```python
""" Script for analysing the Octabuoy in Waqum"""

 """ Importing different functions needed"""
from __future__ import division
from tictoc import tic, toc
import numpy, os
from math import pi
from filelib.waqum_db import open_run_file
from Waqum import Input, Analysis
import matplotlib.pyplot as plt
import matplotlib

#######################################################
"""Function "Tic-Toc" measures time used for analysis"""
tic('Read SIF file')
""" Reads .SIF file with hydro coefficients"""
f = open_run_file('G1.SIF', helper_file='WADAM1.LIS')
f.read_data()
run = f.runs[-1]
toc()

"""Add critical damping"""
coeff = run.add_coefficients('viscous-damping-fraction-of-critical')
coeff.critical_damping_matrix[2,2] = 0.015
coeff.critical_damping_matrix[4,4] = 0.02

"""Defines coefficients"""
displacement = run.geometry.displacement
water_plane_area = run.geometry.water_plane_area
zb = run.geometry.center_of_bouyancy[2]
zg = run.center_of_gravity[2]
rho = run.rho
g = run.g

#######################################################
# Define Input
"""Defines wave heading"""
input = Input()
input.heading = 180

""" Defines duration and timestep"""
input.wave_spectrum_seed = 42
input.dt = 0.1
input.ramp_time = 30
input.transient = 30
input.tmax = 10800

input.dw = 100 # ????
input.verbose = False
input.log = 'mylog.txt'
"""Chooses solver, here Runge Kutta 4"""
input.solver_method = 'Runge-Kutta 4'
input.initial_position = numpy.zeros((6,), float)

#######################################################
# Define Force
"""Non linear damping added"""
class NonLinearDamping(object):
    def __init__(self, B):
        self.B = B

    def set_system(self, system):
        self.system = system
""" Example of added non linear damping, unfortunately not finished in the master
thesis"""
    def excitation(self, t, vel, pos):
        forces = numpy.zeros(6, float)
        forces[2] = - vel[2]**2 * self.B
        return forces*0

"""Mooring line restoring added"""
class Mooring(object):

    def set_system(self, system):
        self.system = system

    def excitation(self, t, vel, pos):
        forces = numpy.zeros(6, float)
        forces[0] = -(-0.0029*pos[0]**6 + 0.4995*pos[0]**5 - 33.281*pos[0]**4 +
1084.1*pos[0]**3 - 17752*pos[0]**2 + 138714*pos[0])
        forces[1] = -(-0.0029*pos[1]**6 + 0.4995*pos[1]**5 - 33.281*pos[1]**4 +
1084.1*pos[1]**3 - 17752*pos[1]**2 + 138714*pos[1])
        return forces

"""Defines decay analysis"""
def decay():
    """ Need to insert a spectrum, otherwise default spectrum is applied"""
    input.wave_spectrum = 'PM'
    input.wave_spectrum_parameters = {'Hs': 0.0, 'Tz': 22.5}
    working_dir = 'Decay-'
    cwd = os.getcwd()
    """Choosing which DOF(s) to preform decay for"""
    for dof in [4]:
        print 'Solving for DOF',dof
        if not os.path.isdir(os.path.join(cwd,working_dir+'%d'%dof)):
            os.makedirs(os.path.join(cwd,working_dir+'%d'%dof))
        os.chdir(os.path.join(cwd,working_dir+'%d'%dof))
        input.initial_position[dof] = 5*numpy.pi/180
        plt.figure()
        """adding damping to the motion control springs"""
        for damp in (0.05,0.05):
            analysis = Analysis(input, run)
            analysis.setup()
            system = analysis.system

            """Adds non linear damping"""
            nl_excitation = NonLinearDamping(0)
            system.add_excitation(nl_excitation)

            """Adds non linear restoring"""
            nl_restoring = NonLinearRestoring()
            system.add_excitation(nl_restoring)

            """ Mooring lines taken into account"""
            mooring=Mooring()
            system.add_excitation(mooring)
