Development and analysis of a combined tuned mass and tuned liquid damping system in a barge-like floating wind turbine structure

Feline Fox

Master Thesis

Delft University of Technology

July 2023

Preface

This thesis is the product of a spontaneous idea from over a year ago. Originally I had asked myself if and how a structure could have unfixed eigenfrequencies and if so, if this could be adapted so that a structures eigenfrequency would never coincide with the excitation frequency. While this was never conceptualized, the idea of variable eigenfrequencies did lead me to liquid mass dampers with time dependent tank shapes. And so, the idea of combining tuned mass dampers and tuned liquid dampers came to life.

I would like to take this chance to thank the people who made this thesis possible. First of all, I would like to thank my supervisors Oriol, Pim and Michael for taking a chance on an original project. While I have not always been vigilant in seeking your help, you have given me valuable advise along the way.

I would further like to thank my friends and family for your encouragement and support. Special thanks to Derk, who has been a big help, both mentally and by assisting me with DelftBlue documentation. All of your help has been much appreciated.

Feline Fox
June 2023
Delft

Abstract

As offshore wind turbines are moving further from the coast, floating wind turbines are being considered. These structures need to be able to endure high dynamic loads while remaining as still as possible in order to minimize internal stresses and maximize the efficiency of the turbine. To find ways to combat large responses, the effectiveness of tuned mass dampers (TMDs) and tuned liquid dampers (TLDs) has been widely researched. While both systems have been shown to be effective, these damping systems are being used separately. This thesis investigates the effectiveness of vertical TMDs combined with a TLD by analysing three barge-turbine structures. The first system is the undamped structure, which works as a control. The second system combines vertical TMDs with a TLD. These damping systems are not directly linked. The third system interdependently combines vertical TMDs with a TLD by stacking the damping systems. This makes the TMD displacement determine the shape of the TLD tank, which causes the system to be significantly nonlinear.

The analysis of the three systems begins with the development of a program that creates the frequency response function for each of these systems. The basis for these programs is a linear modal analysis. The nonlinearity of the third system is accounted for by Newton iteration.

The effectiveness of each damping system is determined by the performance index, which is based on the integral of the frequency response function.

The damping parameters of the independent TMD and TLD are optimized. Due to computational difficulties no parameter optimization is performed for the interdependent damping system. In order to judge possible performance improvement for this system, a rough sensitivity study is performed on the damping parameters of this system.

The results of the optimization and sensitivity study show that the independently damped system performs best of all three systems when using these particular parameters. The interdependently damped system has unknown damping potential.

It is not possible to definitively conclude from this research whether the TMDs and TLD damp more effectively when they work either separately or interdependently. Both damped systems are able to perform better than the undamped system with the correct damping parameters.

The developed programs give reasonable frequency response spectra for all three systems. From this it is concluded that the programs of the three systems work as intended, including the nonlinear part.

Contents

List of Figures

List of Tables

1 | Introduction

1.1 Floating wind turbines

Due to a scarcity of land, more structures are now being built in offshore and coastal areas. This is especially the case for wind turbines, which are pushed as far out of sight as feasible. Apart from aesthetic reasons to build wind parks further away from the coast, this also increases wind quality. Offshore wind resources are of higher quality than on land, and get increasingly consistent when moving further away from the coast [20].

However, moving further away from the coast gives deeper water construction sites. Fixed structures in deep water are not always economical, which is why fixed wind turbines are usually only being build at water depths up to 60 m. Waters deeper than that are where floating structures come into play. Floating structures also have the advantage that they are less dependent on the seabed conditions than fixed structures [45].

Structures in deep water environments are subject to high dynamic loads, such as wind, water waves, drift, currents, and dynamic load of the rotor. These loads can cause large motions of the floating structures, which is undesirable for various reasons. First of all, large acceleration of the barge, causes yet higher loads throughout the structure, making it more expensive to build safe structures as maintenance costs go up and life span goes down. This is especially relevant for cyclic loads that can potentially cause resonance in the structure. Even aside from resonance, these cyclic loads cause fatigue and thus structural accelerations need to be kept down. Lastly, in the specific case of wind turbines, efficiency needs to be kept in mind. Wind turbines are most efficient when in an upright position. When the floating barge rotates, the turbines therefore become less profitable.

1.2 Damping systems

In order to mitigate the dynamic response of structures, tuned mass dampers (TMDs) and tuned liquid dampers (TLDs) have been under investigation. A TMD is a damping system that vibrates at an eigenfrequency that counteracts the vibration of the main structure. Different versions of tuned mass dampers and tuned liquid dampers in floating windmill systems have been analyzed. He, E.M. et. al (2017) studied a 2D floating wind turbine system with a horizontal tuned mass damper in the nacelle. They concluded a significant impact on response vibrations [18]. Yang, J. et. al (2019) studied a similar 2D floating windmill system with horizontal TMD's in the platform, comparing them with a similar system with a TMD attached to the nacelle of the windmill. They concluded that the TMD placed in the platform did indeed effectively improve the dynamic response of the system as well [43].

A TLD is simular to a TMD, but instead of a solid mass, fluid is used. This can be in the form of column TLDs, where the damping largely depends on mass distribution, or in the form of tank TLDs, where apart from mass distribution the damping also depends on a sloshing effect. Sloshing of a liquid in a structure has a pronounced effect on the vibration of a rotating system [15][6].

A study by Ha, M. et. al (2016) concludes that the use of a multilayered tuned liquid damper in the nacelle of a spar type floating turbine is effective for reducing pitch vibration, although its effectiveness is negligible for vertical and horizontal vibration reduction. It is also stated that the effectiveness increases with increasing layers [15]. Placing a multi-column tuned liquid damper in the platform of the system, shows effectiveness against pitch as well [6].

1.3 Combination damping system

The effectiveness of both TMDs and TLDs in the platform of a floating wind turbine has been shown separately. It is worth asking whether improving the effectiveness of these dampers is possible when combining them. Further, does the placement of these systems make a significant difference?

In this thesis, a multi mass vertical TMD is combined with a sloshing TLD as shown in figure 1.1. In this configuration, the TLD is placed directly on top of the TMDs. This effectively makes the TMD displacement determine the shape of the TLD sloshing tank. This is relevant as the eigenfrequency of the TLD depends on the shape of its tank.

There are a couple of reasons to conjecture that the stacked damper configuration is more effective than separate dampers. First of all, with a TLD tuned to the pitch eigenfrequency of the system, the TLD's mass distribution causes the TMDs to displace as well. More pressure is exerted on the TMDs on the highest side of the barge due to the TLD wave. Thus, even with TMDs tuned to a different frequency, during pitch of the platform the TMDs at the low end of the platform go up, while the TMDs at the high end go down. This means that the liquid mass of the TLD will be distributed inversely proportional to the platform's elevation even more so than a TLD on its own, effectively creating a restoring moment with its displaced mass.

This stacked damper placement also causes the TLD liquid to exert pressure on the TMDs below. Most pressure is exerted when a TMD is at its lowest point, increasing its displacement. This amplifies not only the before mentioned effect, but also the effectiveness of the TMDs themself. The increased displacement of both the TLD and the TMDs leads to a higher portion of the energy in the system to go into the dampers instead of the barge system.

Further, the displacement of the TMDs during pitch does not only increase the effectiveness of the TLD, but also causes the TMDs to take energy out of the system so that they behave as dampers at this frequency. This means that the TMDs can function at both heave and pitch frequency when tuned to only the heave eigenfrequency of the system.

Lastly, the damping ability of a TMD depends largely on its mass. TMDs with bigger mass are able to absorb more energy. Since the movement of a TMD in the stacked configuration also moves the mass of the TLD section above it, the TMD mass is effectively bigger when stacked.

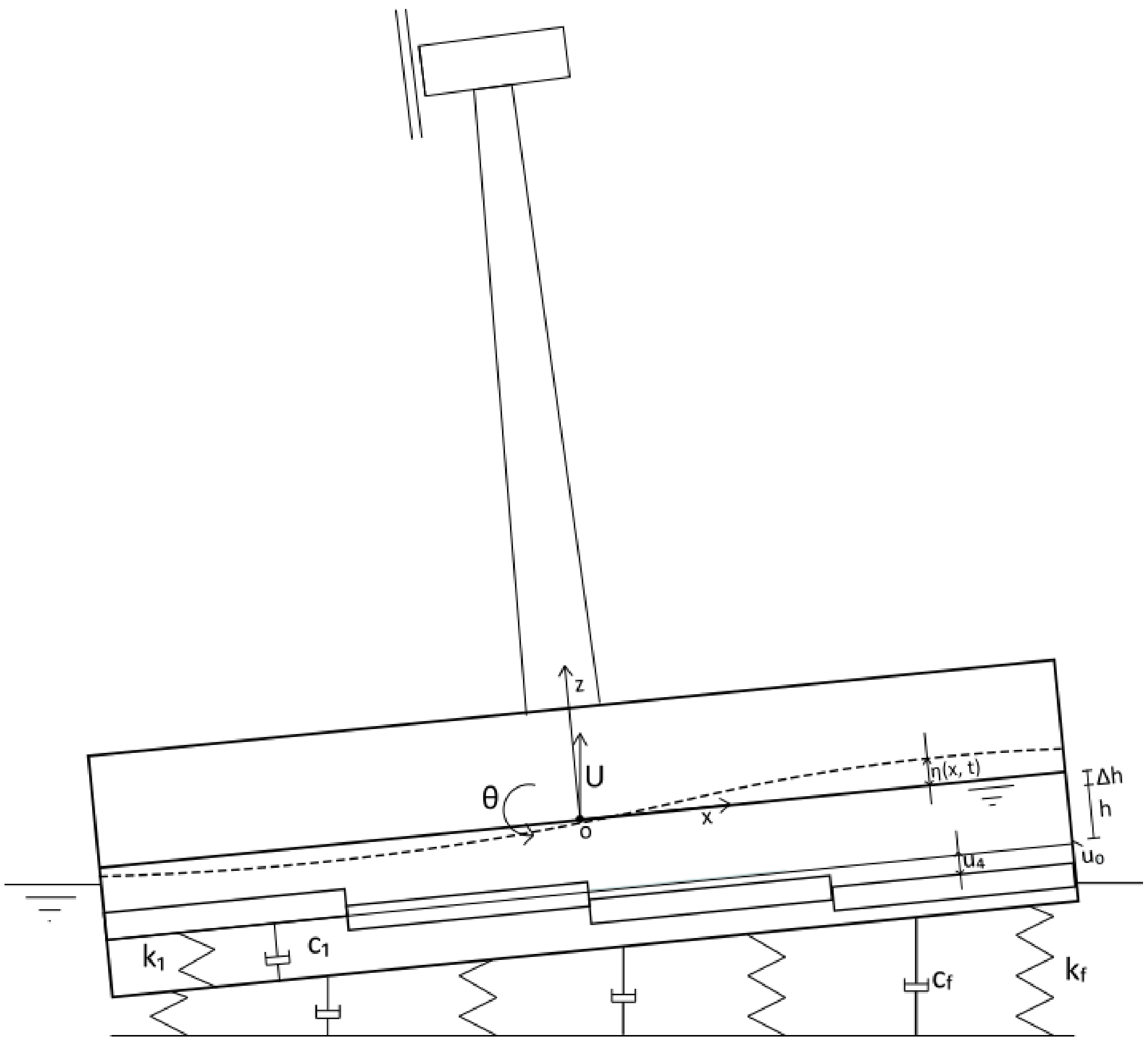

Figure 1.1: Wind turbine on a barge with a combined TMD-TLD system. Here, U is the vertical degree of freedom (DoF) of the barge, θ the rotational DoF, u_4 the DoF of the 4th TMD with u_0 as baseline. µ(x, t) is the wave elevation of the TLD, h + Δh is the resting TLD liquid line. k_1 and c_1 are the TMD stiffness and damping coefficients of the first TMD respectively, while k_f and c_f are the hydrostatic stiffness and hydrodynamic damping of the floating surface respectively.

1.4 Scope and objectives

Since the response of a system can be reduced by the redistribution the liquid of a TLD, it raises the question if magnifying this phenomenon also magnifies the effectiveness of a TLD. The damping effectiveness due to magnified mass redistribution by combining TMDs with TLDs as described above has not yet been researched, however.

The aim of this thesis is twofold.

1. To build an adjustable program to determine the response of the barge described above under a dynamic excitation. Developing an adaptable program enables further research using the same model. Linear modal analysis has established itself as a standard tool for engineers. It provides a reliable approach to determine responses of structures of a linear or low nonlinear degree. However, as nonlinearity increases, linear analysis becomes unreliable. For the program linear modal analysis is used in conjunction with numerical methods to function as nonlinear analysis of the coupled damping system.

2. To find out whether a tuned fluid damper can be more effective when its tank's shape depends

on the displacements of tuned mass dampers. Since the TLD eigenmode and eigenfrequencies depend on the shape of its tank, this will impact the eigenmodes and eigenfrequencies of the complete system.

Three models are created using Python to represent the barge without a damping system, with a damping system in the form of a separate TLD and TMD and finally with a TLD and TMDs that are interdependent. These models can be adjusted to fit different parameters, the amount of eigenmodes accounted for and a specific number of tuned mass dampers. For the sake of simplicity, the effectiveness of this type of damping system is determined using only its first eigenmode. Also, the evaluation is limited to two TMDs. The TMDs are tuned to the heave eigenfrequency of the barge, while the TLD is tuned to the pitch eigenfrequency.

A standard barge [22] will be used as the system's platform. Even though the barge restricts the height of TMD and TLD displacements, the response will be considered unrestricted for simplicity.

To determine the damping efficiency of the proposed system, it needs to be compared to the system without tuned mass dampers and the system with independent TMDs and TLD. Each system has the same total mass and dimensions, independent of the presence of tuned mass dampers and damping fluid. The total mass of the dampers is the same in each case. The stiffness and damping parameters of the mass dampers in each damped system is tuned to the system's optimum. The optimal parameters are calculated numerically by minimizing the performance index [26]. This index is the integral of the frequency response spectrum. The performance index also gives comparable results between the different systems. The optimum is found using a white noise spectrum as load. While the excitement of the barge depends on wave and wind spectra, not white noise, these spectra are situation specific and therefor not relevant in a general case.
The same base structure is used for each system. The chosen system parameters are those of a 5-MW, three bladed NREL wind turbine with variable speed [22].

The equations of motion of the damping liquid are modeled using boundary conditions on the platform and TMDs in conjunction with linear velocity potential theory. In order to determine the effect of the damping liquid, it is important to account for its sloshing effect since this has been found to be a significant factor [28], as well as its mass distribution. If the right amount of liquid is chosen, the liquid behaves as a tuned liquid damper, which is why the damping and mass parameters of the liquid are also optimized for a TLD using optimal design theory for a single degree of freedom mass damper in combination with the first sloshing eigenfrequency as an initial guess [13] [46]. Since TLDs are not nearly as effective for vertical and horizontal damping as for pitch damping [6], the liquid is tuned for the eigenfrequency of the rotational degree of freedom of the platform. The tuned mass dampers are tuned to the heave eigenfrequency.

The response of all systems (undamped, damped with independent TMD-TLD system and damped with interdependent TMD-TLD system) is determined numerically in 2D. For simplicity it assumed that the platform is rigid, so that only the global response will be given. This means that the platform itself can be modeled as a rigid mass with 3 degrees of freedom, namely vertical translation, horizontal translation, and rotation. In this thesis, the horizontal motion is disregarded since the investigated dampers are expected to only mitigate heave and pitch, not surge. The structural deformations are also neglected in this thesis. Due to additional energy dissipation, the response would be slightly lower when local deformations are taken into account, making the rigid case slightly conservative. The wind turbine itself is also modeled as a rigid beam, rigidly connected to the platform.

Lagrange's equation is used to establish the equations of motion of the system with Rayleigh dissipation to account for damping. The response of the system is determined by doing an frequency response analysis of the system, based on linear modal analysis.

In order to do so, additional hydrodynamic mass, damping and stiffness matrices are determined first. The contributing mass, stiffness and damping for the ‘wet’ system are found using velocity potential theory.

The third model (damped with interdependent TMD-TLD system) is nonlinear and its system is solved using the Newton method of iteration.

2 | Theory

The objective of this chapter is to give all necessary theory to find the movement pattern of the floating barge structures investigated in this thesis.

First, fluid-structure interaction will be addressed to find the fluid potential and wave shape functions of the TLD, as well as the hydraulic damping, stiffness and additional mass effects on the structure due to the ocean water.

Next, creating the equation of motion using Lagrange's equation including Rayleigh dissipation is discussed.

In order to find the movement patterns, eigenvalue analysis has to be performed so that the response function of each structure can be found.

Next, the optimization scheme of the parameters of the damping system is discussed. Fair comparison between barge structures can after all only be made when each is designed optimally.

Lastly, iteration is addressed as an approach to solve the non-linear system of the non-linear third model (barge damped with interdependent TMD-TLD system).

2.1 Fluid-structure interaction

The movement of the structure is influenced by fluid interaction. Both the interaction with the TLD fluid and the interaction with the surrounding water need to be modeled. This is possible using potential flow theory.

2.1.1 Potential flow theory

The behaviour of fluid can be described using potential flow theory as long as its flow is irrotational. Assuming that there is zero angular velocity in the fluid, the velocity of the fluid can be expressed as the spatial gradient of the fluid potential.

In case of shallow liquid and absence of high frequency loading, the compressibility of most liquids can be neglected. The governing equation of motion for a non-compressible fluid is:

$$\nabla^2\phi(x,z,t) = 0 \tag{2.1}$$

Where $\nabla = \frac{\partial}{\partial x} + \frac{\partial}{\partial z}$ = nabla operator;
ϕ = scalar velocity potential $[m^2/s$.

The fluid velocity vector $\boldsymbol{v}_f$ and the fluid pressure p_f respectively are related to the fluid potential scalar as follows:

$$\boldsymbol{v}_f(x,z,t) = \nabla\phi(x,z,t) \tag{2.2}$$

$$p_f(x,z,t) = -\rho_f\frac{\partial\phi(x,z,t)}{\partial t} \tag{2.3}$$

Where: ρ_f = fluid density $[kg/m^3]$.

2.1.2 Boundary conditions

There are different boundary conditions for the liquid inside and outside of a floating barge with a TLD. Both will be discussed here.

Fluid inside a tank

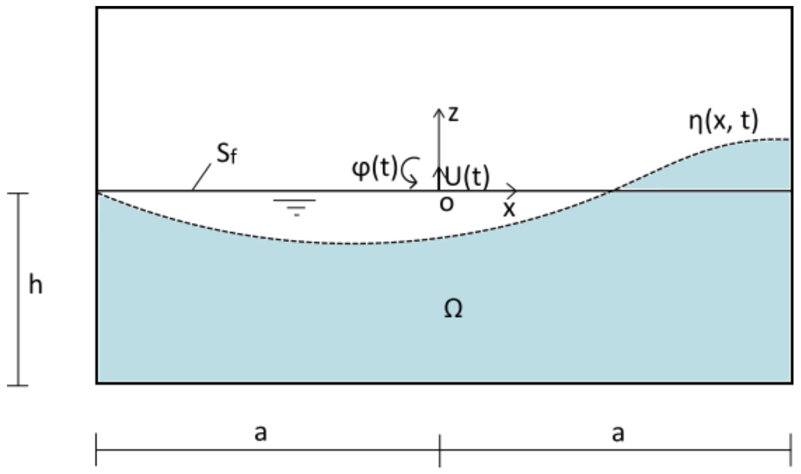

Figure 2.1: Boundary conditions of a two-dimensional fluid domain (Ω) confined to a tank with free surface Sf, wave elevation µ(x, t). The liquid height in rest is h and half the tank width is a.

The two-dimensional fluid domain of a fluid in a tank is shown in figure 2.1. Its boundary conditions can be expressed as follows:

$$\boldsymbol{v}_f(x=-a,z,t) = -z\dot{\theta} \tag{2.4}$$

$$\boldsymbol{v}_f(x=a,z,t) = -z\dot{\theta} \tag{2.5}$$

$$\boldsymbol{v}_f(x,z=-h,t) = \dot{U} + x\dot{\theta} \tag{2.6}$$

$$\frac{\partial \phi(x,z,t)}{\partial z} - \frac{\partial \eta(x,z,t)}{\partial t} = 0 \tag{2.7}$$

$$\frac{\partial \phi(x,z=0,t)}{\partial t} + g\eta(x,z=0,t) = 0 \tag{2.8}$$

Where:	U	= vertical translation of the tank [m];
	θ	= rotation of the tank [rad];
	$\boldsymbol{v}_f(x,z,t)$	= the TLD fluid velocity vector normal to the tank wall [m/s];
	$\eta(x,z,t)$	= free surface elevation [m].

Equations 2.4, 2.5 and 2.6 give the relative velocity continuity normal to the tank wall. The liquid and tank wall velocity are equal at all times, so that the fluid does not pass through the wall, nor do fluid and wall lose contact. Zero horizontal velocity of the tank and rigid walls are assumed, so that only vertical and angular velocity play a role.

The kinematic free boundary condition 2.7 implies that the surface elevation velocity is equal to the vertical flow velocity of the TLD liquid.

Finally, using Bernoulli's law gives condition 2.8, which denotes that the pressure on the free surface level is constant and equal to zero.

Solving the governing equation for these boundary conditions gives the potential and surface wave elevation of the liquid. Setting 2.4, 2.5 and 2.6 equal to zero for a tank in rest, gives the homogeneous solution and gives the eigenvalues of the sloshing liquid.

Fluid outside of a floating structure

The boundary conditions outside of a tank are slightly different, as seen in figure 2.2.

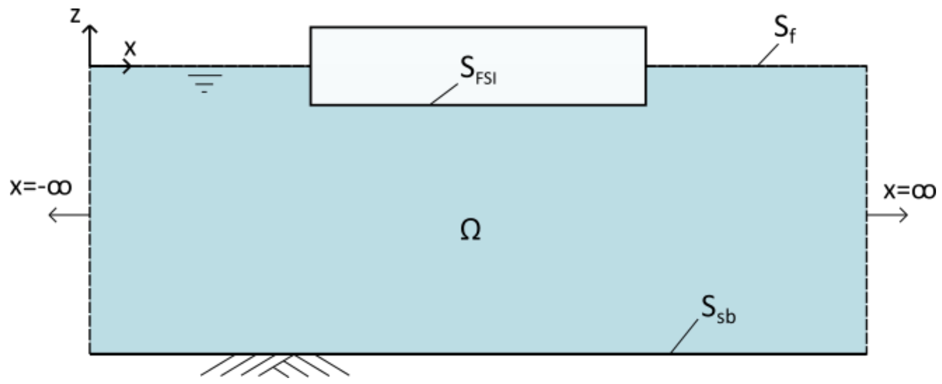

Figure 2.2: Boundary conditions of a two-dimensional fluid domain (Ω) with a floating body. With free surface ($\mathrm{S_f}$), fluid-structure interaction ($\mathrm{S_{FSI}}$) and seabed ($\mathrm{S_{sb}}$) boundary conditions and radiation conditions at infinite distance from the structure.

The boundary conditions can be expressed as follows:

$$\boldsymbol{v}_f(x,z,t)\vec{n} = 0\Bigg|_{S_{sb}} \tag{2.9}$$

$$\boldsymbol{v}_f(x,z,t)\vec{n} = \dot{U}_{FSI}(x)\Bigg|_{S_{FSI}} \tag{2.10}$$

$$\frac{\partial\phi(x,z=0,t)}{\partial z} - \frac{\partial\eta(x,z=0,t)}{\partial t} = 0\Bigg|_{S_f} \tag{2.11}$$

$$\frac{\partial\phi(x,z=0,t)}{\partial t} + g\eta(x,z=0,t) = 0\Bigg|_{S_f} \tag{2.12}$$

$$ \tag{2.13}$$

Where: $\vec{n}$ = outward normal vector to the boundary;
$U_{FSI}(x)$ = translation of the structure [m].

Equation 2.9 gives the boundary condition at the seabed. The velocity normal to the seabed is zero (assuming a rigid seabed), accounting for impermeability. Similarly, the velocity normal to the floating structure is equal to the velocity of the (rigid) structure itself, as expressed in 2.10. The free surface boundary conditions (2.11 and 2.12) are the same here as in the case of the liquid contained in the tank as described above.

The radiation condition is satisfied by using a trial solution with a decreasing amplitude for an absolutely increasing x-coordinate. Assuming that the potential modes can be separated into variables as $\phi_m(x,z,t) = X_m(x)Z_m(z)q_m(t)$, a possible trial solution for $X_m(x)$ could be $X_m(x) = A_m exp(-ik_m x)$, where A_m is the amplitude and k_m is the wave number of the m^{th} mode. More on this in section 2.1.4.

2.1.3 Sloshing inside a tank

Sloshing commonly occurs to liquid inside a tank. The tuned liquid damper used in this thesis is effectively a large tank with a sloshing liquid. The effectiveness of the TLD depends on the sloshing mechanics of this liquid and the interaction with its tank. Similar to the TMD, the sloshing frequency of the TLD needs to be tuned to the frequency of the main structure, in this case the tank itself.

It is possible to describe the potential and wave elevation of a sloshing fluid in a tank with an analytical approximation [27] [37]. In order to do so, first of all, the following trial solutions are proposed:

$$\eta(x,z,t) = \sum f_m(x,z)exp(i\omega_m t) = \sum f_m(x,z)q_m(t) \tag{2.14}$$

$$\phi(x,z,t) = \sum i\omega_m \varphi_m(x,z)exp(i\omega_m t) = \sum \varphi_m(x,z)\frac{\partial q_m(t)}{\partial t} \tag{2.15}$$

Here, $\mathrm{q_m(t)}$ is the generalized surface motion degree of freedom of sloshing mode m (m = 1, 2, 3 ...). It is the amplitude of the wave shape at x=0 for free vibration. When modalized, the $\mathrm{q_m(t)}$ of every sloshing mode depends on the mode's eigenfrequency ω_m. φ_m(x,z) and $\mathrm{f_m}$(x,z) are the modal amplitude functions of the velocity potential and the wave height respectively. Decoupling the modes is possible in this case, since linearity of the subsystem is assumed.

Combining and rewriting the boundary conditions for a tank at rest and the trial solutions gives the eigenvalue problem:

$$\nabla^2\varphi_m(x,z) = 0 \tag{2.16}$$

$$\frac{\partial\varphi_m(x,z)}{\partial\vec{n}}|_{x=-a,x=a,z=-h} = 0 \tag{2.17}$$

$$\frac{\partial\varphi_m(x,z=0)}{\partial z} - \frac{\omega_m^2}{g}\varphi_m(x,z=0) = 0 \tag{2.18}$$

Where $\vec{n}$ is the outward normal vector to the tank wall.

Assuming that every φ_m can be written as a multiplication of two functions with only one variable ($\varphi_m(x,z) = X_m(x)Z_m(z)$), these functions can be approximated using trial functions again. Choosing a function fitting the boundary conditions for $\mathrm{X_m}$(x) and substituting into equation 2.16, gives the approximation to $\mathrm{Z_m}$(z). The following trial solutions for X(x) fit the boundary conditions for the symmetric and anti-symmetric modes respectively:

$$X_m(x) = \begin{cases} cos(\alpha_m x) \\ sin(\beta_m x) \end{cases} \tag{2.19}$$

With:

$$\alpha_m = \frac{(2m-1)\pi}{2a}, \qquad \beta_m = \frac{2\pi}{a} \tag{2.20}$$

Using these possible solutions for $\mathrm{X_m}$(x), then give that the solutions for $\mathrm{Z_m}$(z) are:

$$Z_m(z) = \begin{cases} A_m sinh(\alpha_m z) + cosh(\alpha_m z) & \text{(symmetric)} \\ B_m sinh(\beta_m z) + cosh(\beta_m z) & \text{(anti-symmetric)} \end{cases} \tag{2.21}$$

So that:

$$\varphi_m(x,z) = X(x)Z(z) = \begin{cases} cos(\alpha_m x)(A_m sinh(\alpha_m z) + cosh(\alpha_m z)) & \text{(symmetric)} \\ sin(\beta_m x)(B_m sinh(\beta_m z) + cosh(\beta_m z)) & \text{(anti-symmetric)} \end{cases} \tag{2.22}$$

Substitution of φ_m(x, z) into equation 2.18 gives the modal eigenfrequencies:

$$\omega_m^2 = \begin{cases} A_m\alpha_m g & \text{(symmetric)} \\ B_m\beta_m g & \text{(anti-symmetric)} \end{cases} \tag{2.23}$$

According to Faltinsen et al. (2001) [10], the Rayleigh-Kelvin variational formula can be used to estimate the sloshing eigenfrequencies as:

$$\omega_m^2 = g\frac{\int\int_\Omega (\nabla\varphi_m)^2\,dx\,dz}{\int_{S_f}\varphi_m^2\,dS_f} \tag{2.24}$$

Or, in case of a square tank:

$$\omega_m^2 = g\frac{\int_{-a}^{a}\int_{-h}^{0}(\nabla\varphi_m)^2\,dz\,dx}{\int_{-a}^{a}\varphi_m^2\big|_{z=0}\,dx} \tag{2.25}$$

Combining this with equation 2.23 gives:

$$g\frac{\int_{-a}^{a}\int_{-h}^{0}(\nabla\varphi_m)^2\,dz\,dx}{\int_{-a}^{a}\varphi_m^2\big|_{z=0}\,dx} = \begin{cases} A_m\alpha_m g & \text{(symmetric)} \\ B_m\beta_m g & \text{(anti-symmetric)} \end{cases} \tag{2.26}$$

Here Ω is the static liquid region, and S_f is the static free liquid surface, as shown in figure 2.1.

When combining 2.23 and 2.26, A_m and B_m can be found to be:

$$A_m = \frac{-b_{Sm} - \sqrt{b_{Sm}^2 - 4a_{Sm}c_{Sm}}}{2a_{Sm}} \tag{2.27}$$

$$\text{With:} \tag{2.28}$$

$$a_{Sm} = \int\int_\Omega cos^2(\beta_m x) + sinh^2(\beta_m z)\,dx\,dz \tag{2.29}$$

$$b_{Sm} = 2\int\int_\Omega cosh(\beta_m z) + sinh(\beta_m z)\,dx\,dz - \frac{a}{\beta_m} \tag{2.30}$$

$$c_{Sm} = \int\int_\Omega sin^2(\beta_m x) + sinh^2(\beta_m z)\,dx\,dz \tag{2.31}$$

$$\tag{2.32}$$

$$B_m = \frac{-b_{Am} - \sqrt{b_{Am}^2 - 4a_{Am}c_{Am}}}{2a_{Am}} \tag{2.33}$$

$$\text{With:} \tag{2.34}$$

$$a_{Am} = \int\int_\Omega sin^2(\alpha_m x) + sinh^2(\alpha_m z)\,dx\,dz \tag{2.35}$$

$$b_{Am} = 2\int\int_\Omega cosh(\alpha_m z) + sinh(\alpha_m z)\,dx\,dz - \frac{a}{\alpha_m} \tag{2.36}$$

$$c_{Am} = \int\int_\Omega cos^2(\alpha_m x) + sinh^2(\alpha_m z)\,dx\,dz \tag{2.37}$$

Now the potential function is known, the wave shape function can easily be found from the dynamic boundary condition 2.7:

$$\eta = \int \frac{\delta\phi}{\delta z}\,dt \tag{2.38}$$

2.1.4 Fluid movement outside of a floating structure

The fluid movement outside of a floating barge does not only cause excitation forces on said structure. It also causes hydrostatic stiffness and damping as well as additional mass to the structure. This 'wet' situation gives a different dynamic response than the 'dry' situation. The additional parameters will be discussed here.

