_func(y, t):
    dy1 = y[2-1]
    dy3 = y[4-1]
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

#     j2 = j2_func(y[3-1])

#     dy5 = y[6-1]
#     dy7 = y[8-1]
#     dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
#     dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
#     dy6 /= j1 + je
#     dy8 /= j2
    return [dy1, dy2, dy3, dy4, 0, 0, 0, 0]
```

In [ ]:
```
```

In [132…]
```
# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result3_df(t, result3):
    columns = ['时间 (s)', '浮子垂荡位移 (m)', '浮子垂荡速度 (m/s)', '振子垂荡位移 (m)', '振子垂荡速度 (m/s)',
               '浮子纵摇角位移 (rad)', '浮子纵摇角速度 (rad/s)', '振子纵摇角位移 (rad)', '振子纵摇角速度 (rad/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result3 = pd.DataFrame(result3, columns=columns[1:])
    result3_df = pd.concat([shijian, result3], axis=1)
    result3_df = result3_df.iloc[:, [0, 1, 2, 5, 6, 3, 4, 7, 8]]
    return result3_df
```

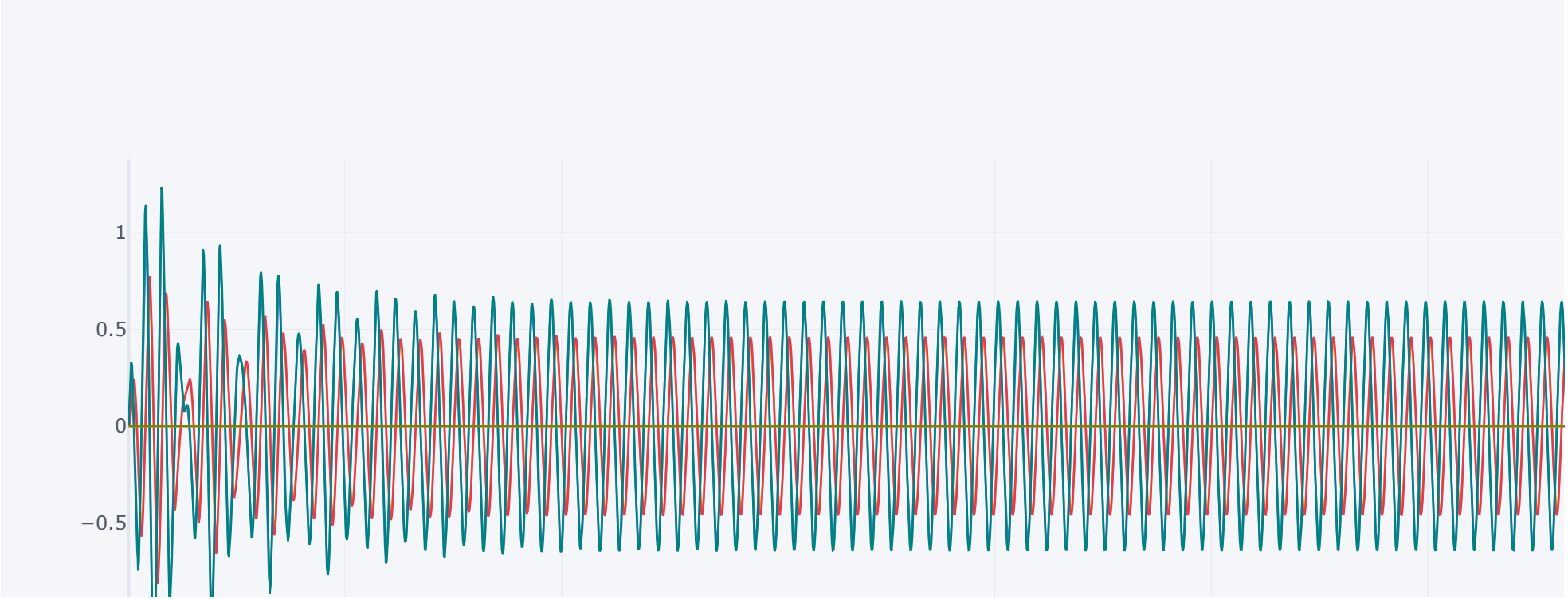
In [132…]
```
# 时间间隔 = 0.2
# _l, _r = 0, 200
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res3 = odeint(ode_func, y0, t)
```

In [ ]:
```
```

In [132…
```
import mitosheet

# mitosheet.sheet(result1_df, result3_df, analysis_to_replay="id-slppdpmjso")
```

In [ ]:
```

```

In [132…
```
result3_df = get_result3_df(t, res3)
result3_df.iplot(x='时间 (s)')
```



In [ ]:
```

```

计算吻合度

```

```

In [132…
```
result1_df
```

Out[1329]:

	时间 (s)	浮子位移 (m)	浮子速度 (m/s)	振子位移 (m)	振子速度 (m/s)
0	0.0	0.000000	0.000000	0.000000	0.000000
1	0.2	0.017848	0.163187	0.005633	0.081593
2	0.4	0.059669	0.243720	0.037968	0.241117
3	0.6	0.111698	0.268485	0.097504	0.336066
4	0.8	0.163602	0.240825	0.164661	0.317472
...	...	...	...	...	...
1996	399.2	0.417679	0.112626	0.442369	0.126207
1997	399.4	0.423633	-0.053477	0.450041	-0.049983
1998	399.6	0.396567	-0.215411	0.422636	-0.222278
1999	399.8	0.338590	-0.360555	0.362288	-0.377248
2000	400.0	0.254223	-0.477595	0.273702	-0.502813

2001 rows × 5 columns

In [133…
```
result3_df
```

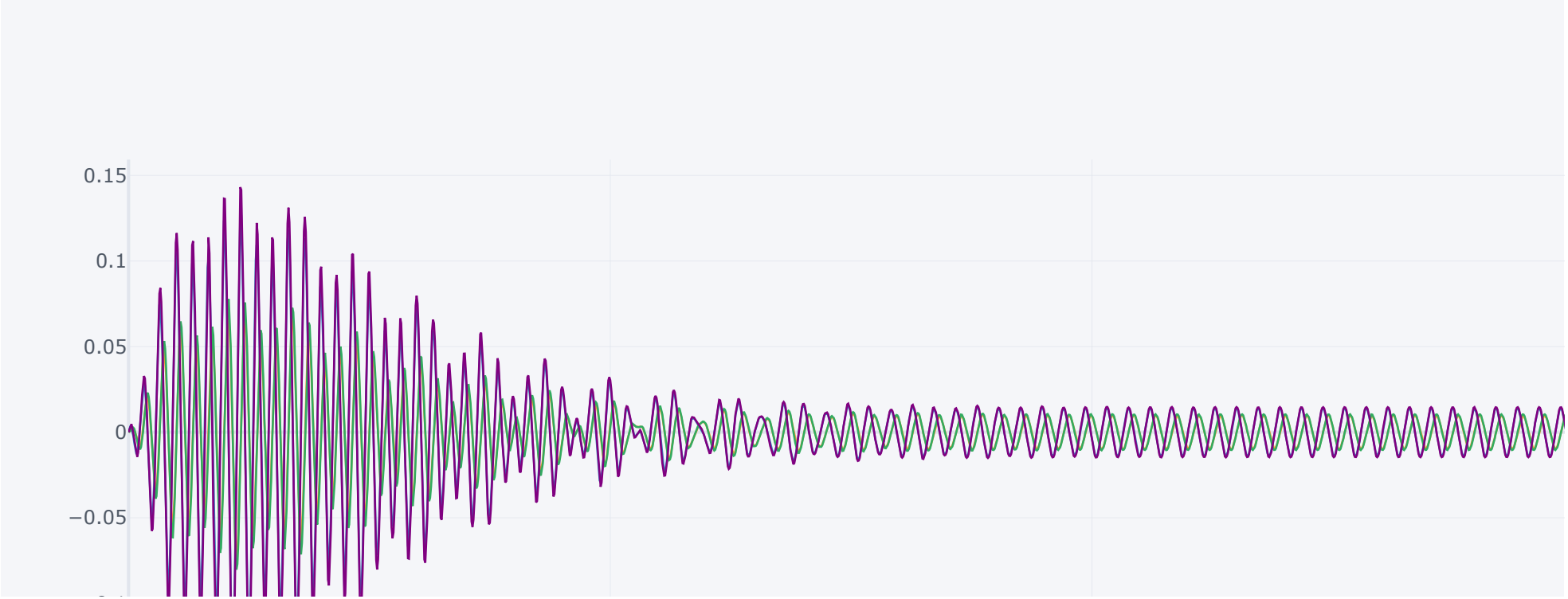
Out[1330]:

	时间 (s)	浮子垂荡位移 (m)	浮子垂荡速度 (m/s)	浮子纵摇角位移 (rad)	浮子纵摇角速度 (rad/s)	振子垂荡位移 (m)	振子垂荡速度 (m/s)	振子纵摇角位移 (rad)	振子纵摇角速度 (rad/s)
0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.0
1	0.2	0.017426	0.159684	0.0	0.0	0.005494	0.079661	0.0	0.0
2	0.4	0.058488	0.239968	0.0	0.0	0.037146	0.236482	0.0	0.0
3	0.6	0.109852	0.265720	0.0	0.0	0.095694	0.331338	0.0	0.0
4	0.8	0.161378	0.240003	0.0	0.0	0.162119	0.315185	0.0	0.0
...	...	...	...	...	...	...	...	...	...
1996	399.2	0.427201	0.116146	0.0	0.0	0.452447	0.130095	0.0	0.0
1997	399.4	0.433479	-0.053781	0.0	0.0	0.460494	-0.050149	0.0	0.0
1998	399.6	0.405969	-0.219515	0.0	0.0	0.432648	-0.226484	0.0	0.0
1999	399.8	0.346816	-0.368139	0.0	0.0	0.371080	-0.385166	0.0	0.0
2000	400.0	0.260631	-0.488069	0.0	0.0	0.280588	-0.513827	0.0	0.0

2001 rows × 9 columns

In [133…