            """ Motion control springs, not used when mooring lines are added"""
            system.add_spring(dof=0, spring_period=320, damping_ratio=damp)
            system.add_spring(dof=1, spring_period=320, damping_ratio=damp)
            system.add_spring(dof=5, spring_period=320, damping_ratio=damp)


            tic('Run analysis')
            analysis.simulate()

            """FFT analysis, generates power spectrum distribution"""
            pxx,freqs = matplotlib.mlab.psd(system.pos_hist[:,dof],Fs=10,NFFT=2**20)

            """Plotting FFT results"""
            max_pxx = -1e10
            max_freq = 0
            for _pxx, freq in zip(pxx,freqs):
                if _pxx > max_pxx:
                    max_pxx = _pxx
                    max_freq = freq
            print 'Decay dof', dof, max_freq, 1/max_freq
            plt.plot(freqs, pxx, label='Damping %.2f'%damp)
            toc()
            last_pxx=0
            eps=1e-11
            x=[]
            for f,p in zip(freqs,pxx):
                if (p>eps and last_pxx<eps) or (p<eps and last_pxx>eps):
                    x.append(f)
                last_pxx=p
        plt.legend()
    plt.show()

""" Run Analysis"""
"""Defines JONSWAP analysis"""
def jonswap(wave_period, significant_waveheight, gamma):
    working_dir = os.path.join('HS-%.2f'%significant_waveheight,'Period-%.2f'%wave_period)
    cwd = os.getcwd()
    if not os.path.isdir(working_dir):
        os.makedirs(working_dir)
    os.chdir(os.path.join(cwd,working_dir))
    input.wave_spectrum = 'JONSWAP'
    input.wave_spectrum_parameters = {'Hs': significant_waveheight, 'gamma': gamma, 'Tp':
wave_period}

    analysis = Analysis(input, run)
    analysis.setup()
    system = analysis.system

    """Adds non linear damping"""
    nl_excitation = NonLinearDamping(0)
    system.add_excitation(nl_excitation)
    """Adds non linear restoring"""
    nl_restoring = NonLinearRestoring()
    system.add_excitation(nl_restoring)
    """ Mooring lines taken into account"""
    mooring=Mooring()
    system.add_excitation(mooring)
    """Adds motion control springs, not used when mooring lines are added"""
    system.add_spring(dof=0, spring_period=320, damping_ratio=0.05)
    system.add_spring(dof=1, spring_period=320, damping_ratio=0.05)
    system.add_spring(dof=5, spring_period=320, damping_ratio=0.05)
#
    tic('Run analysis')
    analysis.simulate()
    toc()


    plot_force = []

    """ Plots results"""
    plt.figure()
    plt.subplot(2,1,1)
    plt.plot(analysis.t, system.pos_hist[:,2], label='Heave')
    plt.legend()

    plt.subplot(2,1,2)
    plt.plot(analysis.t, system.pos_hist[:,4]*180/pi, label='Pitch')
    plt.legend()

    gamma=2.2
    heave=(system.pos_hist[:,2])
    pitch=(system.pos_hist[:,4])
    """Calculates Tz"""
    tz = wave_period*(0.6673+0.05037*gamma-0.006230*gamma**2 + 0.0003341*gamma**3)
    """Calculates MPL and Standard Deviation"""
    print 2*(2*numpy.std(heave)**2*numpy.log(10800/tz))**0.5,
2*(2*numpy.std(pitch*(180/numpy.pi))**2*numpy.log(10800/tz))**0.5
    print numpy.std(heave), numpy.std(pitch*(180/numpy.pi))


    os.chdir(cwd)
"""Prints different features of the analysis for control"""
def info():
    print 'g', g
    print 'rho', rho
    print 'displacement', displacement
    print 'water_plane_area', water_plane_area
    print 'zb', zb
    print 'zg', zg

info()
"""Runs the above defined analyses with different inputs"""
decay()
periods = [9.0, 9.8, 10.4, 13.0, 11.4, 15.4, 14.9, 15.8, 17.2]
HS = [4.2, 3.9, 5.9, 10.0, 6.7, 15.8, 15.2, 17.0, 19.8]
gamma =[2.2]
for wave_period,significant_waveheight  in zip(periods, HS):
    print '-'*20
    jonswap(wave_period,significant_waveheight,gamma)
plt.show()
```