The potential and wave elevation outside of the tank will need to be solved to determine the additional 'wet' parameters. The same trial solutions to the ones in section 2.1.3 are used:

$$\eta(x,z,t) = \sum f_m(x,z)exp(i\omega_m t) = \sum f_m(x,z)q_m(t) \tag{2.39}$$

$$\phi(x,z,t) = \sum i\omega_m \varphi_m(x,z)exp(i\omega_m t) = \sum \varphi_m(x,z)\frac{\partial q_m(t)}{\partial t} \tag{2.40}$$

Together with the boundary conditions this gives the following eigenvalue problem:

$$\nabla^2\varphi_m(x,z) = 0 \tag{2.41}$$

$$\left.\frac{\partial\varphi_m(x,z)}{\partial\vec{n}}\right|_{S_{FSI},S_{sb}} = 0 \tag{2.42}$$

$$\left.\frac{\partial\varphi_m(x,z)}{\partial z}\right|_{S_f} - \left.\frac{\omega_m^2}{g}\varphi_m(x,z)\right|_{S_f} = 0 \tag{2.43}$$

Assuming that separation of variables is accurate, φ_{m}(x, z) can be expressed as $\varphi_m(x,z) = X_m(x)Z_m(z)$. These functions will be approximated as trial functions again, fitting the boundary conditions. This is how the radiation condition will be satisfied. The following trial solutions for $X_m(x)$ fit the radiation conditions:

$$X_m(x) = exp(-\alpha_m x) \tag{2.44}$$

The trial solutions for $\mathrm{X_m}$(x) and $\mathrm{Z_m}$(z) need to satisfy 2.41. This gives a possible $\mathrm{Z_m}$(z):

$$Z_m(z) = A_m cos(\alpha_m z) \tag{2.45}$$

With:

$$\alpha_m = \frac{(2m-1)\pi}{2H} \tag{2.46}$$

Where H is the water depth. So that:

$$\varphi_m(x,z) = A_m cos(\alpha z)exp(-\alpha x) \tag{2.47}$$

Substitution into 2.43 gives the eigenfrequencies:

$$\omega_m^2 = A_m\alpha_m g \tag{2.48}$$

Using the Rayleigh-Kelvin equation again to estimate the eigenfrequencies:

$$\omega_m^2 = g\frac{\int_{-\infty}^{\infty}\int_{-H}^{0}(\nabla\varphi_m)^2\,dz\,dx}{\int_{-\infty}^{\infty}\varphi_m^2\big|_{z=0}\,dx} \tag{2.49}$$

Combining 2.47, 2.48 and 2.49 gives $\mathrm{A_m}$:

$$A_m = \frac{1}{2}(2m-1)\pi - 1 \tag{2.50}$$

With the potential function now known, the wave shape function can be found from the dynamic boundary condition 2.11:

$$\varphi(x,z) = \sum(\frac{1}{2}(2m-1)\pi - 1)cos(\alpha_m z)exp(-\alpha_m x) \tag{2.51}$$

$$\eta(x,z) = \int\frac{\delta\phi}{\delta z}\,dt \tag{2.52}$$

Additional mass and hydrodynamic damping

As a physical body moves through fluid, its inertia is bigger compared to the same body moving through a vacuum. This increased inertia is accounted for with an additional virtual mass and damping load [9] [39]. The hydrodynamic damping caused by the fluid movement due to the bodies own movement is, more specifically, radiation damping. The viscous effects of the fluid is another damping source to be accounted for. As potential flow deals with non-viscous fluids, this effect is included in the form of drag forces on the barge.

First, the theoretical potential flow situation is analysed; As the floating body accelerates, the fluid around it needs to accelerate as well, causing an additional force. This force in the direction of acceleration can be calculated by integrating the hydrodynamic pressure over area of the body perpendicular to the acceleration direction: $F_h = \int p\, dA_h$. Bernoulli's equation for unsteady potential flow gives the pressure as $p = -\rho(\frac{\partial \phi}{\partial t} + \frac{1}{2}|\nabla\phi|^2)$ [40].

The exact potential flow directly bordering a rectangular prism can not be given, since this flow must be turbulent around the corners and potential flow theory demands irrotational flow. Nonetheless, the potential around a rectangular tank will here be approximated with the potential flow as determined in the section above. The potential will be assessed at $z = 0$ to determine vertical movement, where the vertical fluid velocity is equal to the motion of the floating body $\dot{U} + \dot{\theta}x$:

$$\phi(x, z = 0, t) = \sum(\frac{1}{2}(2m-1)\pi - 1)exp(-\alpha x)(\dot{U} + \dot{\theta}x) \tag{2.53}$$

This gives the following pressure and hydrodynamic force on the body:

$$\begin{aligned} p = & -\rho \sum A_m(exp(-\alpha_m x)(\ddot{U} + \ddot{\theta}x) + exp(-2\alpha_m x) \\ & \cdot \alpha_m^2(-\frac{1}{2}\dot{U}^2 + (\frac{1}{2}x^2 - x + \frac{1}{2})\dot{\theta}^2 + (x+1)\dot{\theta}\dot{U})) \end{aligned} \tag{2.54}$$

$$F_h = b\int_{-a}^{a} p\, dx \tag{2.55}$$

$$\begin{aligned} = & \rho b \sum A_m^2\big[exp(-2\alpha_m a)(-\frac{1}{4}\alpha_m\dot{U}^2 + ((-\frac{1}{4}a^2 + a - \frac{1}{2})\alpha_m - \frac{1}{4}a + \frac{1}{4} + \frac{1}{8\alpha_m})\dot{\theta}^2 \\ & - (\frac{1}{2}\alpha_m(a+1) + \frac{1}{4})\dot{U}\dot{\theta}) - exp(2\alpha_m a)(\frac{1}{4}\alpha_m\dot{U}^2 + (\frac{1}{4}\alpha_m(a^2 + 2a + 1) - \frac{1}{4}a - \frac{1}{4} - \frac{1}{8\alpha_m})\dot{\theta}^2 \\ & - ((\frac{1}{2}a - \frac{1}{2})\alpha_m - \frac{1}{4})\dot{\theta}\dot{U})\big] + \rho b\sum A_m\big[exp(-\alpha_m a)(\frac{1}{\alpha_m}\ddot{U} + (\frac{1}{\alpha_m}a - \frac{1}{\alpha_m^2})\ddot{\theta}) \\ & + exp(\alpha_m a)(-\frac{1}{\alpha_m}\ddot{U} - (\frac{1}{\alpha_m}a - \frac{1}{\alpha_m^2})\ddot{\theta})\big] \end{aligned} \tag{2.56}$$

Where b is the width of the rectangular floating area and where A_m and α_m as defined in equations 2.50 and 2.46 respectively. When disregarding the non-linear elements, this simplifies to:

$$F_h = \rho b\sum A_m\big[exp(-\alpha_m a)(\frac{1}{\alpha_m}\ddot{U} + (\frac{1}{\alpha_m}a - \frac{1}{\alpha_m^2})\ddot{\theta}) + exp(\alpha_m a)(-\frac{1}{\alpha_m}\ddot{U} - (\frac{1}{\alpha_m}a - \frac{1}{\alpha_m^2})\ddot{\theta})\big] \tag{2.57}$$

The linear part of the additional hydrodynamic force, gives zero additional damping terms, but the additional mass terms are expressed as follows:

$$m_{h,U} = -\rho b\sum A_m\frac{1}{\alpha_m}\big[exp(-\alpha_m a) - exp(\alpha_m a)\big] \tag{2.58}$$

$$m_{h,\theta} = -\rho b\sum A_m(\frac{1}{\alpha_m}a - \frac{1}{\alpha_m^2})\big[exp(-\alpha_m a) - exp(\alpha_m a)\big] \tag{2.59}$$

$$m_{h,\theta U} = m_{h,U\theta} = 0 \tag{2.60}$$

Which then gives the additional mass ($\mathbf{A}$) matrice:

$$\boldsymbol{A} = \begin{bmatrix} m_{h,U} & m_{h,\theta U} & 0 & \dots & 0 \\ m_{h,U\theta} & m_{h,\theta} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \tag{2.61}$$

As discussed, there are no linear radiation damping terms. There are however drag forces working on the body as well, creating damping. The total drag on a body can be subdivided into friction forces on the body's skin and pressure forces. While the friction forces depend on the Reynolds number, the total drag force can be approximated as constant when the shape of the body causes predominantly pressure drag. This is the case for bodies with sharp edges that cause separation of flow along the sides of the body, thus also for a barge, which is a rectangular prism [44], [41].

The pressure drag can be approximated as $F_{drag} = -\frac{1}{2}bC_d\rho\int_{-a}^{a}(\dot{X}+\dot{\Theta}x)|\dot{X}+\dot{\Theta}x|\,dx$, where C_d is the drag coefficient [14] [2] [8]. Since the linearized equivalent equation of $cos(\omega t)|cos(\omega t)| \approx \frac{8}{3\pi}cos(\omega t)$ and the vertical displacement of the barge is harmonic, the pressure drag is approximated as $F_{drag} = -\frac{8}{6\pi}\hat{U}bC_d\rho\int_{-a}^{a}\dot{X}+\dot{\Theta}x\,dx$. Where $\hat{U}$ is the amplitude of the vertical movement of the barge, which is the same as the amplitude of the velocity potential at $z=0$, namely A_m.

Thus, the total drag force on the floating body is:

$$F_{drag} = -\frac{4}{3\pi}((2m-1)\pi-1)abC_d\rho\dot{X} \tag{2.62}$$

According to Ezoji et al. [8], the drag coefficient C_d can be approximated as:

$$C_d = A(KC)^n \tag{2.63}$$

Where KC is the dimensionless Keulegan-Carpenter number, which describes the relative importance of drag compared to inertia forces, and A and n are non-dimensional coefficients.

The Keulegan-Carpenter number is determined as $KC = \frac{2\pi\hat{U}}{D}$, where $\hat{U}$ is the amplitude of the oscillations and D is the diameter of the floating body. In the case of the floating barge the diameter is $2a$ and the amplitudes of the oscillations at z=0 are A_m.

The n is related to the separation angle of the fluid vortexes around the edges of the body. This number depends on the inner angle of the body's edges δ as $n = \frac{3-2\lambda}{2\lambda-1}, \lambda = 2-\frac{\delta}{\pi}$. In case of a sharp square edge $\delta = 0$, which gives $n = -\frac{1}{3}$.

The A factor is given by the ratio of the body's height h_{plate} versus its width d_{plate}: $A = \frac{h_{body}}{d_{plate}}$. In the case of the floating barge $A = \frac{H}{2a}$, where H is the floating depth of the barge at rest.

This gives a drag coefficient $C_d = \frac{H}{2a}(\frac{2\pi(\frac{1}{2}(2m-1)\pi-1)}{2a})^{-\frac{1}{3}}$

(2.62) gives a heave damping term only:

$$c_{d,U} = \frac{2}{3\pi}((2m-1)\pi-1)b\rho H(\frac{2\pi(\frac{1}{2}(2m-1)\pi-1)}{2a})^{-\frac{1}{3}} \tag{2.64}$$

$$c_{d,\theta} = c_{d,\theta U} = c_{d,U\theta} = 0 \tag{2.65}$$

Which gives the additional damping matrix:

$$\boldsymbol{B} = \begin{bmatrix} c_{d,U} & c_{d,\theta U} & 0 & \dots & 0 \\ c_{d,U\theta} & c_{d,\theta} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{2.66}$$

Hydrostatic stiffness

The hydrostatic stiffness is directly related to the hydrostatic buoyancy pressure p. Archimedes' principle states that the weight of a displaced volume of liquid by a body is equal to the upward pressure on that body. This means that the buoyancy force is:

$$\begin{aligned} F_h &= \int_{-a}^{a} \rho g b(H - U - \theta x)\, dx \\ &= 2\rho g b a H - 2\rho g b a U - \rho g b a^2 \theta \end{aligned} \tag{2.67}$$

With H the floating depth of the body at rest, so that the additional hydrostatic stiffness matrix (**D**) and force vector are respectively:

$$\boldsymbol{D} = \begin{bmatrix} 2\rho g b a & 0 & 0 & \dots & 0 \\ 0 & \rho g b a^2 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad \boldsymbol{F_h} = \begin{bmatrix} 2\rho g b a H \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.68}$$

2.2 Equation of motion

The barge-wind turbine structure under investigation in this thesis is simplified as one rigid body, so that only the global response of the barge-system is given. The local response of the tank walls are therefore disregarded. The system's response is considered in 2D, so that there are 3 degrees of freedom, namely vertical and horizontal translation (heave and surge) and rotation (pitch). Each tuned mass dampers gives one additional translational degree of freedom. The tuned liquid damper gives an additional translational degree of freedom of its equivalent mass. For the analysis of the proposed model, surge is ignored since neither damping system in its original form has been shown to be considerably effective against surge. Presuming that the horizontal motion is minor, lets the horizontal degrees of freedom be omitted.

The equations of motion are found using Lagrange's equation:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} + \frac{\partial \mathcal{R}}{\partial \dot{q}_i} = Q_i \tag{2.69}$$

Where $\mathcal{L}$ is the Lagrangian, $\mathcal{R}$ is the Rayleigh dissipation function, q_i are the generalized degrees of freedom and Q_i is the generalized loading vector. Here, $\mathcal{L} = \mathcal{K} - \mathcal{P}$ in which potential and kinetic energy respectably can be generally expressed as follows:

$$\mathcal{P} = \frac{1}{2}\sum_{i=0}^{n} k_i q_i^2 + g m_i q_i \tag{2.70}$$

$$\mathcal{K} = \frac{1}{2}\sum_{i=0}^{n} m_i \dot{q}_i^2 \tag{2.71}$$

Where:	k_i	$=$ stiffness parameter for DoF q_i $[N/m]$;
	m_i	$=$ mass for DoF q_i $[kg]$;
	n	$=$ number of DoFs.

The damping forces are incorporated into the Rayleigh dissipation function $\mathcal{R}$:

$$\mathcal{R} = \frac{1}{2}\sum_{i=0}^{n} c_i \dot{q}_i^2 \tag{2.72}$$

Where c_i is the damping coefficient for q_i.

2.70, 2.71 and 2.72 are inserted into the Lagrange equation in order to find the equations of motion. This gives the mass-, damping and stiffness matrices **M**, **C** and **K**.
The resulting equation of motion will be in the form:

$$(\boldsymbol{M}+\boldsymbol{A})\ddot{\boldsymbol{x}} + (\boldsymbol{C}+\boldsymbol{B})\dot{\boldsymbol{x}} + (\boldsymbol{K}+\boldsymbol{D})\boldsymbol{x} = \boldsymbol{F} \tag{2.73}$$

Where **M**, **C** and **K** are the mass, damping and stiffness matrices of the dry system respectively. **A**, **B** and **D** are the additional 'wet' mass, hydraulic damping and stiffness matrices respectively as determined in section 2.1.4. Force vector **F** gives the excitation loads, $\mathbf{F_h}$ gives the hydraulic force (2.68) and $\boldsymbol{x}$ is the displacement vector of the system. When analysing only the heave and pitch of the platform, the following force vector and displacement vector are used:

$$\boldsymbol{F} = \begin{bmatrix} F_v(t) \\ M(t) \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} U(t) \\ \theta(t) \\ q(t) \\ u_1(t) \\ \vdots \\ u_n(t) \end{bmatrix} \tag{2.74}$$

Here, n is the number of tuned mass dampers and $F_v(t)$ and M(t) are the external vertical force and moment respectively.

2.3 Modal analysis

A modal analysis consists of decoupling the modes of a system and treating the total response of said system as a superposition of the responses to each excitation mode. This is of course possible only when analysing linear systems. The total response looks as follows:

$$\boldsymbol{x}(t) = \sum_{i=1}^{N} \hat{\Phi}_i u_i(t) \tag{2.75}$$

In which $\hat{\Phi}_i$ is the ith eigenvector, $u_i(t)$ is the corresponding modal coordinate and N is the number of eigenmodes. The modal analysis is exact for non-damped systems, but only an approximation in the case of damped systems. Since eigenfrequencies and eigenmodes of a damped system are very close to their undamped counterpart, this is not a problem.

2.3.1 Linear eigenvalue analysis

An eigenvalue analysis of the system will be performed based on free vibration. This means that a solution to the homogeneous equation of motion in the form of $\boldsymbol{M}\ddot{\boldsymbol{x}} + \boldsymbol{C}\dot{\boldsymbol{x}} + \boldsymbol{K}\boldsymbol{x} = \boldsymbol{0}$ needs to be found. The general solution to this equation can be found in the form of:

$$\boldsymbol{x} = \sum_{i=1}^{N} \boldsymbol{X}_i exp(i\omega_i t) \tag{2.76}$$

Substituting this trial solution into the equation of motion, gives:

$$(-\omega_i^2 \boldsymbol{M} + i\boldsymbol{C} + \boldsymbol{K})\boldsymbol{X}_i = \boldsymbol{0} \tag{2.77}$$

Which only gives a non-trivial solution when:

$$det(-\omega_i^2 \boldsymbol{M} + i\boldsymbol{C} + \boldsymbol{K}) = \boldsymbol{0} \tag{2.78}$$

Solving 2.78, gives the eigenfrequencies ω_i.

Substituting the eigenfrequencies into 2.77 gives the corresponding non-trivial solution for $\boldsymbol{X}_i$ and the response in turn:

$$\boldsymbol{X}_i = (-\omega_i^2 \boldsymbol{M} + i\boldsymbol{C} + \boldsymbol{K})^{-1} \tag{2.79}$$

$$\boldsymbol{x}(t) = \sum_{i=1}^{N} \boldsymbol{X}_i \exp(i\omega_i t) = \boldsymbol{\Phi u} = \sum_{i=1}^{N} \hat{\Phi}_i u_i(t) \tag{2.80}$$

Where $\boldsymbol{\Phi}$ is the eigenmatrix, with the eigenvectors being its columns:

$$\boldsymbol{\Phi} = \begin{bmatrix} \Phi_1 & \Phi_2 & \dots & \Phi_N \end{bmatrix} \tag{2.81}$$

The unit response function $\mathbf{H}(\omega)$ is determined by solving the inhomogeneous equation of motion, with non-specific frequencies and the load amplitude $\hat{\boldsymbol{F}}$:

$$(-\omega_i^2(\boldsymbol{M} + \boldsymbol{A}) + i(\boldsymbol{C} + \boldsymbol{B}) + \boldsymbol{K} + \boldsymbol{D})\boldsymbol{X}_i = \hat{\boldsymbol{F}} \tag{2.82}$$

$$\boldsymbol{H}(\omega) = (-\omega^2(\boldsymbol{M} + \boldsymbol{A}) + i(\boldsymbol{C} + \boldsymbol{B}) + \boldsymbol{K} + \boldsymbol{D})^{-1} \tag{2.83}$$

2.3.2 Nonlinear eigenvalue analysis

In the case of a nonlinear system, the mass $\boldsymbol{M}(\boldsymbol{X})$, damping $\boldsymbol{C}(\boldsymbol{X})$ and stiffness $\boldsymbol{K}(\boldsymbol{X})$ matrices depend on the response amplitude $\boldsymbol{X}(t)$. The response amplitude in turn is depends on the wave of the TLD, not only its amplitude, so that the whole system response amplitude is time dependent. This system can therefor not be solved accurately using linear modal analysis. The first iteration is approximated using the equivalent nonlinear modal analysis in the frequency domain [35] however, where the amplitude dependency has been acknowledged. The mass, damping and stiffness matrices are determined for zero response amplitude:

$$\boldsymbol{H}(\omega, \boldsymbol{X} = \boldsymbol{0}) = (-\omega^2(\boldsymbol{M}(\boldsymbol{0}) + \boldsymbol{A}) + i(\boldsymbol{C}(\boldsymbol{0}) + \boldsymbol{B}) + \boldsymbol{K}(\boldsymbol{0}) + \boldsymbol{D})^{-1} \tag{2.84}$$

Since the system is nonlinearly dependent on the response amplitude, a response change gives the following system change:

$$(-\omega^2(\boldsymbol{M}(\boldsymbol{X} + \Delta\boldsymbol{X}) + \boldsymbol{A}) + i(\boldsymbol{C}(\boldsymbol{X} + \Delta\boldsymbol{X}) + \boldsymbol{B}) + \boldsymbol{K}(\boldsymbol{X} + \Delta\boldsymbol{X}) + \boldsymbol{D})(\boldsymbol{X} + \Delta\boldsymbol{X}) = \hat{\boldsymbol{F}} \tag{2.85}$$

From there, the system is solved by using Newton iteration (see section 2.5) until $\Delta\boldsymbol{X}$ approaches zero, correcting the residual response [30], [29].

The above is a local state description of the nonlinear problem for a particular time and response amplitude. This process can be repeated at any time t in order to determine the time dependency of the frequency response spectrum.

As nonlinear eigenmodes are not necessarily orthogonal and can therefor not be superimposed, each eigenvector at each mode likely contains contributions of all other modes. It has been observed that the nonlinear part of the eigenmode is fairly small in the vicinity of the eigenfrequency, but becomes dominant when farther from resonance [35]. The system does not have fixed eigenmodes and eigenfrequencies. These have to be found for each local state.

2.4 Damping system optimization

Optimal damping of a structure gives minimum displacements of said structure under load. This is specified more practically as the minimum performance index J, which is the mean square displacement response of the structure under a particular dynamic load. The optimal parameters (TMD mass, damping ratio and stiffness) for a single degree of freedom (SDoF) structure with the system subjected to white noise excitation is known [26] [34]. Numeric solutions have been found in the form of:

$$f = \frac{\omega_{TMD}}{\omega_S} \tag{2.86}$$

$$f = \sqrt{\frac{1+\mu/2}{1+\mu}} \tag{2.87}$$

$$\xi = \frac{c_{TMD}}{2m\omega_{TMD}} \tag{2.88}$$

$$\xi = \sqrt{\frac{\mu(1+\frac{3}{4}\mu)}{4(1+\mu)(1+\mu/2}} \tag{2.89}$$

$$\mu = \frac{m}{M} \tag{2.90}$$

Where:	f	= frequency ratio between the TMD and the structure;
	ξ	= damping ratio of the TMD;
	μ	= mass ratio between the TMD and the structure;
	M	= mass of the structure $[kg]$;
	m	= mass of the TMD $[kg]$.

Knowing that the TMD should be in resonance with the structure ($f = 1$), the optimal parameters can be found for the SDoF system.

The optimal paramaters for a SDoF system with a TLD are determined by using the first sloshing frequency ω_{TLD} of the TLD [12]. The equivalent stiffness parameter of a TLD is related to the liquid height. The damping parameter is determined by multiple factors, such as liquid viscosity and potential damping screens in the tank. The design of this damping system are out of the scope of this thesis. So, the TLD is optimized towards the optimal liquid height h_{TLD} and a general damping constant c_{TLD}.

$$\omega_{TLD} = \sqrt{\frac{\pi g}{2a} tanh(\frac{\pi h_{TLD}}{2a})} \tag{2.91}$$

This gives:

$$\omega_S \cdot f = \sqrt{\frac{\pi g}{2a} tanh(\frac{\pi h_{TLD}}{2a})} \tag{2.92}$$

$$h_{TLD} = tanh^{-1}(\frac{2a}{\pi g} f^2 \omega_S^2)\frac{2a}{\pi} \tag{2.93}$$

Since the frequency ratio f and liquid depth h_{TLD} depend in each other via the mass ratio, they are determined by means of iteration.

When determining the optimal parameters for more complex systems, the performance index needs to be minimized by letting its derivative go to zero [26]. The performance index for a white noise is defined as follows:

$$J = \int_0^\infty \boldsymbol{H}(\omega) S_0 \, d\omega \tag{2.94}$$

Where: $\boldsymbol{H}(\omega)$ = unit response function $[m/N]$;
S_0 = white noise spectrum $[N]$.

Calculate the gradients of the performance index over the parameters and the newly calculated parameters as follows:

$$\frac{\partial J_n}{\partial c_n} = \frac{J(c_n + \frac{1}{2}s) - J(c_n - \frac{1}{2}s)}{s}, \qquad c_{n+1} = c_n - \frac{\partial J_n}{\partial c_n} s \tag{2.95}$$

$$\frac{\partial J_n}{\partial k_n} = \frac{J(k_n + \frac{1}{2}s) - J(k_n - \frac{1}{2}s)}{s}, \qquad k_{n+1} = k_n - \frac{\partial J_n}{\partial k_n} s \tag{2.96}$$

$$\frac{\partial J_n}{\partial m_n} = \frac{J(m_n + \frac{1}{2}s) - J(m_n - \frac{1}{2}s)}{s}, \qquad m_{n+1} = m_n - \frac{\partial J_n}{\partial m_n} s \tag{2.97}$$

Where s is the incremental step to determine the parameters. Each original parameter ($n = 0$) is determined as the optimal parameter for a SDoF system in the first frequency of the desired degree of freedom.

The process is repeated with the new parameters until the gradient is lower than the tolerated error ϵ:

$$|\frac{J'_n - J'_{n+1}}{J'_n}| < \epsilon \tag{2.98}$$

Each parameter influences the optimization of the remaining parameters. Therefor, all parameters either need to be optimized in tandem (one at a time) or they need to be optimized using an iteration scheme appropriate for nonlinearity (see section 2.5.

Due to the serialized nature of parameter optimization, these computations are too extensive for a nonlinear system under the time constraints of this research. For this reason a rough sensitivity study is performed on the parameters in this case instead. While this makes a comparison between different systems difficult, a sensitivity study does give an indication about the potential effectiveness of the interdependent damping system.

2.5 Iteration

Iteration can be used for root finding and for finding minima and maxima. Both uses are explained here.

As nonlinear systems are concerned, the Newton method is an appropriate iteration method [33].

2.5.1 Newton method for root finding

In principle, the unit response function H(ω) for the total system can be found for every tank shape in the manner described in this chapter. However, the shape of the tank in the case of the investigated system of this thesis is determined by the displacement of the tuned mass dampers (see section 3.1.3 for a description of the system). This means that the response of the system and the tank shape are interdependent and need to be computed using iteration.

Only one initial guess is needed as a starting point, as well as the derivative at this point. In case of a system of equations this would be the Jacobian. The process to solve a system $\mathbf{G}(\mathbf{x}) = \mathbf{0}$ is defined as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J(\mathbf{x}_k)^{-1}\mathbf{G}(\mathbf{x}_k) \tag{2.99}$$

where: $\mathbf{x}_k$ = unknown vector at the k$^{\text{th}}$ step;
$J(x_k)^{-1}$ = inverse Jacobian at the k$^{\text{th}}$ step;
$\mathbf{G}(\mathbf{x}_k)$ = system at the k$^{\text{th}}$ step

In case of the system specified here, the Jacobian can not be determined analytically. Therefor a finite difference approximation is used for a numerical approach:

$$J_{i,j} = \frac{\partial g_i(x_j)}{\partial x_j} \approx \frac{g_i(x_j + \epsilon) - g_i(x_j)}{\epsilon} \tag{2.100}$$

Where ϵ is the step size and approaches 0.

While this method is not guaranteed to converge to the desired solution or any solution, modified methods exist that do [16][7]. As the iterations by the Newton method give declining $\mathbf{G}_k$ as k increases as long as the step size is small enough, controlling the step size leads to the desired outcome. The global or damped Newton method reduces the step size per iteration by using a damping term r:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - r \cdot J(\mathbf{x}_k)^{-1}\mathbf{G}(\mathbf{x}_k) \tag{2.101}$$

$$\text{or:} \tag{2.102}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \Delta\mathbf{x}_{k,k+1} \tag{2.103}$$

For a regular Newton iteration, $r = 1$. Armijo's rule [19] gives the following procedure to determine the damping term for the modified method, using a temporary vector $\mathbf{G}(\mathbf{t}) = \mathbf{G}(\mathbf{x}_{k+1})$ for $r = 1$:

$$\text{While } |\mathbf{G}(\mathbf{t})| > |\mathbf{G}(\mathbf{x}_k)| :$$

$$\mathbf{t} = \mathbf{x}_k - r \cdot J(\mathbf{x}_k)^{-1}\mathbf{G}(\mathbf{x}_k) \tag{2.104}$$

$$r = \frac{1}{2}r \tag{2.105}$$

$$\mathbf{G}(\mathbf{t}) = \mathbf{G}(\mathbf{x}_k) - r \cdot J(\mathbf{x}_k)^{-1}\mathbf{G}(\mathbf{x}_k) \tag{2.106}$$

This gives the next iteration as:

$$\mathbf{x}_{k+1} = \mathbf{t} \tag{2.107}$$

$$\mathbf{G}(\mathbf{x}_{k+1}) = \mathbf{G}(\mathbf{t}) \tag{2.108}$$

2.5.2 Newton method for optimization

Finding the optimal damping parameters for a system are found iteratively. This can be done one parameter at a time, or simultaneously. The Newton method can be used to find the maxima, minima and saddle-points of a system by looking for the root of the derivative of the system. The final iteration gives a local minimum if and only if the Hessian is positive definite, which needs to be checked. If the Hessian is negative definite, the final iteration gives a maximum. If some eigenvalues of the Hessian are negative and some positive, the critical point that is found is a saddle point.

Since the optimization problem adapts the root finding problem to the derivative of the system, it is defined as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H(\mathbf{x}_k)^{-1}\nabla_k\mathbf{G}(\mathbf{x}_k) \tag{2.109}$$

where: $\mathbf{x}_k$ = unknown vector at the k$^{\text{th}}$ step;
$H(x_k)^{-1}$ = inverse Hessian at the k$^{\text{th}}$ step;
∇_k = $\sum \frac{\partial}{\partial \boldsymbol{x}_{i,k}}$ = the nabla operator at the k$^{\text{th}}$ step;
$\mathbf{G}(\mathbf{x}_k)$ = system at the k$^{\text{th}}$ step.

The Hessian is determined using finite difference approximation:

$$H_{i,j} = \frac{\partial^2 g_i(x_1, .., x_i, .., x_j, .., x_n)}{\partial x_i x_j} \tag{2.110}$$

$$\approx \frac{g_i(x_1, .., x_i + \epsilon, .., x_j + \epsilon, .., x_n) - g_i(x_1, .., x_i + \epsilon, .., x_j, .., x_n)}{\epsilon^2} \tag{2.111}$$

$$- \frac{g_i(x_1, .., x_i, ..x_j + \epsilon, .., x_n) + g_i(x_1, .., x_i, ..x_j, .., x_n)}{\epsilon^2} \tag{2.112}$$

Where step size ϵ approaches 0. Like the root finding process, the optimization process does not always converge, but can be made to by controlling the step size. The damping term r then changes equation 2.109 into:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - r \cdot H(\mathbf{x}_k)^{-1} \nabla_k \mathbf{G}(\mathbf{x}_k) \tag{2.113}$$

To ensure convergence, require the gradient of the system $\nabla \mathbf{G}(\mathbf{x}_k)$ to go down. Starting as an undamped iteration using $r = 1$, r is determined by Armijo's rule as follows:

$$\text{While } |\nabla \mathbf{G}(\mathbf{t})| > |\nabla \mathbf{G}(\mathbf{x}_k)| :$$

$$\mathbf{t} = \mathbf{x}_k - r \cdot H(\mathbf{x}_k)^{-1} \nabla_k \mathbf{G}(\mathbf{x}_k) \tag{2.114}$$

$$r = \frac{1}{2} r \tag{2.115}$$

$$\nabla \mathbf{G}(\mathbf{t}) = \nabla \mathbf{G}(\mathbf{x}_k) - r \cdot H(\mathbf{x}_k)^{-1} \nabla \mathbf{G}(\mathbf{x}_k) \tag{2.116}$$

This gives the next iteration as:

$$\mathbf{x}_{k+1} = \mathbf{t} \tag{2.117}$$

$$\nabla \mathbf{G}(\mathbf{x}_{k+1}) = \nabla \mathbf{G}(\mathbf{t}) \tag{2.118}$$

2.5.3 Sensitivity study

In order to survey whether the optimum for any of the damping parameters found is merely a local minimum, a rough sensitivity analysis is performed for each parameter. This analysis gives the change of the performance index depending on the change of a particular parameter.