```
temp = pd.DataFrame(result1_df.values - result3_df.iloc[:, [0, 1, 2, 5, 6]].values)
temp.iloc[:, 1:].iplot()
```



In [134…
```
wenhe_avg = 1 - abs(result1[1:, :4] - res3[1:, :4]) / abs(result1[1:, :4])
wenhe_avg = wenhe_avg[800:1000, :]
wenhe_avg.mean(0)
```

Out[1346]:
```
array([0.97412908, 0.96171154, 0.97501944, 0.97279231])
```

In [133…
```
wenhe_avg[116:, :]
```

Out[1333]:
```
array([[0.97939823, 0.97585433, 0.97944912, 0.97585713],
       [0.97815144, 0.97531178, 0.9781641 , 0.97532835],
       [0.97750318, 0.97436294, 0.9775064 , 0.97441647],
       ...,
       [0.97629011, 0.98094606, 0.97630945, 0.98107883],
       [0.97570539, 0.97896336, 0.97573182, 0.97901009],
       [0.97479328, 0.9780691 , 0.97483987, 0.97809557]])
```

In [ ]:

问题2参数

In [154…

```python
# TODO 参数

# TODO 4
浮子质量 = 4866  # kg
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m

# TODO 3
question1234 = 2  # """ 问题 1 的参数 """
if question1234 == 1:
    # 问题1: 参数
    # 纵摇附加转动惯量 = 6779.315  # kg·m^2
    # 纵摇兴波阻尼系数 = 151.4388  # N·m·s
    # 纵摇激励力矩振幅 = 1230  # N·m
    入射波浪频率 = 1.4005  # s^{-1}
    垂荡附加质量 = 1335.535  # kg
    垂荡兴波阻尼系数 = 656.3616  # N·s/m
    垂荡激励力振幅 = 6250  # N
elif question1234 == 2:
    # 问题2: 参数
    # 纵摇附加转动惯量 = 7131.29
    # 纵摇兴波阻尼系数 = 2992.724
    # 纵摇激励力矩振幅 = 2560
```

```
    入射波浪频率 = 2.2143
    垂荡附加质量 = 1165.992
    垂荡兴波阻尼系数 = 167.8395
    垂荡激励力振幅 = 4890
elif question1234 == 3:
    # 问题3：参数
    入射波浪频率 = 1.7152
    垂荡附加质量 = 1028.876
    垂荡兴波阻尼系数 = 683.4558
    垂荡激励力振幅 = 3640

    纵摇附加转动惯量 = 7001.914
    纵摇兴波阻尼系数 = 654.3383
    纵摇激励力矩振幅 = 1690
elif question1234 == 4:
    # 问题4：参数
    入射波浪频率 = 1.9806
    垂荡附加质量 = 1091.099
    纵摇附加转动惯量 = 7142.493
    垂荡兴波阻尼系数 = 528.5018
    纵摇兴波阻尼系数 = 1655.909
    垂荡激励力振幅 = 1760
    纵摇激励力矩振幅 = 2140

# TODO 检验：参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量
f = 垂荡激励力振幅
omega = 入射波浪频率

# c = 10000  # 直线阻尼器
c = 36572.41264494
k = 弹簧刚度
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积

# C = 1000  # 旋转阻尼器
L = 纵摇激励力矩振幅
K = 扭转弹簧刚度
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数

时间间隔 = 0.2
_l, _r = 0, 100
```

In [ ]:

模型1

In [155…

```python
def diff_equation(ys, t):
    y1 = y2 = y3 = y4 = 0

    y1 = ys[2-1]
    y3 = ys[4-1]

    y4 = c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1])
    y4 = y4 / m2

    y2 = f * np.cos(omega * t) + k1 * ys[2-1] + k2 * ys[1-1] - (c * (ys[2-1] - ys[4-1]) + k * (ys[1-1] - ys[3-1]))
    y2 = y2 / m1_
    return [y1, y2, y3, y4]

def get_result1_1_df(t, result1_1):
    columns = ['时间 (s)', '浮子位移 (m)', '浮子速度 (m/s)', '振子位移 (m)', '振子速度 (m/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result1_1 = pd.DataFrame(result1_1, columns=columns[1:])
    result1_1_df = pd.concat([shijian, result1_1], axis=1)
    return result1_1_df

# 时间间隔 = 0.2
# _l, _r = 0, 200
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0, 0, 0, 0]
result1 = odeint(diff_equation, y0, t)
result1_df = get_result1_1_df(t, result1)
result1_df
```

Out[1550]:

	时间 (s)	浮子位移 (m)	浮子速度 (m/s)	振子位移 (m)	振子速度 (m/s)
0	0.0	0.000000	0.000000	0.000000	0.000000
1	0.2	0.012624	0.115478	0.007593	0.090335
2	0.4	0.042944	0.178513	0.035153	0.176546
3	0.6	0.079704	0.177141	0.073745	0.196578
4	0.8	0.108945	0.103610	0.108448	0.137276
...	...	...	...	...	...
496	99.2	-0.437578	-0.184216	-0.462618	-0.281388
497	99.4	-0.430871	0.249673	-0.472242	0.186177
498	99.6	-0.341146	0.632366	-0.390776	0.614582
499	99.8	-0.186294	0.890355	-0.234566	0.921454
500	100.0	0.003335	0.974579	-0.034277	1.048291

501 rows × 5 columns

In [ ]:

模型3

In [155…

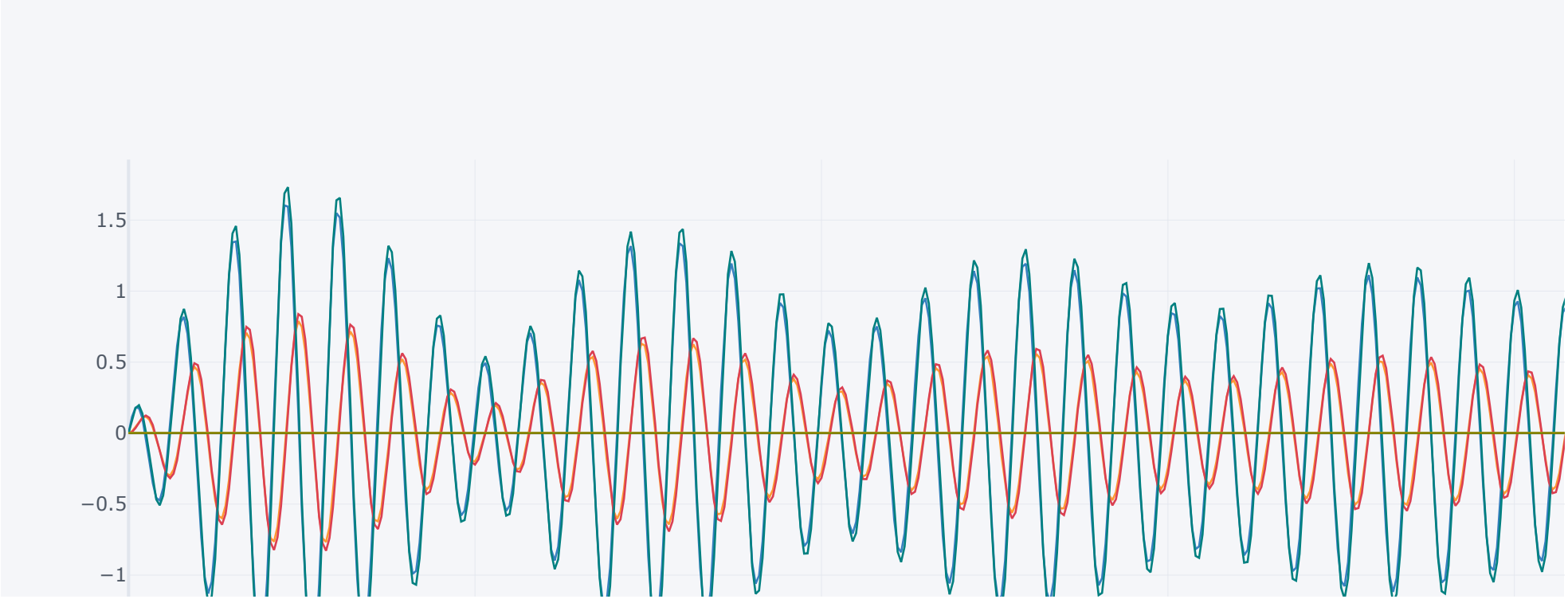
```python
def ode_func(y, t):
    dy1 = y[2-1]
    dy3 = y[4-1]
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

#     j2 = j2_func(y[3-1])

#     dy5 = y[6-1]
#     dy7 = y[8-1]
#     dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
#     dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
#     dy6 /= j1 + je
#     dy8 /= j2
    return [dy1, dy2, dy3, dy4, 0, 0, 0, 0]
```

In [155…

```
# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result3_df(t, result3):
    columns = ['时间 (s)', '浮子垂荡位移 (m)', '浮子垂荡速度 (m/s)', '振子垂荡位移 (m)', '振子垂荡速度 (m/s)',
               '浮子纵摇角位移 (rad)', '浮子纵摇角速度 (rad/s)', '振子纵摇角位移 (rad)', '振子纵摇角速度 (rad/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result3 = pd.DataFrame(result3, columns=columns[1:])
    result3_df = pd.concat([shijian, result3], axis=1)
    result3_df = result3_df.iloc[:, [0, 1, 2, 5, 6, 3, 4, 7, 8]]
    return result3_df

# 时间间隔 = 0.2
# _l, _r = 0, 200
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res3 = odeint(ode_func, y0, t)
```

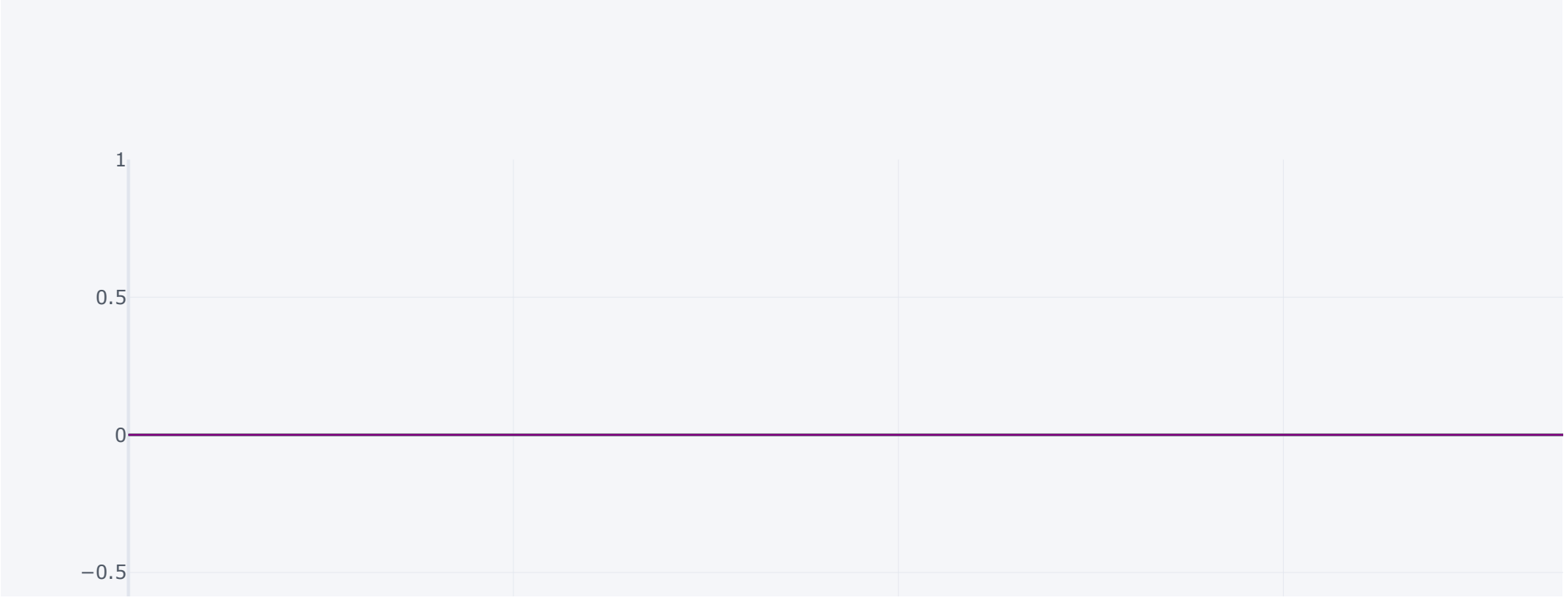
In [155…

```
result3_df = get_result3_df(t, res3)
result3_df.iplot(x='时间 (s)')
```



In [155…

```
temp = pd.DataFrame(result1_df.values - result3_df.iloc[:, [0, 1, 2, 5, 6]].values)
temp.iloc[:, 1:].iplot()
```



In [155…
```
# c = 36572.41264494
data_temp1 = result1_df.iloc[:, 1:].values
data_temp2 = result3_df.iloc[:, [1, 2, 5, 6]].values
deta_v1 = abs(data_temp1[:, 1] - data_temp1[:, 3])[60*5:100*5]
deta_v2 = abs(data_temp2[:, 1] - data_temp2[:, 3])[60*5:100*5]
power1 = c * (deta_v1**2).sum() * 时间间隔 / 40
power2 = c * (deta_v2**2).sum() * 时间间隔 / 40
power1, power2
```

Out[1555]: (227.1230051331009, 227.1230051331009)

In [ ]:

In [156…
```
wenhe_avg = 1 - abs(result1[1:, :4] - res3[1:, :4]) / abs(result1[1:, :4])
wenhe_avg = wenhe_avg[800:1000, :]
wenhe_avg.mean(0)
```

Out[1560]: 1.0

In [156…
```
wenhe_avg = 1 - abs(data_temp1 - data_temp2) / abs(data_temp1)
wenhe_avg = wenhe_avg[1:, :]
wenhe_avg.mean(0)
```

Out[1565]: array([1., 1., 1., 1.])

In [ ]:

In [ ]:

灵敏性分析

In [139…
```
# TODO 4
浮子质量 = 4866  # kg
浮子底半径 = 1  # m
浮子圆柱部分高度 = 3  # m
浮子圆锥部分高度 = 0.8  # m
振子质量 = 2433  # kg
振子半径 = 0.5  # m
振子高度 = 0.5  # m
海水的密度 = 1025  # kg/m^3
重力加速度 = 9.8  # m/s^2
弹簧刚度 = 80000  # N/m
弹簧原长 = 0.5  # m
扭转弹簧刚度 = 250000  # N·m
静水恢复力矩系数 = 8890.7  # N·m