The interdependent TMD-TLD system is analyzed for the optimal parameters of the independent TMD-TLD system. The optimal parameters for these systems are not the same. The effect of varying damping parameters on the efficiency of the interdependent system are analyzed. The parameters checked for are the damping constant c_{TMD} of the TMD's, the stiffness constant k_{TMD} of the TMD's and the damping constant c_{TLD} of the TLD.

In favor of simplicity each parameter is adjusted separately to analyze the impact of its change on the system. The performance index is determined for each parameter within a range of the optimized parameter. All other parameters are kept constant when analysing any one parameter.

While it is easier to judge the impact of each parameter this way, the parameters of the TMD and TLD influence each other's effectiveness. To get a clearer picture of the sensitivity, a series of parameter combinations need to be computed and plotted. This is out of the scope of this thesis.

3 | Model set-up

The general theory to calculate the motion of a floating barge have been discussed in chapter 2. However, the method to compute specific barge structures can deviate slightly. In this chapter, the model set up of three different barge structures with increasing complexity will be discussed. The first two barge structures can be analysed straightforwardly, whereas the last structure (the structure under investigation in this thesis) will need some more attention as it behaves nonlinearly. The first two structures are mainly explored in order to have a reference point for the latter structure.

All systems have the same main structure consisting of a floating barge and a wind turbine. The barge structure is based on a preliminary design by the Department of Naval Architecture and Marine Engineering at the Universities of Glasgow and Strathclyde [3]. The wind turbine used for these systems was developed as a representational turbine for concept studies known as the 'NREL offshore 5-MW baseline wind turbine' [22].

3.1 System description

3.1.1 Undamped barge

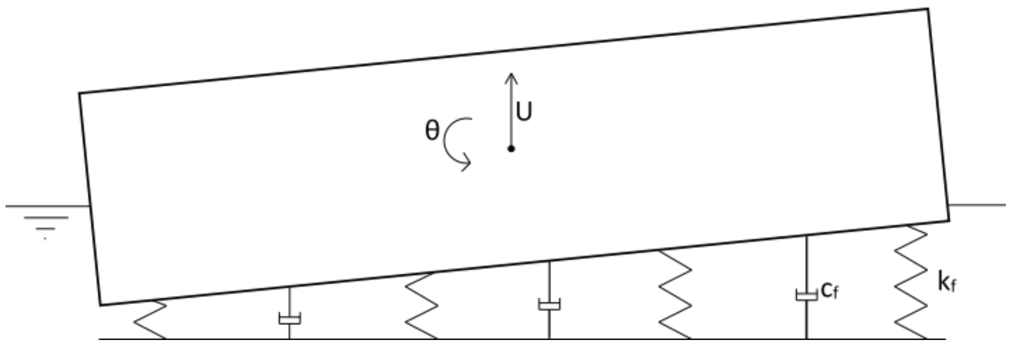

Figure 3.1: Basic barge, no structural damping system. The hydrostatic stiffness coefficient is shown as k_f and the hydrodynamic damping coefficient as c_f.

The first barge to be analysed is the basic barge structure as shown in figure 3.1. Since this structure only includes the barge itself, it is treated as a foundation for the other two systems. Everything discussed here, also holds true for the damped structures.

First of all, the barge of each structure will be regarded as a floating stiff body. The barge itself has two degrees of freedom: the vertical translation U and the rotation θ. Any horizontal translation by the barge is disregarded. The water the barge floats on is modeled as a continuous base with a stiffness matrix **D** and a damping matrix **B**. **D** and **B** are the hydrostatic stiffness and hydrodynamic damping matrices respectively. The barge does not have a stiffness matrix **K**, since it is modeled as infinitely stiff so that no potential energy is present in the form of structural deformation.

3.1.2 Barge with separate TMD and TLD system

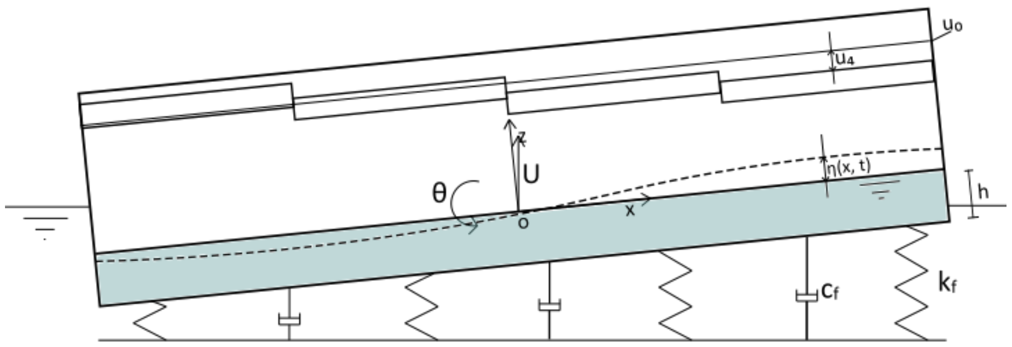

Figure 3.2: Barge with independent TMD and TLD system. Here, u_4 is the DoF of the 4th TMD with u_0 as baseline. µ(x, t) is the wave elevation of the TLD, h is the resting TLD liquid line and k_f and c_f are the hydrostatic stiffness and hydrodynamic damping of the floating surface respectively.

The system shown in figure 3.2 has a damping system consisting of multiple TMDs and a TLD. The TLD is placed at the bottom of the barge so that the TMDs do not interfere with the TLD's tank shape and therefore not with its eigenfrequencies. Granted that the distance between TLD and TMD is large enough, the TMDs and TLD do not affect eachother.

For readability of the figure, the springs and dampers of the TMDs are left out of the figure. However, each TMD has a spring stiffness of k_i and a damping coefficient of c_i, where i is the ith TMD from left to right. Each TMD has one vertical degree of freedom u_i, with the TMD height in rest marked in the figure as u_0.

The TLD has one representational degree of freedom q(t), which is the wave shape amplitude at $x = 0$ (see paragraph 2.1.3). In order to find the sloshing modes, the local coordinates x and z are used. The water height in rest is h, while the free surface elevation is denoted as η.

3.1.3 Barge with interdependent TMD and TLD system

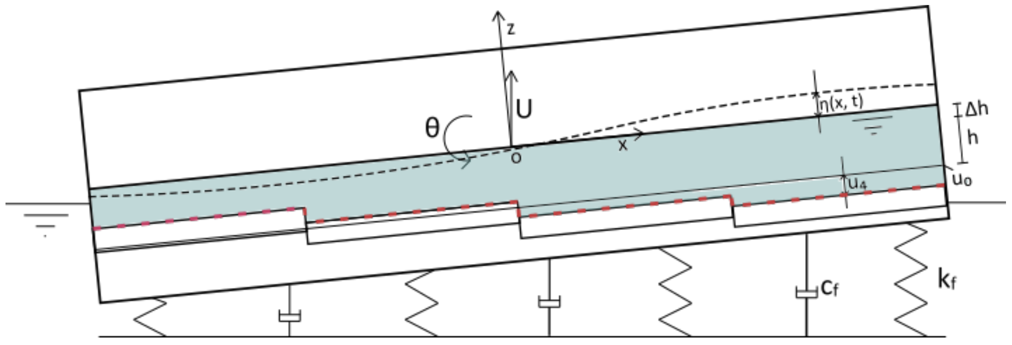

Figure 3.3: Barge with interdependent TMD and TLD system. Here, u_4 is the DoF of the 4th TMD with u_0 as baseline. µ(x, t) is the wave elevation of the TLD, h + ∆h is the resting TLD liquid line and k_f and c_f are the hydrostatic stiffness and hydrodynamic damping of the floating surface respectively.

The barge system discussed here is the most complex one and the main focus of this thesis. The barge structure is shown in figure 3.3. In this case, the TLD is placed directly over the TMDs. The purpose of the proposed TMD - TLD combination is to enhance the effect of each damping system. Here, the TMDs are also the bottom of the TLD tank. The weight distribution of the TLD liquid creates a restoring moment. Letting the bottom of the TLD tank be be related to the pressure of the TLD so that it is reversely related to the displacement of the platform, increases the restoring moment created by a standard TLD. Similarly, the increased pressure on the TMDs by the damping liquid gives a bigger response by the TMDs, which in turn increases their restoring forces onto the platform.

The wave shape and sloshing modes of the liquid depend on the tank shape. Thus, in case of a time dependent tank shape, the Lagrangian and therefore the mass- and stiffness matrices are time dependent. Also, the response amplitude of the TMDs is dependent on the wave of the TLD, not just its amplitude and is therefor time dependent. In principle, the unit response function H(ω) for the total system can be found for every tank shape in the manner described in 2.3.1. However, the shape of the tank in the case of this model is determined by the displacement of the tuned mass dampers. This means that the response of the system and the tank shape are interdependent and need to be computed using iteration.

The boundary conditions change after every iteration. Apart from the changing eigenfrequency and eigenmodes of the liquid, the liquid pressure on the TMDs changes depending on the height of the liquid over the TMD.

The volume of damping liquid stays the same, so that the level water height changes per iteration. This needs to be accounted for to prevent invalid large integrations over the liquid volume in case of a big response of the TMDs near the eigenfrequencies. This means that depending on the TMD movement, the 'resting' water height changes with a Δh, as shown in figure 3.3.

When calculating the fluid potential of the TLD, volume integration is performed. In the case of multiple TMDs, this integration is done as a sum of the integration over each compartment of a TMD. Volume integration over the resting liquid is done only where the level water height is higher than the TMD in question. In case of integration over the volume under the wave shape, integration over x is limited to where the wave is higher than the TMD in question. This to prevent negative integration values in case of a TMD displacement higher than the liquid surface. See figure 3.4 for additional clarification.

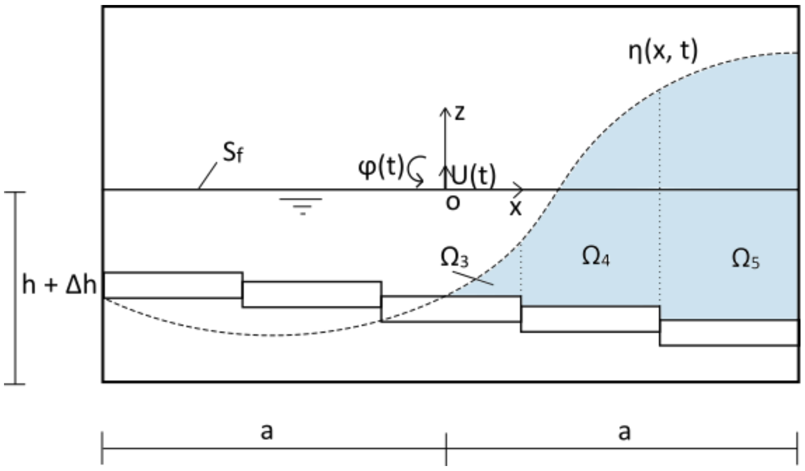

Figure 3.4: Volume integration is performed over sections of the fluid domain. In the situation depicted, integration starts from the intersection of the wave shape η(x, t) with the third TMD. Integration before the intersection would give negative results.

3.2 Assumptions and boundary conditions

Some assumptions are applied to the liquid in and outside of the structure, which influence the potential flow computations.

First of all, while the abrupt angles in the structure would cause turbulence, this is ignored. The liquid movement inside the TLD and outside the structure are assumed to be irrotational so that linear velocity potential theory can be applied. For limited liquid velocities, this is still a good approximation.

Secondly, all used fluid is deemed homogeneous and incompressible. This means most notably that the water pressure at the seabed does not influence the water density there despite the water depth. Increased density at this depth could slightly alter the velocity potential, but is not significant considering the size of the fluid domain.

A number of simplifications are made with regards to the structure and its situation as well.

First of all, the most important simplification of the models is the use of a stiff body for the barge-turbine structure. This modification reduces the amount of degrees of freedom, namely the local degrees of freedom are disregarded. The energy dissipation due to movement in these local degrees of freedom is omitted in the stiff models, making them more conservative. However, the reduction of degrees of freedom also means less eigenfrequencies and less chance at resonance.

Also, horizontal movement is disregarded as the implemented damping systems are not expected to affect surge of the barge system.

Thirdly, the displacement response of each model depends on the type of loading. Each of the models is subjected to a white noise load instead of a characteristic ocean wave load. The reason for this is the influence of the parameters of the main structure on the response to certain loading. Shifting its eigenvalues by the additional dampers could either positively or negatively influence the response to loading other than white noise. The effectiveness of the damping systems loaded by white noise, shows their potential for damping capacity rather then their damping capacity in a more specific loading situation. Where these systems used in real life, the main structure would be designed in conjunction with the damping systems.

Lastly, the effect of mooring cables is not included in the models.

Regarding the boundary conditions: the stiff walls of the barge give boundary conditions conforming to the continuity of liquid velocity normal to the tank walls by only using global displacements of the structure, not local deformations. See 2.1.2 for the complete set of boundary conditions of both the TLD liquid in a given tank and the ocean water.

The boundary conditions for the model with the interdependent TMD and TLD system, are time dependent, because the tank shape is time dependent. The boundary condition concerning liquid velocity continuity at the bottom of the tank, now depends on the velocity of the TMDs instead of only the velocity of the barge. Thus, to account for the velocity at each TMD, equation 2.6 becomes:

$$\boldsymbol{v}_f(x, z = -h + u_i(t), t) = \dot{u}_i \tag{3.1}$$

And in order to determine the potential for each tank shape, the modal amplitudes of the TLD need to be calculated using an altered liquid domain. Equations 2.27 to 2.37 become:

$$A_m = \frac{-b_{Sm} - \sqrt{b_{Sm}^2 - 4a_{Sm}c_{Sm}}}{2a_{Sm}} \tag{3.2}$$

With: (3.3)

$$a_{Sm} = \sum_{i=0}^{n} \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} cos^2(\beta_m x) + sinh^2(\beta_m z)\, dx\, dz \tag{3.4}$$

$$b_{Sm} = \sum_{i=0}^{n} 2 \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} cosh(\beta_m z) + sinh(\beta_m z)\, dx\, dz - \frac{a}{\beta_m} \tag{3.5}$$

$$c_{Sm} = \sum_{i=0}^{n} \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} sin^2(\beta_m x) + sinh^2(\beta_m z)\, dx\, dz \tag{3.6}$$

$$B_m = \frac{-b_{Am} - \sqrt{b_{Am}^2 - 4a_{Am}c_{Am}}}{2a_{Am}} \tag{3.7}$$

With: (3.8)

$$a_{Am} = \sum_{i=0}^{n} \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} sin^2(\alpha_m x) + sinh^2(\alpha_m z)\, dx\, dz \tag{3.9}$$

$$b_{Am} = \sum_{i=0}^{n} 2 \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} cosh(\alpha_m z) + sinh(\alpha_m z)\, dx\, dz - \frac{a}{\alpha_m} \tag{3.10}$$

$$c_{Am} = \sum_{i=0}^{n} \int_{-h+u_i(t)}^{0} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} cos^2(\alpha_m x) + sinh^2(\alpha_m z)\, dx\, dz \tag{3.11}$$

As explained in paragraph 3.1.3, the potential flow of the TLD is determined by these amplitudes and can give a response of the TMDs different to the ones used in the first place. This means that the potential and the response calculations need to be iterated until the used tank shape is near enough the displacement equivalent of the TMDs.

The system is not linear, but will be approximated as such for the modal analyses.

3.3 Model parameters

3.3.1 Barge and wind turbine parameters

While the barge in the models is based on a particular preliminary design, this design includes a moon pool for wave energy extraction, which is not adopted for the models in this thesis. This barge is chosen as it was designed specifically to support the reference wind turbine [3].

One additional parameter that was not specified for the designed barge is its seabed depth, as it should be useful at varying depth. Since floating wind turbines become appealing for ocean depths of 60m and over, the models will be analysed at an ocean depth of H = 60m. This is relevant for determining the potential flow outside of the structure and therefor the additional hydrodynamic parameters.

The adopted wind turbine is the 'NREL offshore 5-MW baseline wind turbine', which is a conceptual turbine [22]. It was developed to represent utility-scale multi-megawatt turbines for offshore wind technology. Its features are a composite of representational properties obtained from turbine manufacturers.

Since large part of the mass of the barge consists of added ballast, this will be replaced by the mass dampers. While the total amount of ballast is unknown, the mass for all TMD's combined is set to 10% of the total mass. The total mass stays consistent independent of the TLD or TMD mass.

All necessary parameters for the base structure for the models in this thesis is summarized in table 3.3.1.

Barge	
Size (W x L x H)	40 x 40 x 10
Mass	5 452 000 kg
Depth seabed	60 m
Wind turbine	
Hub height	87.7 m
Turbine radius	63 m
Blade structural and aerodynamic damping ratio	0.5 %
Tower structural damping ratio	1 %
Mass	697 460 kg
Vertical centre of mass	64 m
Mass dampers	
Mass ratio TMD to total mass	1/10
Number of TMD's	2
Mass per TMD	272 600 kg

Table 3.1: Parameters base structure

3.3.2 Additional hydrodynamic and hydrostatic parameters

From the derivation in section 2.1.4, the additional hydrodynamic mass and damping matrices and the hydrostatic stiffness matrix and force vector are known.

The additional hydrodynamic mass and damping parameters for the barge-turbine system are the following for each modal mode m:

$$\alpha_m = \frac{(2m-1)\pi}{120} \tag{3.12}$$

$$A_m = \frac{1}{2}(2m-1)\pi - 1 \tag{3.13}$$

$$m_{h,U} = -4\cdot 10^4 \sum (60 - \frac{120}{(2m-1)\pi})(exp(-\frac{(2m-1)\pi}{6}) - exp(\frac{(2m-1)\pi}{6})) \tag{3.14}$$

$$\begin{aligned} m_{h,\theta} =& -4\cdot 10^4 \sum (1200 + \frac{61\cdot 120}{(2m-1)\pi} + \frac{14400}{(2m-1)^2\pi^2}) \\ & \cdot (exp(-\frac{(2m-1)\pi}{6}) - exp(\frac{(2m-1)\pi}{6})) \end{aligned} \tag{3.15}$$

$$m_{h,\theta U} = m_{h,U\theta} = 0 \tag{3.16}$$

$$c_{d,U} = \sum \frac{4}{3\pi}((2m-1)\pi - 1)800\rho\frac{1}{4}(\frac{2\pi(\frac{1}{2}(2m-1)\pi - 1)}{40})^{-\frac{1}{3}} \tag{3.17}$$

$$c_{d,\theta} = \sum c_{d,\theta U} = c_{d,U\theta} = 0 \tag{3.18}$$

Which then gives the additional mass (**A**) and damping (**B**) matrices:

$$\boldsymbol{A} = \begin{bmatrix} m_{h,U} & m_{h,\theta U} & 0 & \dots & 0 \\ m_{h,U\theta} & m_{h,\theta} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad \boldsymbol{B} = \begin{bmatrix} c_{h,U} & c_{h,\theta U} & 0 & \dots & 0 \\ c_{h,U\theta} & c_{h,\theta} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{3.19}$$

The additional stiffness matrix **D** and the added hydrostatic force vector $\mathbf{F_h}$ on the barge for the base structure are found from equations 2.68. $\mathbf{F_h}$ depends on the floating depth H of the barge, which can be determined using Archimedes' principle as $H = (M_{Barge} + M_{Turbine})/(2\rho ab) = 3.84m$. This gives:

$$\boldsymbol{D} = \begin{bmatrix} 1.57 \cdot 10^7 & 0 & 0 & \dots & 0 \\ 0 & 1.57 \cdot 10^8 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad \boldsymbol{F_h} = \begin{bmatrix} 6.03 \cdot 10^7 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.20}$$

3.3.3 Parameters damping systems

The parameters of the damping systems are determined by optimization. As discussed in section 2.4, optimization of multi degree of freedom systems is done through iteration. The starting parameters for this iteration are given by the optimized parameters of an equivalent single degree of freedom system. These starting parameters are the same for both the model with independent TLD and TMD system and the model with interdependent TLD and TMD system. These starting parameters are determined after the modal analysis of the undamped structure is finished, see paragraph 4.3.1. After all, the eigenfrequencies and mode shapes need to be determined before optimization is possible.

3.3.4 Subscribed load

Each model is subjected to the same normalized force in a range of frequencies. Since the response amplitude for very high amplitudes goes to zero for all models, this range is limited to $\omega = \langle 0, 3]$ rad/s. The models are subjected to a positive translation force $\mathrm{F_v}$ and a positive moment M with the magnitude of the system's dead weight. Thus:

$$\boldsymbol{F_{load}} = \begin{bmatrix} F_v \\ M \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 5.452 \cdot 10^3 kN \\ 5.452 \cdot 10^3 kNm \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.21}$$

3.4 Equations of motion

The equations of motion of the three systems are constructed as explained in section 2.2.

3.4.1 Undamped barge

Without the stiffness and damping from the surrounding fluid, the potential energy $\mathcal{P}$ and the kinetic energy $\mathcal{K}$ of the undamped system (section 3.1.1) are:

$$\mathcal{P} = MgU \tag{3.22}$$

$$\mathcal{K} = \frac{1}{2}M\dot{U}^2 + \frac{1}{2}M\int_{-a}^{a}\frac{1}{2a}\dot{\theta}^2 x^2\,dx = \frac{1}{2}M\dot{U}^2 - \frac{1}{6}M\dot{\theta}^2 a^2 \tag{3.23}$$

Where: M = mass of the structure $[kg]$;
U = vertical translation $[m]$;
θ = rotation $[rad]$;
a = half the barge width $[m]$.

This gives the following mass matrix **M** and force vector **F**:

$$\boldsymbol{M} = \begin{bmatrix} M & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{6}a^2M & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} = \begin{bmatrix} 6,149 \cdot 10^6 & 0 & 0 & \dots & 0 \\ 0 & 4.099 \cdot 10^8 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{3.24}$$

$$\boldsymbol{F} = \begin{bmatrix} -Mg \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} -6.033 \cdot 10^7 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.25}$$

The complete equation of motion with the given matrices then is:

$$(\boldsymbol{M} + \boldsymbol{A})\ddot{\boldsymbol{x}} + \boldsymbol{B}\dot{\boldsymbol{x}} + \boldsymbol{D}\boldsymbol{x} = \boldsymbol{F} + \boldsymbol{F_h} + \boldsymbol{F_{load}} \tag{3.26}$$

3.4.2 Independently damped barge

The model of the independently damped barge (section 3.1.2) has 2 + n degrees of freedom, namely vertical translation U of the barge (heave), rotation θ of the barge (pitch) and translation of the n TMDs u_i. Without the influence of the floating surface, the potential energy $\mathcal{P}$, the kinetic energy $\mathcal{K}$ and the Rayleigh dissipation $\mathcal{R}$ are determined as follows:

$$\begin{aligned} \mathcal{P} =& \sum_{i=0}^{n}(\frac{1}{2}k_i(u_i - (U + \theta(-a + (2a\frac{i}{n-1}))))^2 + m_ig(u_i + \theta(-a + (2a\frac{i}{n-1}))) + MgU \\ &+ \rho g b \int_{-a}^{a}\int_{0}^{\eta}(z + U + \theta x)\,dz\,dx \\ =& \sum_{i=0}^{n}(\frac{1}{2}k_i(u_i - (U + \theta(-a + (2a\frac{i}{n-1}))))^2 + m_ig(u_i + \theta(-a + (2a\frac{i}{n-1})) + MgU \\ &+ \rho g b \int_{-a}^{a}(\frac{1}{2}\eta + U + \theta x)\eta\,dx \end{aligned} \tag{3.27}$$

$$\mathcal{K} = \sum_{i=0}^{n}\frac{1}{2}m_i\dot{u}_i^2 + \frac{1}{2}M\dot{U}^2 + \frac{1}{6}Ma^2\dot{\theta}^2 + \frac{1}{2}\rho b\int_{-h}^{0}\int_{-a}^{a}(\dot{U} + \dot{\theta}x + \frac{\partial\phi}{\partial z})^2 + (\frac{\partial\phi}{\partial x})^2\,dx\,dz \tag{3.28}$$

$$\mathcal{R} = \frac{1}{2}\sum_{i=0}^{n}c_i(\dot{u}_i - (\dot{U} + \dot{\theta}(-a + (2a\frac{i}{n-1}))))^2 + \frac{1}{2}bc_{TLD}\sum_{i=0}^{n}\int_{-h}^{0}\int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)}(\frac{\partial\phi}{\partial z})^2 + (\frac{\partial\phi}{\partial x})^2\,dx\,dz \tag{3.29}$$

Where:	U	= vertical translation $[m]$;
	θ	= rotation $[rad]$;
	a	= half the barge width $[m]$;
	k_i	= stiffness of the ith TMD $[N/m]$;
	c_i	= damping coefficient of the ith TMD $[Ns/m]$;
	c_{TLD}	= equivalent damping coefficient of the TLD $[Ns/m]$;
	M	= mass structure $[kg]$;
	m_i	= mass of the ith TMD $[kg]$;
	u_i	= displacement of the ith TMD $[m]$;
	μ	= wave elevation function $[m]$;
	ϕ	= velocity potential function $[m^2/s]$;
	n	= number of TMDs.

This gives the following mass (**M**), damping (**C**) and stiffness (**K**) matrices and load vector (**F**) for n = 2:

$$\boldsymbol{M} = \begin{bmatrix} 6.981\cdot10^5 & 0 & 0 & 3.075\cdot10^5 & 0 \\ 0 & 2.571\cdot10^8 & 1.269\cdot10^5 & 6.149\cdot10^6 & 0 \\ 0 & 1.269\cdot10^5 & 1.323\cdot10^4 & 0 & 0 \\ 3.075\cdot10^5 & -6.149\cdot10^6 & 0 & 3.075\cdot10^5 & 0 \\ 3.075\cdot10^5 & 6.149\cdot10^6 & 0 & 0 & 3.075\cdot10^5 \end{bmatrix} \tag{3.30}$$

$$\boldsymbol{C} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7.456\cdot10^3 & 0 & 0 \\ 0 & 0 & 0 & 1.805\cdot10^5 & 1.805\cdot10^5 \\ 0 & 0 & 0 & 1.805\cdot10^5 & 1.805\cdot10^5 \end{bmatrix} \tag{3.31}$$

$$\boldsymbol{K} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3.537\cdot10^5 & 0 & 0 \\ 0 & 3.537\cdot10^5 & 1.029\cdot10^3 & 0 & 0 \\ 0 & 0 & 0 & 9.265\cdot10^5 & 0 \\ 0 & 0 & 0 & 0 & 9.265\cdot10^5 \end{bmatrix} \tag{3.32}$$

$$\boldsymbol{F} = \begin{bmatrix} 6.033\cdot10^7 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.33}$$

3.4.3 Interdependently damped barge

The model of the interdependently damped barge (section 3.1.3) has 2 + n degrees of freedom, namely vertical translation U of the barge (heave), rotation of the barge θ and translation of the n TMDs u_i. Because the damping systems (TMDs and TLD) are interdependent in this model, the potential and kinetic energy of the TLD differs slightly from the model with independent damping. The total distance the fluid gets displaced now changes as a result of the TMD displacements as well. As the TMD response is not yet known, this is done iteratively, using the previously estimated response of the TMDs $u_{i,k-1}$ to determine the kth iteration step.

With the influence of the floating surface omitted, the potential energy $\mathcal{P}$, the kinetic energy $\mathcal{K}$ and the Rayleigh dissipation $\mathcal{R}$ are determined as follows:

$$\begin{aligned} \mathcal{P} =& \sum_{i=0}^{n}(\frac{1}{2}k_i u_{i,k}^2 + m_i g(u_{i,k} + U_k + \theta_k(-a + (2a\frac{i}{n-1}))) + MgU_k \\ &+ \rho g b \sum_{i=0}^{n} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} (\frac{1}{2}(\eta + \Delta h) + U_k + \theta_k(-a + (2a\frac{i}{n-1})))(\eta + \Delta h)\, dx \end{aligned} \tag{3.34}$$

$$\begin{aligned} \mathcal{K} =& \sum_{i=0}^{n} \frac{1}{2} m_i (\dot{u}_{i,k} + \dot{U}_k + \dot{\theta}_k(-a + (2a\frac{i}{n-1})))^2 \\ &+ \frac{1}{2}\rho b \sum_{i=0}^{n} \int_{-h+u_{i,k-1}}^{\Delta h} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} (\dot{u}_{i,k} + \dot{U}_k + \dot{\theta}_k(-a + (2a\frac{i}{n-1})) + \frac{\partial \phi}{\partial z})^2 + (\frac{\partial \phi}{\partial x})^2\, dx\, dz \\ &+ \frac{1}{2}M\dot{U}_k^2 + \frac{1}{6}Ma^2\dot{\theta}_k^2 \end{aligned} \tag{3.35}$$

$$\mathcal{R} = \frac{1}{2}\sum_{i=0}^{n} c_i \dot{u}_{i,k}^2 + \frac{1}{2} b c_{TLD} \sum_{i=0}^{n} \int_{-h+u_{i,k-1}}^{\Delta h} \int_{-a+2\frac{a}{n}i}^{-a+2\frac{a}{n}(i+1)} (\frac{\partial \phi}{\partial z})^2 + (\frac{\partial \phi}{\partial x})^2\, dx\, dz \tag{3.36}$$

Where:	U_k	= vertical translation at the k$^{\text{th}}$ iteration step $[m]$;
	θ_k	= rotation at the k$^{\text{th}}$ iteration step $[rad]$;
	a	= half the barge width $[m]$;
	k_i	= stiffness of the i$^{\text{th}}$ TMD $[N/m]$;
	h	= height TLD liquid $[m]$;
	Δh	= displacement of TLD level $[m]$;
	c_i	= damping coefficient of the i$^{\text{th}}$ TMD $[Ns/m]$;
	c_{TLD}	= equivalent damping coefficient of the TLD $[Ns/m]$;
	M	= mass structure $[kg]$;
	m_i	= mass of the i$^{\text{th}}$ TMD $[kg]$;
	$u_{i,k}$	= displacement of the i$^{\text{th}}$ TMD at the k$^{\text{th}}$ iteration step $[m]$;
	μ	= wave elevation function $[m]$;
	ϕ	= velocity potential function $[m^2/s]$;
	n	= number of TMDs .

4 | Results

Modal analysis of each of the systems is performed in this chapter. The response spectrum in the frequency domain is given for each system, as well as the response in the time domain.

4.1 Additional hydrodynamic and hydrostatic parameters

The additional hydrodynamic mass and damping, and the hydrostatic stiffness and external force are the same for each system. They are given in section 3.3.2. For the first eigenmode, the additional mass (**A**) and damping (**B**) matrices are:

$$\boldsymbol{A} = \begin{bmatrix} 9.556 \cdot 10^5 & 0 & 0 & \dots & 0 \\ 0 & 2.187 \cdot 10^8 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{4.1}$$

$$\boldsymbol{B} = \begin{bmatrix} 4.009 \cdot 10^6 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{4.2}$$

The additional hydrostatic stiffness matrix **D** and hydrostatic force vector $\mathbf{F_h}$ are also the same for all models and are given in section 3.3.2

4.2 Undamped system

The model as described in paragraph 3.1.1 is analysed here.

4.2.1 Response spectrum

The response spectrum of the undamped system to the normalized force shows that the barge has two eigenfrequencies, see figure 4.1. The first peak gives the pitch eigenfrequency $\omega = 0.389$ rad/s^2 of the barge. This response has been multiplied by the height of the turbine mast to give the approximate horizontal displacement of the hub. The second peak gives the heave eigenfrequency $\omega = 1.486$ rad/s^2 of the barge.