# TODO 3
# 问题3：参数
入射波浪频率 = 1.7152
垂荡附加质量 = 1028.876
垂荡兴波阻尼系数 = 683.4558
垂荡激励力振幅 = 3640

纵摇附加转动惯量 = 7001.914
纵摇兴波阻尼系数 = 654.3383
```

```
纵摇激励力矩振幅 = 1690

# TODO 问题3：参数
m1, m2, me = 浮子质量, 振子质量, 垂荡附加质量
j2_func = lambda y3: ((0.969588 + y3)**3 - (0.469558 + y3)**3) * m2 / 1.5  # y3
j1, je = 33000, 纵摇附加转动惯量

c = 10000
k = 80000  # 80000
f = 垂荡激励力振幅
k1 = -垂荡兴波阻尼系数
k2 = -海水的密度 * 重力加速度 * S浮子底面积

C = 1000
L = 纵摇激励力矩振幅
K = 250000  # 250000
K1 = -纵摇兴波阻尼系数
K2 = -静水恢复力矩系数
```

In [141…

```
def ode_func(y, t, k=k, K=K):
    dy1 = y[2-1]
    dy3 = y[4-1]
    print(k, K)
    print(c * (y[2-1] - y[4-1] * np.cos(y[7-1])))
    print(k * (y[1-1] - y[3-1] * np.cos(y[7-1])))
    print(C * (y[6-1] - y[8-1]))
    print(K * (y[5-1] - y[7-1]))
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])

    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]
```

In [139…

```
时间间隔 = 0.2
_l, _r = 0, 100
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
```

```
res3 = odeint(ode_func, y0, t)

# TODO 保存结果
波浪频率 = 入射波浪频率
波浪周期 = 1 / 波浪频率

def get_result3_df(t=t, result3=res3):
    columns = ['时间 (s)', '浮子垂荡位移 (m)', '浮子垂荡速度 (m/s)', '振子垂荡位移 (m)', '振子垂荡速度 (m/s)',
               '浮子纵摇角位移 (rad)', '浮子纵摇角速度 (rad/s)', '振子纵摇角位移 (rad)', '振子纵摇角速度 (rad/s)']
    shijian = pd.DataFrame(t, columns=columns[:1])
    result3 = pd.DataFrame(result3, columns=columns[1:])
    result3_df = pd.concat([shijian, result3], axis=1)
    result3_df = result3_df.iloc[:, [0, 1, 2, 5, 6, 3, 4, 7, 8]]
    return result3_df
def save_result3_df(result3_df=get_result3_df()):
    print()
    result3 = result3_df.iloc[:int(40 * 波浪周期 / 时间间隔) + 1:int(0.2 / 时间间隔), :]
#     print(result3)
    result3.to_csv('result3.csv', encoding='utf_8_sig')

    idx = list(map(lambda x: int(x / 时间间隔), [10, 20, 40, 60, 100]))
    result3_paper = result3_df.iloc[idx, :]
#     print(result3_paper)
    result3_paper.to_csv('result3-paper.csv', encoding='utf_8_sig')
    return result3_df

# result3_df = save_result3_df()
```

绘图

In [139…

```
# result3_df.iplot(x='时间 (s)')
```

In [139…

```
# def plot_plotly_xv(title, t=t, y=res3, svg_name=None):
#     trace1 = go.Scatter(x=t, y=y[:, 0], name="$浮子垂荡位移 ~ X_1$", yaxis='y1')
# #     trace2 = go.Scatter(x=t, y=y[:, 1], name="$浮子垂荡速度 ~ V_1$", yaxis='y2')
#     trace3 = go.Scatter(x=t, y=y[:, 2], name="$振子垂荡位移 ~ X_2$", yaxis='y1')
# #     trace4 = go.Scatter(x=t, y=y[:, 3], name="$振子垂荡速度 ~ V_2$", yaxis='y2')
#     fig = go.Figure(data=[trace1, trace3])
# #     fig = go.Figure(data=[trace1, trace2, trace3, trace4])
#     fig.update_layout(
#         width=1000,
#         height=600,
#         xaxis=dict(title='$时间 (s)$'),
#         yaxis=dict(title='$垂荡位移 (m)$'),
#         yaxis2=dict(title='$垂荡速度 (m/s)$', anchor='x', overlaying='y', side='right'),
```

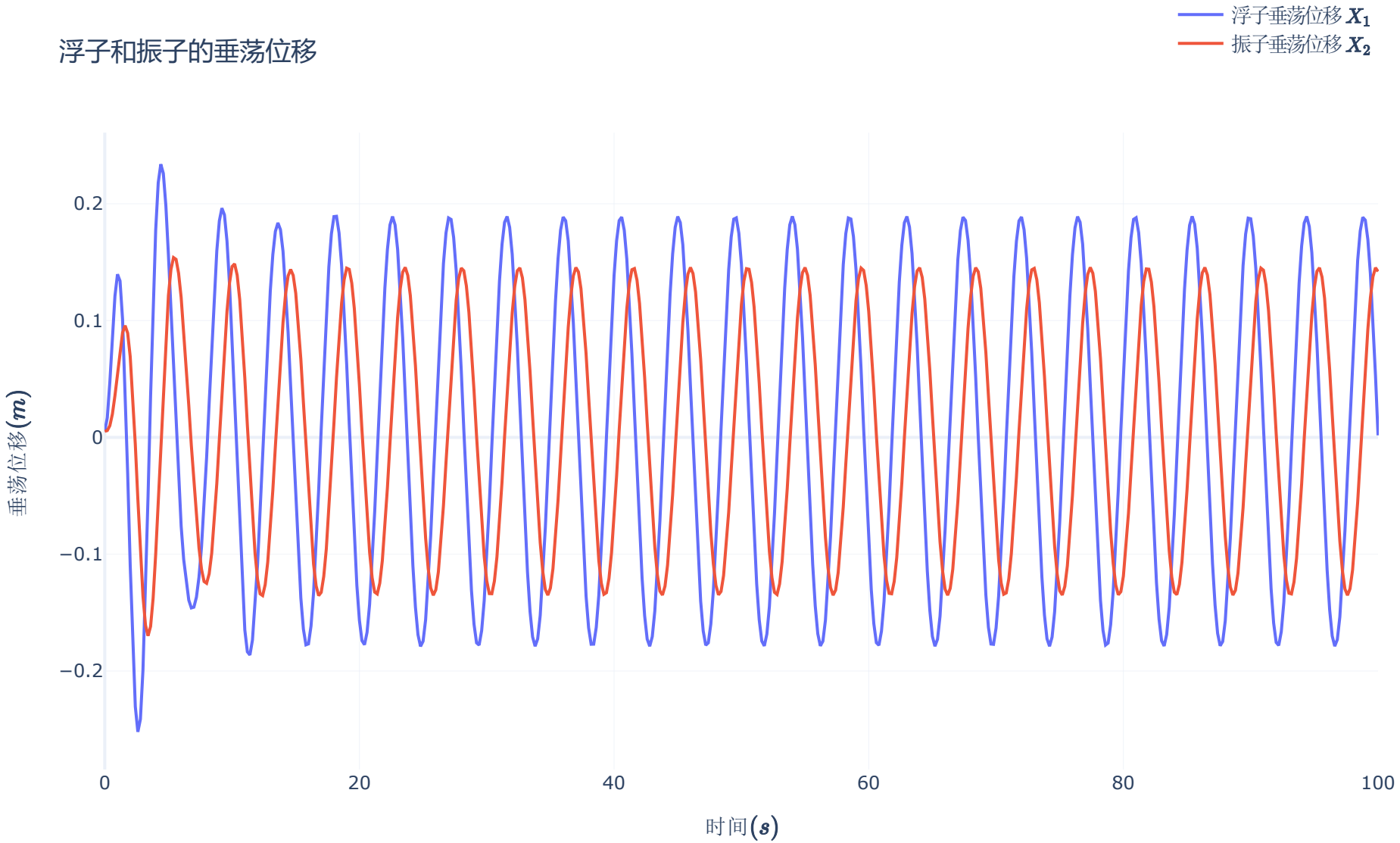
```
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         title=title,
#         template='plotly_white',
#     )
#     if svg_name is not None:
#         fig.write_image(IMG_SVG / svg_name)
#     fig.show()
#     return None
# def plot_plotly_rw(title, t=t, y=res3, svg_name=None):
#     trace1 = go.Scatter(x=t, y=y[:, 4], name="$浮子纵摇角位移~\\theta_1$", yaxis='y1', line=dict(color='rgb(232,137,189)'))
# #     trace2 = go.Scatter(x=t, y=y[:, 5], name="$浮子纵摇角速度~ \omega_1$", yaxis='y2', line=dict(color='rgb(103,194,163)'))
#     trace3 = go.Scatter(x=t, y=y[:, 6], name="$振子纵摇角位移~\\theta_2$", yaxis='y1', line=dict(color='rgb(252,140,99)'))
# #     trace4 = go.Scatter(x=t, y=y[:, 7], name="$振子纵摇角速度~ \omega_2$", yaxis='y2', line=dict(color='rgb(142,160,201)'))
#     fig = go.Figure(data=[trace1, trace3])
# #     fig = go.Figure(data=[trace1, trace2, trace3, trace4])
#     fig.update_layout(
#         width=1000,
#         height=600,
#         xaxis=dict(title='$时间 (s)$'),
#         yaxis=dict(title='$纵摇角位移 (rad)$'),
#         yaxis2=dict(title='$纵摇角速度 (rad/s)$', anchor='x', overlaying='y', side='right'),
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         title=title,
#         template='plotly_white',
#     )
#     if svg_name is not None:
#         fig.write_image(IMG_SVG / svg_name)
#     fig.show()
#     return None
```

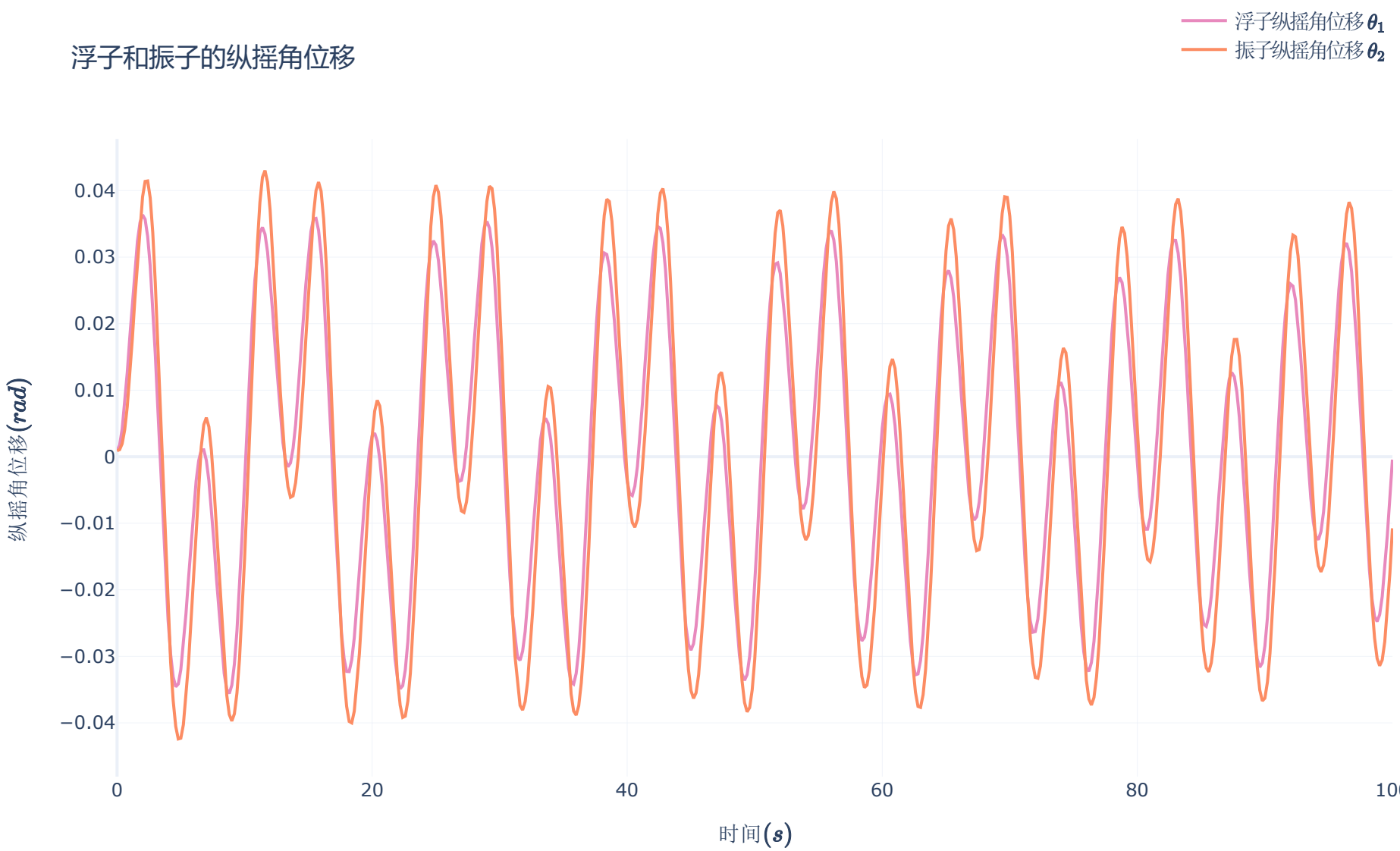
相差

In [146…

```
c = 2000  # 10000
k = 1600  # 80000

C = 3600  # 1000
K = 5000  # 250000
me = 1 * 垂荡附加质量

def ode_func(y, t, k=k, K=K):
    dy1 = y[2-1]
    dy3 = y[4-1]
#     print(k, K)
#     print(c * (y[2-1] - y[4-1] * np.cos(y[7-1])))
#     print(k * (y[1-1] - y[3-1] * np.cos(y[7-1])))
```

```
#     print(C * (y[6-1] - y[8-1]))
#     print(K * (y[5-1] - y[7-1]))
#     print('-'*50)
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])

    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]

t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res3 = odeint(ode_func, y0, t)
plot_plotly_xv('浮子和振子的垂荡位移', t=t, y=res3, svg_name='灵敏性分析-浮子和振子的垂荡位移.svg')
plot_plotly_rw('浮子和振子的纵摇角位移', t=t, y=res3, svg_name='灵敏性分析-浮子和振子的纵摇角位移.svg')
```


浮子和振子的垂荡位移
浮子垂荡位移 X_1
振子垂荡位移 X_2
0.2
0.1
0
−0.1
−0.2
垂荡位移(m)
0
20
40
60
80
100
时间(s)



In [ ]:
```
t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
y0 = [0 for _ in range(8)]
res3 = odeint(ode_func, y0, t)
```

In [141…

diff k K

In [146…