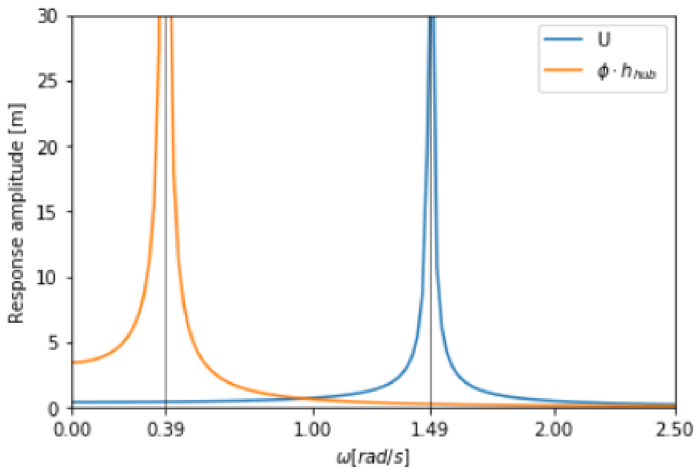

Figure 4.1: Frequency response functions of the heave U [m] and pitch Θ [rad] of the undamped system.

The integral of both response spectra gives the performance index J, see chapter 2.4. This gives $J = 12.32$ for the undamped system.

4.3 Barge with separate TMD and TLD system

The model as described in paragraph 3.1.2 is analysed here.

4.3.1 Optimization first step

The first step in finding the optimal system parameters is described in chapter 2.4. As described here, the optimal parameters for a single degree of freedom system under white noise loading are known. Using the eigenfrequencies found for the undamped system, estimates are made for the optimal stiffness parameters and damping parameters of the TMD's and the TLD. The equivalent stiffness of a TLD is related to its fluid height h, which also determines the weight of the TLD. The TMD's are tuned to the vertical translation eigenfrequency ω = 1.486 rad/s, while the TLD is mostly responsive to rotation and is therefore tuned to the rotation eigenfrequency ω = 0.389 rad/s.

Combined with the given TMD to barge mass ratio (table 3.3.1), this gives the starting parameters as seen in table 4.3.1.

Starting parameters	
k_{TMD}	533.74 kN/m
c_{TMD}	137.01 kNs/m
h_{TLD}	0.181 m
c_{TLD}	12.65 kNs/m

Table 4.1: Starting parameters for the optimization of the damped system with separate TMD's and TLD.

4.3.2 Optimization

As described in chapter 2.4, the performance index J are minimized as a measure for the optimal stiffness and damping parameters of the TMD's and the TLD. This can be done by looking for a minimum one parameter at a time in a serialized fashion or all parameters at once using an appropriate iteration scheme. In this section serial optimization is performed as described in section 2.4 and combined optimization is performed using the Newton method as described in section 2.5.

Serial optimization

The algorithm for serial optimization can be found in appendix B.6.

Starting with the optimal damping parameters for a single degree of freedom system, the progression of the performance index to the optimal parameters for the actual system reaches convergence in very few steps. This is clear from the convergence charts 4.2.

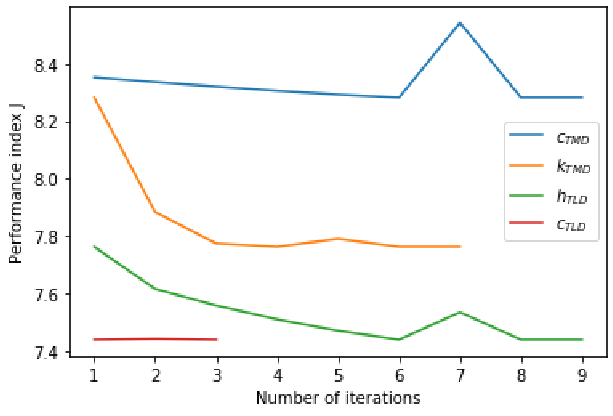

Figure 4.2: Convergence of the performance index J to its optimum using serial optimization, depending on the parameters of the TMD's and TLD.

A possible problem arises, as it is not known whether the minimum found is only a local minimum. A rough sensitivity study is done to combat this. From figures 4.4, 4.6 it is clear that there is a distinct minimum for the TMD and TLD damping parameters c_{TMD}, c_{TLD}.

This is not the case for the TMD stiffness parameter k, see figure 4.3. The minimum found by starting from the SDOF optimum is a local minimum, but there are distinct peaks and valleys. Within the studied range the local minimum suffices, though it is unknown if a lower performance index may be found. The multiple minima along different k_{TMD} are caused by the changes in eigenfrequency as the stiffness changes.

A clear patern of maxima and minima can also be seen for the TLD height h (figure 4.5. The equivalent stiffness of the TLD depends in part on the liquid heigth, as well as its mass. Therefor the eigenfrequency of the TLD is highly susceptible to changes in h. There seems to be a more apparent optimum for the TLD height than for the TMD stiffness within the range of the sensitivity study. When taking the mass of the system into account, any minima found for larger TLD heights are not relevant, as the TLD weight can not be bigger than the total weight.

The minima in the sensitivity figures coincide with the parameters found, which suggests that the parameters found are adequately close to optimal. See table 4.3.2 for the final values.

Optimized parameters	
k_{TMD}	503.81 kN/m
c_{TMD}	87.13 kNs/m
h_{TLD}	0.313 m
c_{TLD}	11.85 kNs/m

Table 4.2: Optimized parameters for the damped system with separate TMD's and TLD using serial optimization.

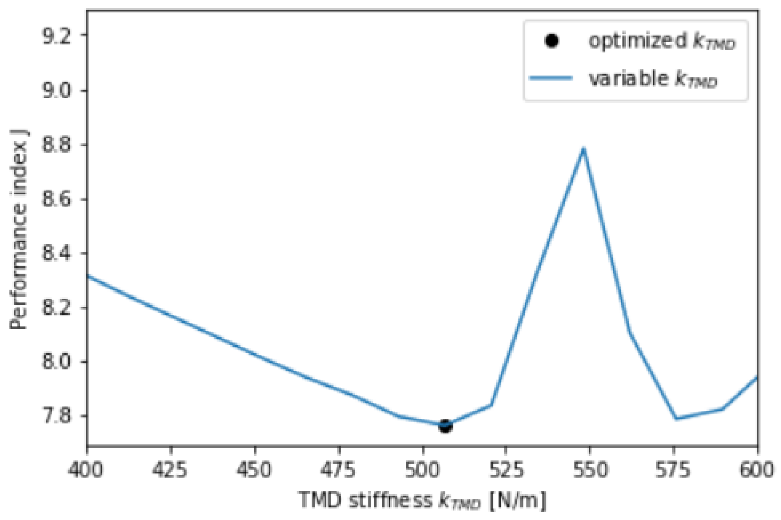

Figure 4.3: Performance index depending on the TMD stiffness constant $k_{TMD}[kN/m]$. Best performance index for $k_{TMD} = 503.81kN/m$

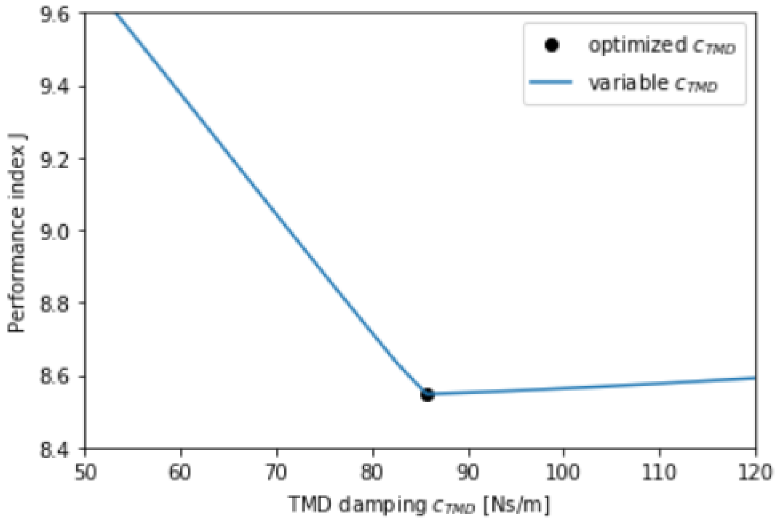

Figure 4.4: Performance index depending on the TMD damping parameter $c_{TMD}[kNs/m]$. Best performance index for $c_{TMD} = 87.13kNs/m$

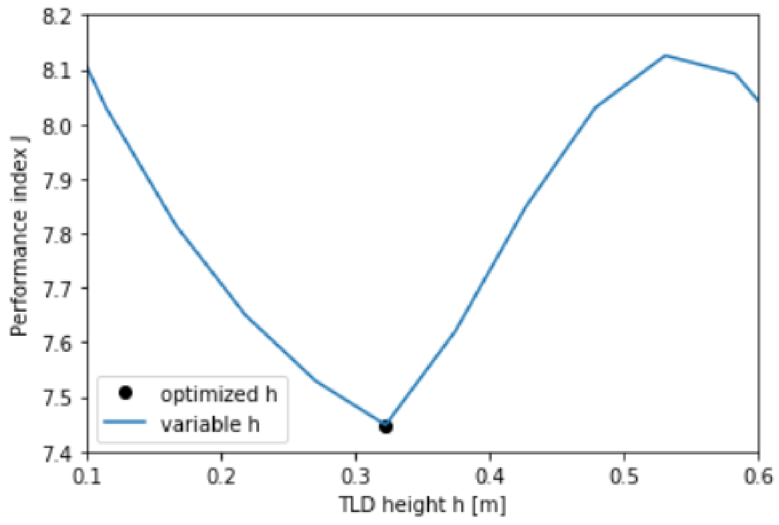

Figure 4.5: Performance index depending on the TLD liquid height h [m]. Best performance index for h = 0.313 m

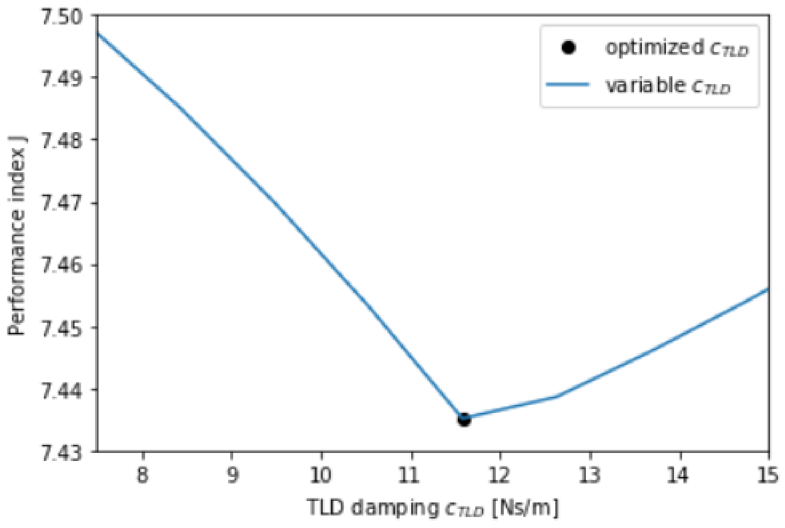

Figure 4.6: Performance index depending on the TLD equivalent damping parameter $c_{TLD}[kNs/m]$. Best performance index for $c_{TLD} = 11.85kNs/m$

Newton optimization

The algorithm for Newton optimization can be found in appendix B.7. As discussed in section 2.4, the Newton method can be used for optimization of a system by looking for gradients of the system of zero. In this case the function to analyse is the performance index $J(\mathbf{X})$, giving the system $\nabla J(\mathbf{X}) = \mathbf{0}$. Equation 2.109 then becomes:

$$\mathbf{X}_{k+1} = \mathbf{X}_k - H(\mathbf{X}_k)^{-1}\nabla\mathbf{J}(\mathbf{X}_k) \tag{4.3}$$

where: $\mathbf{X}_k = \begin{bmatrix} c_{TMD} & k_{TMD} & h & c_{TLD} \end{bmatrix}_k^T$ = Parameter vector at the kth step;
$H(x_k)^{-1}$ = Inverse Hessian at the kth step;
$\nabla\mathbf{J}(\mathbf{x}_k)$ = Performance index gradient at the kth step.

Similarly to the serial optimization, the initial parameters are the optimal damping parameters for a single degree of freedom system. The number of iterations to reach convergence is 25 in this case (figure 4.7), where the serial optimization used less than 10 iterations per parameter (figure 4.2). While the number of convergence steps for this optimization scheme is higher than the number of iterations of the individual parameters in the serial optimization scheme, the total amount of iterations is comparable as only one set of iterations needs to be performed.

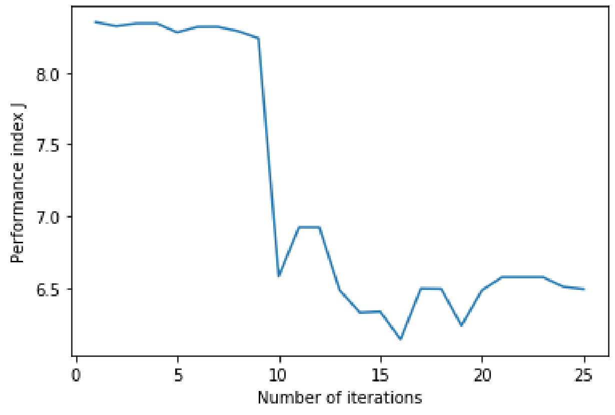

Figure 4.7: Convergence of the performance index J to its optimum using Newton optimization, depending on the parameters of the TMD's and TLD.

Figure 4.7 shows that the performance index per iteration step does not gradually go down, but fluctuates before reaching convergence when using the Newton method. It is evident that the critical point that is found is not the global minimum. Inspection of the Hessian of the system at convergence gives both positive and negative eigenvalues (λ = 6.78647680; 3.72361138e^{-09}; $-1.05920466e^{-09}$; $-9.77932808e^{-09}$), which means that the optimization process has led to a saddle point.

Moreover, while the Newton optimization gives a lower performance index than the serial optimization ($J = 6.49$), the given parameters are not all positive (see table 4.3.2). Thus, the resulting parameters can not be physically applied. Thus, henceforth the parameters from the serial optimization are used.

Optimized parameters	
k_{TMD}	517.55 kN/m
c_{TMD}	-366.60 kNs/m
h_{TLD}	0.942 m
c_{TLD}	12.78 kNs/m

Table 4.3: Optimized parameters for the damped system with separate TMD's and TLD using Newton optimization.

4.3.3 Frequency response spectra

The response using the starting parameters as seen in table 4.3.1 gives a performance index $J = 8.92$. The response spectra (figure 4.8) show that the response at both the heave eigenfrequency and the rotation eigenfrequency are blunted by the damping system. However, it has also created additional rotation response peaks. The TMD response is clearly connected to the heave response, while the TLD response is connected to the pitch response (figure 4.9).

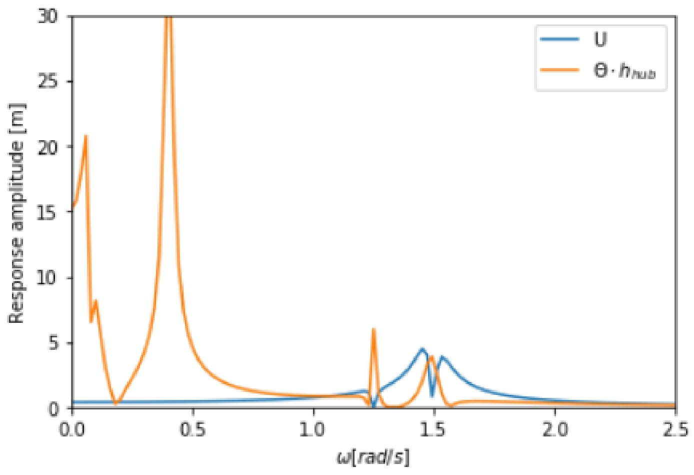

Figure 4.8: The heave U and rotation Θ response spectra of the independent damping system, using non-optimized damping parameters.

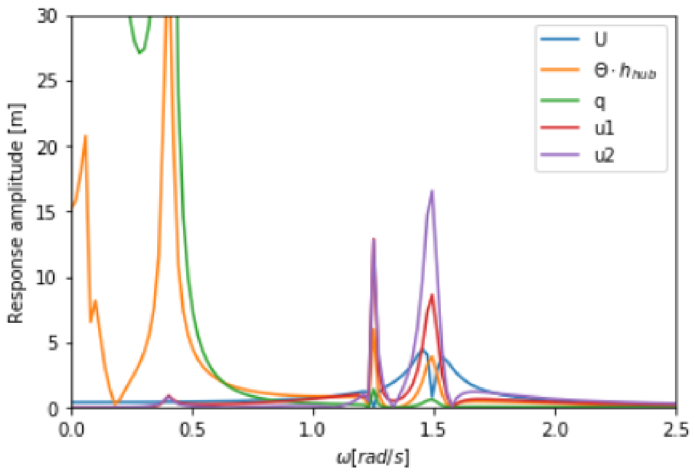

Figure 4.9: The response spectra of all degrees of freedom of the independent damping system, using non-optimized damping parameters.

The system with optimized parameters gives a response spectrum with similar eigenfrequencies (table 4.5), but smaller peaks, see figures 4.10, 4.11. The optimization gives a performance index J= 7.38.

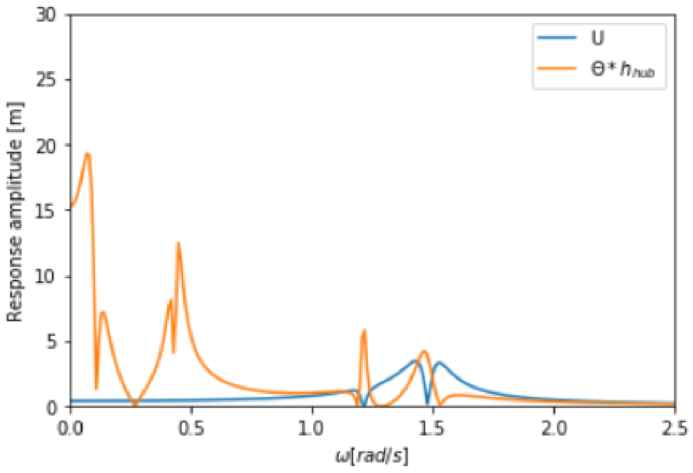

Figure 4.10: The heave U and rotation Θ response spectra of the independent damping system, using optimized damping parameters.

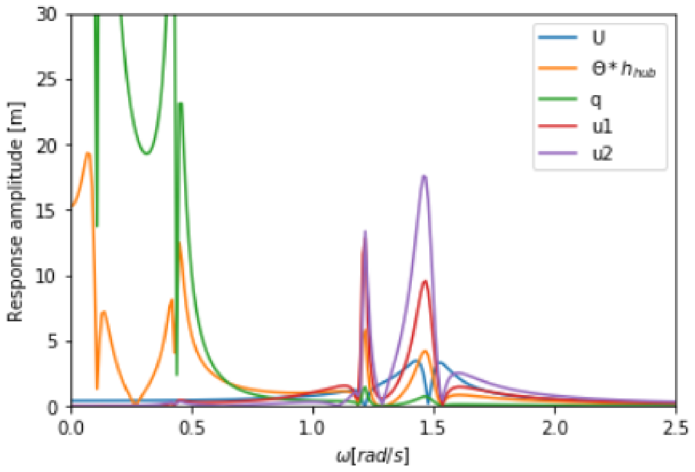

Figure 4.11: The response spectra of all degrees of freedom of the independent damping system, using optimized damping parameters.

A direct comparison (figures 4.12, 4.13) shows that most performance improvement comes from the pitch response.

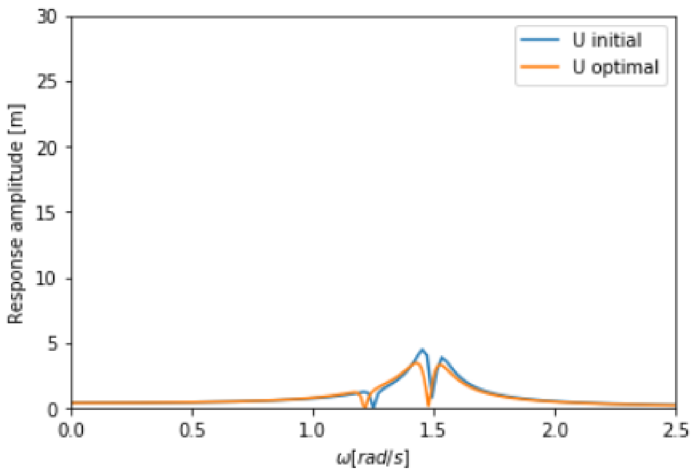

Figure 4.12: The heave U response spectra of the independently damped system with the initial parameters and the optimized parameters.

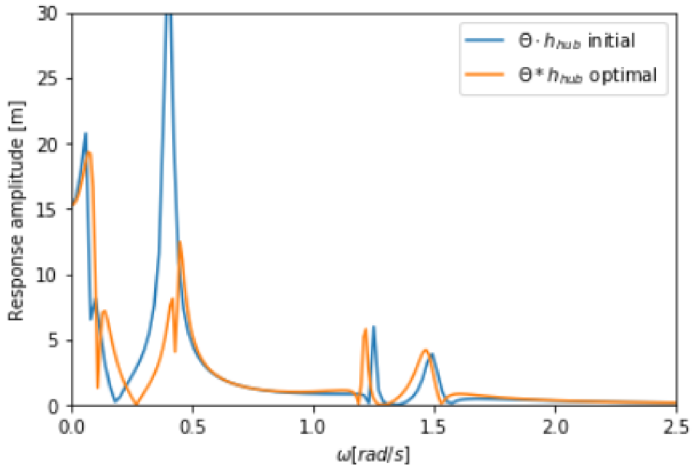

Figure 4.13: The pitch Θ response spectra of the independently damped system with the initial parameters and the optimized parameters.

The eigenshape of the TLD does not change when the liquid height does as its shape depends on the shape of its tank. However, its amplitude does change. See figure 4.14. The increased amplitude causes a higher pitch damping contribution.

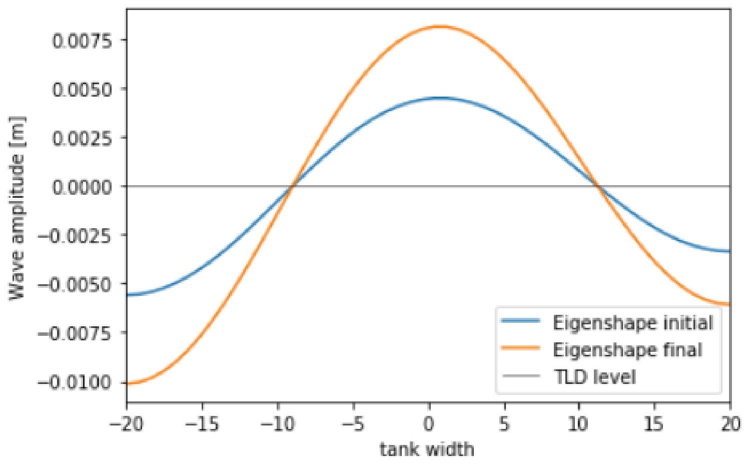

Figure 4.14: The eigenshape of the TLD of model 2.

4.4 Barge with interdependent TMD and TLD system

The model as described in paragraph 3.1.3 is analysed here.

4.4.1 Initial parameters

The optimized parameters (table 4.3.2) for the independent TMD-TLD damping system are used for the interdependent TMD-TLD damping system as well. See paragraphs 4.3.1 and 4.3.3 for the computation.

4.4.2 Iteration

In order to find the response spectrum of the system, the displacements of the barge system need to be found for a range of frequencies of applied unit force. The first guess for the displacement at any frequency will simply be the solution to the equations of motion with zero displacement for all the degrees of freedom. This solution will be different from the solution to the independent TLD-TMD system, since the TMD's are subjected to the pressure from the TLD. This pressure is uniform, as this first approximation will be gotten from the system at rest.

As discussed in chapter 2.5, the Newton method is used to find the response of the barge, the TLD and the TMD's. For the barge system, the system $\mathbf{G}(\mathbf{X}) = (-\omega^2(\boldsymbol{M}+\boldsymbol{A}) + i(\boldsymbol{C}+\boldsymbol{B}) + \boldsymbol{K} + \boldsymbol{D})\boldsymbol{X} - \hat{\mathbf{F}} = \mathbf{0}$ needs to be solved (see section 2.3.1). The following iteration scheme is used:

$$\mathbf{X}_{k+1} = \mathbf{X}_k - J(\mathbf{X}_k)^{-1}\mathbf{G}(\mathbf{X}_k) \tag{4.4}$$

Where $\mathbf{X}_k$	= response amplitude vector at the kth step;
$J(X_k)^{-1}$	= inverse Jacobian at the kth step;
$\mathbf{G}(\mathbf{X}_k)$	= system at the kth step

Using the finite difference method (equation 2.100) gives the Jacobian:

$$J_{i,j} = \frac{\boldsymbol{G}_i(X_1, X_2 \dots X_j + \epsilon \dots X_n) - \boldsymbol{G}_i(X_1, X_2 \dots X_j \dots X_n)}{\epsilon} \tag{4.5}$$

Where X_n is the displacement of the final TMD.

Since the Lagrangian and therefor the response of the system only changes depending on the displacements of the TMD's in the previous iteration but not on the displacements of the other degrees of freedom (see section 4.4 and appendix C.7), this gives:

$$J = \begin{bmatrix} 0 & 0 & 0 & \frac{\boldsymbol{G}_1(X_4+\epsilon ... X_n) - \boldsymbol{G}_1(X_4 ... X_n)}{\epsilon} & \dots & \frac{\boldsymbol{G}_1(X_4 ... X_p+\epsilon) - \boldsymbol{G}_1(X_4 ... X_n)}{\epsilon} \\ 0 & 0 & 0 & \frac{\boldsymbol{G}_2(X_4+\epsilon ... X_n) - \boldsymbol{G}_2(X_4 ... X_n)}{\epsilon} & \dots & \frac{\boldsymbol{G}_2(X_4 ... X_n+\epsilon) - \boldsymbol{G}_2(X_4 ... X_n)}{\epsilon} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \frac{\boldsymbol{G}_n(X_4+\epsilon ... X_n) - \boldsymbol{G}_n(X_4 ... X_n)}{\epsilon} & \dots & \frac{\boldsymbol{G}_n(X_4 ... X_n+\epsilon) - \boldsymbol{G}_n(X_4 ... X_n)}{\epsilon} \end{bmatrix} \tag{4.6}$$

The first iteration step $\mathbf{X}_1 = \mathbf{X}_0 - J(\mathbf{X}_0)^{-1}\mathbf{G}(\mathbf{X}_0)$ uses $\mathbf{X}_0 = \mathbf{H}(\omega, \mathbf{0})\hat{\mathbf{F}}$.

4.4.3 Frequency response spectra

The first iteration is based on a zero initial displacement at time $t = 1$ s. This gives $\mathbf{X}_0 = \mathbf{H}(\omega, \mathbf{0})\hat{\mathbf{F}}$ and the spectra in figures 4.15, 4.16. This iteration suggests a performance index J = 12.76. The eigenfrequencies of the TMD's and TLD lay closer to the pitch eigenfrequency than the heave frequency of the barge (see table 4.5).

As the first iteration is based on zero displacement of the TMD's, the eigenshape of the TLD is the same as that of the optimized model 2 (see figure 4.14).

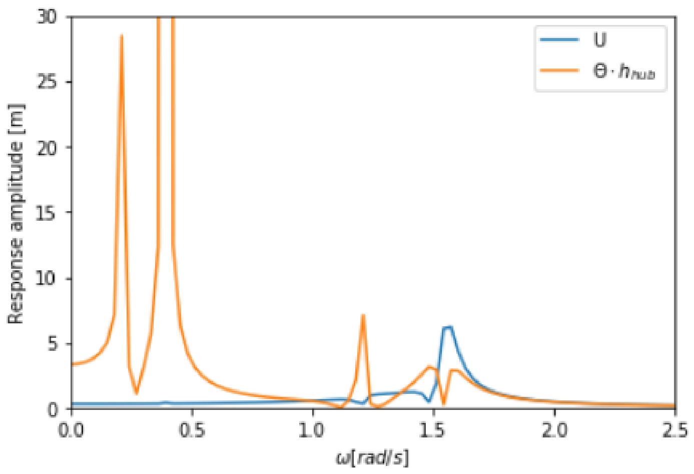

Figure 4.15: The heave U and pitch Θ response spectra of the interdependent damping system, first iteration (using non-optimized damping parameters).

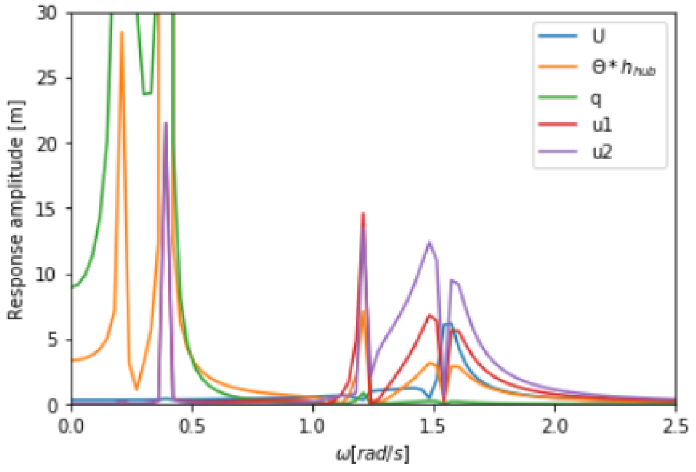

Figure 4.16: The response spectra of all degrees of freedom of the interdependent damping system, first iteration (using non-optimized damping parameters).

Response spectra

The response spectra of the interdependently damped barge with parameters optimized to the independently damped system give a performance index J= 12.85. The response spectra (figures 4.17 and 4.18) show a blunted response around heave resonance, but not around pitch resonance. The response of the TMDs is clearly linked to the pitch response (through the response q of the TLD) as well as to the heave response. The response of the TLD is related to the pitch response.

The eigenfrequencies of the system depend on the displacement of the system. Thus, they are not constant throughout the frequency response spectrum and are not conclusively determined.

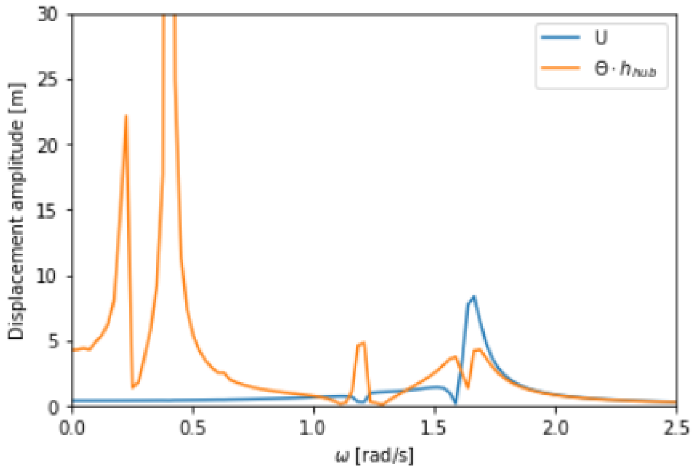

Figure 4.17: The heave U and pitch Θ response spectra of the interdependent damping system at t= 1 s, final iteration (using non-optimized damping parameters).