```python
def ode_func(y, t, k, K, c=c, C=C):
    dy1 = y[2-1]
    dy3 = y[4-1]
#     print(k, K)
#     print(c * (y[2-1] - y[4-1] * np.cos(y[7-1])))
#     print(k * (y[1-1] - y[3-1] * np.cos(y[7-1])))
#     print(C * (y[6-1] - y[8-1]))
#     print(K * (y[5-1] - y[7-1]))
#     print('-'*50)
    dy4 = c * (y[2-1] - y[4-1] * np.cos(y[7-1])) + k * (y[1-1] - y[3-1] * np.cos(y[7-1]))
    dy2 = f * np.cos(omega * t) + k1 * y[2-1] + k2 * y[1-1] - dy4
    dy2 /= m1 + me
    dy4 /= m2 * np.cos(y[7-1])

    j2 = j2_func(y[3-1])

    dy5 = y[6-1]
    dy7 = y[8-1]
    dy8 = C * (y[6-1] - y[8-1]) + K * (y[5-1] - y[7-1])
    dy6 = L * np.cos(omega * t) + K1 * y[6-1] + K2 * y[5-1] - dy8
    dy6 /= j1 + je
    dy8 /= j2
    return [dy1, dy2, dy3, dy4, dy5, dy6, dy7, dy8]
```

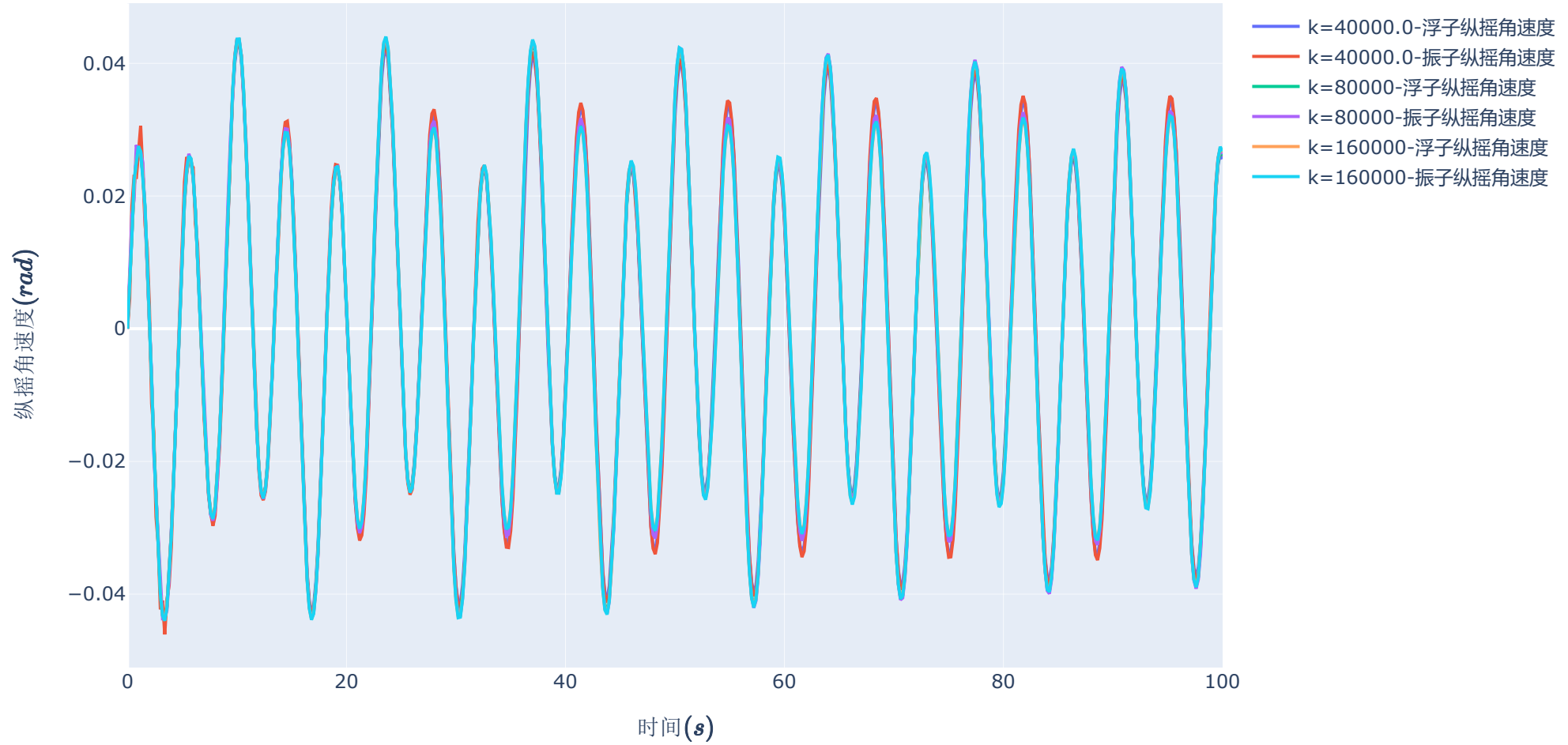
In [147…

```python
data_X = []
data_V = []
data_theta = []
data_omega = []

for c, C, k, K in zip([10000 * 0.5, 10000, 10000 * 2],
                      [1000 * 0.5, 1000, 1000 * 2],
                      [80000 * 0.5, 80000, 80000 * 2],
                      [250000 * 0.5, 250000, 250000 * 2]):
    时间间隔 = 0.2
    波浪频率 = 入射波浪频率
    波浪周期 = 1 / 波浪频率
    _l, _r = 0, 100
    t = np.linspace(_l, _r, num=int(_r / 时间间隔) + 1)
    y0 = [0 for _ in range(8)]
    res3 = odeint(ode_func, y0, t, args=(k, K))

    fuzi_columnes = [f'k={k}-浮子垂荡位移', f'k={k}-浮子垂荡速度', f'k={k}-浮子纵摇角位移', f'k={k}-浮子纵摇角速度']
    zhenzi_columnes = [f'k={k}-振子垂荡位移', f'k={k}-振子垂荡速度', f'k={k}-振子纵摇角位移', f'k={k}-振子纵摇角速度']

    data_X.append(go.Scatter(x=t, y=res3[:, 0], name=fuzi_columnes[0]))
```

```
    data_V.append(go.Scatter(x=t, y=res3[:, 1], name=fuzi_columnes[1]))
    data_theta.append(go.Scatter(x=t, y=res3[:, 4], name=fuzi_columnes[2]))
    data_omega.append(go.Scatter(x=t, y=res3[:, 5], name=fuzi_columnes[3]))

    data_X.append(go.Scatter(x=t, y=res3[:, 2], name=zhenzi_columnes[0]))
    data_V.append(go.Scatter(x=t, y=res3[:, 3], name=zhenzi_columnes[1]))
    data_theta.append(go.Scatter(x=t, y=res3[:, 6], name=zhenzi_columnes[2]))
    data_omega.append(go.Scatter(x=t, y=res3[:, 7], name=zhenzi_columnes[3]))

fig_data_X = go.Figure(data=data_X)
fig_data_X.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$垂荡位移 (m)$'),
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         title=title,
#         template='plotly_white'
)
fig_data_V = go.Figure(data=data_V)
fig_data_V.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$垂荡速度 (m)$'),
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         title=title,
#         template='plotly_white'
)
fig_data_theta = go.Figure(data=data_theta)
fig_data_theta.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$纵摇角位移 (rad)$'),
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
#         title=title,
#         template='plotly_white'
)
fig_data_omega = go.Figure(data=data_omega)
fig_data_omega.update_layout(
        width=1000,
        height=600,
        xaxis=dict(title='$时间 (s)$'),
        yaxis=dict(title='$纵摇角速度 (rad)$'),
#         legend=dict(y=1.22, yanchor="top", x=1, xanchor="right"),
```

```
#         title=title,
#         template='plotly_white'
)
```

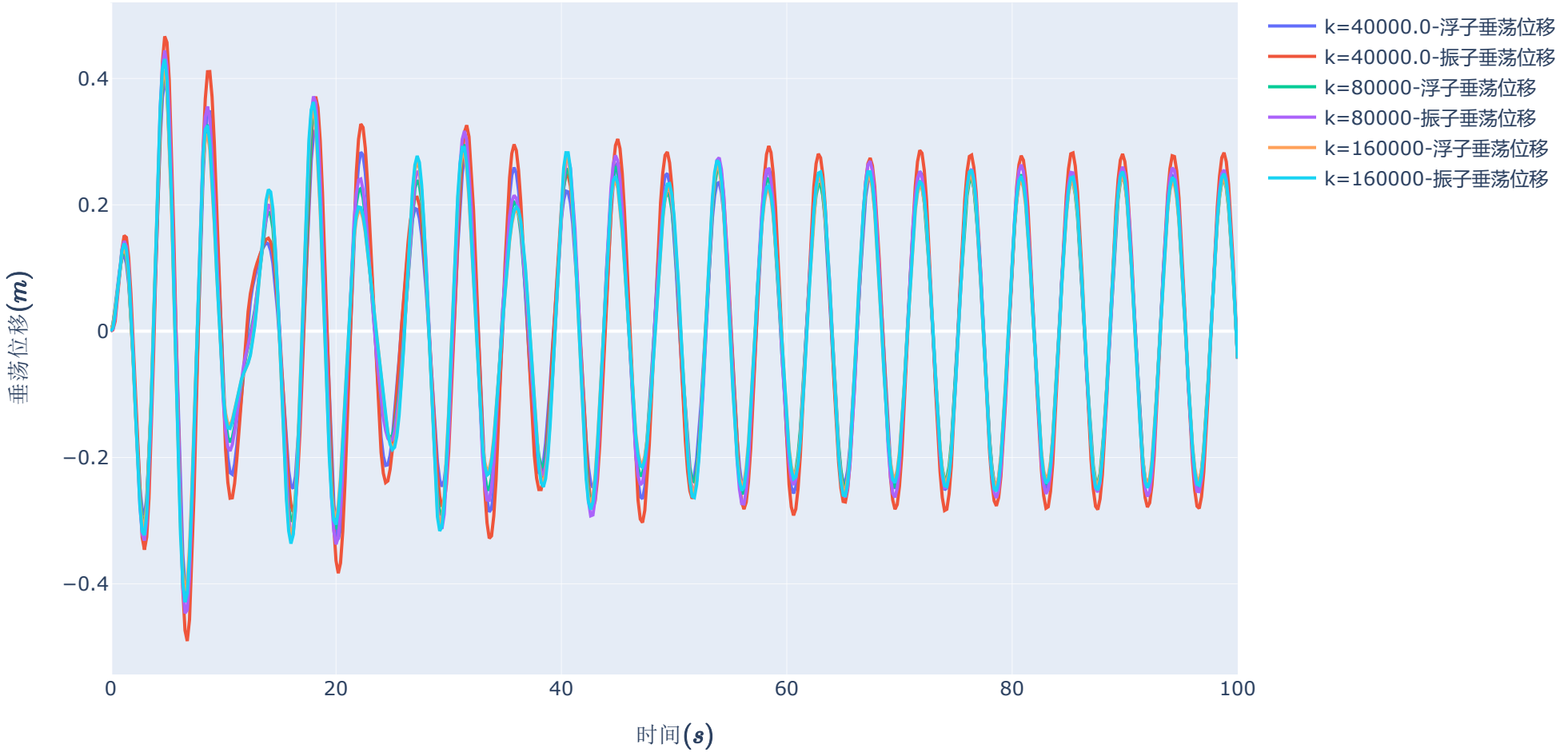
```
None
```

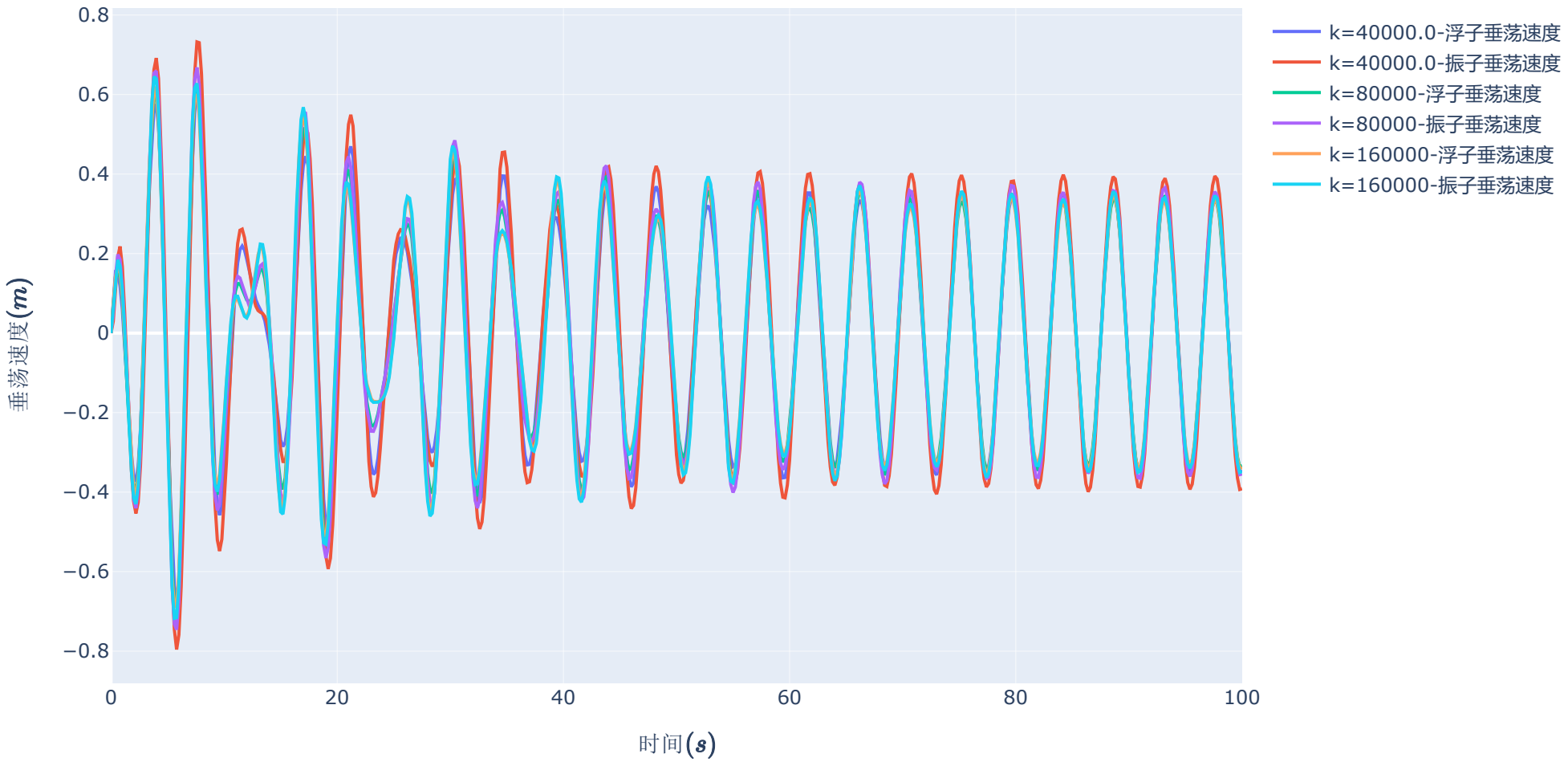


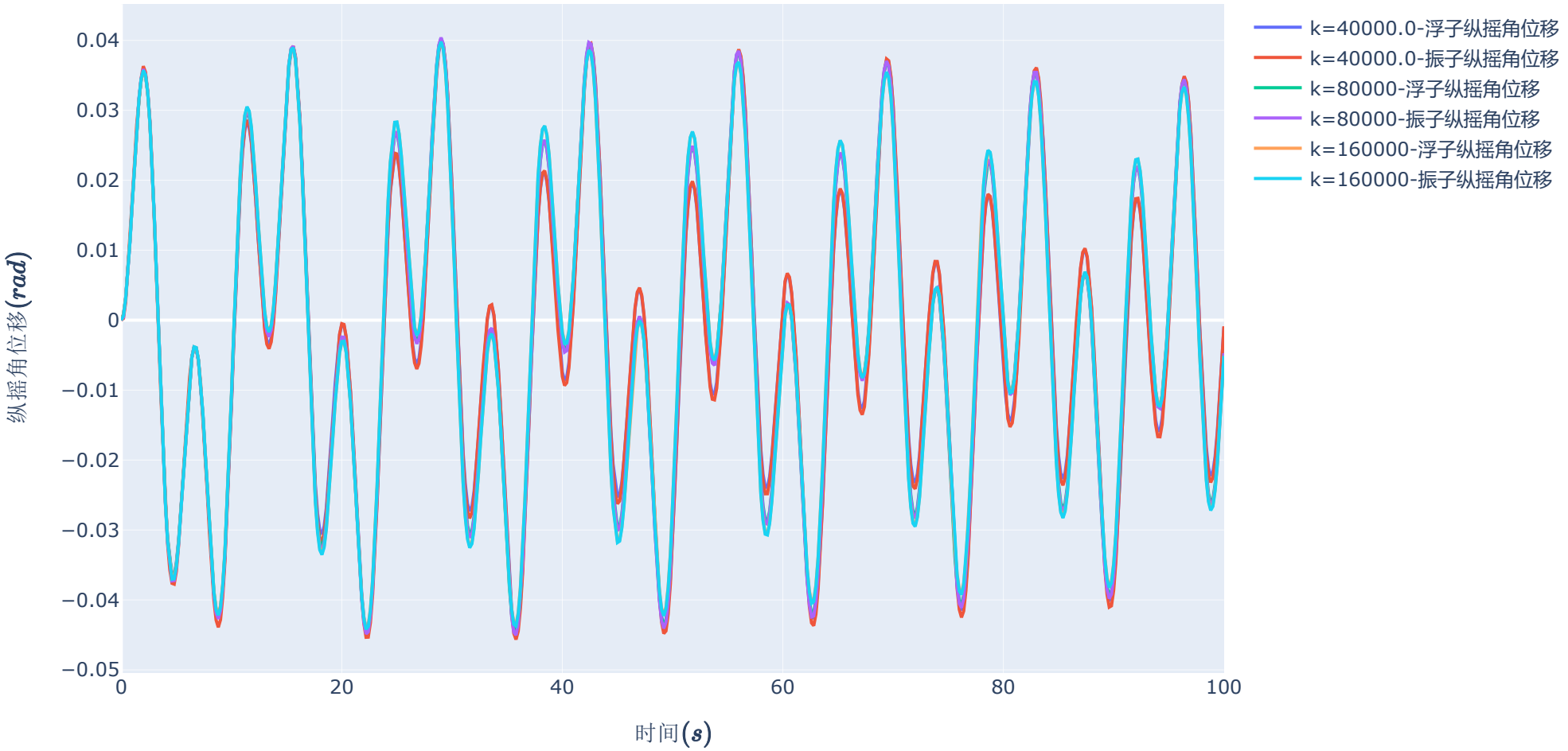
In [ ]:

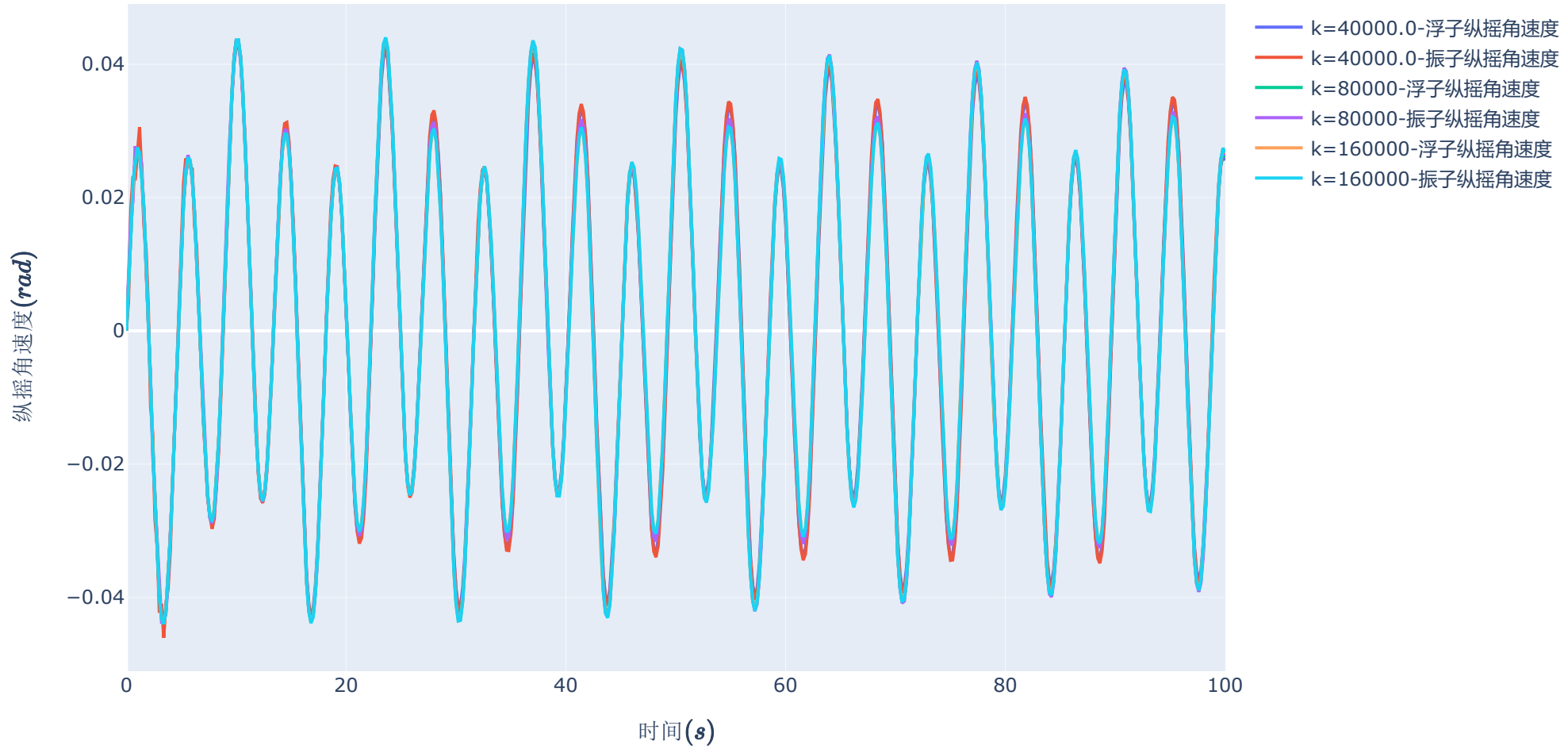
In [147…