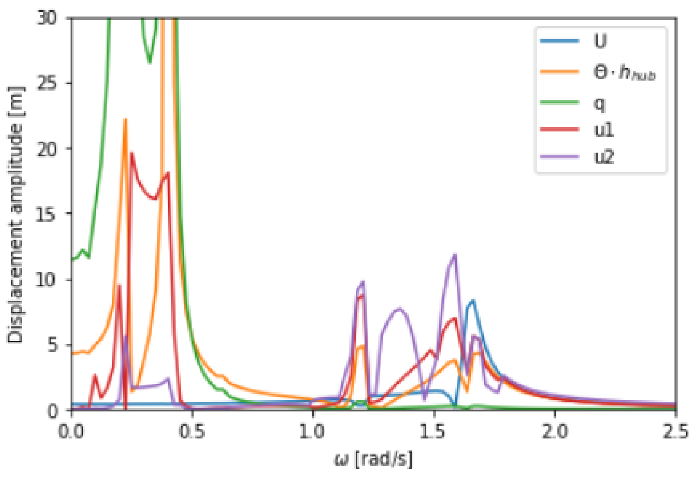

Figure 4.18: The response spectra of all degrees of freedom of the interdependent damping system at t= 1 s, final iteration (using non-optimized damping parameters).

The initial guess of the frequency response spectra is compared to the final iteration in figures 4.19 and 4.20. There is a clear shift in resonance frequencies, although the overall shape of the spectra is similar.

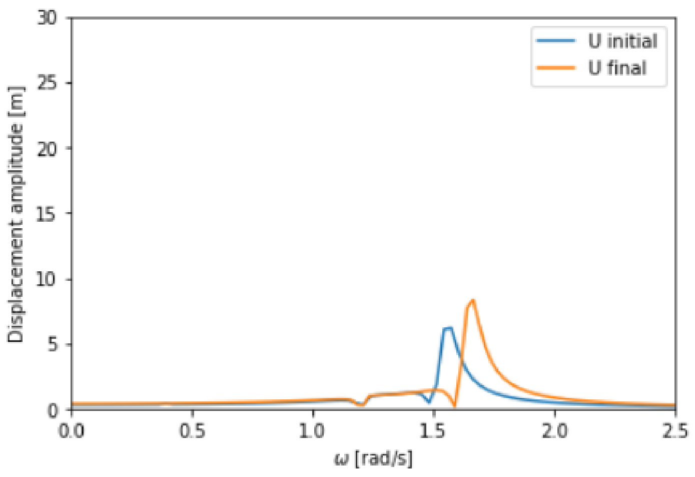

Figure 4.19: The heave U response spectra of the interdependently damped system. The initial guess of the spectrum compared to the fully iterated spectrum.

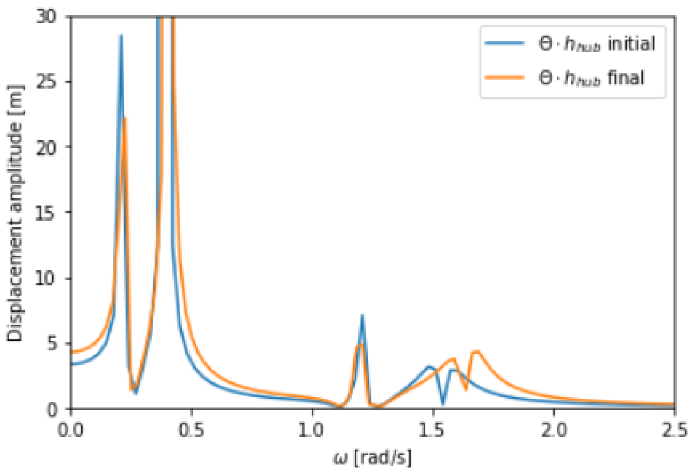

Figure 4.20: The pitch Θ response spectra of the interdependently damped system. The initial guess of the spectrum compared to the fully iterated spectrum.

Sensitivity

As explained in section 2.5.3, a rough sensitivity study is performed for each parameter at a time.

The sensitivity study for the TMD damping parameter (figure 4.21) shows a clear decrease of the performance index as the damping coefficient of the TMDs goes up. Its optimum is not in the range of the sensitivity study. The lowest performance index found in this range is J = 12.48.

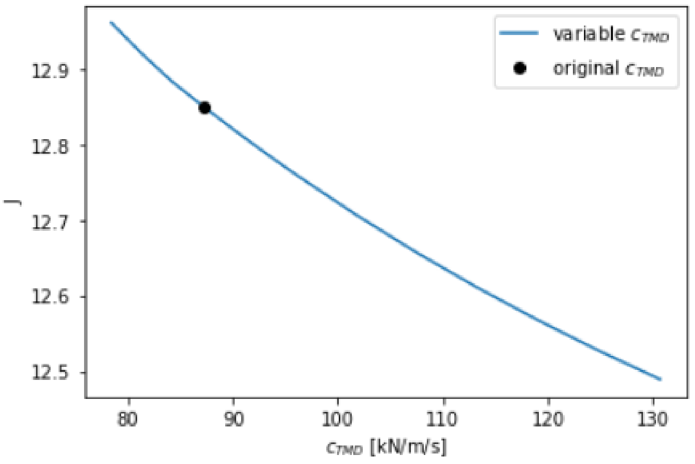

Figure 4.21: Performance index depending on the TMD damping parameter $c_{TMD}[kNs/m]$. No minimum of the performance index found.

The sensitivity study for the TMD stiffness parameter does include a minimum in its range, see figure 4.22. It is unknown if this is a local minimum or the global optimum. The minimum found gives a performance index J = 11.96.

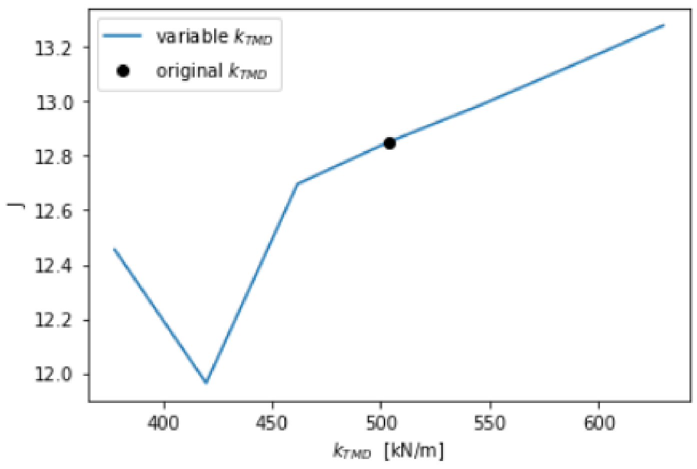

Figure 4.22: Performance index depending on the TMD stiffness parameter $k_{TMD}[kN/m]$, with a minimum J for $k_{TMD} = 419.84kN/m$.

The sensitivity study of TLD liquid height (figure 4.23) shows a clear peak with valleys. The current liquid height is clearly not the optimal height in combination with the other parameters. The minimum performance index found in the range of this analysis is J = 10.85.

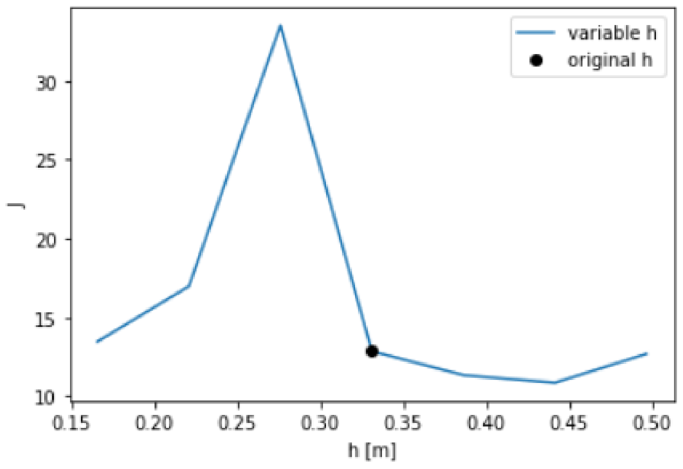

Figure 4.23: Performance index depending on the TLD liquid height h [m], with a minimum J for $h = 0.441m$.

Figure 4.24 shows low sensitivity of the TLD damping parameter within the range of the analysis. The performance index does not change significantly over this range and remains J = 12.85.

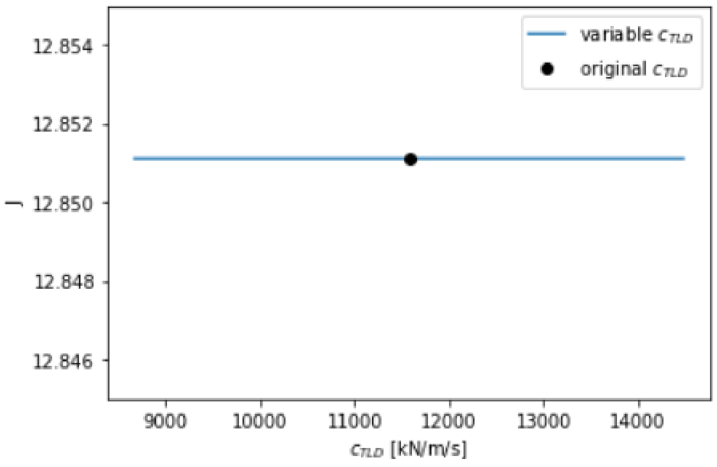

Figure 4.24: Performance index depending on the TLD damping parameter $c_{TLD}[kN/m/s]$, without a distinct minimum for J.

4.5 Comparison

In this section the three different systems are compared. The frequency response spectra, eigenfrequencies and the performance index J of each system is set side by side. Both damped systems show a damped heave response (figure 4.25). These models also both show additional pitch response peaks (figure 4.26). Figures 4.11 and 4.18 show that the TMDs of model 2 do not respond to pitch exitation, while the TMDs of model 3 do respond.

The eigenfrequencies of the models are shown in table 4.5. The final (iterated) version of model 3 does not have constant eigenfrequencies and are therefor not shown. Table 4.5 shows the performance index based on the total heave and pitch response.

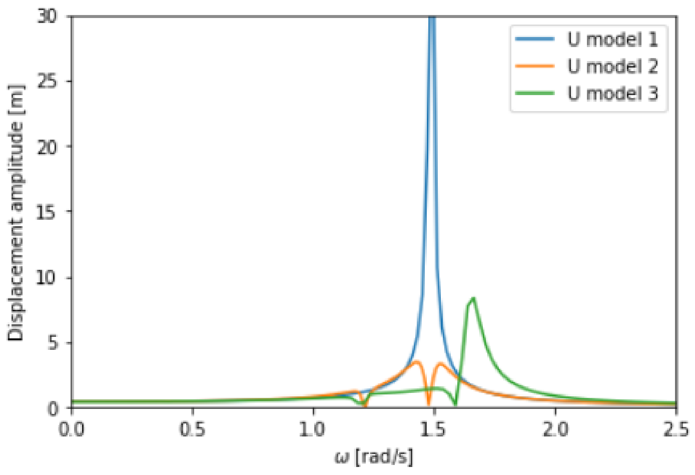

Figure 4.25: Frequency response spectra of the heave U for model 1, optimized model 2 and completely iterated model 3.

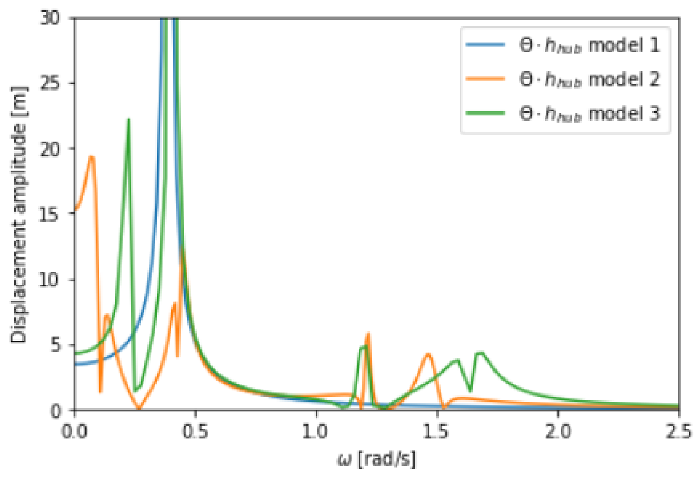

Figure 4.26: Frequency response spectra of the pitch Θ for model 1, optimized model 2 and completely iterated model 3.

	Eigenfrequencies [rad / s]				
Model 1	0.389	1.486	-	-	-
Model 2 (non-optimized)	0.108	0.443	1.412	1.736	1.771
Model 2 (optimized)	0.107	0.439	1.175	1.280	1.535
Model 3 - iteration 1 (non-optimized)	0.218	0.394	1.176	1.280	1.585
Model 3	N/A	N/A	N/A	N/A	N/A

Table 4.4: Eigenfrequencies of each system

	Performance index J
Model 1	12.32
Model 2 (non-optimized)	8.92
Model 2 (optimized)	7.38
Model 3 - iteration 1 (non-optimized)	12.76
Model 3 - final iteration (non-optimized)	12.85

Table 4.5: The total performance index for each system

5 | Discussion and conclusion

5.1 Discussion results

5.1.1 Program development

As stated in section 1.4, one of the objectives of this project was the development of adjustable programs to determine the frequency response of the three different barge systems as described in chapter 3.3.3.

The analyses of the models show convincing frequency response functions. The two models of the damped systems show blunted responses as expected. This effect is lacking for the pitch response of the interdependently damped model, but this does not seem to be caused by divergence during the iteration due to the nonlinear effects, as this is also the case for the first iteration. The peaks of the pitch response is still in accordance with the completely undamped model. The nonlinearity problem for the interdependent damping system has been resolved as the interdependently damped model reaches convergence for all frequencies. Judging on the analysis performed in this thesis, it looks like the modeling programs are successful.

The optimization of the models is also successful, if accounted for the limitations as described in section 5.2. The fast convergence to the respective optima of the parameters (see figure 4.2) shows the advantage of running the serialized optimization as opposed to determining the performance for a range of parameters.

5.1.2 Effectiveness interdependent damping system

The second aim of this research is the investigation of the effectiveness of an interdependent damping system as described in section 3.1.3 as compared to an independent damping system (section 3.1.2) and an undamped system (section 3.1.1).

From this research can not be deduced decisively whether or not the interdependent damping system is more effective than the independent damping system. The comparison of the two damping systems can only happen effectively if both systems are tuned optimally. As the interdependently damped model lacks optimization, any direct comparison falls short.

However, the sensitivity studies of the interdependently damped model give some insight into the potential of the damping system. While the performance index of this system is higher than that of both the undamped and the independently damped model, it is clear from the sensitivity studies that the interdependently damped model leaves a lot on the table. The study of the TLD liquid height h (figure 4.5) shows that even optimization of this parameter alone gives a significantly lower performance index ($\Delta J = 2.00$, which is a 15.6 % decrease of the performance index). In fact, there is considerable possibility for improvement with regards to the TMD damping and stiffness parameters as well.

Figure 4.13 shows the effect optimization has on the pitch response of the independently damped model. Its response dramatically decreases after optimization. As the pitch response peaks of the interdependently damped model are sharp and look mostly undamped (figure 4.20), a similar reaction to parameter optimization would result in a significantly lower performance index for the interdependently damped model as well.

5.2 Research limitations

The set-up of the models operates under certain limitations, which have to be taken into account when assessing the accuracy of the results.

First of all, the modeling of fluid-structure interaction is simplified in multiple ways. By applying linear potential flow theory, the flow of the TLD and the surrounding ocean water is assumed to behave irrotationally, not taking into account the nonlinear effects from turbulence. Any nonlinear effects due to fluid separation from the structure is disregarded as well. The compressibility of fluid is neglected likewise.

The nonlinear terms are also omitted from the additional hydraulic mass, damping and stiffness matrices. These matrices have furthermore been determined based on free vibration so that the hydrodynamic terms include only the movement of the barge itself, not that of the surrounding fluid.

Further, the degrees of freedom are limited. The horizontal motion of the barge is disregarded and the structure is considered to be ridged so that local deformations in the structure are neglected. The analysis of the models is only determined for the first eigenmode of the TLD and with merely two TMDs to aid simplicity. A higher number of TLD eigenmodes gives more accurate results, but is also more computationally demanding. Since only the first TLD mode is tuned to the pitch frequency of the barge, if the TLD is modeled with multiple modes, the TLD sloshing movement likely does not fully contribute to the pitch damping anymore. This decreases the damping contribution of the TLD. The models are programmed in such a way that the number of modes and the number of TMDs is adjustable.

Lastly, constraints on the barge such as mooring cables are not included in the model. Mooring lines have been shown to increase platform stability, but their effect is difficult to accurately predict [17], [32]. There are furthermore no height constraints on the displacement of the TMDs and the TLD.

Apart from the limitations of the models, the research itself has deficiencies. In particular, the lack of optimization for the interdependently damped model is an issue as it prohibits the one on one comparison between the three models. While the sensitivity analysis for this model gives some insight into its damping potential, it does not give a definitive answer about the effectiveness of the damping system.

Also, the optimization of the independently damped model does not necessarily lead to the best parameter combination possible. From the serialized and the Newton optimization only the serialized optimization gives usable results. The individual parameters are optimized while keeping the rest of the parameters at their original values. Changing any of these parameters would result in a different optimization of the other parameters. Thus, going through the same optimization process using different parameters as a starting point would likely give a different outcome. As it is, the values found with the serial optimization process are an acceptable estimation.

5.3 Recommendations

5.3.1 Improvement program and research

First and foremost, improvement of the optimization is useful for a better damping result, for both damped models. Not only is the current optimization process not suitable to the interdependently damped model, it also does not necessarily give the best overall parameters for the systems. The effectiveness of each parameter depends on the other parameters, which is why the optimization in this study is done one parameter at a time. This also implies multiple possible parameter compositions. It is worth investigating whether optimizing all parameters simultaneously followed by iteration of this process leads to a lower performance index. It is not likely that fixed point iteration would lead to convergence of the optimization however, a more intricate iteration scheme might be necessary. While the Newton method as described in chapter 2.5 seems a more prudent scheme for a multivariate optimization like this, it does not give the desired results. Since the parameter range can not be controlled with this method, some of the optimized parameters are outside of the range of physical possibility. Apart from that, it can not be guaranteed that the critical point the iterations converge to is a minimum and not to either a maximum or a saddle point. In the case of the independently damped system the optimization converges to a saddle point and furthermore gives a negative damping parameter of the TMDs and a TLD liquid height that would lead to a TLD mass bigger than the system mass (see section 4.3.2 and table 4.3.2).

Also, the nonlinearity of the third system poses the problem of a time dependent frequency response. The potentially significant time-dependence of the eigenfrequencies and eigenmodes needs to be analysed as well. If their time-dependence is indeed significant, this could affect the response of the system either negatively or positively.

However, this might not be the case. The nonlinearity of the system shows itself in a shift of the resonance frequencies (figures 4.19, 4.20) between the first and final iteration of the system. As this is the nonlinearity is biggest at the largest response amplitude, the time dependent displacement of the TLD is not expected to be large enough to significantly increase nonlinearity further. However, the spectrum is likely to deform more when a larger force amplitude is applied to the system since this increases the displacements.

Increasing the degrees of freedom of the models can give a better understanding of the behaviour of the physical system. Placing the models in a 3D space and giving up the rigid body assumption strongly increases their complexity.

Adding local deformations to the model causes some energy dissipation. More importantly it causes additional eigenfrequencies and eigenmodes. If the energy of the system is distributed over more eigenfrequencies, damping of the base frequency could be less effective. If this is the case it could be that increasing the number of TMDs and varying their mass is beneficial to the damping performance of the system.

Including nonlinear effects of the fluid-structure interaction increases both accuracy and complexity of the models as well. The shallowness of the TLD causes a large portion of the liquid to contribute to wave forming. The bigger the contribution of the waves, the higher the contribution of nonlinearity to the energy dissipation of the TLD. The breaking of waves is the primary cause of energy dissipation in sloshing [25], [36]. When using nonlinear flow, vorticity is also taken into account. Vortex shedding and turbulence cause additional energy dissipation as well. This means that a nonlinear model would show more damping at high response amplitudes. However, since the damping constant for the TLD in the current model is used as an adaptable design variable, this does not change the outcome much. While the equivalent damping parameter is not constant anymore, its magnitude is most relevant during pitch frequency and the damping parameter can be designed as such. The damping constant can be increased with the use of damping screens, but can be lowered only by changing the properties of the damping fluid.

When using nonlinear flow theory the eigenmodes of the TLD can not be superimposed anymore.

5.3.2 Improvement damping system

The damping systems of the two damped models can be further improved by experimenting with a different number of tuned mass dampers. The floating structure vibrates at different modes. It might therefore be beneficial to use TMDs with varying parameters throughout the platform. It remains to be seen whether tuning TMDs to the system's secondary eigenmodes is effective, or takes too much damping potential away from the primary eigenmode.

Having different TMDs designated for different degrees of freedom could also make a difference. While the TMDs toward the centre of the platform are best used for heave motion, the outer TMDs could be useful for combating pitch.

5.3.3 Nonlinear analysis approach

This research used modal analysis in the frequency spectrum with Newton method iteration to analyse the nonlinear model. This method is computationally heavy and could either be modified or replaced by other analysis methods for faster computation.

Some possible alternative analysis methods are described here.

Coupling of linear and nonlinear substructures is described in various literature [21], [5], [11], [24], [23]. Here, pre-modeled substructures are combined using a coupling matrix. The linear and nonlinear parts of the system can be modeled separately, which allows for cheaper computations.

Similarly, the describing function method where a nonlinear system is described as a linear system with an additional nonlinearity matrix [31], [38], [21] allows for a largely linear computation.

Finally, nonlinear systems can be described most accurately in the time domain using Runge-Kutta iteration [42], [4]. This method does have stability concerns as the initial displacement guess gets farther from the final iteration [21].

5.3.4 Further research

The efficiency of the damping systems depends on the applied load to the barge-turbine system. Looking at specific locations gives the relevant wave states and wind loads necessary to determine the performance index of either system. Further research into the stochastic loading on the barge-turbine system is needed to validate the interdependent damping system in specific design situations.

Further, the effect of damping of the barge movement on the lifetime and serviceability of the turbine needs to be established to verify the concrete usefulness of the system. Any damping of the hull increases efficiency of the turbine. Since a rigid body is assumed, damping of the barge also damps the movement of the hull, this is not necessarily the case when using a flexible tower. Disproportionate damping of the barge with respect to the turbine could increase tension in the tower, decreasing its lifespan. This needs to be examined.

5.4 Conclusion

5.4.1 Program development

The program for all three models is based on linear modal analysis, which is an already established analytic theory. The interdependently damped model is further developed using Newton iteration to account for nonlinearity. Since the frequency response spectra of the nonlinear system found by the program are reasonable and match the shape of the linear systems, it can be concluded that the program is successful.

5.4.2 Effectiveness interdependent damping system

Lack of optimization for the parameters of the interdependently damped model prevents the direct comparison of the efficacy of the two damped systems.

The independently damped system proves to be effective, decreasing the performance index significantly compared to the undamped system.

Optimizing the TLD liquid height alone gives the interdependently damped system an advantage over the undamped system. Whether complete optimization improves the response enough to edge out the independently damped system can not be inferred from the current results.

A | Appendix model 1

A.1 Input

```
import numpy as np
import scipy
import sympy

A= np.zeros((2,2))  # added hydraulic mass
B= np.zeros((2,2))  # added hydraulic damping
C= np.zeros((2,2))  # added hydraulic stiffness

a= 20        # half of barge width
L= a*2
b= 2 * a     # barge depth
rho= 1000    # water density
g= 9.81
h_hub= 87.74 # hub height

M_b= 6149460                  # Mass of system
H= M_b / (rho * b * 2 * a)  # Floating depth
k_m= rho * g                  # Hydrostatic stiffness
c_w= 0                        # Hydrodynamic damping

A[0,0]= 9.556 * 10**5
A[1,1]= 2.187 * 10**8

B[0,0]= 2/(3 * np.pi)* ( np.pi - 1) * b * rho * H * (2 * np.pi *( 1/2 *
    np.pi - 1) / (2 * a))**(1/3)

C[0,0]= 2 * rho * g * b * a
C[1,1]= rho * g * b * a**2
```

A.2 Lagrangian

```
def Lagrangian(rho, a, M, g):
    X= Symbol('X')
    T= Symbol('T')
    U= sympy.Function('U')        # Heave
    phi= sympy.Function('phi')    # Pitch

    vert= U(T) + phi(T) * X
    # Kinetic energy without tmds
    K= M / 2 * (U(T).diff(T)**2 + 1/3 * a**2 * phi(T).diff(T)**2)

    # Potential energy without tmds
    P= M * g * U(T)

    L= K - P
    return L
```

A.3 Response

```
omega= 1
Load= np.array([M_b, M_b])  # Excitation

def displacement(omega, M, a, A, B, C, Load):
    X= Symbol('X')
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')

    L= Lagrangian(rho, a, M, g)

    Lagrange_phi= L.diff(Derivative(phi(T), T)).diff(T) - L.diff(phi(T)
        )
    Lagrange_U= L.diff(Derivative(U(T), T)).diff(T) - L.diff(U(T))

    Mass_matrix= np.zeros(((2), (2)))

    Mass_matrix[0, 0:2] = [Lagrange_U.coeff(Derivative(U(T), (T,2))),
        Lagrange_U.coeff(Derivative(phi(T), (T,2)))]
    Mass_matrix[1, 0:2] = [Lagrange_phi.coeff(Derivative(U(T), (T,2))),
         Lagrange_phi.coeff(Derivative(phi(T), (T,2)))]

    Stiff_matrix= np.zeros(((2), (2)))

    Stiff_matrix[0, 0:2] = [Lagrange_U.coeff(U(T)), Lagrange_U.coeff(
        phi(T))]
    Stiff_matrix[1, 0:2] = [Lagrange_phi.coeff(U(T)), Lagrange_phi.
        coeff(phi(T))]

    Force_vec= -np.array([sympy.Poly(Lagrange_U).TC(), sympy.Poly(
        Lagrange_phi).TC()])

    Eigenmatrix= np.linalg.inv(-omega**2 * (Mass_matrix + A) + 1j *
        omega * B + Stiff_matrix + C)

    disp= Eigenmatrix @ Load

    return disp.real + disp.imag
```

B | Appendix model 2

B.1 Input

```
import numpy as np
import scipy
import sympy
from scipy import integrate
from sympy import Symbol, Poly, diff, Derivative
from sympy.utilities.lambdify import lambdify
from sympy.abc import t, x, z, T, X, Z
from copy import copy
import json
from sympy.parsing.sympy_parser import parse_expr

# In case of a re-run
try:
    f= open("model2.json", "r+")
    data= json.load(f)
    f.close()

except FileNotFoundError:
    # Starting without tld:
    data= {"h": 0, "c_tld": 0}

    jdata = json.dumps(data, indent=4)
    f= open("model2_3.json", "w")
    f.write(jdata)
    f.close()

c_tld= data['c_tld']      # Equivalent damping coefficient (TLD)
h= data['h']              # Liquid height in rest (TLD)

m= 1                  # Number of modes
a= 20                 # Half of tank width
b= 2 * a              # Tank depth
rho= 1000             # Water density
g= 9.81               # Gravitational constant
h_hub= 87.74          # Hub height
n= 2                  # Number of TMDs
M_total= 6149460      # System mass

def M_structure(h): # Mass barge + turbine (without TLD and TMD's)
    return M_total * .9 - (rho * 2 * a * b * h)
```

```
M = M_structure(h)
M_tmd= np.ones(n) * 0.1 * M_total / n # Mass of the TMDs

H= M_total / (rho * b * 2 * a) # Floating depth

A= np.zeros((n+3, n + 3))  # Hydraulic mass matrix
A[0,0]= 9.556 * 10**5
A[1,1]= 2.187 * 10**8

B= np.zeros((n+3, n + 3))  # Hydraulic damping matrix
B[0, 0]= 2/(3 * np.pi)* ( np.pi - 1) * b * rho * H * (2 * np.pi *( 1/2
    * np.pi - 1) / (2 * a))**(1/3)

D= np.zeros((n+3, n + 3))  # Hydraulic stiffness matrix
D[0,0]= 2 * rho * g * b * a
D[1,1]= rho * g * b * a**2

# SDoF optimal parameters:
m_ratio= 1/8                  # Mass ratio of total TMD/structure
f= np.sqrt((1 + m_ratio/2) / (1 + m_ratio)) # Frequency ratio TMD/
    structure
eps= np.sqrt((m_ratio * (1 + 3/4 * m_ratio)) / (4 *  (1 + m_ratio) * (1
     + m_ratio / 2)))                        # Damping ratio
om_s= np.sqrt(D[0,0]/(M_total * .8))     # Heave eigenfrequency
om_tmd= f * om_s                         # TMD eigenfrequency
k_tmd= om_tmd ** 2 * M_tmd               # TMD stiffness
c_tmd= 2 * M_tmd * om_tmd * eps          # TMD damping coefficients
```

B.2 TLD sloshing

For the definitions of these functions, see section 2.1.3

```
def alpha(a, m):
    alpha= np.zeros(m)
    for i in range(m):
        alpha[i] = (2 * (i+1) - 1) * np.pi / (2 * a)
    return alpha

def beta(a, m):
    beta= np.zeros(m)
    for i in range(m):
        beta[i] = (i + 1) * np.pi / a
    return beta

def B_A(a, h, m):
    a_A= np.zeros(m)
    b_A= np.zeros(m)
    c_A= np.zeros(m)
    B_A= np.zeros(m)
    omega2_A= np.zeros(m)
    for i in range(m):
        a_A[i] = integrate.dblquad(lambda x, z: np.sin(alpha(a, m)[i] *
            x)**2 + np.sinh(alpha(a, m)[i] * z)**2, -h, 0, lambda x: -a
           , lambda x: a)[0]
        b_A[i] = 2 * integrate.dblquad(lambda x, z: np.sinh(alpha(a, m)
           [i] * z) * np.cosh(alpha(a, m)[i] * z), -h, 0, lambda x: -a,
            lambda x: a)[0] - a / alpha(a, m)[i]
        c_A[i] = integrate.dblquad(lambda x, z: np.cos(alpha(a, m)[i] *
            x)**2 + np.sinh(alpha(a, m)[i] * z)**2, -h, 0, lambda x: -a
           , lambda x: a)[0]

        B_A[i] = (-b_A[i] - np.sqrt(b_A[i]**2 - 4 * a_A[i] * c_A[i]))/
           (2 * a_A[i])
        omega2_A[i] = B_A[i] * alpha(a, m)[i] * g
    return B_A

def B_S(a, h, m):
    a_S= np.zeros(m)
    b_S= np.zeros(m)
    c_S= np.zeros(m)
    B_S= np.zeros(m)
    omega2_S= np.zeros(m)

    for i in range(m):
        a_S[i] = integrate.dblquad(lambda x, z: np.cos(beta(a, m)[i] *
           x)**2 + np.sinh(beta(a, m)[i] * z)**2, -h, 0, lambda x: -a,
           lambda x: a)[0]
        b_S[i] = 2 * integrate.dblquad(lambda x, z: np.sinh(beta(a, m)[
           i] * z) * np.cosh(beta(a, m)[i] * z), -h, 0, lambda x: -a,
           lambda x: a)[0] - a / beta(a, m)[i]
        c_S[i] = integrate.dblquad(lambda x, z: np.sin(beta(a, m)[i] *
           x)**2 + np.sinh(beta(a, m)[i] * z)**2, -h, 0, lambda x: -a,
           lambda x: a)[0]

        B_S[i] = (-b_S[i] - np.sqrt(b_S[i]**2 - 4 * a_S[i] * c_S[i]))/
           (2 * a_S[i])
        omega2_S[i] = B_S[i] * beta(a, m)[i] * g
    return B_S


def Omega_A(alpha, B_A, g, m):
    omega2_A= np.zeros(m, dtype= 'complex_')
    for i in range(m):
        omega2_A[i] = B_A[i] * alpha[i] * g
    return np.sqrt(omega2_A)

def Omega_S(beta, B_S, g, m):
    omega2_S= np.zeros(m, dtype= 'complex_')
    for i in range(m):
        omega2_S[i] = B_S[i] * beta[i] * g
    return np.sqrt(omega2_S)
```

```
# Numerical assymetric mode function
def mode_func_nA(x, z, alpha, B_A):
    num= (np.sin(alpha * x) * (np.cosh(alpha * z) + B_A * np.sinh(alpha
        * z)))
    return num

# Symbolic assymetric mode function
def mode_func_sA(alpha, B_A):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    sym= (sympy.sin(alpha * X) * (sympy.cosh(alpha * Z) + B_A * sympy.
       sinh(alpha * Z)))
    return sym

def mode_func_nS(x, z, beta, B_S):
    num= (np.cos(beta * x) * (np.cosh(beta * z) + B_S * np.sinh(beta *
       z)))
    return num

def mode_func_sS(beta, B_S):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    sym= (sympy.cos(beta * X) * (sympy.cosh(beta * Z) + B_S * sympy.
       sinh(beta * Z)))
    return sym

def potential_nA(x, z, t, omega, alpha, B_A):
    nump= mode_func_nA(x, z, alpha, B_A) * (1j * omega * np.exp(1j *
       omega * t)).real
    return nump

def potential_sA(alpha, B_A):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    q= sympy.Function('q')
    symp= mode_func_sA(alpha, B_A) * Derivative(q(T), T)
    return symp

def potential_nS(x, z, t, omega, beta, B_S):
    nump= mode_func_nS(x, z, beta, B_S) * (1j * omega * np.exp(1j *
       omega * t)).real
    return nump

def potential_sS(beta, B_S):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    q= sympy.Function('q')
    symp= mode_func_sS(beta, B_S) * Derivative(q(T), T)
    return symp

def wave_shape_sA(alpha, B_A):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    p= potential_sA(alpha, B_A)
    dp_dz= p.diff(Z)
    shape= sympy.integrate(dp_dz, T)
    shape= shape.subs(Z, 0)
    return shape

def wave_shape_nA(x, t, omega, alpha, B_A):
    z= Symbol('z')
    X= Symbol('X')
    T= Symbol('T')
    potential= 1j * omega * sympy.exp(1j * omega * T) * (sympy.sin(
       alpha * X) * (sympy.cosh(alpha * z) + B_A * sympy.sinh(alpha * z
       )))
    df_dz= potential.diff(z)
    f= sympy.re(sympy.integrate(df_dz, T))
    p= lambdify([z, X, T], f, 'numpy')
    return p(0, x, t)

def wave_shape_sS(beta, B_S):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    p= potential_sS(beta, B_S)
    dp_dz= p.diff(Z)
    shape= sympy.integrate(dp_dz, T)
    shape= shape.subs(Z, 0)
    return shape

def wave_shape_nS(x, t, omega, beta, B_S):
    z= Symbol('z')
    X= Symbol('X')
    T= Symbol('T')
    potential= 1j * omega * sympy.exp(1j * omega * T) * (sympy.cos(beta
        * X) * (sympy.cosh(beta * z) + B_S * sympy.sinh(beta * z)))
    df_dz= potential.diff(z)
    f= sympy.re(sympy.integrate(df_dz, T))
    p= lambdify([z, X, T], f, 'numpy')
    return p(0, x, t)

def dpotential_dx(a, h, m):
    X= Symbol('X')
    dpot_dx = 0
    for i in range(m):
        dpot_dx += (potential_sA(alpha(a, m)[i], B_A(a, h, m)[i])).diff
            (X) + (potential_sS(beta(a, m)[i], B_S(a, h, m)[i])).diff(X)
    return dpot_dx
```

```
def dpotential_dz(a, h, m):
    Z= Symbol('Z')
    dpot_dz = 0
    for i in range(m):
        dpot_dz += (potential_sA(alpha(a, m)[i], B_A(a, h, m)[i])).diff
            (Z) + (potential_sS(beta(a, m)[i], B_S(a, h, m)[i])).diff(Z)
    return dpot_dz
```

B.3 Equation of motion

```
def Lagrangian(rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx, dpot_dz):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    U= sympy.Function('U')                              # DoF heave
    phi= sympy.Function('phi')                          # DoF pitch
    q= sympy.Function('q')                              # DoF TLD wave
        amplitude
    u= [sympy.Function('u%d' % i) for i in range(n)] # DoF TMD

    vert= U(T) + phi(T) * X

    wave= 0
    for i in range(m):
        wave+= wave_shape_sA(alpha(a, m)[i], B_A(a, h, m)[i]) +
            wave_shape_sS(beta(a, m)[i], B_S(a, h, m)[i]) + h

    # Kinetic energy without tmds
    K= sympy.expand(1/2 * rho * b * sympy.integrate(sympy.integrate((
        dpot_dx) ** 2 + (vert.diff(T) + dpot_dz) ** 2, (Z, -h, 0)), (X,
        -a, a))) + M / 2 * (U(T).diff(T)**2 + 1/3 * a**2 * phi(T).diff(T
        )**2)

    # Potential energy without tmds
    P= M * g * U(T)
    for i in range(m):
        P += sympy.expand(g * rho * b * sympy.integrate((1/2 * wave +
            vert) * wave, (X, -a, a)))

    # Additional energy tmds:
    for i in range(n):
        K += sympy.expand(1/2 * M_tmd[i] * (u[i](T).diff(T) + U(T).diff
            (T) + phi(T).diff(T) * (- a + (2 * a * i) / (n - 1)))**2)

        P += sympy.expand(1/2 * k_tmd[i] * u[i](T)**2 + M_tmd[i] * g *
            (u[i](T) + U(T) + phi(T) * (- a + (2 * a * i) / (n - 1))))

    L= K - P
    return L
```

```
def Reighley(a, h, c_tmd, c_tld, n, dpot_dx, dpot_dz, m):
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')
    q= sympy.Function('q')
    u= [sympy.Function('u%d' % i) for i in range(n)]

    wave= 0
    for i in range(m):
        wave+= wave_shape_sA(alpha(a, m)[i], B_A(a, h, m)[i]) +
            wave_shape_sS(beta(a, m)[i], B_S(a, h, m)[i]) + h
    # Reighley dissipation without tmds
    R= 1/ 2 * c_tld * b * sympy.integrate(wave.diff(T)**2, (X, -a, a))

    # Additional energy tmds:
    for i in range(n):
        R += sympy.expand(1/2 * c_tmd[i] * (u[i](T).diff(T))**2)

    return R

def Mass_matrix(Lagrange, Reighley, n):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')
    q= sympy.Function('q')
    u= [sympy.Function('u%d' % i) for i in range(n)]

    L= Lagrange
    R= Reighley

    Lagrange_phi= L.diff(Derivative(phi(T), T)).diff(T) - L.diff(phi(T)
        ) + R.diff(Derivative(phi(T), T))
    Lagrange_U= L.diff(Derivative(U(T), T)).diff(T) - L.diff(U(T)) + R.
        diff(Derivative(U(T), T))
    Lagrange_q= L.diff(Derivative(q(T), T)).diff(T) - L.diff(q(T)) + R.
        diff(Derivative(q(T), T))
    Lagrange_u= [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i]= L.diff(Derivative(u[i](T), T)).diff(T) - L.diff(
            u[i](T)) + R.diff(Derivative(u[i](T), T))

    Mass_matrix= np.zeros(((n + 3), (n + 3)))

    Mass_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), (T,2))),
        Lagrange_U.coeff(Derivative(phi(T), (T,2))), Lagrange_U.coeff(
        Derivative(q(T), (T,2)))]
    Mass_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), (T,2))),
         Lagrange_phi.coeff(Derivative(phi(T), (T,2))), Lagrange_phi.
        coeff(Derivative(q(T), (T,2)))]
    Mass_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), (T,2))),
       Lagrange_q.coeff(Derivative(phi(T), (T,2))), Lagrange_q.coeff(
       Derivative(q(T), (T,2)))]

    for i in range(n):
        Mass_matrix[0, 3 : i + 3]= Lagrange_U.coeff(Derivative(u[i](T),
            (T, 2)))
        Mass_matrix[1, 3 : i + 3]= Lagrange_phi.coeff(Derivative(u[i](T
           ), (T, 2)))
        Mass_matrix[2, 3 : i + 3]= Lagrange_q.coeff(Derivative(u[i](T),
            (T, 2)))

        Mass_matrix[i + 3, 0]= Lagrange_u[i].coeff(Derivative(U(T), (T,
            2)))
        Mass_matrix[i + 3, 1]= Lagrange_u[i].coeff(Derivative(phi(T), (
           T, 2)))
        Mass_matrix[i + 3, 2]= Lagrange_u[i].coeff(Derivative(q(T), (T,
            2)))
        for j in range(n):
            Mass_matrix[i + 3, j + 3]= Lagrange_u[i].coeff(Derivative(u
               [j](T), (T, 2)))

    return Mass_matrix

def Stiff_matrix(Lagrange, Reighley, n):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')
    q= sympy.Function('q')
    u= [sympy.Function('u%d' % i) for i in range(n)]

    L= Lagrange
    R= Reighley

    Lagrange_phi= L.diff(Derivative(phi(T), T)).diff(T) - L.diff(phi(T)
       ) + R.diff(Derivative(phi(T), T))
    Lagrange_U= L.diff(Derivative(U(T), T)).diff(T) - L.diff(U(T)) + R.
       diff(Derivative(U(T), T))
    Lagrange_q= L.diff(Derivative(q(T), T)).diff(T) - L.diff(q(T)) + R.
       diff(Derivative(q(T), T))
    Lagrange_u= [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i]= L.diff(Derivative(u[i](T), T)).diff(T) - L.diff(
            u[i](T)) + R.diff(Derivative(u[i](T), T))
    Stiff_matrix= np.zeros(((n + 3), (n + 3)))

    Stiff_matrix[0, 0:3] = [Lagrange_U.coeff(U(T)), Lagrange_U.coeff(
       phi(T)), Lagrange_U.coeff(q(T))]
    Stiff_matrix[1, 0:3] = [Lagrange_phi.coeff(U(T)), Lagrange_phi.
```

```
        coeff(phi(T)), Lagrange_phi.coeff(q(T))]
    Stiff_matrix[2, 0:3] = [Lagrange_q.coeff(U(T)), Lagrange_q.coeff(
        phi(T)), Lagrange_q.coeff(q(T))]

    for i in range(n):
        Stiff_matrix[0, 3 : i + 3]= Lagrange_U.coeff(u[i](T))
        Stiff_matrix[1, 3 : i + 3]= Lagrange_phi.coeff(u[i](T))
        Stiff_matrix[2, 3 : i + 3]= Lagrange_q.coeff(u[i](T))

        Stiff_matrix[i + 3, 0]= Lagrange_u[i].coeff(U(T))
        Stiff_matrix[i + 3, 1]= Lagrange_u[i].coeff(phi(T))
        Stiff_matrix[i + 3, 2]= Lagrange_u[i].coeff(q(T))
        for j in range(n):
            Stiff_matrix[i + 3, j + 3]= Lagrange_u[i].coeff(u[j](T))
    return Stiff_matrix

def Damp_matrix(Lagrange, Reighley, n):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')
    q= sympy.Function('q')
    u= [sympy.Function('u%d' % i) for i in range(n)]

    L= Lagrange
    R= Reighley

    Lagrange_phi= L.diff(Derivative(phi(T), T)).diff(T) - L.diff(phi(T)
        ) + R.diff(Derivative(phi(T), T))
    Lagrange_U= L.diff(Derivative(U(T), T)).diff(T) - L.diff(U(T)) + R.
        diff(Derivative(U(T), T))
    Lagrange_q= L.diff(Derivative(q(T), T)).diff(T) - L.diff(q(T)) + R.
        diff(Derivative(q(T), T))
    Lagrange_u= [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i]= L.diff(Derivative(u[i](T), T)).diff(T) - L.diff(
            u[i](T)) + R.diff(Derivative(u[i](T), T))

    Damp_matrix= np.zeros(((n + 3), (n + 3)))

    Damp_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), T)),
        Lagrange_U.coeff(Derivative(phi(T), T)), Lagrange_U.coeff(
        Derivative(q(T), T))]
    Damp_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), T)),
        Lagrange_phi.coeff(Derivative(phi(T), T)), Lagrange_phi.coeff(
        Derivative(q(T), T))]
    Damp_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), T)),
        Lagrange_q.coeff(Derivative(phi(T), T)), Lagrange_q.coeff(
        Derivative(q(T), T))]
    for i in range(n):
        Damp_matrix[0, 3 : i + 3]= Lagrange_U.coeff(Derivative(u[i](T))
            )
        Damp_matrix[1, 3 : i + 3]= Lagrange_phi.coeff(Derivative(u[i](T
            )))
        Damp_matrix[2, 3 : i + 3]= Lagrange_q.coeff(Derivative(u[i](T))
            )

        Damp_matrix[i + 3, 0]= Lagrange_u[i].coeff(Derivative(U(T)))
        Damp_matrix[i + 3, 1]= Lagrange_u[i].coeff(Derivative(phi(T)))
        Damp_matrix[i + 3, 2]= Lagrange_u[i].coeff(Derivative(q(T)))
        for j in range(n):
            Damp_matrix[i + 3, j + 3]= Lagrange_u[i].coeff(Derivative(u
                [i](T), T))

    return Damp_matrix
```

B.4 Response

```
def Response(Lagrange, Reighley, omega, Mass_m, Stiff_m, Damp_m, A, B,
    D, n, load= None):
    X= Symbol('X')
    Z= Symbol('Z')
    T= Symbol('T')
    U= sympy.Function('U')
    phi= sympy.Function('phi')
    q= sympy.Function('q')
    u= [sympy.Function('u%d' % i) for i in range(n)]

    L= Lagrange
    R= Reighley

    Lagrange_phi= L.diff(Derivative(phi(T), T)).diff(T) - L.diff(phi(T)
        ) + R.diff(Derivative(phi(T), T))
    Lagrange_U= L.diff(Derivative(U(T), T)).diff(T) - L.diff(U(T)) + R.
        diff(Derivative(U(T), T))
    Lagrange_q= L.diff(Derivative(q(T), T)).diff(T) - L.diff(q(T)) + R.
        diff(Derivative(q(T), T))
    Lagrange_u= [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i]= L.diff(Derivative(u[i](T), T)).diff(T) - L.diff(
            u[i](T)) + R.diff(Derivative(u[i](T), T))

    Eigenmatrix= np.linalg.inv(-omega**2 * (Mass_m + A) + 1j * omega *
        (Damp_m + B) + (Stiff_m + D))

    if load is None:
        Load= np.zeros(3 + n, dtype= 'complex_').transpose()
        Load[0]= M_total
        Load[1]= M_total
    else:
        Load= np.zeros(3 + n, dtype= 'complex_').transpose() + load
        Load[0] += M_total
        Load[1] += M_total

    disp= (Eigenmatrix @ Load)

    return disp
```

B.5 Initial guess parameters

```
def h_tld(Lagrange, Reighley, n):
    Stiff_m= Stiff_matrix(Lagrange, Reigh, n)
    Mass_m = Mass_matrix(Lagrange, Reigh, n)
    Damp_m = Damp_matrix(Lagrange, Reigh, n)
    f_ratio= np.sqrt((1 + m_ratio/2) / (1 + m_ratio))
    f_eigen= np.sqrt(scipy.linalg.eigvals(Stiff_m + D, b= Mass_m + A))
    print('Eigen:', f_eigen)
    h= np.arctanh((f_eigen[1] * f_ratio)**2 * 2 * a / (np.pi * g)) * 2
       * a / np.pi
    return abs(h.real)

def Mass_tld(h):
    return  h * 2 * a * b * rho

dpot_dx= dpotential_dx(a, h, m)
dpot_dz= dpotential_dz(a, h, m)
Lagrange= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx,
   dpot_dz)
Reigh= Reighley(a, h, c_tmd, c_tld, n, dpot_dx, dpot_dz, m)
Stiff_m= Stiff_matrix(Lagrange, Reigh, n)
Mass_m = Mass_matrix(Lagrange, Reigh, n)
Damp_m = Damp_matrix(Lagrange, Reigh, n)
f_eigen= np.sqrt(scipy.linalg.eigvals(Stiff_m + D, b= Mass_m + A))

# Calculate initial c and k
c_tmd= abs(2 * M_tmd * np.sqrt(f_eigen[0]) * eps)
k_tmd= abs(f_eigen[0] * M_tmd)

h = h_tld(Lagrange, Reigh, n)

M = M_structure(h)

M_tld= Mass_tld(h)
m_ratio_tld= M_tld / M_total # Mass ratio TLD/total structure
f_tld= 1 / (1 + m_ratio_tld) # frequency ratio TLD/total structure

data['c_tld']= abs(np.sqrt(m_ratio_tld / (1 + m_ratio_tld)) * 2 * M_tld
    * f_eigen[1] * f_tld).tolist()
data['c_tmd']= abs(c_tmd).tolist()
data['k_tmd']= abs(k_tmd).tolist()
```

```
jdata= json.dumps(data, indent=4)

f= open("model2.json", "w")
f.write(jdata)
f.close()
```

B.6 Serial optimization

```
# Performance index
def J(Lagrange, Reigh, A, B, D, n, load= None):
    omega= np.linspace(10**-5, 5, 100)

    S0= 1   # White noise spectrum (constant)

    Mass_m= Mass_matrix(Lagrange, Reigh, n)
    Stiff_m= Stiff_matrix(Lagrange, Reigh, n)
    Damp_m= Damp_matrix(Lagrange, Reigh, n)

    H_U= np.zeros(len(omega))
    H_phi= np.zeros(len(omega))
    dis= np.zeros((len(omega), n + 3))
    for i in range(len(omega)):
        dis[i] = Response(Lagrange, Reigh, omega[i], Mass_m, Stiff_m,
            Damp_m, A, B, D, n, load= None)

    for i in range(len(omega)):
        H_U[i] = dis[i][0]
        H_phi[i] = dis[i][1]

    # Performance of U and phi:
    E_U= S0 * np.trapz(abs(H_U), x= omega)
    E_phi= S0 * np.trapz(abs(H_phi * h_hub), x= omega)

    return E_U + E_phi

# Iterations of the parameters
def c_tmd_new(s_c, c_tmd, n):
    return c_tmd + np.ones(n) * s_c

def k_tmd_new(s_k, k_tmd, n):
    return k_tmd + np.ones(n) * s_k

def h_new(h, sh):
    return abs(h.real + sh)

def c_tld_new(s_tld, c_tld, n):
    return c_tld + s_tld

# Determining the stepsize for the parameter iterations
```

```
s_u= 10**4  # upper bound s
s_l= 0      # lower bound s

golden= (3 - np.sqrt(5)) / 2 # golden ratio
s1= s_l + (1 - golden) * (s_u - s_l)
s2= s_l + golden * (s_u - s_l)

# Upper bound
c_tmd_1= c_tmd_new(s1, c_tmd, n)
k_tmd_1= k_tmd_new(s1, k_tmd, n)
Lagrange_1= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_1, m, n, dpot_dx,
    dpot_dz)
Reigh_1= Reighley(a, h, c_tmd_1, c_tld, n, dpot_dx, dpot_dz, m)

J_1= J(Lagrange_1, Reigh_1, A, B, D, n, load= None)

# Lower bound
c_tmd_2= c_tmd_new(s2, c_tmd, n)
k_tmd_2= k_tmd_new(s2, k_tmd, n)
Lagrange_2= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_2, m, n, dpot_dx,
    dpot_dz)
Reigh_2= Reighley(a, h, c_tmd_2, c_tld, n, dpot_dx, dpot_dz, m)
J_2= J(Lagrange_2, Reigh_2, A, B, D, n, load= None)
tol= 10
max_it= 20

for i in range(max_it):
    if abs(s_u - s_l) < tol:
        break
    else:
        if J_1 < J_2:
            s_u= s1
            s1= s2
            J_2= J_1
            s1= s_l + (1 - golden) * (s_u - s_l)
            # Upper bound
            c_tmd_1= c_tmd_new(-s1, c_tmd, n)
            k_tmd_1= k_tmd_new(-s1, k_tmd, n)
            Lagrange_1= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_1, m,
                n, dpot_dx, dpot_dz)
            Reigh_1= Reighley(a, h, c_tmd_1, c_tld, n, dpot_dx, dpot_dz
                , m)
            J_1= J(Lagrange_1, Reigh_1, A, B, D, n, load= None)
            # Lower bound
            c_tmd_2= c_tmd_new(-s2, c_tmd, n)
            k_tmd_2= k_tmd_new(-s2, k_tmd, n)
            Lagrange_2= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_2, m,
                n, dpot_dx, dpot_dz)
            Reigh_2= Reighley(a, h, c_tmd_2, c_tld, n, dpot_dx, dpot_dz
                , m)
            J_2= J(Lagrange_2, Reigh_2, A, B, D, n, load= None)
        else:
            s_l= s2
            s2= s1
            J_1= J_2
            s1= s_l + golden * (s_u - s_l)

s= (s_l + s_u)/2

data["s"]= s

jdata= json.dumps(data, indent=4)

f= open("model2.json", "w")
f.write(jdata)
f.close()

# Determining the optimal TMD damping parameter

f= open("model2.json", "r+")
data= json.load(f)
f.close()

s= data["s"]

c_tmd_n= c_tmd_new(s, c_tmd, n)  # new iteration of c_tmd

Reigh_new= Reighley(a, h, c_tmd_n, c_tld, n, dpot_dx, dpot_dz, m)

J_old = J(Lagrange, Reigh, A, B, D, n, load= None)
J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
J_left= J_old
J_right= J_new

conv_J_ctmd= np.array([J_old])
conv_ctmd= np.array([c_tmd])

tol= 10**-4         # tolerance
c_previous= np.zeros(n)
max_iterations= 20
solution_c= False
for i in range(max_iterations):
    if abs((J_old - J_new)/ J_old) <= tol:
        if J_old < J_new:
            c_tmd= c_tmd
            J_final= J_old
            Reigh= Reighley(a, h, c_tmd, c_tld, n, dpot_dx, dpot_dz, m)
        else:
            c_tmd= c_tmd_n
            J_final= J_new
            Reigh= Reigh_new
        solution_c = True
        conv_J_ctmd= np.append(conv_J_ctmd, J_final)
        conv_ctmd= np.append(conv_ctmd, np.array([c_tmd]), axis=0)
```

```
            break
        elif np.all(c_previous == c_tmd_n):
            if J_old < J_new:
                c_tmd= c_tmd
                J_final= J_old
                Reigh= Reighley(a, h, c_tmd, c_tld, n, dpot_dx, dpot_dz, m)
            else:
                c_tmd= c_tmd_n
                J_final= J_new
                Reigh= Reigh_new
            solution_c = True
            conv_J_ctmd= np.append(conv_J_ctmd, J_final)
            conv_ctmd= np.append(conv_ctmd, np.array([c_tmd]), axis=0)
            break
        else:
            c_previous= c_tmd
            if J_left < J_right:
                if i == 0:
                    c_tmd= c_tmd
                else:
                    c_tmd= c_tmd_n
                c_tmd_n= c_tmd_new(-s, c_tmd, n)
                J_old= J_left
                J_right= J_left
                Reigh_new= Reighley(a, h, c_tmd_n, c_tld, n, dpot_dx,
                    dpot_dz, m)
                J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
                J_left= J_new
            else:
                c_tmd= c_tmd_n
                c_tmd_n= c_tmd_new(s, c_tmd, n)
                J_old= J_right
                J_left= J_right
                Reigh_new= Reighley(a, h, c_tmd_n, c_tld, n, dpot_dx,
                    dpot_dz, m)
                J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
                J_right= J_new
            conv_J_ctmd= np.append(conv_J_ctmd, J_new)
            conv_ctmd= np.append(conv_ctmd, np.array([c_tmd_n]), axis=0)
            continue

if solution_c == False:
    print('no convergence, c_tmd:', c_tmd)
else:
    print('optimal c_tmd:', c_tmd, '\n', 'J:', J_final)

data["J"]= J_final
data["Reighley"]= str(Reigh)
data["c_tmd"]= c_tmd.tolist()

jdata= json.dumps(data, indent=4)
```

```
f= open("model2.json", "w")
f.write(jdata)
f.close()

# Determining the optimal TMD stiffness parameter

f= open("model2.json", "r+")
data= json.load(f)
f.close()

k_tmd_n= k_tmd_new(s, k_tmd, n)
Lagrange= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx,
    dpot_dz)
Lagrange_new= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_n, m, n, dpot_dx
    , dpot_dz)
Reigh= parse_expr(data["Reighley"])

c_tmd= np.array(data["c_tmd"])

J_old = data["J"]
J_new= J(Lagrange_new, Reigh, A, B, D, n, load= None)
J_left= J_old
J_right= J_new

conv_J_ktmd= np.array([J_old])
conv_ktmd= np.array([k_tmd])

tol= 10**-4 # tolerance
max_iterations= 20
k_previous = np.zeros(n)
solution_k= False
for i in range(max_iterations):
    if abs((J_old - J_new)/ J_old) <= tol:
        if J_old < J_new:
            k_tmd= k_tmd
            J_final= J_old
            Lagrange= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd, m, n,
                dpot_dx, dpot_dz)
        else:
            k_tmd= k_tmd_n
            J_final= J_new
            Lagrange= Lagrange_new
        solution_k = True
        conv_J_ktmd= np.append(conv_J_ktmd, J_final)
        conv_ktmd= np.append(conv_ktmd, np.array([k_tmd]), axis= 0)
        break
    elif np.all(k_previous == k_tmd_n):
        if J_old < J_new:
            k_tmd= k_tmd
            J_final= J_old
            Lagrange= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd, m, n,
                dpot_dx, dpot_dz)
        else:
            k_tmd= k_tmd_n
            J_final= J_new
            Lagrange= Lagrange_new
        solution_k = True
        conv_J_ktmd= np.append(conv_J_ktmd, J_final)
        conv_ktmd= np.append(conv_ktmd, np.array([k_tmd]), axis= 0)
        break
    else:
        k_previous= k_tmd
        if J_left < J_right:
            if i == 0:
                k_tmd= k_tmd
            else:
                k_tmd= k_tmd_n
            k_tmd_n= k_tmd_new(-s, k_tmd, n)
            J_old= J_left
            J_right= J_left
            Lagrange_new= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_n, m
                , n, dpot_dx, dpot_dz)
            J_new= J(Lagrange_new, Reigh, A, B, D, n, load= None)
            J_left= J_new
        else:
            k_tmd= k_tmd_n
            k_tmd_n= k_tmd_new(s, k_tmd, n)
            J_old= J_right
            J_left= J_right
            Lagrange_new= Lagrangian(rho, a, h, M, g, M_tmd, k_tmd_n, m
                , n, dpot_dx, dpot_dz)
            J_new= J(Lagrange_new, Reigh, A, B, D, n, load= None)
            J_right= J_new
        conv_J_ktmd= np.append(conv_J_ktmd, J_new)
        conv_ktmd= np.append(conv_ktmd, np.array([k_tmd_n]), axis= 0)
        continue

if solution_k == False:
    print('no convergence')
else:
    print('optimal k_tmd:', k_tmd, '\n', 'J:', J_final)

data["J"]= J_final
data["Lagrangian"]= str(Lagrange)
data["k_tmd"]= k_tmd.tolist()

jdata= json.dumps(data, indent=4)

f= open("model2.json", "w")
f.write(jdata)
f.close()

# Determining the optimal TLD height
```

```
f= open("model2.json", "r+")
data= json.load(f)
f.close()

sh= s/c_tmd[0] * h # Iteration stepsize for h, scaled to c_tmd

h_n= h_new(h, sh)  # new iteration of h
c_tmd= np.array(data["c_tmd"])
dpot_dx= dpotential_dx(a, h_n, m)
dpot_dz= dpotential_dz(a, h_n, m)
Lagrange_new= Lagrangian(rho, a, h_n, M_structure(h_n), g, M_tmd, k_tmd
    , m, n, dpot_dx, dpot_dz)
J_old = data['J']
J_new= J(Lagrange_new, Reigh_new, A, B, D, n, load= None)
J_left= copy.copy(J_old)
J_right= copy.copy(J_new)

conv_J_h= np.array([J_old])
conv_h= np.array([h])

tol= 10**-4 # tolerance
h_previous= 0
max_iterations= 20
solution_h= False
for i in range(max_iterations):
    if abs((J_old - J_new)/ J_old) <= tol:
        if J_old < J_new:
            h = h
            J_final= J_old
            Reigh= Reigh
            Lagrange= Lagrange
        else:
            h= h_n
            J_final= J_new
            Reigh= Reigh_new
            Lagrange= Lagrange_new
            M = M_structure(h)
            M_tld= Mass_tld(h)
        solution_h = True
        conv_J_h= np.append(conv_J_h, J_final)
        conv_h= np.append(conv_h, h)
        break
    elif np.all(h_previous == h_n):
        if J_old < J_new:
            h = h
            J_final= copy.copy(J_old)
            Reigh= copy.copy(Reigh)
            Lagrange= copy.copy(Lagrange)
        else:
            h = copy.copy(h_n)
            J_final= copy.copy(J_new)
            Reigh= copy.copy(Reigh_new)
```

```
                Lagrange= copy.copy(Lagrange_new)
                M = M_structure(h)
                M_tld= Mass_tld(h)
            solution_h = True
            conv_J_h= np.append(conv_J_h, J_final)
            conv_h= np.append(conv_h, h)
            break
    else:
        h_previous= copy.copy(h)
        if J_left < J_right:
            if i == 0:
                h= h
            else:
                h= copy.copy(h_n)
            h_n= h_new(h, -sh)
            J_old= copy.copy(J_left)
            J_right= copy.copy(J_left)
            M_tld= Mass_tld(h_n)
            dpot_dx= dpotential_dx(a, h_n, m)
            dpot_dz= dpotential_dz(a, h_n, m)
            Reigh_new= Reighley(a, h_n, c_tmd, c_tld, n, dpot_dx,
               dpot_dz, m)
            Lagrange_new= Lagrangian(rho, a, h_n, M_structure(h_n), g,
               M_tmd, k_tmd, m, n, dpot_dx, dpot_dz)
            J_new= J(Lagrange_new, Reigh_new, A, B, D, n, load= None)
            J_left= copy.copy(J_new)
        else:
            h= copy.copy(h_n)
            h_n= h_new(h, sh)
            J_old= copy.copy(J_right)
            J_left= copy.copy(J_right)
            M_tld= Mass_tld(h_n)
            dpot_dx= dpotential_dx(a, h_n, m)
            dpot_dz= dpotential_dz(a, h_n, m)
            Reigh_new= Reighley(a, h_n, c_tmd, c_tld, n, dpot_dx,
               dpot_dz, m)
            Lagrange_new= Lagrangian(rho, a, h_n, M_structure(h_n), g,
               M_tmd, k_tmd, m, n, dpot_dx, dpot_dz)
            J_new= J(Lagrange_new, Reigh_new, A, B, D, n, load= None)
            J_right= copy.copy(J_new)
        conv_J_h= np.append(conv_J_h, J_new)
        conv_h= np.append(conv_h, h_n)
        continue

if solution_h == False:
    print('no convergence')
else:
    print('optimal h:', h, '\n', 'J:', J_final)

data["J"]= J_final
data["Reighley"]= str(Reigh)
data['h']= h.real
```

```
jdata= json.dumps(data, indent=4)

f= open("model2.json", "w")
f.write(jdata)
f.close()

# Determining the optimal TLD equivalent damping parameter

f= open("model2.json", "r+")
data= json.load(f)
f.close()

s = s/c_tmd[0] * c_tld  # Iteration stepsize for c_tld, scaled to c_tmd
M = M_structure(h)

c_tld_n= c_tld_new(s, c_tld, n)  # new iteration of c_tld

c_tmd= np.array(data["c_tmd"])
k_tmd= np.array(data["k_tmd"])
Reigh= Reighley(a, h, c_tmd, c_tld, n, dpot_dx, dpot_dz, m) #
   parse_expr(data["Reighley"])

dpot_dx= dpotential_dx(a, h, m)
dpot_dz= dpotential_dz(a, h, m)
Reigh_new= Reighley(a, h, c_tmd, c_tld_n, n, dpot_dx, dpot_dz, m)
Lagrange= parse_expr(data["Lagrangian"])

J_old = data['J']
J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
J_left= copy.copy(J_old)
J_right= copy.copy(J_new)

conv_J_ctld= np.array([J_old])
conv_ctld= np.array([c_tld])

tol= 10**-4 # tolerance
c_tld_previous= 0
max_iterations= 20
solution_c_tld= False
for i in range(max_iterations):
    if abs((J_old - J_new)/ J_old) <= tol:
        if J_old < J_new:
            c_tld= c_tld
            J_final= J_old
            Reigh= Reigh
        else:
            c_tld= c_tld_n
            J_final= J_new
            Reigh= Reigh_new
        solution_c_tld = True
        conv_J_ctld= np.append(conv_J_ctld, J_final)
        conv_ctld= np.append(conv_ctld, c_tld)
        break
    elif np.all(c_tld_previous == c_tld_n) and i > 2:
        if J_old < J_new:
            c_tld= c_tld
            J_final= J_old
            Reigh= Reigh
        else:
            c_tld= c_tld_n
            J_final= J_new
            Reigh= Reigh_new
        solution_c_tld = True
        conv_J_ctld= np.append(conv_J_ctld, J_final)
        conv_ctld= np.append(conv_ctld, c_tld)
        break
    else:
        c_tld_previous= copy.copy(c_tld)
        if J_left < J_right:
            if i == 0:
                c_tld= c_tld
            else:
                c_tld= copy.copy(c_tld_n)
            c_tld_n= c_tld_new(-s, c_tld, n)
            J_old= copy.copy(J_left)
            J_right= copy.copy(J_left)
            Reigh_new= Reighley(a, h, c_tmd, c_tld_n, n, dpot_dx,
                dpot_dz, m)
            J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
            J_left= copy.copy(J_new)
        else:
            c_tld= copy.copy(c_tld_n)
            c_tld_n= c_tld_new(s, c_tld, n)
            J_old= copy.copy(J_right)
            J_left= copy.copy(J_right)
            Reigh_new= Reighley(a, h, c_tmd, c_tld_n, n, dpot_dx,
                dpot_dz, m)
            J_new= J(Lagrange, Reigh_new, A, B, D, n, load= None)
            J_right= copy.copy(J_new)
        conv_J_ctld= np.append(conv_J_ctld, J_new)
        conv_ctld= np.append(conv_ctld, c_tld_n)
        continue

if solution_c_tld == False:
    print('no convergence')
else:
    print('optimal c_tld:', c_tld, '\n', 'J:', J_final)

data["J"]= J_final
data["Reighley"]= str(Reigh)
data["c_tld"]= c_tld
data['h']= abs(h)
```

```
jdata= json.dumps(data, indent=4)

f= open("model2.json", "w")
f.write(jdata)
f.close()
```

B.7 Newton optimization

```
def Optimization(c_tmd, k_tmd, h, c_tld, rho, a, M, g, M_tmd, m, A, B,
    D, n, max_error= 0.01, max_iterations= 25, load= None):
     # Finding initial interval:
     dpot_dx= dpotential_dx(a, h, m)
     dpot_dz= dpotential_dz(a, h, m)
     x0= np.array([np.array(c_tmd), np.array(k_tmd), h, c_tld])
     # Create Jacobian:
     Jac= np.zeros(4)
     Hes= np.zeros((4, 4))

     eps = np.array([[np.array([0.1, 0.1]), 0, 0, 0], [0, np.array([0.1,
          0.1]), 0, 0], [0, 0, 0.1, 0], [0, 0, 0, 0.1]]) * max_error

     solution = False

     f_dic= {}
     f0 = J(Lagrangian(rho, a, x0, M, g, M_tmd, m, n, dpot_dx, dpot_dz),
          Reighley(a, x0, n, dpot_dx, dpot_dz, m), A, B, D, n, load= None
         )
     data['J('+ "c_tmd=" + str(x0[0]) + "k_tmd=" + str(x0[1]) + "c_tld="
         + str(x0[3]) + "h_tld=" + str(x0[2]) + ')'] = f0
     for dx in range(4): # number of parameters
          f_dic['f' + str(dx)] = J(Lagrangian(rho, a, x0 + eps[dx], M, g,
               M_tmd, m, n, dpot_dx, dpot_dz), Reighley(a, x0 + eps[dx], n
              , dpot_dx, dpot_dz, m), A, B, D, n, load= None)
          print('fi= ', f_dic['f' + str(dx)])
          Jac[dx] = (f_dic['f' + str(dx)] - f0)/ eps[3][3]
     for dx in range(4):
          for df in range(4):
               f_dic['fij' + str(df) + str(dx)] = J(Lagrangian(rho, a, x0
                   + eps[dx] + eps[df], M, g, M_tmd, m, n, dpot_dx, dpot_dz
                   ), Reighley(a, x0 + eps[dx] + eps[df], n, dpot_dx,
                   dpot_dz, m), A, B, D, n, load= None)
               Hes[df][dx] = (f_dic['fij' + str(df) + str(dx)] - f_dic['f'
                    + str(dx)] - f_dic['f' + str(df)] + f0)/ eps[3][3] ** 2
     delta_x = np.linalg.pinv(Hes) @ Jac
     r = 1 # damping factor
     _t = x0 - r * delta_x
     f_t = J(Lagrangian(rho, a, _t, M, g, M_tmd, m, n, dpot_dx, dpot_dz)
         , Reighley(a, _t, n, dpot_dx, dpot_dz, m), A, B, D, n, load=
        None)
```

```
while (f_t > f0):
    r = 1/2 * r
    f_t = J(Lagrangian(rho, a, x0 - r * delta_x, M, g, M_tmd, m, n,
        dpot_dx, dpot_dz), Reighley(a, x0 - r * delta_x, n, dpot_dx
       , dpot_dz, m), A, B, D, n, load= None)
x1 = x0 - r * delta_x
f1 = J(Lagrangian(rho, a, x1, M, g, M_tmd, m, n, dpot_dx, dpot_dz),
    Reighley(a, x1, n, dpot_dx, dpot_dz, m), A, B, D, n, load= None
   )
data['J('+ "c_tmd=" + str(x1[0]) + "k_tmd=" + str(x1[1]) + "c_tld="
    + str(x1[3]) + "h_tld=" + str(x1[2]) + ')'] = f1
for dx in range(4): # number of parameters
    f_dic['f' + str(dx)] = J(Lagrangian(rho, a, x1 + eps[dx], M, g,
        M_tmd, m, n, dpot_dx, dpot_dz), Reighley(a, x1 + eps[dx], n
       , dpot_dx, dpot_dz, m), A, B, D, n, load= None)
    Jac[dx] = (f_dic['f' + str(dx)] - f1)/ eps[3][3]
for dx in range(4):
    for df in range(4):
        f_dic['fij' + str(df) + str(dx)] = J(Lagrangian(rho, a, x1
            + eps[dx] + eps[df], M, g, M_tmd, m, n, dpot_dx, dpot_dz
            ), Reighley(a, x1 + eps[dx] + eps[df], n, dpot_dx,
            dpot_dz, m), A, B, D, n, load= None)
        Hes[df][dx] = (f_dic['fij' + str(df) + str(dx)] - f_dic['f'
             + str(dx)] - f_dic['f' + str(df)] + f1)/ eps[3][3] ** 2
delta_x = np.linalg.pinv(Hes) @ Jac
x0 = copy(x1)
r = 1 # damping factor
_t = x1 - r * delta_x
f_t = J(Lagrangian(rho, a, _t, M, g, M_tmd, m, n, dpot_dx, dpot_dz)
   , Reighley(a, _t, n, dpot_dx, dpot_dz, m), A, B, D, n, load=
   None)
while (f_t > f0):
    r = 1/2 * r
    print('r:', r)
    f_t = J(Lagrangian(rho, a, x1 - r * delta_x, M, g, M_tmd, m, n,
        dpot_dx, dpot_dz), Reighley(a, x1 - r * delta_x, n, dpot_dx
       , dpot_dz, m), A, B, D, n, load= None)
x2 = x1 - r * delta_x
conv_J= np.array([f0, f1])
conv_par = np.array([x0, x1, x2])
for i in range(max_iterations):
    if solution == True:
        break
    for j in range(len((x1))):
        solution = True
        Optimum = (1/2 * (x1 + x2)) #.real + (1/2 * (x1 + x2)).imag
        try:
            if (all(abs(x1 - x2)[j] > np.ones(len((x1)[j])) *
               max_error)):
                solution = False
                continue
        except (TypeError):
            if (abs(x1 - x2)[j] > max_error):
                solution = False
                continue
    if solution == True:
        break
    solution = True
    Optimum = (1/2 * (x1 + x2)) #.real + (1/2 * (x1 + x2)).imag
    for j in range(len(Jac)):
        try:
            if (abs(Jac)[j] > np.ones(n) * max_error).any():
                solution = False
                continue
        except (TypeError):
            if (abs(Jac)[j] > max_error).any():
                solution = False
                continue
    if solution == True:
        break
    if np.array_equal(x0, x2): # or any(abs(disp2) > 10 * abs(disp1
       )):
        print(2)
        break
    else:
        f0= copy(f1)
        x0 = copy(x1)
        x1 = copy(x2)
        f1 = J(Lagrangian(rho, a, x1, M, g, M_tmd, m, n, dpot_dx,
            dpot_dz), Reighley(a, x1, n, dpot_dx, dpot_dz, m), A, B,
             D, n, load= None)
        data['J('+ "c_tmd=" + str(x1[0]) + "k_tmd=" + str(x1[1]) +
            "c_tld=" + str(x1[3]) + "h_tld=" + str(x1[2]) + ')'] =
            f1
        for dx in range(4): # number of parameters
            f_dic['f' + str(dx)] = J(Lagrangian(rho, a, x1 + eps[dx
               ], M, g, M_tmd, m, n, dpot_dx, dpot_dz), Reighley(a,
                x1 + eps[dx], n, dpot_dx, dpot_dz, m), A, B, D, n,
               load= None)
            Jac[dx] = (f_dic['f' + str(dx)] - f1)/ eps[3][3]
        for dx in range(4):
            for df in range(4):
                f_dic['fij' + str(df) + str(dx)] = J(Lagrangian(rho
                    , a, x1 + eps[dx] + eps[df], M, g, M_tmd, m, n,
                    dpot_dx, dpot_dz), Reighley(a, x1 + eps[dx] +
                    eps[df], n, dpot_dx, dpot_dz, m), A, B, D, n,
                    load= None)
                Hes[df][dx] = (f_dic['fij' + str(df) + str(dx)] -
                    f_dic['f' + str(dx)] - f_dic['f' + str(df)] + f1
                    )/ eps[3][3] ** 2
        delta_x = np.linalg.pinv(Hes) @ Jac
        r = 1 # damping factor
        _t = x1 - r * delta_x
        f_t = J(Lagrangian(rho, a, _t, M, g, M_tmd, m, n, dpot_dx,
            dpot_dz), Reighley(a, _t, n, dpot_dx, dpot_dz, m), A, B,
             D, n, load= None)
        while (f_t > f0) & (r > 0.001):
            r = 1/2 * r
            f_t = J(Lagrangian(rho, a, x1 - r * delta_x, M, g,
                M_tmd, m, n, dpot_dx, dpot_dz), Reighley(a, x1 - r *
                delta_x, n, dpot_dx, dpot_dz, m), A, B, D, n, load=
                None)
        x2 = x1 - r * delta_x
        conv_J = np.append(conv_J, [f1])
        conv_par= np.append(conv_par, np.array([x2]), axis= 0)

data['convergence J'] = conv_J
data['convergence parameters'] = conv_par

if solution == False:
    Optimum= x0.real + x0.imag

data["Optimum[c_tmd, k_tmd, h, c_tld]"] = Optimum.tolist()
data['J optimal'] = f1.tolist()

return f1, Optimum
```

C | Appendix model 3

The script for model 3 is run in DelftBlue [1].

C.1 Input

```
import numpy as np
import scipy
import sympy
from scipy.linalg import eigh
from numpy.linalg import inv, eig
from scipy import integrate, signal
from sympy import Symbol, solve, Poly, diff, Derivative, I
from sympy.utilities.lambdify import lambdify
from sympy.abc import t, x, z, T, X, Z
import json
from sympy.parsing.sympy_parser import parse_expr
import copy
import mpi4py
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()


data = {}
# Importing starting values (optimum for model 2)
if rank == 0:
    f = open("model2.json", "r+")
    data_model2 = json.load(f)
    f.close()

# In case of a re-run
    try:
        f = open("model3.json", "r+")
        data = json.load(f)
        f.close()
    except FileNotFoundError:
        data = data_model2
        jdata = json.dumps(data, indent=4)
        f = open("model3.json", "w")
        f.write(jdata)
        f.close()

    h = data["h"]
    c_tld = data["c_tld"]
    k_tmd = np.array(data["k_tmd"])
    c_tmd = np.array(data["c_tmd"])
else:
    data = None
    h= data_model2["h"]
    c_tld= data_model2["c_tld"]
    k_tmd= np.array(data_model2["k_tmd"])
    c_tmd= np.array(data_model2["c_tmd"])

data = comm.bcast(data, root=0)
h = comm.bcast(h, root=0)
c_tmd = comm.bcast(c_tmd, root=0)
k_tmd = comm.bcast(k_tmd, root=0)
c_tld = comm.bcast(c_tld, root=0)

# potential flow rectangular tank
m = 1           # number of modes
a = 20          # half of tank width
b = 2 * a       # tank depth
rho = 1000      # water density
g = 9.81        # gravitational acceleration
h_hub = 87.74   # hub height
n = 2           # number of TMDs

M_total = 6149460

def M_structure(h):   # Mass barge + turbine
    return M_total * .9 - (rho * 2 * a * b * h)

if rank == 0:
    M = M_structure(h)
else:
    M = 0
M = comm.bcast(M, root=0)

M_tmd = np.ones(n) * 0.1 * M_total / n  # TMD mass
H = M_total / (rho * b * 2 * a)         # Floating depth

A = np.zeros((n+3, n + 3))  # Hydraulic mass matrix
A[0, 0] = 9.556 * 10**5
A[1, 1] = 2.187 * 10**8

B = np.zeros((n+3, n + 3))  # Hydraulic damping matrix
B[0, 0] = 2/(3 * np.pi) * (np.pi - 1) * b * rho * H * \
    (2 * np.pi * (1/2 * np.pi - 1) / (2 * a))**(1/3)

D = np.zeros((n+3, n + 3))  # Hydraulic stiffness matrix
D[0, 0] = 2 * rho * g * b * a
```

```
D[1, 1] = rho * g * b * a**2
```

C.2 TLD sloshing

The TLD functions for the initial iteration are the same as those for model 2 B.2 and are not repeated here. The new functions also work for the initial iteration, but are slower due to an increase in symbolic integrations. The faster functions are used where possible.

```
def B_A(disp, a, h, m, n):  # disp: response of the previous iteration
    a_A = np.zeros(m)
    b_A = np.zeros(m)
    c_A = np.zeros(m)
    B_A = np.zeros(m)

    for i in range(m):
        for j in range(n):
            if -h + disp[j + 3] < delta_h:
                a_A[i] += integrate.dblquad(lambda x, z: np.sin(alpha(a
                    , m)[i] * x)**2 + np.sinh(alpha(a, m)[i] * z)**2, -h
                     + disp[j + 3], delta_h, lambda x: -a + 2 * a / n *
                    j, lambda x: -a + 2 * a / n * (j + 1))[0]
                b_A[i] += 2 * integrate.dblquad(lambda x, z: np.sinh(
                    alpha(a, m)[i] * z) * np.cosh(alpha(a, m)[i] * z), -
                    h + disp[j + 3], delta_h, lambda x: -a + 2 * a / n *
                     j, lambda x: -a + 2 * a / n * (j + 1))[0] - a /
                    alpha(a, m)[i]
                c_A[i] += integrate.dblquad(lambda x, z: np.cos(alpha(a
                    , m)[i] * x)**2 + np.sinh(alpha(a, m)[i] * z)**2, -h
                     + disp[j + 3], delta_h, lambda x: -a + 2 * a / n *
                    j, lambda x: -a + 2 * a / n * (j + 1))[0]

        B_A[i] = (-b_A[i] - np.sqrt(b_A[i]**2 - 4 *
                  a_A[i] * c_A[i])) / (2 * a_A[i])
    return B_A


def B_S(disp, a, h, m, n):  # disp: response of the previous iteration
    a_S = np.zeros(m)
    b_S = np.zeros(m)
    c_S = np.zeros(m)
    B_S = np.zeros(m)

    for i in range(m):
        for j in range(n):
            if -h + disp[j + 3] < delta_h:
                a_S[i] += integrate.dblquad(lambda x, z: np.cos(beta(a,
                     m)[i] * x)**2 + np.sinh(beta(a, m)[i] * z)**2, -h +
                     disp[j + 3], delta_h, lambda x: -a + 2 * a / n * j,
                     lambda x: -a + 2 * a / n * (j + 1))[0]
                b_S[i] += 2 * integrate.dblquad(lambda x, z: np.sinh(
                    beta(a, m)[i] * z) * np.cosh(beta(a, m)[i] * z), -h
                    +disp[j + 3], delta_h, lambda x: -a + 2 * a / n * j,
                     lambda x: -a + 2 * a / n * (j + 1))[0] - a / beta(a
                    , m)[i]
                c_S[i] += integrate.dblquad(lambda x, z: np.sin(beta(a,
                    m)[i] * x)**2 + np.sinh(beta(a, m)[i] * z)**2, -h +
                    disp[j + 3], delta_h, lambda x: -a + 2 * a / n * j,
                    lambda x: -a + 2 * a / n * (j + 1))[0]

        B_S[i] = (-b_S[i] - np.sqrt(b_S[i]**2 - 4 *
                   a_S[i] * c_S[i])) / (2 * a_S[i])

    return B_S

def dpotential_dx(disp, a, h, m, n):
    X = Symbol('X')

    global delta_h   # change in waterheight from rest position

    d_tmd = disp[3: 3 + n]
    delta_h = 1/n * np.sum(d_tmd)
    count = 0
    for i in range(n):
        if d_tmd[i] > h + delta_h:
            np.delete(d_tmd, [i])
        else:
            count += 1
    delta_h = 1/count * np.sum(d_tmd)

    dpot_dx = 0
    for i in range(m):
        dpot_dx += (potential_sA(alpha(a, m)[i], B_A0(a, h, m, n)[i])).
            diff(X) + (potential_sS(beta(a, m)[i], B_S0( a, h, m, n)[i])
            ).diff(X)
    return dpot_dx


def dpotential_dz(disp, a, h, m, n):
    Z = Symbol('Z')

    global delta_h   # change in waterheight from rest position

    d_tmd = disp[3: 3 + n]
    delta_h = 1/n * np.sum(d_tmd)
    count = 0
    for i in range(n):
        if d_tmd[i] > h + delta_h:
            np.delete(d_tmd, [i])
        else:
            count += 1
    delta_h = 1/count * np.sum(d_tmd)

    dpot_dz = 0
    for i in range(m):
        dpot_dz += (potential_sA(alpha(a, m)[i], B_A0( a, h, m, n)[i]))
            .diff(Z) + (potential_sS(beta(a, m)[i], B_S0( a, h, m, n)[i
            ])).diff(Z)
    return dpot_dz
```

C.3 Initial equation of motion

```
def L0(rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx0, dpot_dz0):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U= sympy.Function('U')                                    # DoF heave
    phi= sympy.Function('phi')                                # DoF pitch
    q= sympy.Function('q')                                    # DoF TLD wave
        amplitude
    u= [sympy.Function('u%d' % i) for i in range(n)] # DoF TMD

    vert = U(T) + phi(T) * X

    # kinetic energy without tmds
    K = sympy.expand(1/2 * rho * b * sympy.integrate(sympy.integrate((
        dpot_dx0) ** 2 + (vert.diff(T) + dpot_dz0)** 2, (Z, -h, 0)), (X,
        -a, a))) + M / 2 * (U(T).diff(T)**2 + 1/3 * a**2 * phi(T).diff(
        T)**2)

    # potential energy without tmds
    P = M * g * U(T)
    wave = 0
    for i in range(m):
        wave += wave_shape_sA(alpha(a, m)[i], B_A0(a, h, m, n)[i]) +
            wave_shape_sS(beta(a, m)[i], B_S0(a, h, m, n)[i]) + h

    for i in range(m):
        P += sympy.expand(g * rho * b * sympy.integrate(1 / 2 * (wave +
            vert) * wave, (X, -a, a)))

    # additional energy tmds:
    for i in range(n):
        K += sympy.expand(1/2 * M_tmd[i] * (u[i](T).diff(T) + U(T).diff
            (T) + phi(T).diff(T) * (- a + (2 * a * i) / (n - 1)))**2)

        P += sympy.expand(1/2 * k_tmd[i] * u[i](T)**2 + M_tmd[i] * g *
            (u[i](T) + U(T) + phi(T) * (- a + (2 * a * i) / (n - 1))))

    L = K - P

    return L

def R0(a, b, h, c_tmd, c_tld, m, n, dpot_dx, dpot_dz):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U= sympy.Function('U')                              # DoF heave
    phi= sympy.Function('phi')                          # DoF pitch
    q= sympy.Function('q')                              # DoF TLD wave
        amplitude
    u= [sympy.Function('u%d' % i) for i in range(n)] # DoF TMD

    # reighley dissipation tld (without tmds)
    R = 1/2 * b * c_tld * sympy.integrate(abs(sympy.integrate(dpot_dx
        ** 2 + dpot_dz ** 2 , (Z, -h, 0))), (X, -a, a))

    # additional dissipation tmds:
    for i in range(n):
        R += sympy.expand(1/2 * c_tmd[i] * (u[i](T).diff(T))**2)

    return R
```

C.4 Equation of motion

```
def L(disp, omega, t, rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx,
    dpot_dz):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U= sympy.Function('U')                              # DoF heave
    phi= sympy.Function('phi')                          # DoF pitch
    q= sympy.Function('q')                              # DoF TLD wave
        amplitude
    u= [sympy.Function('u%d' % i) for i in range(n)] # DoF TMD

    omega_A = Omega_A(alpha(a, m), B_A0(a, h, m, n), g, m)
    omega_S = Omega_S(beta(a, m), B_S0(a, h, m, n), g, m)

    global delta_h   # change in waterheight from rest position

    d_tmd = disp[3: 3 + n]
    delta_h = 1/n * np.sum(d_tmd)
    count = 0
    for i in range(n):
        if d_tmd[i] > h + delta_h:
            np.delete(d_tmd, [i])
        else:
            count += 1
    delta_h = 1/count * np.sum(d_tmd)

    vert = U(T) + phi(T) * X  # vertical response of the barge

    # kinetic energy without tmds
    K = M / 2 * (U(T).diff(T)**2 + 1/3 * a**2 * phi(T).diff(T)**2)
    for j in range(n):
        if d_tmd[i] < h + delta_h:
            K += sympy.expand(1/2 * rho * b * sympy.integrate(sympy.
                integrate((dpot_dx) ** 2 + (u[j](T).diff(T) + vert.diff(
                T) + dpot_dz) ** 2, (Z, -h + d_tmd[j], delta_h)), (X, -a
                 + 2 * a / n * j, -a + 2 * a / n * (j + 1))))

    # Potential energy without tmds:
    # The liquid might not cover the whole length of the barge. The
        integral needs to be taken over the covered length only,
        otherwise (significant) negative integrals are taken, which
        causes errors and overflow in the code.
    wave_shape_A = 0
    wave_shape_S = 0
    P = sympy.expand(M * g * U(T))
    global intersection
    intersection = None  # The possible boundary between 'dry' tmd's
        and 'wet' tmd's
    for i in range(m):
        wave_shape_A += wave_shape_sA(alpha(a, m)[i], B_A0(a, h, m, n)[
            i])
        wave_shape_S += wave_shape_sS(beta(a, m)[i], B_S0(a, h, m, n)[i
            ])
    wave = wave_shape_A + wave_shape_S #+ h + delta_h

    for i in range(m):
        for j in range(n):
            try:  # In case of an intersection
                intersection = sympy.solve(sympy.re((wave_shape_A.subs(
                    q(T), sympy.exp(
                     I * t * omega_A[i])) + wave_shape_S.subs(q(T),
                        sympy.exp(I * t * omega_S[i])))) - disp[j + 3],
                        X)
                if -a + 2 * a / n * j < intersection[0] < -a + 2 * a /
                    n * (j + 1):
                     if (sympy.re(wave_shape_A.subs(q(T), sympy.exp(I *
                        t * omega_A[i])) + wave_shape_S.subs(q(T), sympy
                        .exp(I * t * omega_S[i]))).subs(X, -a + 2 * a /
                        n * j)) > disp[j + 3]:
                         P += rho * g * b * sympy.integrate((1/2 * (wave
                              + delta_h) + vert) * (wave + delta_h), (X,
                             -a + 2 * a / n * j, intersection))
                     else:
                         P += rho * g * b * sympy.integrate((1/2 * (wave
                              + delta_h) + vert) * (wave + delta_h), (X,
                             intersection, -a + 2 * a / n * (j + 1)))
                elif sympy.re((wave_shape_A.subs(q(T), sympy.exp(I * t
                    * omega_A[i])) + wave_shape_S.subs(q(T), sympy.exp(I
                     * t * omega_S[i]))).subs(X, -a + 2 * a / n * j)) >
                    disp[j + 3]:
                     P += rho * g * b * sympy.integrate((1/2 * (wave +
                        delta_h) + vert) * (wave + delta_h), (X, -a + 2
                        * a / n * j, -a + 2 * a / n * (j + 1)))
                else:
                    P += 0
            except:  # In case there are no intersections
                P += rho * g * b * sympy.integrate((1/2 * (wave +
                    delta_h) + vert) * (wave + delta_h), (X, -a + 2 * a
                    / n * j, -a + 2 * a / n * (j + 1)))

    # Additional energy tmds:

    for i in range(n):
        K += sympy.expand(1/2 * M_tmd[i] * (u[i](T).diff(T) + (U(T).
            diff(T) + phi(T).diff(T) * (- a + (2 * a * i) / (n - 1))))
            **2)

        P += sympy.expand(1/2 * k_tmd[i] * u[i](T)**2 + M_tmd[i] * g *
            (u[i](T) + (U(T) + phi(T) * (- a + (2 * a * i) / (n - 1)))))

    L = K - P

    return L


def R(disp, a, b, h, c_tmd, c_tld, m, n, dpot_dx, dpot_dz):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U= sympy.Function('U')                                  # DoF heave
    phi= sympy.Function('phi')                              # DoF pitch
    q= sympy.Function('q')                                  # DoF TLD wave
        amplitude
    u= [sympy.Function('u%d' % i) for i in range(n)] # DoF TMD

    global delta_h   # change in waterheight from rest position
    d_tmd = disp[3: 3 + n]
    delta_h = 1/n * np.sum(d_tmd)
    count = 0
    for i in range(n):
        if d_tmd[i] > h + delta_h:
            np.delete(d_tmd, [i])
        else:
            count += 1
    delta_h = 1/count * np.sum(d_tmd)

    # reighley dissipation tld (without tmds)
    R = 0
    for i in range(n):
        R += 1/2 * b * c_tld * sympy.integrate(abs(sympy.integrate(
            dpot_dx ** 2 + dpot_dz ** 2 , (X, -a + 2*a*i/n, -a + 2 * a *
             (i + 1) / n))), (Z, -h + disp[3 + i], delta_h))

    # additional energy tmds:
    for i in range(n):
        R += sympy.expand(1/2 * c_tmd[i] * (u[i](T).diff(T))**2)

    return R
```

C.5 Initial response

```
def Response0(omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m,
    n, A, B, D, dpot_dx0, dpot_dz0, load=None):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U = sympy.Function('U')
    phi = sympy.Function('phi')
    q = sympy.Function('q')
    u = [sympy.Function('u%d' % i) for i in range(n)]

    omega_A = Omega_A(alpha(a, m), B_A0(a, h, m, n), g, m)
    omega_S = Omega_S(beta(a, m), B_S0(a, h, m, n), g, m)

    # Lagrangians:
    La = L0(rho, a, h, M, g, M_tmd, k_tmd, m, n, dpot_dx0, dpot_dz0)
    Re = R0(a, b, h, c_tmd, c_tld, m, n, dpot_dx0, dpot_dz0)

    Lagrange_phi = La.diff(Derivative(phi(T), T)).diff(T) - La.diff(phi
        (T)) + Re.diff(Derivative(phi(T), T))
    Lagrange_U = La.diff(Derivative(U(T), T)).diff(T) - La.diff(U(T)) +
        Re.diff(Derivative(U(T), T))
    Lagrange_q = La.diff(Derivative(q(T), T)).diff(T) - La.diff(q(T)) +
        Re.diff(Derivative(q(T), T))
    Lagrange_u = [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i] = La.diff(Derivative(u[i](T), T)).diff(T) - La.
            diff(u[i](T)) + Re.diff(Derivative(u[i](T), T))

    # Mass matrix:

    Mass_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Mass_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), (T, 2))),
        Lagrange_U.coeff(Derivative(phi(T), (T, 2))), Lagrange_U.coeff(
        Derivative(q(T), (T, 2)))]
    Mass_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), (T, 2)))
        , Lagrange_phi.coeff(Derivative(phi(T), (T, 2))), Lagrange_phi.
        coeff(Derivative(q(T), (T, 2)))]
    Mass_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), (T, 2))),
        Lagrange_q.coeff(Derivative(phi(T), (T, 2))), Lagrange_q.coeff(
        Derivative(q(T), (T, 2)))]

    for i in range(n):
        Mass_matrix[0, 3: i + 3] = Lagrange_U.coeff(Derivative(u[i](T),
            (T, 2)))
        Mass_matrix[1, 3: i + 3] = Lagrange_phi.coeff(Derivative(u[i](T
            ), (T, 2)))
        Mass_matrix[2, 3: i + 3] = Lagrange_q.coeff(Derivative(u[i](T),
            (T, 2)))

        Mass_matrix[i + 3, 0] = Lagrange_u[i].coeff(Derivative(U(T), (T
            , 2)))
        Mass_matrix[i + 3, 1] = Lagrange_u[i].coeff(Derivative(phi(T),
            (T, 2)))
        Mass_matrix[i + 3, 2] = Lagrange_u[i].coeff(Derivative(q(T), (T
            , 2)))
        for j in range(n):
            Mass_matrix[i + 3, j + 3] = Lagrange_u[i].coeff(Derivative(
                u[j](T), (T, 2)))

    # Stiffness matrix:

    Stiff_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Stiff_matrix[0, 0:3] = [Lagrange_U.coeff(U(T)), Lagrange_U.coeff(
        phi(T)), Lagrange_U.coeff(q(T))]
    Stiff_matrix[1, 0:3] = [Lagrange_phi.coeff(U(T)), Lagrange_phi.
        coeff(phi(T)), Lagrange_phi.coeff(q(T))]
    Stiff_matrix[2, 0:3] = [Lagrange_q.coeffU(T)), Lagrange_q.coeff(phi
        (T)), Lagrange_q.coeff(q(T))]

    for i in range(n):
        Stiff_matrix[0, 3: i + 3] = Lagrange_U.coeff(u[i](T))
        Stiff_matrix[1, 3: i + 3] = Lagrange_phi.coeff(u[i](T))
        Stiff_matrix[2, 3: i + 3] = Lagrange_q.coeff(u[i](T))

        Stiff_matrix[i + 3, 0] = Lagrange_u[i].coeff(U(T))
        Stiff_matrix[i + 3, 1] = Lagrange_u[i].coeff(phi(T))
        Stiff_matrix[i + 3, 2] = Lagrange_u[i].coeff(q(T))
        for j in range(n):
            Stiff_matrix[i + 3, j + 3] = Lagrange_u[i].coeff(u[j](T))

    # Damping matrix

    Damp_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Damp_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), T)),
        Lagrange_U.coeff(Derivative(phi(T), T)), Lagrange_U.coeff(
        Derivative(q(T), T))]
    Damp_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), T)),
        Lagrange_phi.coeff(Derivative(phi(T), T)), Lagrange_phi.coeff(
        Derivative(q(T), T))]
    Damp_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), T)),
        Lagrange_q.coeff(Derivative(phi(T), T)), Lagrange_q.coeff(
        Derivative(q(T), T))]
    for i in range(n):
        Damp_matrix[0, 3: i + 3] = Lagrange_U.coeff(Derivative(u[i](T))
            )
        Damp_matrix[1, 3: i + 3] = Lagrange_phi.coeff(Derivative(u[i](T
            )))
        Damp_matrix[2, 3: i + 3] = Lagrange_q.coeff(Derivative(u[i](T))
            )

        Damp_matrix[i + 3, 0] = Lagrange_u[i].coeff(Derivative(U(T)))
        Damp_matrix[i + 3, 1] = Lagrange_u[i].coeff(Derivative(phi(T)))
        Damp_matrix[i + 3, 2] = Lagrange_u[i].coeff(Derivative(q(T)))
        for j in range(n):
            Damp_matrix[i + 3, j + 3] = Lagrange_u[i].coeff(Derivative(
                u[i](T), T))

    # Unit force
    Force_vec = np.zeros(3 + n, dtype='complex_').transpose()
    Force_vec[0:2] = M_total

    Eigenmatrix = np.linalg.inv(-omega**2 * (Mass_matrix + A) + 1j *
        omega * (Damp_matrix + B) + (Stiff_matrix + D))

    if load is None:
        Load = np.ones(3 + n, dtype='complex_').transpose()
    else:
        Load = np.ones(3 + n, dtype='complex_').transpose() + load

    disp = np.empty(n + 3, dtype='complex_')
    for i in range(3 + n):
        disp[i] = (Eigenmatrix @ (Force_vec + Load))[i]

    return  disp.real + disp.imag
```

C.6 System G(X) = 0

```
def G(disp, omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m, n
    , A, B, D, dpot_dx, dpot_dz, load=None, max_error=0.01):
    X = Symbol('X')
    Z = Symbol('Z')
    T = Symbol('T')
    U = sympy.Function('U')
    phi = sympy.Function('phi')
    q = sympy.Function('q')
    u = [sympy.Function('u%d' % i) for i in range(n)]

    global delta_h   # change in waterheight from rest position

    d_tmd = disp[3: 3 + n]
    delta_h = 1/n * np.sum(d_tmd)
    count = 0
    for i in range(n):
        if d_tmd[i] > h + delta_h:
            np.delete(d_tmd, [i])
        else:
            count += 1
    delta_h = 1/count * np.sum(d_tmd)

    omega_A = Omega_A(alpha(a, m), B_A0(a, h, m, n), g, m)
    omega_S = Omega_S(beta(a, m), B_S0(a, h, m, n), g, m)

    # Lagrangians:
    La = L(disp, omega, t, rho, a, h, M, g,
           M_tmd, k_tmd, m, n, dpot_dx, dpot_dz)
    Re = R(disp, a, b, h, c_tmd, c_tld, m, n, dpot_dx, dpot_dz)

    Lagrange_phi = La.diff(Derivative(phi(T), T)).diff(T) - La.diff(phi
        (T)) + Re.diff(Derivative(phi(T), T))
    Lagrange_U = La.diff(Derivative(U(T), T)).diff(T) - La.diff(U(T)) +
         Re.diff(Derivative(U(T), T))
    Lagrange_q = La.diff(Derivative(q(T), T)).diff(T) - La.diff(q(T)) +
         Re.diff(Derivative(q(T), T))
    Lagrange_u = [0 for i in range(n)]

    for i in range(n):
        Lagrange_u[i] = La.diff(Derivative(u[i](T), T)).diff(T) - La.
            diff(u[i](T)) + Re.diff(Derivative(u[i](T), T))

    # Mass matrix:

    global Mass_matrix

    Mass_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Mass_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), (T, 2))),
        Lagrange_U.coeff(Derivative(phi(T), (T, 2))), Lagrange_U.coeff(
        Derivative(q(T), (T, 2)))]
    Mass_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), (T, 2)))
        , Lagrange_phi.coeff(Derivative(phi(T), (T, 2))), Lagrange_phi.
        coeff(Derivative(q(T), (T, 2)))]
    Mass_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), (T, 2))),
        Lagrange_q.coeff(Derivative(phi(T), (T, 2))), Lagrange_q.coeff(
        Derivative(q(T), (T, 2)))]

    for i in range(n):
        Mass_matrix[0, 3: i + 3] = Lagrange_U.coeff(Derivative(u[i](T),
             (T, 2)))
        Mass_matrix[1, 3: i + 3] = Lagrange_phi.coeff(Derivative(u[i](T
            ), (T, 2)))
        Mass_matrix[2, 3: i + 3] = Lagrange_q.coeff(Derivative(u[i](T),
             (T, 2)))

        Mass_matrix[i + 3, 0] = Lagrange_u[i].coeff(Derivative(U(T), (T
            , 2)))
        Mass_matrix[i + 3, 1] = Lagrange_u[i].coeff(Derivative(phi(T),
            (T, 2)))
        Mass_matrix[i + 3, 2] = Lagrange_u[i].coeff(Derivative(q(T), (T
            , 2)))
        for j in range(n):
            Mass_matrix[i + 3, j + 3] = Lagrange_u[i].coeff(Derivative(
                u[j](T), (T, 2)))

    # Stiffness matrix:

    global Stiff_matrix

    Stiff_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Stiff_matrix[0, 0:3] = [Lagrange_U.coeff(U(T)), Lagrange_U.coeff(
        phi(T)), Lagrange_U.coeff(q(T))]
    Stiff_matrix[1, 0:3] = [Lagrange_phi.coeff(U(T)), Lagrange_phi.
        coeff(phi(T)), Lagrange_phi.coeff(q(T))]
    Stiff_matrix[2, 0:3] = [Lagrange_q.coeff(U(T)), Lagrange_q.coeff(
        phi(T)), Lagrange_q.coeff(q(T))]

    for i in range(n):
        Stiff_matrix[0, 3: i + 3] = Lagrange_U.coeff(u[i](T))
        Stiff_matrix[1, 3: i + 3] = Lagrange_phi.coeff(u[i](T))
        Stiff_matrix[2, 3: i + 3] = Lagrange_q.coeff(u[i](T))

        Stiff_matrix[i + 3, 0] = Lagrange_u[i].coeff(U(T))
        Stiff_matrix[i + 3, 1] = Lagrange_u[i].coeff(phi(T))
        Stiff_matrix[i + 3, 2] = Lagrange_u[i].coeff(q(T))
        for j in range(n):
            Stiff_matrix[i + 3, j + 3] = Lagrange_u[i].coeff(u[j](T))

    # Damping matrix

    Damp_matrix = np.zeros(((n + 3), (n + 3)), dtype='complex_')

    Damp_matrix[0, 0:3] = [Lagrange_U.coeff(Derivative(U(T), T)),
        Lagrange_U.coeff(Derivative(phi(T), T)), Lagrange_U.coeff(
        Derivative(q(T), T))]
    Damp_matrix[1, 0:3] = [Lagrange_phi.coeff(Derivative(U(T), T)),
        Lagrange_phi.coeff(Derivative(phi(T), T)), Lagrange_phi.coeff(
        Derivative(q(T), T))]
    Damp_matrix[2, 0:3] = [Lagrange_q.coeff(Derivative(U(T), T)),
        Lagrange_q.coeff(Derivative(phi(T), T)), Lagrange_q.coeff(
        Derivative(q(T), T))]

    for i in range(n):
        Damp_matrix[0, 3: i + 3] = Lagrange_U.coeff(Derivative(u[i](T))
            )
        Damp_matrix[1, 3: i + 3] = Lagrange_phi.coeff(Derivative(u[i](T
            )))
        Damp_matrix[2, 3: i + 3] = Lagrange_q.coeff(Derivative(u[i](T))
            )

        Damp_matrix[i + 3, 0] = Lagrange_u[i].coeff(Derivative(U(T)))
        Damp_matrix[i + 3, 1] = Lagrange_u[i].coeff(Derivative(phi(T)))
        Damp_matrix[i + 3, 2] = Lagrange_u[i].coeff(Derivative(q(T)))
        for j in range(n):
            Damp_matrix[i + 3, j +
                        3] = Lagrange_u[i].coeff(Derivative(u[i](T), T)
                            )

    wave_shape = 0
    for i in range(n):
        for j in range(m):
            wave_shape += wave_shape_sA(alpha(a, m)[j], B_A0(a, h, m, n
                )[j]).subs(q(T), sympy.exp(I * omega_A[j] * t)) +
                wave_shape_sS(beta(a, m)[j], B_S0(a, h, m, n)[j]).subs(q
                (T), sympy.exp(I * omega_S[j] * t)) + h + delta_h

    if load is None:
        Load = np.zeros(3 + n, dtype='complex_').transpose()
        Load[0:2] = M_total  # Normalized unit load
    else:
        Load = load
        Load[0:2] += M_total  # Normalized unit load

    eom = (-omega**2 * (Mass_matrix + A) @ disp + 1j * omega * (
       Damp_matrix + B) @ disp + (Stiff_matrix + D) @ disp)

    return (eom.real + eom.imag - Load)
```

C.7 Iteration

```
def iteration(omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m,
    n, A, B, D, max_error= 0.01, max_iterations= 25, load= None):
    # Finding initial interval:
    dpot_dx0= dpotential_dx0(a, h, m, n)
    dpot_dz0= dpotential_dz0(a, h, m, n)
    d0= Response0(omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld
        , m, n, A, B, D, dpot_dx0, dpot_dz0, load= None)
    x0= copy.copy(d0)

    # Create Jacobian:
    Jac= np.zeros((n + 3, n + 3))

    eps = np.zeros((n + 3, n + 3)) # The stepsize for the approximate
        derivative
    for i in range(n + 3):
        eps[i][i] = max_error * 0.01

    solution = False
    for j in range(2):
```

```
    if solution == True:
        break
    elif j > 0:
        x0 = -copy.copy(d0)
    f0= G(x0, omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld
       , m, n, A, B, D, dpotential_dx(x0, a, h, m, n),
       dpotential_dz(x0, a, h, m, n), load=None)
    f_dic= {}
    for dx in range(n):
        f_dic['f' + str(dx)] = G(x0 + eps[dx + 3], omega, t, rho, a
            , h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m, n, A, B, D,
            dpotential_dx(x0 + eps[dx + 3], a, h, m, n),
            dpotential_dz(x0 + eps[dx + 3], a, h, m, n), load=None)
        for df in range(n + 3):
             Jac[df][dx + 3] = (f_dic['f' + str(dx)][df] - f0[df])/
                 eps[df][df]
    delta_x = np.linalg.pinv(Jac) @ f0
    r = 1 # damping factor
    _t = x0 - r * delta_x
    f_t = G(_t, omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd,
       c_tld, m, n, A, B, D, dpotential_dx(_t, a, h, m, n),
       dpotential_dz(_t, a, h, m, n), load=None)
    while (f_t > disp2).any() :
        r = 1/2 * r
        _t = x0 - r * delta_x
        f_t = G(_t, omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd,
             c_tld, m, n, A, B, D, dpotential_dx(_t, a, h, m, n),
            dpotential_dz(_t, a, h, m, n), load=None)
    delta_x = x0 - _t
    f1 = f_t
    x1 = x0 - r * delta_x
    for dx in range(n):
        f_dic['f' + str(dx)] = G(x1 + eps[dx + 3], omega, t, rho, a
            , h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m, n, A, B, D,
            dpotential_dx(x1 + eps[dx + 3], a, h, m, n),
            dpotential_dz(x1 + eps[dx + 3], a, h, m, n), load=None)
        for df in range(n + 3):
             Jac[df][dx + 3] = (f_dic['f' + str(dx)][df] - f1[df])/
                 eps[df][df]
    delta_x = np.linalg.pinv(Jac) @ f1
    r = 1 # damping factor
    _t = x1 - r * delta_x
    f_t = G(_t, omega, t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd,
       c_tld, m, n, A, B, D, dpotential_dx(_t, a, h, m, n),
       dpotential_dz(_t, a, h, m, n), load=None)
    while (f_t > disp2).any() :
        r = 1/2 * r
        f_t = G(x1 - r * delta_x, omega, t, rho, a, h, M, g, M_tmd,
             c_tmd, k_tmd, c_tld, m, n, A, B, D, dpotential_dx(x1 -
            r * delta_x, a, h, m, n), dpotential_dz(x1 - r * delta_x
            , a, h, m, n), load=None)
    delta_x = x1 - _t
```

```
        f1 = f_t
        x0 = copy.copy(x1)
        x2 = x1 - r * delta_x
        for i in range(max_iterations):
            if (abs((x2 - x1)) <= max_error).all() or ((abs(f1) <
               max_error).all()):
                solution = True
                response = 1/2 * (x1 + x2)
                break
            # If there is a cycle between two points:
            elif (all(x0 == x2)):
                break
            elif any(np.isnan(x2)):
                break
            else:
                f0= copy.copy(f1)
                x0 = copy.copy(x1)
                x1 = copy.copy(x2)
                f1= G(x1, omega, t, rho, a, h, M, g, M_tmd, c_tmd,
                   k_tmd, c_tld, m, n, A, B, D, dpotential_dx(x1, a, h,
                    m, n), dpotential_dz(x1, a, h, m, n), load=None)
                # Update the Jacobian
                for dx in range(n):
                    f_dic['f' + str(dx)] = G(x1 + eps[dx + 3], omega, t
                        , rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m
                        , n, A, B, D, dpotential_dx(x1 + eps[dx + 3], a,
                         h, m, n), dpotential_dz(x1 + eps[dx + 3], a, h,
                         m, n), load=None)
                    for df in range(n + 3):
                        Jac[df][dx + 3] = (f_dic['f' + str(dx)][df] -
                           f1[df])/ eps[df][df]
                delta_x = np.linalg.pinv(Jac) @ f1
                r = 1 # damping factor
                _t = x1 - r * delta_x
                f_t = G(_t, omega, t, rho, a, h, M, g, M_tmd, c_tmd,
                   k_tmd, c_tld, m, n, A, B, D, dpotential_dx(_t, a, h,
                    m, n), dpotential_dz(_t, a, h, m, n), load=None)
                 while (f_t > f1).any():
                    r = 1/2 * r
                    _t = x1 - r * delta_x
                    f_t = G(_t, omega, t, rho, a, h, M, g, M_tmd, c_tmd
                        , k_tmd, c_tld, m, n, A, B, D, dpotential_dx(_t,
                         a, h, m, n), dpotential_dz(_t, a, h, m, n),
                        load=None)
                delta_x = x1 - _t
                f1 = f_t
                x2 = x1 - r * delta_x

    if solution == False:
        response= d0

    data["G(omega=" + str(omega) + ", t=" + str(t) + "c_tmd=" + str(
            c_tmd) + "k_tmd=" + str(k_tmd) + "c_tld=" + str(c_tld) + "h_tld
            =" + str(h) + ")"] = response.tolist()

    # Saving the wave shape
    q = sympy.Function('q')
    omega_A = Omega_A(alpha(a, m), B_A(response, a, h, m, n), g, m)
    omega_S = Omega_S(beta(a, m), B_S(response, a, h, m, n), g, m)
    wave_shape = 0
    for j in range(m):
        wave_shape += wave_shape_sA(alpha(a, m)[j], B_A(response, a, h,
             m, n)[j]).subs(q(T), sympy.exp(I * omega_A[j] * t)) +
            wave_shape_sS(beta(a, m)[j], B_S(response, a, h, m, n)[j]).
            subs(q(T), sympy.exp(I * omega_S[j] * t))

    width = np.linspace(-a, a, 50)
    w = []
    for j in range(len(width)):
        w = np.append(w, int(sympy.re((wave_shape - h - delta_h).subs(X
            , width[j]).evalf()) + sympy.im((wave_shape - h - delta_h).
            subs(X, width[j]).evalf())))

    data['wave_shape(Omega:' + str(omega) + ', t:' + str(t) + "c_tmd="
        + str(c_tmd) + "k_tmd=" + str(k_tmd) + "c_tld=" + str(c_tld) + "
        h_tld=" + str(h) + ')'] = w.tolist()

    if solution == False:
        print('No covergence reached, omega=' + str(omega) + 't=' + str
            (t))
    else:
        print('rank:', rank, 'Convergence! response=', response)

    return response.real + response.imag

# Performance index
def J(t, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m, n, A, B, D,
    omega_0, omega_end, omega_num, max_error=0.01, max_iterations=25,
    load=None):

    omega = np.linspace(omega_0, omega_end, omega_num)
    S0 = 1   # white noise spectrum (constant)

    H_U = np.zeros(len(omega))
    H_phi = np.zeros(len(omega))

    try:  # if the full response matrix already exsists, it doesn't
        have to be calculated again
        data["responses(Omega=(" + str(omega_0) + "," + str(omega_end)
            + "," + str(omega_num) + "), t=" + str(t) + ", c_tmd=" + str
            (c_tmd[0]) + ", k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(
            c_tld) + "h_tld=" + str(h) + ")"]
    except KeyError:
        data["responses(Omega=(" + str(omega_0) + "," + str(omega_end)
            + "," + str(omega_num) + "), t=" + str(t) + ", c_tmd=" + str
            (c_tmd[0]) + ", k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(
            c_tld) + "h_tld=" + str(h) + ")"] = []

    dis = np.zeros((len(omega), n + 3))

    for i in range(len(omega)):
        try: # if any response array already exsists, it doesn't have
            to be calculated again
             data["responses(Omega=(" + str(omega_0) + "," + str(
                omega_end) + "," + str(omega_num) + "), t=" + str( t) +
                ", c_tmd=" + str(c_tmd[0]) + ", k_tmd=" + str(k_tmd[0])
                + "c_tld=" + str(c_tld) + "h_tld=" + str(h) + ")"][i]
        except (KeyError, IndexError):
            try:
                dis[i] = iteration(omega[i], t, rho, a, h, M, g, M_tmd,
                    c_tmd, k_tmd, c_tld, m, n, A, B, D, max_error=0.01,
                    max_iterations=25, load=None)
                data["responses(Omega=(" + str(omega_0) + "," + str(
                   omega_end) + "," + str(omega_num) + "), t=" + str( t
                   ) + ", c_tmd=" + str(c_tmd[0]) + ", k_tmd=" + str(
                   k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" + str(h
                   ) + ")"].append(dis[i].tolist())
            except np.linalg.LinAlgError:
                print('singular:', i)
                data["responses(Omega=(" + str(omega_0) + "," + str(
                   omega_end) + "," + str(omega_num) + "), t=" + str( t
                   ) + ", c_tmd=" + str(c_tmd[0]) + ", k_tmd=" + str(
                   k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" + str(h
                   ) + ")"].append(dis[i - 1].tolist())

    for i in range(len(omega)):
        H_U[i] = dis[i][0]
        H_phi[i] = dis[i][1]

    data["H_U(Omega=(" + str(omega_0) + "," + str(omega_end) + "," +
       str(omega_num) + "), t=" + str(t) + ", c_tmd=" + str(c_tmd[0]) +
        ", k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" +
        str(h) + ")"] = H_U.tolist()
    data["H_Phi(Omega=(" + str(omega_0) + "," + str(omega_end) + "," +
       str(omega_num) + "), t=" + str(t) + ", c_tmd=" + str(c_tmd[0]) +
        ", k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" +
        str(h) + ")"] = H_phi.tolist()

    # optimum for U and phi wanted:
    E_U = S0 * np.trapz(abs(H_U), x=omega)
    # the horizontal response of the hub is most important here
    E_phi = S0 * np.trapz(abs(H_phi * h_hub), x=omega)

    data["J(Omega=(" + str(omega_0) + "," + str(omega_end) + "," + str(
       omega_num) + "), t=" + str(t) + ", c_tmd=" + str(c_tmd[0]) + ",
       k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" +
            str(h) + ")"] = E_U + E_phi

        return E_U + E_phi


omega_0 = 10**-5
omega_end = 3
omega_num = 100


# One parameter at a time:
results = J(time, rho, a, h, M, g, M_tmd, c_tmd, k_tmd, c_tld, m, n, A,
     B, D, omega_0, omega_end, omega_num, max_error=0.01, max_iterations
    =25, load=None)

comm.Barrier()

# Sensitivity of any parameter (here only shown for c_tmd):

c_tmd_sensitive = comm.scatter(np.linspace(c_tmd[0] - 0.1 * c_tmd[0],
    c_tmd[0] + 0.1 * c_tmd[0], 25), root=0)

# k_tmd_sensitive = comm.scatter(np.linspace(k_tmd[0] - 0.1 * k_tmd[0],
     k_tmd[0] + 0.1 * k_tmd[0], 25), root=0)
# h_sensitive= comm.scatter(np.linspace(h * .95, h*1.05, 25))
# c_tld_sensitive = comm.scatter(np.linspace(c_tld[0] - 0.1 * c_tld[0],
     c_tmd[0] + 0.1 * c_tld[0], 25), root=0)

results = J(time, rho, a, h, M, g, M_tmd, c_tmd_sensitive * np.ones(n),
     k_tmd, c_tld, m, n, A, B, D, omega_0, omega_end, omega_num,
    max_error=0.01, max_iterations=25, load=None)

comm.Barrier()


J = comm.reduce(results, op=MPI.SUM, root=0)

if rank == 0:
    for i in range(1, size, 1):
        data_i = comm.recv(source=i, tag=11)
        for key in data_i:
            if key in data:
                data[key] = data[key]
            else:
                data[key] = data_i[key]

    data["J_total(" + ", c_tmd=" + str(c_tmd[0]) + ", k_tmd=" +
         str(k_tmd[0]) + "c_tld=" + str(c_tld) + "h_tld=" + str(h) + ")
            "] = J / size
    jdata = json.dumps(data, indent=4)
    f = open("model3.json", "w")
    f.write(jdata)
    f.close()
    print("J =" + str(J / size), "for c_tmd=" +
          str(c_tmd[0]) + ", k_tmd=" + str(k_tmd[0]) + "c_tld=" + str(
             c_tld) + "h_tld=" + str(h) + ")")
else:
    data_i = data
    comm.send(data_i, dest=0, tag=11)
```

Bibliography

[1] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 1)*. `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1`. 2022.

[2] Song An and Odd M Faltinsen. "An experimental and numerical study of heave added mass and damping of horizontally submerged and perforated rectangular plates". In: *Journal of Fluids and Structures* 39 (2013), pp. 87–101.

[3] Gunjit Bir and Jason Jonkman. "Aeroelastic instabilities of large offshore and onshore wind turbines". In: *Journal of Physics: Conference Series*. Vol. 75. 1. IOP Publishing. 2007, p. 012069.

[4] Zhifu Cao et al. "Dynamic sensitivity-based finite element model updating for nonlinear structures using time-domain responses". In: *International Journal of Mechanical Sciences* 184 (2020), p. 105788.

[5] YH Chong and M Imregun. "Coupling of non-linear substructures using variable modal parameters". In: *Mechanical Systems and Signal Processing* 14.5 (2000), pp. 731–746.

[6] Christophe Coudurier, Olivier Lepreux, and Nicolas Petit. "Modelling of a tuned liquid multi-column damper. Application to floating wind turbine for improved robustness against wave incidence". In: *Ocean Engineering* 165 (2018), pp. 277–292.

[7] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.

[8] Mojtaba Ezoji, Naser Shabakhty, and Longbin Tao. "Hydrodynamic damping of solid and perforated heave plates oscillating at low KC number based on experimental data: A review". In: *Ocean Engineering* 253 (2022), p. 111247.

[9] Odd Faltinsen. *Sea loads on ships and offshore structures*. Vol. 1. Cambridge university press, 1993.

[10] Odd M Faltinsen and Alexander N Timokha. "An adaptive multimodal approach to nonlinear sloshing in a rectangular tank". In: *Journal of Fluid Mechanics* 432 (2001), pp. 167–200.

[11] JV Ferreira and DJ Ewins. "Nonlinear receptance coupling approach based on describing functions". In: *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*. Citeseer. 1996, pp. 1034–1040.

[12] Yozo Fujino et al. "Tuned liquid damper (TLD) for suppressing horizontal motion of structures". In: *Journal of Engineering Mechanics* 118.10 (1992), pp. 2017–2030.

[13] H Gao, KCS Kwok, and B Samali. "Optimization of tuned liquid column dampers". In: *Engineering structures* 19.6 (1997), pp. 476–486.

[14] Hanbin Gu et al. "Drag, added mass and radiation damping of oscillating vertical cylindrical bodies in heave and surge in still water". In: *Journal of Fluids and Structures* 82 (2018), pp. 343–356.

[15] Minho Ha and Cheolung Cheong. "Pitch motion mitigation of spar-type floating substructure for offshore wind turbine using multilayer tuned liquid damper". In: *Ocean Engineering* 116 (2016), pp. 157–164.

[16] William W Hager. *Applied numerical linear algebra*. Vol. 87. SIAM, 2022.

[17] Matthew Hall, Brad Buckham, and Curran Crawford. "Evaluating the importance of mooring line model fidelity in floating offshore wind turbine simulations". In: *Wind energy* 17.12 (2014), pp. 1835–1853.

[18] Er-Ming He, Ya-Qi Hu, and Yang Zhang. "Optimization design of tuned mass damper for vibration suppression of a barge-type offshore floating wind turbine". In: *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment* 231.1 (2017), pp. 302–315.

[19] Michael Hintermüller and Room John-von Neumann Haus. "Nonlinear optimization". In: *Vorlesungsskript Universität Berlin* (2010).

[20] Yinlong Hu et al. "Load mitigation for a barge-type floating offshore wind turbine via inerter-based passive structural control". In: *Engineering Structures* 177 (2018), pp. 198–209.

[21] Sen Huang. "Dynamic analysis of assembled structures with nonlinearity". PhD thesis. Imperial College London, 2007.

[22] Jason Jonkman et al. *Definition of a 5-MW reference wind turbine for offshore system development.* Tech. rep. National Renewable Energy Lab.(NREL), Golden, CO (United States), 2009.

[23] Taner Kalaycıoğlu and H Nevzat Özgüven. "FRF decoupling of nonlinear systems". In: *Mechanical Systems and Signal Processing* 102 (2018), pp. 230–244.

[24] Taner Kalaycıoğlu and H Nevzat Özgüven. "Nonlinear structural modification and nonlinear coupling". In: *Mechanical Systems and Signal Processing* 46.2 (2014), pp. 289–306.

[25] Alberto Lago, Dario Trabucco, and Antony Wood. "Chapter 4 - An introduction to dynamic modification devices". In: *Damping Technologies for Tall Buildings.* Ed. by Alberto Lago, Dario Trabucco, and Antony Wood. Butterworth-Heinemann, 2019, pp. 107–234. ISBN: 978-0-12-815963-7. DOI: `https://doi.org/10.1016/B978-0-12-815963-7.00004-X`. URL: `https://www.sciencedirect.com/science/article/pii/B978012815963700004X`.

[26] Chien-Liang Lee et al. "Optimal design theories and applications of tuned mass dampers". In: *Engineering structures* 28.1 (2006), pp. 43–53.

[27] Yuchun Li and Zhuang Wang. "An approximate analytical solution of sloshing frequencies for a liquid in various shape aqueducts". In: *Shock and Vibration* 2014 (2014).

[28] JS Love and MJ Tait. "A preliminary design method for tuned liquid dampers conforming to space restrictions". In: *Engineering structures* 40 (2012), pp. 187–197.

[29] Nicholas F Morris. "The use of modal superposition in nonlinear dynamics". In: *Computers & Structures* 7.1 (1977), pp. 65–72.

[30] Robert E Nickell. "Nonlinear dynamics by mode superposition". In: *Computer Methods in Applied Mechanics and Engineering* 7.1 (1976), pp. 107–129.

[31] Mehmet Bülent Özer, H Nevzat Özgüven, and Thomas J Royston. "Identification of structural non-linearities using describing functions and the Sherman–Morrison method". In: *Mechanical Systems and Signal Processing* 23.1 (2009), pp. 30–44.

[32] Hong-Duc Pham et al. "Dynamic modeling of nylon mooring lines for a floating wind turbine". In: *Applied Ocean Research* 87 (2019), pp. 1–8.

[33] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics.* Vol. 37. Springer Science & Business Media, 2010.

[34] Fahim Sadek et al. "A method of estimating the parameters of tuned mass dampers for seismic applications". In: *Earthquake Engineering & Structural Dynamics* 26.6 (1997), pp. 617–635.

[35] Hugo Ramon Elizalde Siller. "Non-linear modal analysis methods for engineering structures". PhD thesis. University of London, 2004.

[36] LM Sun et al. "Modelling of tuned liquid damper (TLD)". In: *Journal of Wind Engineering and Industrial Aerodynamics* 43.1-3 (1992), pp. 1883–1894.

[37] M.J. Tait. "Modelling and preliminary design of a structure-TLD system". In: *Engineering Structures* 30.10 (2008), pp. 2644–2655. ISSN: 0141-0296. DOI: `https://doi.org/10.1016/j.engstruct.2008.02.017`. URL: `https://www.sciencedirect.com/science/article/pii/S0141029608000667`.

[38] Omer Tanrikulu et al. "Forced harmonic response analysis of nonlinear structures using describing functions". In: *AIAA journal* 31.7 (1993), pp. 1313–1320.

[39] A.H. Techet. *2.016 Hydrodynamics*. 2005.

[40] A.H. Techet. *Derivation of added mass around a sphere*. 2005.

[41] OG Tietjens. *Applied Hydro-and Aerodynamics: Based on Lectures of L. Prandtl*. McGraw-Hill, 1934.

[42] Tianyu Wang et al. "Seismic response prediction of structures based on Runge-Kutta recurrent neural network with prior knowledge". In: *Engineering Structures* 279 (2023), p. 115576.

[43] Jiawen Yang, EM He, and YQ Hu. "Dynamic modeling and vibration suppression for an offshore wind turbine with a tuned mass damper in floating platform". In: *Applied Ocean Research* 83 (2019), pp. 21–29.

[44] Alberto Zasso et al. "Wind induced dynamics of a prismatic slender building with 1: 3 rectangular section". In: *Proceedings of the BBAA VI International Colloquium on Bluff Bodies Aerodynamics & Applications*. 2008.

[45] Yangming Zhang, Xiaowei Zhao, and Xing Wei. "Robust structural control of an underactuated floating wind turbine". In: *Wind Energy* 23.12 (2020), pp. 2166–2185.

[46] Zili Zhang, Biswajit Basu, and Søren RK Nielsen. "Real-time hybrid aeroelastic simulation of wind turbines with various types of full-scale tuned liquid dampers". In: *Wind Energy* 22.2 (2019), pp. 239–256.