```
fig_data_X.write_image(IMG_SVG / "灵敏性分析-垂荡位移.svg")
fig_data_V.write_image(IMG_SVG / "灵敏性分析-垂荡速度.svg")
fig_data_theta.write_image(IMG_SVG / "灵敏性分析-纵摇角位移.svg")
fig_data_omega.write_image(IMG_SVG / "灵敏性分析-纵摇角速度.svg")

fig_data_X.show()
fig_data_V.show()
fig_data_theta.show()
fig_data_omega.show()
```




0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
垂荡速度(m)
0
20
40
60
80
100
时间(s)
k=40000.0-浮子垂荡速度
k=40000.0-振子垂荡速度
k=80000-浮子垂荡速度
k=80000-振子垂荡速度
k=160000-浮子垂荡速度
k=160000-振子垂荡速度


k=40000.0-浮子纵摇角位移
k=40000.0-振子纵摇角位移
k=80000-浮子纵摇角位移
k=80000-振子纵摇角位移
k=160000-浮子纵摇角位移
k=160000-振子纵摇角位移
纵摇角位移(rad)
时间(s)
0.04
0.03
0.02
0.01
0
−0.01
−0.02
−0.03
−0.04
−0.05
0
20
40
60
80
100


0.04
0.02
0
−0.02
−0.04
0
20
40
60
80
100
纵摇角速度(rad)
时间(s)
k=40000.0-浮子纵摇角速度
k=40000.0-振子纵摇角速度
k=80000-浮子纵摇角速度
k=80000-振子纵摇角速度
k=160000-浮子纵摇角速度
k=160000-振子纵摇角速度

In [ ]:

END

In [ ]: