Floating Spar Optimization

A constraint set investigation for the use of simulated annealing for substructure optimization

Torsten Giles Pietersz

Floating Spar Optimization

A constraint set investigation for the use of simulated annealing for substructure optimization

by

Torsten Giles Pietersz

Academic Supervisor:	O. Colomés
Academic Supervisor:	Z. Gao
Company Supervisor:	M. Livingstone
Project Duration:	February, 2022 - November, 2022
TU Delft faculty:	Faculty of Mechanical Engineering, Delft
NTNU department:	Department of Marine Technology, Trondheim

Cover: Hywind Pilot Site by Aleseund University College [5]

Preface

This thesis attempts to develop a design algorithm for a floating substructure of a wind turbine, the algorithm is a non-gradient-based optimizer which optimizes for mass. The research is carried out through the TU-Delft, NTNU and DNV. The latter is highly involved in the requirements set for the tool itself.

The author would like to acknowledge Zhen Gao, Michael Livingstone and Oriol Colomés. All three of you have significantly impacted my development during this thesis. Thank you for your time, expertise and, most of all, your patience.

Torsten Giles Pietersz
Delft, December 2022

Summary

This report serves as a thesis for the European Wind Energy Master's program. The project is set up in cooperation with DNV, aiming to develop a preliminary design tool for their clientele. Ideally, the tool will become a part of the greater software package offered by DNV called 'Renewables Architect' (RA). This framework has been designed specifically for Multi-disciplinary Design, Analysis and Optimization (MDAO).

The thesis consists of four chapters; 'Introduction', 'Numerical load and response', 'Optimization' and 'Results'. The introduction begins with background information on what role floating wind will play in the offshore wind industry and why this is a relevant topic. A general overview of floaters is presented with a comparison of the advantages of each type of floater. As well a chronological review of approachs to preliminary design. The numerical load and response chapter is divided into theory and methodology. Theory begins with the calculation of forcing and starts with hydrodynamics forcing. Hydrodyamic forcing is based on wave kinematics, as such airy wave theory and regular wave kinematics are the first to be explored. After this, the superposition principle is introduced, which leads to irregular wave kinematics. Both Morison's equations and the diffraction approach of Maccamy Fuchs are presented for calculating the hydrodynamic forces. Next, aerodynamics are reviewed in 2.1.1, which begins with the simple principles of lift force on a blade and goes into blade element momentum theory. Afterwhich wind kinematics are discussed, covering the Kaimal spectrum and variations of its usage. The following part of the numerical load and response theory, is the Fatigue damage. Where using Miner's rule, two methods for damage calculation are presented; rainflow counting for time domain simulations and Dirlik's method for frequency domain simulations. After the theory is discussed, the numerical load and response methodology is discussed. The methodology begins with an exploration of the considered site. This will form the basis for fatigue damage calculations' environmental load cases later. The Norway North Sea site 15 is used as a reference location. A joint conditional distribution calculates the frequency of occurrence for the relevant environmental condition set. Next the approach to modelling the floating wind turbine is discussed in 2.2.2. The structural model section begins with a detailed review of the 15MW reference tower and turbine. This section accounts for any calculations made on the turbine and tower. After the spar floater is discussed, some of the spar's design is based on the values calculated from the turbine and tower. Finally, the three elements, spar, turbine, and tower, are combined into one global coordinate system in the whole system class, where more calculations are presented. With the model set up, the next section in the methodology chapter is the forcing calculation on the model. The decision is made to model in the time domain. The forcing section discusses how wind and wave kinematics are generated using Kaimal and Jonswap spectra, respectively. A logarithmic depth distribution is used for wave kinematics for computational efficiency. The hydrodynamic forcing is calculated with Morison's equations. For aerodynamic forcing, an average thrust coefficient is introduced that allows for the calculation of thrust based on the blade-swept area. The response calculation is then presented in the methodology chapter, where the equations of motion are set up in matrix form, and terms in all of the matrices are then derived for the mass, added mass, stiffness and damping matrix. The stiffness matrix considers hydrodynamic stiffness and mooring stiffness. The terms in the damping matrix are derived from aerodynamic and hydrodynamic damping. Hydrodynamic damping only considers viscous damping terms. The model is then tested for a set of regular and irregular wave and wind conditions to analyze the system's behaviour. The fatigue calculation is in the next section, where the substructure is divided into welded areas that will be investigated for lifetime fatigue. The bending stress analysis is presented for the cross-sectional stress of the substructure.

The Optimization chapter begins with an exploration of the theory behind optimization. This is a broad review of both gradient and gradient-free methods. First, considering the gradient-based methods, steepest descent, conjugate gradient and quasi-newton approach are all discussed in this section. Gradient-free methods have a broader range as they are based on theoretically infinite heuristic logic.

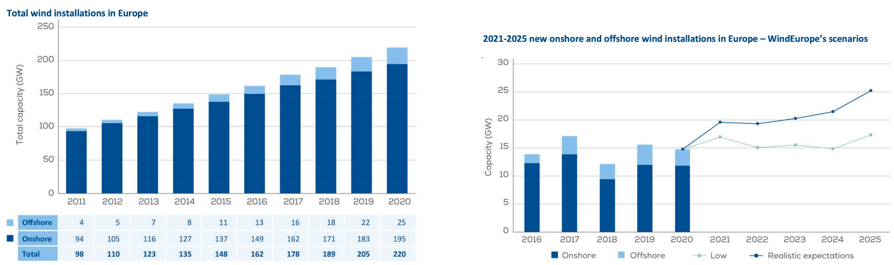
As such, two algorithms are described; Genetic Algorithms and Simulated Annealing. The methodology chapter covers how the theory was implemented and motivates design and simulation choices. The decision is made to use the simulated annealing algorithm, which is further investigated in the methodology section. The methodology section explains how each step in the algorithm is performed and investigates the effects of temperature range on the metropolis criteria. The stopping criteria are also introduced together with their respective parameters. A simple objective function is used to run through the optimization algorithm to increase understanding of the algorithm. The optimization algorithm is then used on a set of test functions known to be challenging to solve. Finally, the optimization problem for this research is formulated. The design space is visualized and compared to the most similar test function. After which, the constraints used by the optimizer are presented. The first constraint set consists of logical design constraints determined by the spar geometry. After which, the constraint set also includes limitations on the extreme response in time domain simulation. In the results chapter, it is found that this slows down the process so much that it is not feasible to include fatigue damage in the constraint function. Furthermore, it is found that global fatigue damage wouldn't be a governing constraint. The results chapter discusses the importance of mooring stiffness in this optimization. Present the results from constraining the optimizer with the time domain response and compare the fatigue lifetime of the optimal designs. It is found that global fatigue is not a governing design consideration. Finally, a discussion is presented where the work is critiqued, and recommendations are made for further work.

Contents

1
Introduction

1.1. Floating Wind

There is an increasing worldwide demand for electricity and a growing concern about the environmental impact of energy exploitation on the earth. That increasing concern has led many governments to express clear goals for what proportion of energy production from renewable sources. One of the consequences has been a massive increase in the production, deployment and operation of wind turbines [1]. The majority of wind installations installed in Europe are onshore. Due to stronger resources and increased space availability, it is advantageous to exploit wind resources offshore. This is not a recent development. The first offshore wind farm (OWF) was built in 1991 off the coast of Denmark. Since then, many OFWs have been constructed all around the globe [26]. The expectation is that the number of offshore installations will only increase [42].



(a) Historic onshore and offshore installed power in Europe. *Source: Wind Europe* [42].

(b) Forecast installed wind energy in Europe. *Source: Wind Europe* [42].

When considering the production of an offshore wind turbine, it is essential to pay attention to the support structure. This foundation will generally make up around 35% of the cost of the total wind farm [25]. The most common support structure for offshore wind turbines is the monopile. Representing 80% of the installed wind turbines, the monopile is dominating [25]. In general, monopiles are designed for depths of around 35 metres. For example, wind farm projects developed before 2020; 90% of the 9 GW Dogger Bank development is below 35m, and about 50% of the 4 GW Hornsea development is below 40m water depth [40]. Space-framed structures like tripods and lattice frames (e.g. jackets) are used in water depths of around 50 metres [53]. According to Jonkman and Matha (2011), there is a need to go into deeper water to harness much of the vast offshore resources worldwide. This sentiment is a logical conclusion when coastal bathymetry worldwide is considered, where the ocean shelves near the coast quickly drop to depths above 60 metres. Going further offshore also solves the divisive issue of the aesthetic nature of wind turbines, as when wind farms are far offshore, they can not be seen from the coast. These extreme depths make the majority of bottom fixed support structures infeasible, and floating support structures will become the solution [38].

1.2. Substructures General Overview

The goal of the floater is to offer support and stability to the wind turbine atop. In general, the mechanisms to do this fall into one of three categories; mooring-, waterplane- and ballast stabilized.

- **TLP**
  The *tensioned leg platform* is stabilized by a mooring mechanism. The mooring lines underneath the support structure are tensioned to create a stabilizing moment when the structure is angled. The tension is created by the high amount of buoyancy of the platform.
- **Semi Submersible platform**
  The *Semi submersible platform* is stabilized by the restoring moment caused by the significant second moment of inertia. They typically consist of multiple legs at a distance from the turbine. These legs cover a relatively large waterplane area, adding to the moment of inertia.
- **Barge type platform**
  The Barge is an extension of semi-submersible type substructures. A barge is typically a shallow drafted concrete or steel floating hull with a shallow draft. It uses a large water plane area to achieve stability. An example is the moon pool barge characterized by a square and ring-shaped floating platform with a central pool that absorbs the wave loads [15].
- **Spar Type** The *Spar type* support structure is stabilized by adding a ballast at the bottom of the support structure. This counterweights the pitching and rolling response that the structure may encounter.

The spar, semi-submersible, and TLP-type substructures have been reviewed by Liu (2016) [50] who defined the relative advantages as shown in table 1.1. From table 1.1 it becomes clear the spar has disadvantages when considering construction and installation. This comes from the difficulty of towing out a spar type. Furthermore, due to their size, the substructure spars come with more challenging logistics for fabrication and transportation. Spars are easy to anchor and show minor sensitivity to wave motions.

Table 1.1: Comparison floating foundation concepts *Source: Liu (2016)*. +,0,- stand for advantageous, neutral and disadvantaged, respectively. [50]

+ Relative advantage 0 Neutral -Relative disadvantage	TLP	Spar	Semi-submersible
Pitch stability	Mooring	Ballast	Buoyancy
Natural periods	+	0	−
Coupled motion	+	0	−
Wave sensitivity	0	+	−
Turbine weight	0	−	+
Moorings	+	−	−
Anchors	−	+	+
Construction and installation	−	−	+
Maintenance	+	0	−

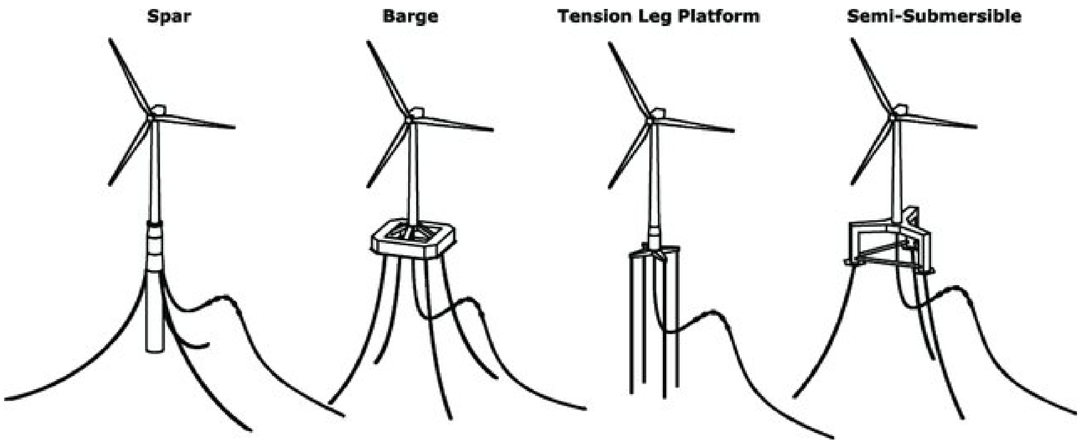


Figure 1.2: Schematic of the four floater types. *Source: Matti Scheu (2018)* [74].

1.3. Preliminary Design

For decades floating support structures have been commonly used in the oil and gas industry. The first mobile submersible unit was commissioned in 1948 [33]. However, offshore wind is a developing technology; there are currently three operational farms: Hywind Scotland, Kincardine (Scotland) and Windfloat Atlantic (Portugal). Kincardine, the largest one with 50 MW capacity, was commissioned on the 19th of October 2021. Hywind Scotland is the first floating wind farm and remains the only one to implement spar-type floaters. Choosing and designing the support structure has yet to be standardized. In 2002, van Hees proposed a two-stage approach which aimed at determining what type of support structure to use by comparing the dimensions, stability and resonance periods of 8 substructures. The preliminary sizing was done using QUESTOR, an in-house software, which uses the stability and maximum inclining moment with a maximum dynamic inclination angle of 10°. The most suited structure (SemiSub: Tri-Floater) is then structurally, hydrodynamically and hydrostatically analyzed to refine further its sizing [12]. Wayman (2006) took a similar two-stage approach but with only four types of substructures. The preliminary sizing began with the pitch stability criterion, after which the cost is estimated, and the substructures are compared. [81]. Wayman then chooses the cheapest solutions to perform a dynamic and static stability check, to minimize the waves' action on the structure. The aim is to reduce the production cost of wind turbines. Therefore the cost is the most important contribution to the comparison. Lefebvre(2012) nearly copies the approach of Wayman. Still, instead of basing the maximum moment on the air pressure of the dry part of the structure, as Van Hees did, Lefebvre bases the preliminary design on the moment created by the maximum amount of thrust produced by the rotor [45].

Another possibility for sizing the support structure is using a design algorithm that in some way optimizes the performance of the wind turbine [22],[70],[31],[47]. This style of design can cover multiple facets of the floating wind turbine system. From optimizing the mooring of the structure [11] to optimizing the controller, tower and floater at the same time [31]. When optimizing multiple systems at the same time, this is called multidisciplinary design optimization. In 2014 a review was done about the design optimization of wind turbine support structures [61]; a prominent contribution to the conclusion was the limitations set by the computational expense for modelling the wind turbine structure. Muskulus advised to start looking at frequency domain modelling and to purely use design optimization of wind turbine support structures in the preliminary design phase. An opposing thought would be to further limit the model's complexity for the time domain simulation.

The remainder of this thesis will focus on spar-type floaters as shown in 1.3 . Not only are they well-documented for the oil & gas industry, but they are also well-researched for offshore floating wind systems. In a 2020 Ocean Engineering publication, the spar-type floater was also found to be the most cost-effective at deeper draughts [36]. The simple nature of spars has the added benefit of simplifying the hydrodynamic calculations.

Figure 1.3: Spar Floating wind turbine: Hywind prototype Statoil Hydro. Image property of Siemens, taken from NS energy publication [66]

1.4. Research Question

This thesis aims to combine optimization and numerical load response modeling for the preliminary design of a spar type substructure. For the investigation of combining optimization with the preliminary design of spar-type substructure, this thesis will be based around the development of a response model and the development of a non-gradient optimizer. Finally, these two will be combined to investigate constraint sets relevant to this specific engineering problem. Investigating constraint sets for a non-gradient-based optimizer for the optimization of the spar-type substructure of a floating wind turbine will give insights into the applicability of the simulated annealing algorithm on such engineering problems. And will eventually answer the following research question:

- What are viable constraint sets when using a simulated annealing algorithm for the optimization of a spar-type floating substructure

Furthermore, some subquestions will be answered as well:

- What is the role of mooring stiffness in the optimization
- What constraints govern the optimal design of a spar-type substructure

2

Numerical load and response

This chapter will cover the numerical load and response calculations of this research. Beginning by going over the theory covered regarding these calculations, after which the methodology used will be introduced, where the model will be tested in a set of predictable conditions to investigate the response and comment on the validity of the developed model.

2.1. Theory: Numerical loads and response

This section is meant to review the relevant literature investigated for the numerical load and response part of this research. The theory behind forcing on a floating wind turbine is discussed. This includes wind- and wave-kinematics and dynamics. Discussing Morrison's equations, spectral approaches and BEM theory. Fatigue is discussed using ideas for both time domain and frequency domain simulations.

2.1.1. Forcing

Forcing on a floating wind turbine comes from the exposure to wind and wave loads. The theory behind these two is discussed in the next few subsections. In principle the approach to forcing calculation is no different than that of a bottom founded substructure. However for floating offshore wind turbines the hydrodynamic loads have a more significant effect on the response characteristics.

Hydrodynamics

Wave Kinematics Hydrodynamics play a big part in the behaviour of an offshore floating wind turbine. Hydrodynamic loading is caused by the interaction between sea and floating wind turbine. As such the flow of the fluid needs to be modeled. A common approach to calculating flow is using airy wave theory. An extensive history of water wave theory is covered by Craik (2004)[16].

Regular Wave Kinematics Airy wave theory or linear wave theory is commonly used to model the wave kinematics of a regular wave. The theory uses potential flow to describe the motion of gravity waves. Airy wave theory was first correctly publicized in the 19th century and presumes the fluid to be inviscid, irrotational and incompressible. Furthermore the assumption is made that there is a uniform mean flow depth. The velocity potential is defined as:

$$\phi = -\frac{\omega H}{2k}\frac{\cosh k(z+h)}{\sinh kh}\sin(\omega t - kx) \tag{2.1}$$

Where ω is the angular velocity of the wave, H is the wave height taken from lowest to highest point, k is the wave number. h is the depth at still water level (SWL), z is the depth being considered, t the time and x the position considered on the x-axis. Equation 2.1 is a potential equation, meaning that it's derivative to any spatial direction correlates to the velocity in that direction. As such the wave velocity and acceleration in x direction, u and $\dot{u}$ respectively, can be defined as:

$$u = \frac{\partial \phi}{\partial x} = \frac{\omega H}{2}\frac{\cosh k(z+h)}{\sinh kh}\cos(\omega t - kx) \tag{2.2}$$

$$\dot{u} = \frac{\partial u}{\partial t} = -\frac{\omega^2 H}{2} \frac{\cosh k(z+h)}{\sinh kh} \sin(\omega t - kx) \tag{2.3}$$

It is noted that the equations for water horizontal particle velocity 2.2 an acceleration 2.3 are functions of depth. As such they can be used to make a velocity profile along the length of the spar or depth of the ocean.

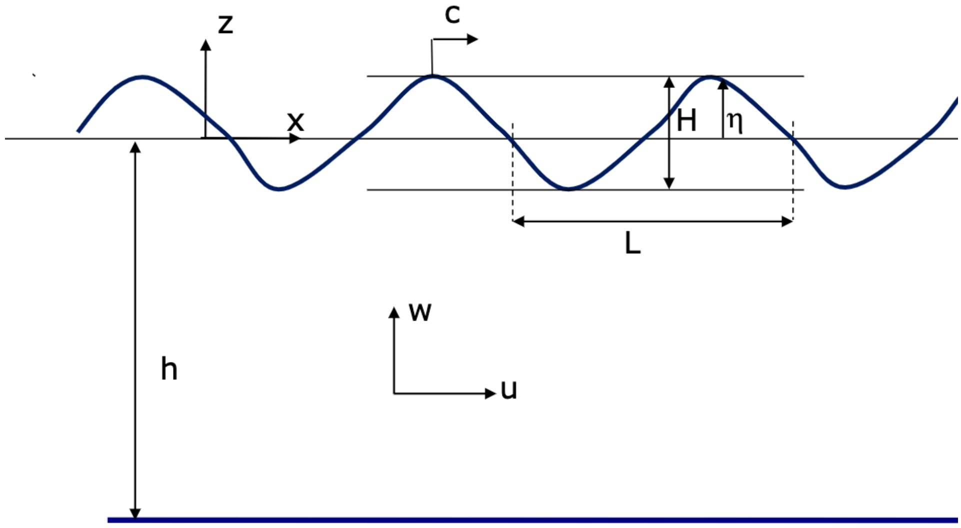


Figure 2.1: Schematic for hydrodynamic parameterization

The wave number k describes the spatial frequency of a wave, given in radians per meter.

$$k = \frac{1}{\lambda} \tag{2.4}$$

Where λ is the wavelength which can be calculated from the frequency and speed of the wave.

$$\lambda = \frac{v}{f} \tag{2.5}$$

Where v and f are wave speed and frequency given in m/s and Hz respectively. However there is a logical problem with this approach; equations 2.2 and 2.3 are used to calculate the wave speed but also take the wave number as an input. To get around this the dispersion relation of water waves can be solved for k. The dispersion relation is defined as:

$$\omega^2 = gk \tanh(kh) \tag{2.6}$$

Where ω is the angular velocity which can be converted from the frequency f. Solving equation 2.6 for k and using equations 2.2 and 2.3 a velocity profile can be made for a regular wave of given wave height H at a depth h.

Irregular Wave Kinematics Although regular waves do mimic nature, it is a closer approximation to reality to consider irregular waves. The shape ζ of an irregular wave can be considered to be a superposition of many partial regular waves. [87].

$$\zeta(x,t) = \sum_{i=1}^{n} c_i \cos(k_i x - \omega_i t + \varepsilon_i) \tag{2.7}$$

Where n is the number of partial waves, i is the number of wave components and the subscript $_i$ points to each component. c_i is the amplitude of the i^{th} partial wave. ω_i is the circular frequency of the partial wave and k_i is the wave number. t is time and ε is a randomly generated phase angle of the partial wave. The randomly generated wave phases have a uniform probibility distribution of $1/(2\pi)$

in the range of $(0, 2\pi)$. The amplitudes c_i are generated from a spectrum that describes the spectral density.

$$c_i = \sqrt{2S_{\zeta\zeta i}\delta\omega_i} \tag{2.8}$$

Where $S_{\zeta\zeta i}$ describes the seaway spectrum, a common ocean wave spectrum is the JONSWAP spectrum. Developed after the collection of north sea data. The JONSWAP spectrum is an artificially modified Pierson-Mosckowitz spectrum to improve the fit of the spectrum to data collected from the north sea during the Joint North Sea Wave Observation Project.

The Pierson-Moskovitch spectrum is defined as:

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \text{exp}\left(-\beta\left(\frac{\omega_0}{\omega}\right)^4\right) \tag{2.9}$$

Where α is $8.1 \cdot 10^3$ and β is 0.74. ω_0 is $\frac{g}{U_{19.5}}$ where $U_{19.5}$ is the wind speed at 19.5 meters high. The height at which the initial measurement were done by Pierson and Mosckowitz. The Jonswap Spectrum is practically the same but multiplied by an external factor γ^r. This ensures that the spectrum is never fully developed, so even over long term time intervals the spectrum keeps changing.

$$\begin{aligned} S_j(\omega) &= \frac{\alpha g^2}{\omega^5} \text{exp}\left[-\frac{5}{4}\left(\frac{\omega_p}{\omega}\right)^4\right]\gamma^r \\ r &= \text{exp}\left[-\frac{(\omega-\omega_p)^2}{2\sigma^2\omega_p^2}\right] \end{aligned} \tag{2.10}$$

Where using the data collected from the Joint North Sea Observation Project

$$\begin{aligned} \alpha &= 0.076\left(\frac{U_{10}^2}{Fg}\right)^{0.22} \\ \omega_p &= 22\left(\frac{g^2}{U_{10}F}\right)^{1/3} \\ \gamma &= 3.3 \\ \sigma &= \begin{cases} 0.07 & \omega \leq \omega_p \\ 0.09 & \omega > \omega_p \end{cases} \end{aligned} \tag{2.11}$$

Where ω_p is the peak angular frequency, U_{10} is the wind speed at 10 meters and F is the fetch distance. The fetch distance can be defined as the distance over which wind can travel undisturbed over water in a constant direction. When the two spectra are compared to each other (see fig:2.2) it becomes clear that the JONSWAP spectrum has a far higher peak spectral density at peak frequency $f = f_p$. Besides using the spectrum for wave height it can then also be used to obtain the wave kinematics of irregular waves which later will prove neccesary for the calculation of the wave loading, which is dependent on said kinematics.

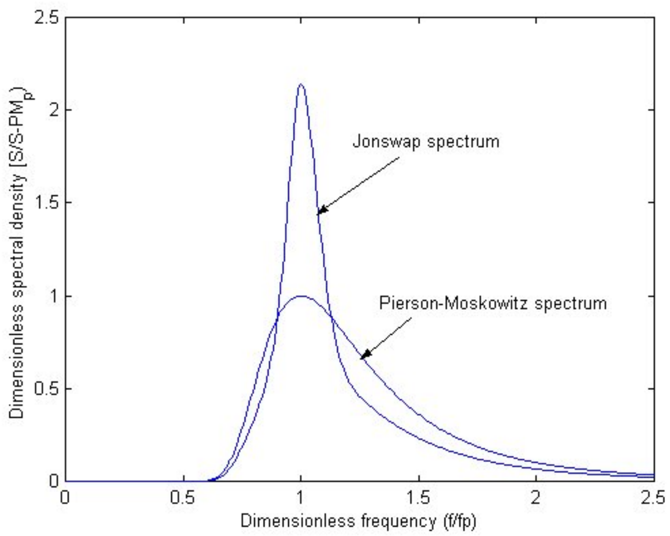


Figure 2.2: Comparison between Pierson-Moskowitz and JONSWAP spectrum. Taken from Abankwa N (2010) [2]

Using this spectrum the motion behaviour of an irregular wave can be modeled that closely resembles the north sea. Or more specifically the findings of Joint North Sea Observation Project.

Morisons equation There are two common practices for calculating the hydrodynamic forcing; Morisons and Maccamy Fuchs'. The Morisons equations are based on a 1950's paper written by JR Morison [59], it is a semi empirical approach to calculating the hydrodynamic forces. Through experimental and analytical analyses the paper showed that hydrodynamic forcing can be seen as the sum of inertia and drag forces. The equation presumes that the acceleration of flow over one location on the body is uniform. For a spar type floater, or any other cylindrical structure that is mounted vertically in the water, the equation will only be valid if the diameter of the cylinder is significantly smaller than the wavelength. When this is not the case diffraction needs to be taken into account.

$$F_{morison} = \underbrace{\rho_{sea} C_m V \dot{u}}_{F_I} + \underbrace{\frac{1}{2}\rho_{sea} C_d A u|u|}_{F_D}, \tag{2.12}$$

$$C_m = 1 + C_a \tag{2.13}$$

Where ρ_{sea} is the density of sea water, V is the volume of the body and $\dot{u}$ is the wave acceleration expressed in equation 2.3. C_m is the inertia and added mass coefficient and C_d is the drag coefficient. u represents the wave speed as expressed in equation 2.2. A is the cross-sectional area of the body perpendicular to the flow direction. This forcing is calulated as a depedent on the wave speed and acceleration. This wave speed can be calculated every depth underwater. This is done by considering a strip of the structure and investigating the water velocity at that depth and in doing so developing a forcing profile over each strip of the slender structure.
In 1945 R.Maccamy and R. Fuchs published a report of two main parts, an exact mathematical solution to the linearized problem of water waves of small steepness incident on a circular cylinder. The second part was an attempt to verify the computation using cylindrical piles in a wave basin which resulted in good approximation of the forces [52].
Still using airy wave theory but in this case also considering a new boundary for solving the potential equation. Maccamy & Fuchs decided to introduce the boundary condition that if there is a cylinder in the fluid, then the fluid could not move through that cylinder. Instead the waves hitting the fluid will reflect and scatter back. Taking this into consideration the velocity potential can be expressed as:

$$\phi = \phi_w + \phi_s \tag{2.14}$$

Where ϕ is the total potential, ϕ_w is the 'incident wave' potential as expressed in equation 2.1 and ϕ_s is the 'scattered wave' potential. The body surface boundary condition, where the waves hit and reflect off the structure can be expressed as:

$$\frac{\partial \phi_s}{\partial n} = -\frac{\partial \phi_w}{\partial n} \tag{2.15}$$

The pressure at any given depth can then be expressed as

$$p = -\rho g z - \rho \frac{\partial \phi}{\partial t} \tag{2.16}$$

The pressure needs to be calculated around the radius of the cylindrical structure in the fluid. Afterwhich the hydrodynamic force at any given depth on the structure can be calculated by integrated the pressure over the radius of the structure . The force is then expressed as:

$$F_z = 2 * \int_0^{\pi} p(\theta) \text{ a } \cos(\pi - \theta) d\theta \tag{2.17}$$

$$\mathrm{F_Z} = \frac{2\rho \mathrm{gH}}{\mathrm{k}} \frac{\cosh \mathrm{k(d+z)}}{\cosh \mathrm{kd}} (\mathrm{ka}) \cos(\sigma \mathrm{t} - \delta) \tag{2.18}$$

$$C_m = \frac{4A(ka)}{\pi (ka)^2} \tag{2.19}$$

$$A(ka) = \frac{1}{\sqrt{[J_1'^2(ka) + Y_1'^2(ka)}} \tag{2.20}$$

$$\delta = -\tan^{-1} \left[Y_1'(ka) / J_1'(ka) \right] \tag{2.21}$$

$$ka = \pi \cdot \frac{D}{L} \tag{2.22}$$

The scattered potential ϕ_s is calculated using a set of bessel functions J_1 and Y_1, where the subscript is meant to indicate the order of bessel function.

Aerodynamics

The aerodynamic loading is caused by the interaction between wind kinematics and the structure. One of the challenges of modeling the aerodynamic behaviour is the measurement of relevant wind data. Aerodynamic loading is the largest contributor to the overturning moment of a floating wind turbine. In 2012 Schubel did a review on wind turbine blade design which includes a summary of blade loads [76] which is where the image below is taken from. Hansen's 2015 book on aerodynamics offers a comprehensive review on the matter [44]. The calculation of the aerodynamic loads is based on blade element momemtum theory (BEM) which find it's roots in the 1D momentum theory. Essentially cutting the blade up into a set amount of slices and interpolating the aerodynamic behaviour of slice each over the entire blade. The derivation of the entire theory would be too cumbersome for the scope of this report but a summarized version is presented below.

A wind turbine transforms the kinetic energy in the wind into mechanical energy and finally in the turbine itself, using a generator this will be transformed into electrical energy. It is common to find a two dimensional approach to calculating the aerodynamics. This presumes that the flow of air moves across the blade. For this calculation the blade can be presumed to be infinitely long and only one 2-D slice is considered.

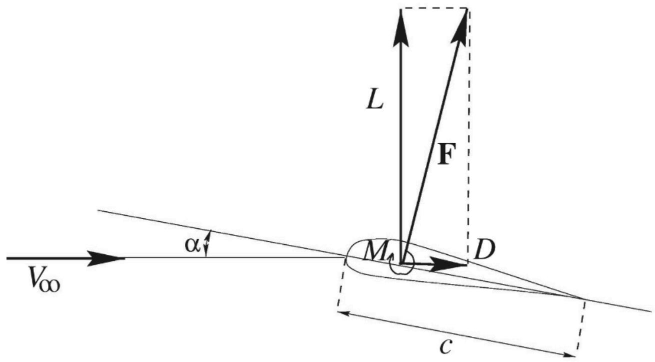


Figure 2.3: Schematic of lift and drag forces *Source: Hansen (2015) [44]*

$$C_l = \frac{L}{1/2\rho_{air}V_\infty^2 c} \tag{2.23}$$

$$C_d = \frac{D}{1/2\rho_{air}V_\infty^2 c} \tag{2.24}$$

The lift and drag coefficient can be determined using equations 2.23 and 2.24. Where c is the cord length, ρ_{air} is the air density in kg/m^3 and V_∞ is the wind speed. This offers the forcing along the blade of the turbine and therefore for the full lift or drag force would have to be integrated over the blade. As the geometry of the blade is smooth and changing throughout, so does the lift and drag characteristic along the blade.

The thrust on a wind turbine can be calculated used 1-dimensional infinitely bladed wind turbine. By comparing the wind speed and air pressure before and after. If the flow is presumed to be frictionless there will be no change in the internal energy. So the shaft power can be related to the mass flow of air.

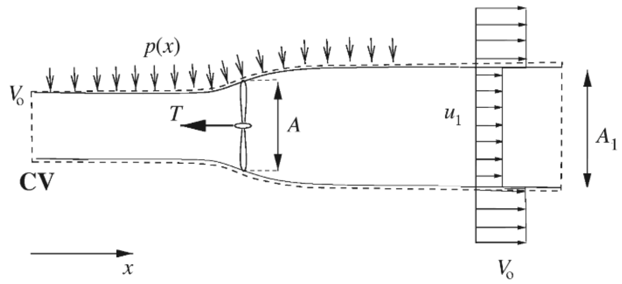


Figure 2.4: Schematic infinitely bladed wind turbine *Source: Hansen (2015) [44]*

The mass flow is defined as:

$$\dot{m} = \rho_{air} \cdot u \cdot A \tag{2.25}$$

The thrust and power can be calculated as follows:

$$T = \rho_{air} u A \left(V_o - u_1\right) = \dot{m}\left(V_o - u_1\right). \tag{2.26}$$

$$P = \frac{1}{2}\rho_{air} u A \left(V_o^2 - u_1^2\right). \tag{2.27}$$

The axial induction factor is the amount of wind that doesn't make it to the turbine.

$$u = (1-a)V_o. \tag{2.28}$$

Using the definition of the axial induction factor a, the expressions for thrust and power become:

$$\begin{aligned} P &= 2\rho_{air} V_o^3 a(1-a)^2 A \\ T &= 2\rho_{air} V_o^2 a(1-a) A \end{aligned} \tag{2.29}$$

These can be made dimensionless to create a set of thrust and power coefficients for each blade part.

$$\begin{aligned} C_p &= \frac{P}{\frac{1}{2}\rho_{air} V_o^3 A} \\ C_T &= \frac{T}{\frac{1}{2}\rho_{air} V_o^2 A} \end{aligned} \tag{2.30}$$

For modelling where there is not enough computational power available to compute the thrust over each element of the blade or even consider each blade, an average C_T can be used to get a simplified relationship between wind speed and thrust. This simplified version presumes a quasi-static nature where the changes in thrust are an instant reaction to experienced wind speed changes.

$$T = C_T \cdot \frac{1}{2}\rho_{air} V_o^2 A \tag{2.31}$$

Wind Kinematics To calculate the loading on the wind turbine, the wind has to be modelled. When modelling a constant wind speed, the wind speed that is experienced by the turbine can be expressed as

$$V_o = V_{constant} - \dot{x} \tag{2.32}$$

Where $V_{constant}$ is the wind speed and $\dot{x}$ is the motion of the turbine in x direction. It is presumed that they are positive along the same axis.

Modelling the wind speed to mimic a specific environment requires a spectral approach similar to the irregular wave kinematics discussed in 2.1.1.

In 1972 J Wyngaard described the behaviour of spectra and cospectra of turbulence in the surface layer. Based on the fluctuation of temperature and wind from data obtained in the 1968 AFCRL Kansas experiments [39]. To this day, the spectrum described in that paper is still widely used for modelling wind. The Kaimal spectrum is expressed as:

$$\left[\frac{nS_u(n)}{u_*^2}\right]_{f=4} = 0.12\phi_\epsilon^{2/3} \tag{2.33}$$

$$\phi_e{}^{2/3} = \left\{ \begin{array}{ll} 1 + 0.5|z/L|^{2/3}, & -2 \leqslant z/L \leqslant 0 \\ 1 + 2 \cdot 5|z/L|^{3/5}, & 0 \leqslant z/L \leqslant +2 \end{array} \right\}. \tag{2.34}$$

Where z/L is a dimensionless length known as the stability parameter, it comes from the Monin-Obukhov similarity theory, z describes the height at which the wind is being considered, and L is the Obukhov length [73]. The spectrum is normalized over the square of friction velocity u_*.

However, an array of adjusted Kaimal spectra is used throughout the industry. For example, the spectrum was defined in the 2016 paper of Abrous [4].

$$s(f) = \frac{\left[\ln (h/z_0)^2\right]^{-1} \cdot l \cdot v_w}{\left(1 + 1.5\left(f \cdot l/v_w\right)\right)^{\frac{5}{3}}} \tag{2.35}$$

While in the investigation of the Kaimal spectrums applicability in the offshore wind industry by Cheynet in 2017 [14], the IEC Kaimal model was used. This model normalizes the spectrum over the wind speed standard deviation σ_u^2:

$$\frac{fS_u(f)}{\sigma_u^2} = \frac{4fL_u/\bar{u}}{(1 + 6fL_u/\bar{u})^{5/3}} \tag{2.36}$$

$$L_u = 8.1\Lambda_1$$
$$\Lambda_1 = \begin{cases} 0.7z & \text{if } z \leq 60 \text{ m} \\ 42 \text{ m} & \text{if } z \geq 60 \text{ m} \end{cases} \tag{2.37}$$

2.1.2. Fatigue Damage

Calculation of fatigue is crucial to the design of a floating wind turbine system. It determines the lifetime of the structure and will therefore be a determining factor in the structural design decisions made on any such structure. This subsection explores how fatigue can be calculated by first looking at Miner's rule, then expanding on rainflow cycle counting and finally discussing Dirliks method which is used for damage calculation in the frequency domain.

Miner's Rule

The basis of both Dirliks Method and rainflow counting is a hypothesis presented by both Palmgren in 1924 [67] on ballbearings and Miner in 1945 [57] on ordinary structural components. This widely accepted hypothesis is referred to as 'Miner's Rule'. The hypothesis assumes that if a structure is exposed to n_i cycles at stress level S_i, the structure would fail at N_i cycles. The fraction of reduced life is precisely proportional to n_i[19]. In other words the structure will fail when:

$$\sum \frac{n_i}{N_1} = 1 \tag{2.38}$$

Or in a situation where there are k amount of stress levels. The damage fraction can be calculated as:

$$\sum_{i=1}^{k} \frac{n_i}{N_i} = C \tag{2.39}$$

Where n_i is the number of cycles accumulated at stress S_i. C is the fraction of life consumed after exposure to the cycles at the set of k stress levels. N_i is the number of available cycles in the components lifetime at stress level S_i

Indexing the stress levels, the damage D on any component can be calculated with the:

$$D_i = n_i \times S_i \tag{2.40}$$

The presumption is made that the critical failure damage is the same across all stress ranges. This implies that if the failure damage we're to be 100, then it would take 10 cycles at stress level 10 or 20 cycles at stress level 5.

$$D_{failure} = N_i \times S_i \tag{2.41}$$

Which means the equation 2.39 can be rewritten as:

$$\sum_{i=1}^{k} \frac{n_i \times S_i}{N_i \times S_i} = C \Rightarrow \frac{\sum_{i=1}^{k} n_i \times S_i}{W_{\text{Fallure}}} = C \tag{2.42}$$

Rainflow Counting

Rainflow counting is a method for calculating the fatigue on a structure from unsteady cyclic loading. The rainflow method is a widely used algorithm, and as such, in cooperation with multiple industries, the method was standardized in 1994 [6]. The rainflow method is a technique that makes it possible to store service measurements for fatigue analysis through cycle counting, both fatigue life prediction and simulation testing. The procedure begins with a preliminary treatment of the loading, sampling, extraction of extremes and quantifying values into classes. It normally consists of monitoring a specific variable (i.e. stress) over time. Computationally this sequence can come from a loading model that adequately represents the real evolution of stress or experimentally from a single variable measurement over time. The rainflow counting method aims to bin a time series of loading into a set stress range and count the cycle of each stress range. The first step is putting the loading series through a hysteresis filter to filter out small variations that do not constitute a new stress cycle. This has also been referred to as a 'peak-valley' filter. In other words, the rainflow counting method only counts the extremes of the loading sequence. To speed up the analysis, splitting the possible extremes into 64 classes with constant interval steps in between is common. So rather than having however many unique possible stresses, there are 64 bins on the stress axes where the nearest points all fall too. A different term for this is 'binning'. To determine the damage, the rainflow counting method extracts stress cycles by always looking at four successive points, as these would be considered one stress cycle. Depending on the relative difference in stress between these points, the rainflow method will either move to the next point or consider these four points in one cycle. If a cycle is found, or in other words, if the middle stresses are bound by the extreme stresses and one cycle is counted, the middle stress is removed, the next four points are taken and so forth. The algorithm of which taken from [6] is shown in figure 2.5

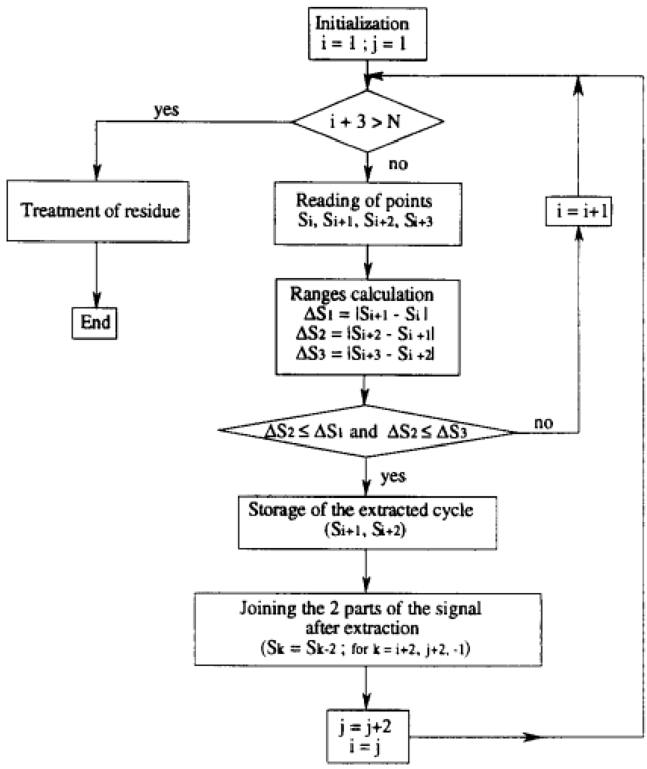


Figure 2.5: Rainflow algorithm proposed 1994 by C. Amzallag et al. *Source: Amazllag et al (1994)* [6]

Dirliks Method

Although the rainflow counting method is beneficial, when it comes to optimization, there is a preference for being able to compute quickly. Therefore it is common to see frequency-domain approaches where no time series are being produced. Therefore the 4-point counting method used in the rainflow method is no longer an option. In 2013 Mršnik [60] reviewed multiple methods to calculate fatigue within the frequency domain, and found that besides Dirlik's method [19], the Tovo–Benasciutti 2005 [9] and Zhao–Baker 1992 [88] methods should be considered as methods for fatigue analysis in the frequency domain. Nonetheless, Zhao-Baker and Tovo-Benasciutti methods have not yet been used in optimization publications reviewed for this thesis.

The Dirlik method is based on two different groups of spectra which have been numerically simulated in the time domain. Early in the development of the Dirlik method, it was decided that it was too complex to derive the distribution of rainflow cycles from a PSD function $G(\omega)$ in closed form. Instead, a Monte Carlo approach was used to generate a sample stress history $s(t)$ from $G(\omega)$. Then the rainflow algorithm was used on $s(t)$ to extract the cycles, and the probability density function of rainflow counted ranges. This allowed calculating the fatigue damage for any given material constants in an S/N curve. The Monte Carlo simulation is run twice, from which three probability density functions are taken, one exponential and two Raleigh. The Dirlik method approximates the cycle-amplitude distribution. The rainflow cycle amplitude probability density function (PDF) estimate is defined as:

$$p_a(s) = \frac{1}{\sqrt{m_0}}\left[\frac{G_1}{Q}e^{\frac{-2}{2}} + \frac{G_2 Z}{R^2}e^{\frac{z^2}{2R^2}} + G_3 Z e^{\frac{-z^2}{2}}\right] \tag{2.43}$$

Where the damage is expressed as

$$\bar{D}^{DK} = C^{-1} v_p m_0^{\frac{k}{2}}\left[G_1 Q^k \Gamma(1+k) + (\sqrt{2})^k \Gamma\left(1+\frac{k}{2}\right)\left(G_2|R|^k + G_3\right)\right] \tag{2.44}$$

Where

$$G_1 = \frac{2\left(x_m - \alpha_2^2\right)}{1+\alpha_2^2}, G_2 = \frac{1-\alpha_2 - G_1 + G_1^2}{1-R}, G_3 = 1 - G_1 - G_2 \tag{2.45}$$

$$Q = \frac{1.25\left(\alpha_2 - G_3 - G_2 R\right)}{G_1}, R = \frac{\alpha_2 - x_m - G_1^2}{1-\alpha_2 - G_1 + G_1^2} \tag{2.46}$$

and

$$\alpha_2 = \frac{m_2}{\sqrt{m_0 m_4}}, x_m = \frac{m_1}{m_0}\sqrt{\frac{m_2}{m_4}}, v_p = \frac{1}{2\pi}\sqrt{\frac{m_4}{m_2}} \tag{2.47}$$

In the damage equation, 2.44, both C and m are parameters of the initial part of the SN curve, defined as the region less than 10^7 cycles.

2.2. Methodology: Numerical loads and response

For the scope of this thesis, it has been decided to develop a time domain simulation. The methodology of this is presented in this section of the thesis. Beginning with the introduction of the environmental conditions considered for this research. After this, the structural model is introduced, and the forcing calculations are made, leading to the response and fatigue calculations.

2.2.1. Environmental Conditions

To investigate the effect of constraints on the optimization of a spar type substructure a test case is considered. The 15MW reference turbine is placed 32 kilometers off the coast of Norway. The environmental data is taken from a study done by Z.Gao [49] (2015). The data was generated using a hindcast model from the Kopadistrian University of Athens. From this study site no. 15 is is used as this is the largest distance from shore in the north sea. The depth however, has been adjusted to allow for the size of spar that would be necesary to sustain a turbine and tower of the size described in the 15 MW reference turbine [27].

Site no	Area	Name	depth (m)	shore(km)	50-year U_w at 10m (m/s)	50-year H_s (m)	Mean value of T_p (s)
15	North Sea	North Sea Center	~~29~~ 400*	300	27.2	8.66	6.93

Table 2.1: Site Conditions for fatigue calculations taken from [49]
*Depth has been adjusted from 29 to 400 meters

The wind measurements are measured at 10-min averages whereas the wind developed for re-sposne calculations develops instantaneous wind. The kaimal spectrum used to develop instantaneous wind will give wind time series with the 10-min average is given. As such a probability density plot is needed for the occurrence of each wind speed. The wind speed characteristics are not separate from the wave conditions, as such the joint probability of wind speed U_w, significant wave height H_s, and significant wave period Tp can be expressed as:

$$f_{U_\mathrm{w}, H_\mathrm{s}, T_\mathrm{p}}(u, h, t) \approx f_{U_\mathrm{w}}(u) \cdot f_{H_\mathrm{s}|U_\mathrm{w}}(h \mid u) \cdot f_{T_\mathrm{p}|H_\mathrm{s}}(t \mid h) \tag{2.48}$$

This is the simplified version proposed by Z.Gao [49], where the distribution of T_p is only conditioned on the wave height. The distribution of the wind speed is expressed using a weibull distribtution:

$$f_{U_\mathrm{w}}(u) = \frac{\alpha_\mathrm{U}}{\beta_\mathrm{U}}\left(\frac{u}{\beta_\mathrm{U}}\right)^{\alpha_\mathrm{U}-1} \cdot \exp\left[-\left(\frac{u}{\beta_\mathrm{U}}\right)^{\alpha_\mathrm{U}}\right] \tag{2.49}$$

The conditional distribution of H_s on U_w is also given with a weibull distribution:

$$f_{H_s|U_\mathrm{w}}(h \mid u) = \frac{\alpha_\mathrm{HC}}{\beta_\mathrm{HC}} \left(\frac{h}{\beta_\mathrm{HC}}\right)^{\alpha_\mathrm{HC}-1} \cdot \exp\left[-\left(\frac{h}{\beta_\mathrm{HC}}\right)^{\alpha_\mathrm{HC}}\right] \tag{2.50}$$

Where α_HC and β_HC are the shape and scale parameters respectively. These are fitted using power functions.

$$\alpha_\mathrm{HC} = a_1 + a_2 \cdot u^{a_3} \tag{2.51}$$

$$\beta_\mathrm{HC} = b_1 + b_2 \cdot u^{b_3} \tag{2.52}$$

a_1, a_2, a_3 and b_1, b_2, b_3 are parameters that have been fitted to measrurements of the site. The parameters for Site 15 are shown in the table 2.2:

Distributions	Parameter	Associated equation	Site 15
Marginal U_w	α_U	2.49	2.299
	β_U	2.49	8.920
Conditional H_s given U_w	a_1	2.51	1.755
	a_2	2.51	0.184
	a_3	2.51	1
	b_1	2.52	0.534
	b_2	2.52	0.007
	b_3	2.52	1.435

Table 2.2: Parameters for marginal distribution U_w and conditional distribution of H_s at given U_w

Lastly is the conditional distribution of the wave period at a given wave height. This is expressed as a function of mean and standard deviation of the natural logarithm of period.

$$f_{T_\mathrm{p}|H_\mathrm{s}}(t \mid h) = \frac{1}{\sqrt{2\pi}\sigma_\mathrm{LTC} t} \cdot \exp\left[-\frac{1}{2}\left(\frac{\ln(t) - \mu_\mathrm{LTC}}{\sigma_\mathrm{LTC}}\right)^2\right] \tag{2.53}$$

Where the means and standard are approximated by:

$$\mu_\mathrm{LTC} = c_1 + c_2 \cdot h^{c_3} \tag{2.54}$$

$$\sigma^2_\mathrm{LTC} = d_1 + d_2 \cdot \exp\left(d_3 h\right) \tag{2.55}$$

The parameters for this distribution are presented in 2.3

Distributions	Parameter	Associated equation	Site 15
Conditional Distribution T_p given H_s	c_1	2.54	1.578
	c_2	2.54	0.222
	c_3	2.54	0.674
	d_1	2.55	0.008
	d_2	2.55	0.227
	d_3	2.55	-0.956

Table 2.3: Parameters for Conditional Distribution of T_p at a given H_s

The joint distribution consists of marginal distribution of the wind speed, a conditional distribution for wave height at a given wind speed, and a conditional distribution for wave period with a given wave height.

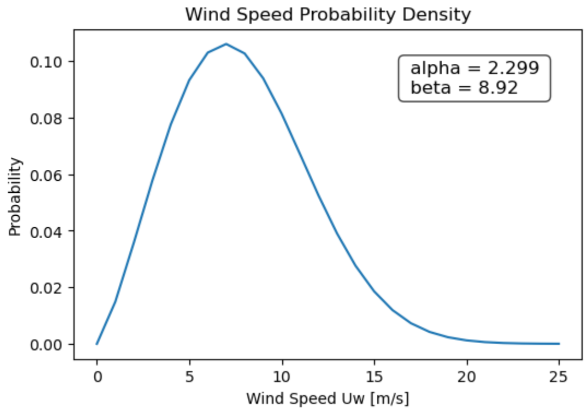
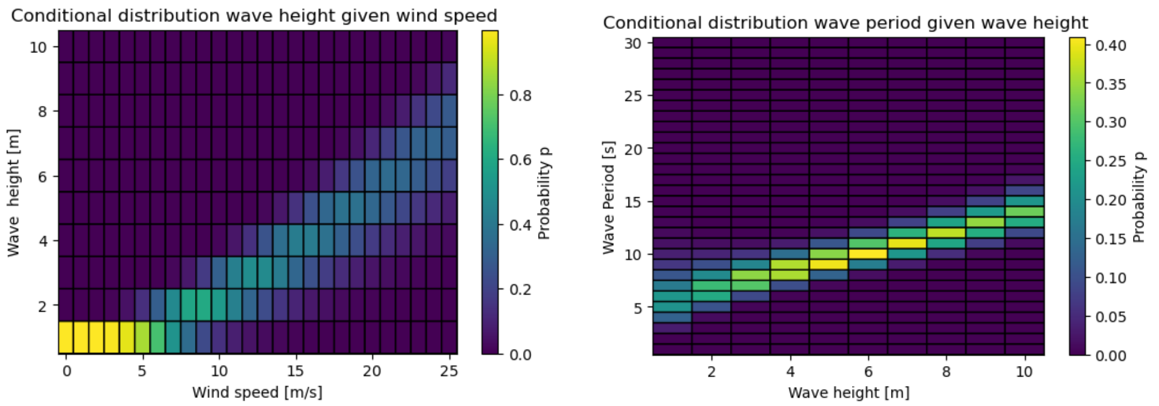


Figure 2.6: Probability density plot using f_{U_w} 2.49 with bin size of $1m/s$ for site 15

The weibull distribution is plotted for a wind speed of 0 to 25 meters. The wind speed with the highest probability of 0.106 is $7m/s$. The first wind speed above 7 to have a probability density smaller than 0.01 is $17m/s$. As such the fatigue calculations will be done from cut in wind speed of 3 up to 17 m/s. To limit the computational expense a stepsize of 2 is used for the wind speed.
To determine the wind wave conditions relevant to the fatigue lifetime damage the figures below are investigated. The conditional distribution of wave height given wind speed and of wave period given wave height are presented in figure 2.7. The figure on the left 2.7a shows the distribution of significant wave height H_s at a given wind speed U. Any collumn in this figure can be summed to 1 as it accounts for probability density for each wave height occurring at that given wind speed. Between wind speeds of 0 and $4m/s$ the chances of there being a significant wave height above $1m$ is negligible. As the wind speed increases there is a larger variation of possible significant wave height occurring. At wind speeds between 5 and 17 m/s it is much more important to consider multiple wind wave conditions as the probability of occurrence is shared over around 3 wave heights. For example at a wind speed of $10m/s$ the significant wave heights that can be expected to occur are 1,2 and 3 meter. This information is used to determine what wave wind conditions are relevant to this test case.



(a) Conditional wave height distribution given wind speed using 2.50 and bin size of $1m/s$

(b) Conditinoal wave period distribution given wave height, used 2.53 and bin size of $1s$

Figure 2.7: Conditional probability densities for Norway site 15

The next step would be to analyze what wave periods to consider, the function for the conditional distribution of wave period given wave height (eq: 2.53) gives the distribution of wave period for any given wave height and is presented in 2.7b.

Using the figures above the wind wave conditions for the lifetime fatigue have been determined. While trying to limit the amount of simulations required to evaluate a design the decision is made to use up to 4 most likely wave periods for each wave height, and up to 3 most like wave height per wind speed. Using the probability density plot of the wind (fig: 2.6) the decision is made to consider wind speeds from 3 to 17 m/s with intervals of $2m/s$. This covers the cut in wind speed and ignores wind speeds that occur less than 1% of the time. The wave period distribution for a wave height of $1m$ is wide spread. As such it would be computationally very heavy to run for every wave period with that given wave height. That's why the decision is made to only consider the wave loading with a slow wave period of 6 and 7 seconds. This approach leads to a very large table which can be found in the appendix. A small part of it is presented here to illustrate some further design choices.

Wind speed u [m/s]	f_{uw}	Wave height h [m]	$f_{hs\|uw}$	Wave Period t [m/s]	$f_{hs\|tp}$
11	0.06703	2	0.432	5	0.218
				6	0.298
				7	0.2199
		3	0.428	6	0.157
				7	0.342
				8	0.287
				9	0.137
13	0.03902	2	0.2264	5	0.218
				6	0.298
				7	0.2199
		3	0.4725	6	0.157
				7	0.342
				8	0.287
				9	0.137
		4	0.2541	7	0.134
				8	0.3679
				9	0.322
				10	0.132

Table 2.4: Design environmental conditions using joint probabilities for wind speed, wave height, and wave period.

From table 2.4 it becomes clear that by considering the wave period t the amount of environmental conditions to consider to evaluate the lifetime fatigue damage increases on average by three. Furthermore, the most probable wave period for a given wave height is concentrated around one area. For the sake of computational expense, the average weighted probability of the wave period will be taken for each wind wave condition. The weighted average of the wave period can be calculated with the following:

$$T_{use} = \frac{\sum t_i \cdot f_{hs|tp_i}}{\sum f_{hs|tp_i}} \tag{2.56}$$

The probability density of the wind speed is normalized for the considered probabilities, after which the freq of occurrence is calculated by multiplying the normalized marginal distribution of the wind speed with the conditional distribution of the wave height at a given wind speed and of the normalized distribution of wave period at a given wave height. This results in the following environmental conditions for the fatigue lifetime damage calculation.

Wind speed u [m/s]	f_{uw}	f_{uw}norm	Wave height h [m]	$f_{hs\|uw}$	$fs_{hs\|uw}$ norm	Wave Period t [m/s]	Freq Occurrence
3	0.05767	0.11941442	1	0.8033	1	25	0.11941442
5	0.09329	0.19317099	1	0.858	0.867543	25	0.16758414
			2	0.131	0.132457	6	0.02558686
7	0.10609	0.21967532	1	0.513	0.521872	25	0.11464236
			2	0.47	0.478128	6	0.10503296
9	0.09395	0.19453762	1	0.2252	0.22538	25	0.04384495
			2	0.605	0.605484	6	0.11778949
			3	0.169	0.169135	7.43	0.03290318
11	0.06703	0.13879571	2	0.432	0.502326	6.001	0.06972064
			3	0.428	0.497674	7.437	0.06907507
13	0.03902	0.08079679	2	0.2264	0.237566	6.001	0.01919454
			3	0.4725	0.495803	7.437	0.04005927
			4	0.2541	0.266632	8.47252	0.02154298
15	0.01861	0.03853481	2	0.1004	0.102043	6.001	0.0039322
			3	0.3271	0.332452	7.437	0.01281099
			4	0.4144	0.421181	8.47252	0.01623013
			5	0.142	0.144324	9.429	0.00556148
17	0.00728	0.01507434	3	0.1758	0.184315	7.437	0.00277843
			4	0.369	0.386874	8.47252	0.00583186
			5	0.3291	0.345041	9.429	0.00520126
			6	0.0799	0.08377	10.35	0.00126278

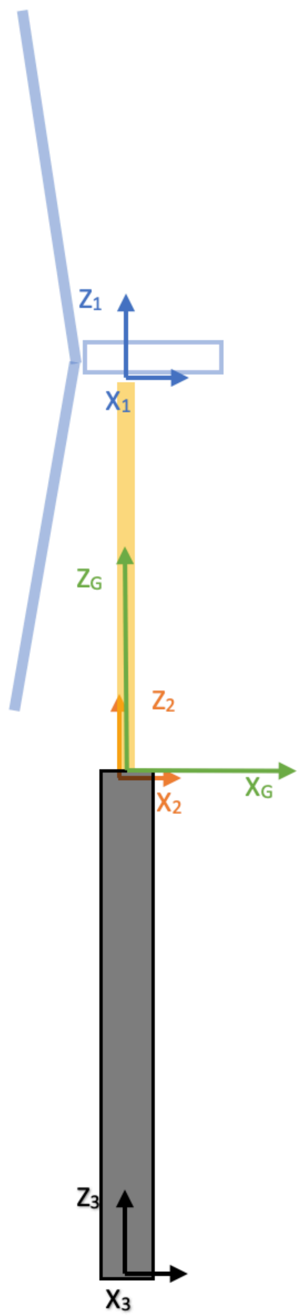
Table 2.5: Environmental conditions used for fatigue damage calculation

2.2.2. Structural Model

The structural model is considered to consist of three rigidly connected parts; the tower, the RNA assembly and the substructure (see figure:2.8). This section will explore how these parts have been modeled and in doing so give insight into what simplifications have been made. For the modeling of the structure in the optimization the IEA 15MW reference wind turbine has been used [27] as such any results from the optimization have a bias to the tower and turbine combination.

A Cartesian coordinate system is used with the origin at the SWL and it's positive z-direction facing upwards. Meaning that any negative z-coordinate can be considered to be submerged in water, presuming still water conditions. Each part will first be considered in it's own local coordinate system, afterwhich the inertia's are adjusted for the global coordinate system. The research considers a 3 degree of freedom system; surge(x-translation),heave (z-translation) and pitch (rotation about the y axis). This choice is motivated by computational efficiency which will be discussed later on. In a 6 degree of freedom system the subscripts for these degrees of freedom would be 1, 3 and 5. In the rest of report these same subscripts will be used to refer to surge(1), heave(3) and pitch(5).

Positive rotation is considered to be counter-clockwise. The same holds true for positive moments.



Tower

In the defenition of the 15MW IEA reference turbine [27] the tower used was meant to be planted on a monopile. The substructure will be designed such that the same tower would fit a top the spar. The material properties weights and distances can be found in the table 2.6

The tower was designed as an isotropic steel tube. With a thickness ranging from 23 to 45 millimetres. The

tower defined in [27] begins 15 metres above SWL. The transition piece sits between the SWL and the start of the tower. For the sake of coding efficiency the height, diameter and thicknesses of the transition piece are included in the tower and thus seen as one piece. The exact data can be found in appendix B.
Using those coordinates and the material properties, the mass can be calculated per section and then summed to get the total mass of the tower.

$$A_i = \pi \cdot ((\frac{D_i}{2})^2 - (\frac{D_i}{2} - t_i)^2 \tag{2.57}$$

$$\delta h = h_{Tower_{i+1}} - h_{Tower_i} \tag{2.58}$$

$$m_i = A_i \cdot \rho_{steel} \cdot \delta h \tag{2.59}$$

$$m_{tower} = \sum_{i=1}^{k} m_i \tag{2.60}$$

Where A_i is the area per section, D_i is the outer diameter of that section, and t_i is the thickness. δh is the height difference within a section. This alternates between 0.0001 and 5 metres, as can be seen in B.2. Calculated with the difference in tower height between two sections, $h_{Tower_{i+1}}$ and h_{Tower_i}. The sectional mass can then be calculated with the sectional area A_i, the density of the steel ρ_{steel} and δh. The sectional masses m_i are also used to calculate the tower's centre of gravity. As the tower is practically modelled as straight-edged buckets that progressively go down in diameter every 5 metres, the z-coordinate of the centre of gravity of each section is halfway up that section.

$$z_{cm_{Tower}} = \frac{\sum_{i=1}^{k} \frac{h_{Tower_{i+1}} + h_{Tower_i}}{2} \cdot m_i}{m_{tower}} \tag{2.61}$$

The moment of inertia of the tower is calculated by first calculating the moment of inertia for each section with the origin of each being in the centroid of the section piece, after which the parallel axis theorem [3] to calculate the moment of the tower around the SWL. The moment of inertia of each is calculated with the formula for a thin-walled cylinder 2.62

$$I_x = I_y = \frac{1}{12} m \left[3\left(r_1^2 + r_2^2\right) + h^2\right] \tag{2.62}$$

$$I_{x_{tower}} = \sum_{i=1}^{k} (I_{x_i} + m_i \cdot z_{cm_i}^2) \tag{2.63}$$

Description	Value	Unit
$YoungsModulus(E)$	$2.00E^{11}$	Pa
$ShearModulus(G)$	$7.93E^{10}$	Pa
ρ_{steel}	$7.85E^3$	kg/m^3
m_{tower}	$0.97E^6$	kg
z_{cm}	57.8	m
z_{hub}	150	m
$I_{x_{tower}}$	$4.81E^9$	$kg \cdot m^2$

Table 2.6: Calculated values of the tower

Turbine

The turbine is considered a mass with inertia, as such, information on the turbine contains the weight of the rotor nacelle assembly and the blades. The moment of inertia of the RNA is given. However the weight of the blades are not included in the given value, the IEA reference turbine does not mention anything about the moment of inertia of the blades. To make up for this, the moment of inertia of the RNA is adjusted with the blades modelled as distanced point masses using the parallel axis theorem [3].

$$I_{xx_{RNA}} = I_{xx_{RNA_{IEA}}} + 3 \cdot m_{blade} \cdot x^2_{cm_{blade}} \tag{2.64}$$

Where $I_{xx_{RNA}}$ is the moment of inertia of the RNA, including the blades, $I_{xx_{RNA_{IEA}}}$ represents the given moment of inertia in the definition of the reference turbine, m_{blade} is the mass of the blades. $x_{cm_{blade}}$ is the location of the blade's centre of gravity, which is considered to be the distance to the centre of the RNA.

The coordinate of the centre of gravity of the RNA, including blades will not need to change as the blades are symmetrical and, in this case considered to be stiff. The given inertia values of the table

The moment of inertia of each component of the RNA is given in B.1 and has the origin of the coordinate system at the tower top. Using the parallel axis theorem [3] this makes the expression for moment of inertia over the SWL($I_{XX_{SWL}}$):

$$I_{xx_{SWL}} = I_{xx_{RNA}} + (m_{RNA} + 3 \cdot m_{blade}) \cdot z^2_{hub} \tag{2.65}$$

where $I_{xx_{RNA}}$ is the moment of inertia of the RNA including blades, m_{RNA} and m_{blade} is the mass of the RNA and blade respectively. $z_{cm_{RNA}}$ is the z-coordinate of the center of mass of the RNA.

Name	Amount	Unit
$Mass_{RNA}$	$820E^3$	kg
$Mass_{Blade}$	$65E^3$	kg
$Mass_{Total}$	$1015E^5$	kg
z_{hub}	150	m
z_{cm}	153.97	m
$I_{XX_{rna_{IEA}}}$	$126.03E^4$	$kg \cdot m^2$
$I_{XX_{rna}}$	$152E^6$	$kg \cdot m^2$
$I_{XX_{rna_{SWL}}}$	$242E^8$	$kg \cdot m^2$
V_{rated}	11	m/s
L_{blade}	150	m

Table 2.7: Values used from the IEA reference turbine regarding the turcentressembly

Spar Floater

The spar floater is the substructure of the floating turbine. The goal is to optimize the design of the spar for cost; which in this case is correlated to mass. For floating wind turbines, the substructure design greatly influences the response characteristics of the system. In most cases, the turbine and tower will be designed first, which creates the requirements for the substructure. As such both the tower and turbine form prerequisite information for designing the spar. The spar is presumed to be of one length with a set diameter where the steel thickness is the same over the length of the spar. The three design variables of the spar are then length L_{spar}, diameter D_{spar} and steel thickness t_{spar}

From these, the water plane area and water plane moment of inertia can be calculated.

$$A_{wp} = \pi \cdot \frac{D_{spar}}{2}^2 \tag{2.66}$$

$$I_t = \frac{\pi}{4} \cdot \frac{D_{spar}}{2}^4 \tag{2.67}$$

The spar is considered to be moored at a third of the length of the draught where $z_{moor} = -\frac{1}{3} \cdot L_{spar}$ calculations in the design of the spar can be divided into three classes; geometrics, stability and structural array.

Geometrics Geometrics contains the calculations regarding the mass, inertia and centre of gravity. The mass is calculated by first considering the necessary ballast. The ballast can be calculated using Archimedes's principle [10]. The necessary ballast can be equated to the submerged volume.

$$\nabla = \rho_{sea} \cdot \underbrace{(L_{spar} - z_{freeboard}) \cdot (\frac{D_{spar}}{2})^2 \cdot \pi}_{V_0} \tag{2.68}$$

$$m_{ballast} = \nabla - (m_{spar} + m_{turbine} + m_{tower}) \tag{2.69}$$

Where ∇ is the weight of a submerged volume of water. V_0 is the underwater volume, $\rho_{seawater}$ is the density of seawater, L_{spar} is the total length of the spar and $z_{freeboard}$ is the amount of freeboard. All $m_{subscript}$ are masses where the subscript indicates what the mass is from. Although self-evident, it should be noted that m_{spar} only considers the steel mass. The total mass of the substructure $m_{substructure}$ is a simple sum.

$$m_{substructure} = m_{spar} + m_{ballast} \tag{2.70}$$

Ballasting a spar is often done with heavy materials like cement. In [48], the ballast density was a design variable ranging from 1281 to $2082 kg/m^3$. The industry now also knows super dense materials like Magnadense [64], which reach densities up to $5100 kg/m^3$. However, as Leimeister et al. pointed out [46], this recent dense material is similar in price to cement. As such, the multidisciplinary optimization that was run by Leimeister et al. found that the optimal design used Magnadense [64]. For purposes of this research Magnadense is always used and $\rho_{ballast}$ is $5100 kg/m^3$. This allows the calculation of the centre of gravity of the ballast.

$$z_{ballast} = \frac{m_{ballast}}{\rho_{ballast}} \cdot \frac{1}{A_{wp}} \tag{2.71}$$

Where $z_{ballast}$ amount of depth that the ballast will take up.

$$z_{cm_{ballast}} = -L + 0.5 \cdot zballast \tag{2.72}$$

Which is essential for calculating the moment of inertia of the spar. Using the parallel axis theorem, the moment of inertia of the spar floater is calculated by considering the steel structure of the spar to be a hollow cylinder(2.62) and the ballast to be a solid cylinder. Then using the parallel axis theorem, the moment of inertia of the substructure in the global coordinate system can be written as:

$$Ix_{ballast} = \frac{1}{2} \cdot m_{ballast} \cdot (\frac{D}{2} - t)^2 \tag{2.73}$$

$$Ix_{spar} = Ix_{ballast} \cdot m_{ballast} \cdot z^2_{cm_{ballast}} + Ix_{steel} + m_{steel} \cdot z^2_{cm_{steel}} \tag{2.74}$$

Where Ix_{spar} is the moment of inertia of the spar within the global coordinate system. $Ix_{ballast}$ is the ballast moment of inertia taken around its centre of mass in equation (2.73).Ix_{steel} is the moment of inertia of the steel structure of the spar calculated with the equation, being considered as a hollow cylinder it is estimated with equation (2.62). m and z_{cm} refer to the mass and centre of mass in the global coordinate system, respectively.

Full System

The full system considers all three parts; turbine, tower and spar. Given that all the inertia has been calculated in the same coordinate system, these can be summed to calculate the inertia of the full system within the global coordinate system.

$$I_{fs_{xx}} = I_{xx_{spar}} + I_{xx_{turbine}} + I_{xx_{tower}} \tag{2.75}$$

The mass of the full system m_{fs} is the full system mass defined as the sum of masses of the spar, ballast, turbine and tower.

$$m_{fs} = m_{spar} + m_{turbine} m_{tower} \tag{2.76}$$

Allowing for the calculation of the z-coordinate of the centre of mass:

$$z_c m = \frac{m_{spar} * z_{cm_{spar}} + m_{tower} \cdot z_{cm_{tower}} + m_{turbine} * zcm_{turbine}}{m_f s} \tag{2.77}$$

With the full system put together, the intact stability calculations are possible. The intact stability refers to the undamaged stability of the structure. A measure for the initial stability is the metacentric height which is the distance between the centre of gravity and the metacentre. The metacentre is the point at which a vertical line through the centre at the heeled position would cross the vertical line at the initial position. The metacentric height is defined as:

$$GM = BM - CG - CB \tag{2.78}$$

where GM is the metacentric height, CG is the centre of gravity as calculated in 2.77. CB is the centre of buoyancy, and presuming still water conditions, this would be at half the depth of the spar.

$$CB = -\frac{(L_{spar} - z_{freeboard})}{2} \tag{2.79}$$

BM is the distance between the centre of buoyancy and metacentric height. It can be calculated as:

$$BM = \frac{I_{fs_{xx}}}{\nabla} \tag{2.80}$$

Where ∇ represents the displaced volume of the substructure, which is just the submerged volume of the substructure.

$$\nabla = \rho_{sw} \cdot (L_{spar} - z_{freeboard}) \cdot r^2 \cdot \pi \tag{2.81}$$

2.2.3. Forcing Calculations

This section is a detailed review of how the aero- and hydrodynamic forcing is calculated in this thesis. The theory of these calculations is covered in section 2.1.1. This section will explain how that is implemented within the model developed for this research. As time domain simulations will be part of the spar design evaluation it is imperative that the optimizer is able to run the simulation for a given set of designs. The conditions under which each design is going to be evaluated will be discussed in the constraint section of the optimization review 3.2. The optimizer will vary the design of the spar, it will not change the environment in which the spar is modelled. As such the wind and wave kinematics are precalculated using the methods discussed in section 2.1.1 and 2.1.1. As both regular and irregular wave conditions, steady and unsteady wind conditions will be evaluated these kinematics are calculated once beforehand and then passed on through the rest of the evaluation. The irregular wind and wave conditions are generated using a reverse Fourier transform. For wind that means getting the spectral amplitudes from the Kaimal spectral densities 2.33, by taking the square root of double the spectral density multiplied by the frequency step size.

$$a = \sqrt{2 \cdot S_u \cdot df} \tag{2.82}$$

Where S_u is the spectral density, for wind using a Kaimal spectrum this is $S_{u_{kaimal}}$ described by:

$$S_{u_{kaimal}} = 4 \cdot I^2 v_w \cdot l * ((\frac{1 + 6 * (f) * l}{v_w}))^{(}\frac{-5}{3}) \tag{2.83}$$

Where I is the turbulence intensity, v_w is the wind speed, l is the turbulence length and f is the frequency. As such using a reverse Fourier transform over frequency range f gives a time series of the wind speed.

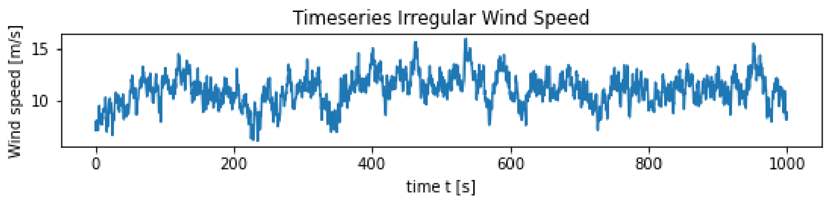


Figure 2.9: Irregular wave height time series using average wind speed of 11 m/s. Turbulence intensity of 0.14% and a turbulence length of 340.2 meters

The irregular wave is essentially a superposition of a set of regular waves. In this model unidirectional waves are considered. Using the amplitudes from the JONSWAP spectrum and a reverse Fourier transform the wave height time series of an irregular wave can be generated.

Where in this case the spectral density is defined as:

$$S_{U_{JS}}(f) = 0.3125 \cdot Hs^2 \cdot Tp \cdot ((\frac{f}{fp})^{-5}) \cdot exp(-1.25 \cdot ((\frac{f}{fp})^{-4})) \cdot \Gamma \tag{2.84}$$

$$\Gamma = (1 - 0.287 \cdot log(\gamma)) \cdot \gamma^{exp(-0.5 \cdot (\frac{\frac{f}{fp} - 1}{\sigma})^2)} \tag{2.85}$$

Where γ is the gamma parameter of the JONSWAP spectrum, fp is the peak frequency defined as 1 over the peak frequency T_p and σ is defined as in 2.11.

$$\sigma = \begin{cases} 0.07 & f \leq f_p \\ 0.09 & f > f_p \end{cases} \tag{2.86}$$

The amplitudes for this spectral density are calculated using equation 2.82 and filling it into equation 2.84. An example of that time-series with a significant wave height of 10 meters and a wave period of 10 seconds is presented below.

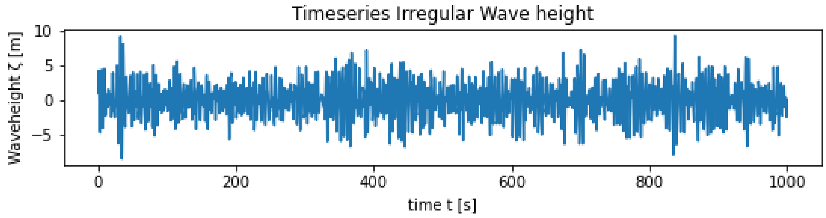


Figure 2.10: Irregular wave height time series using $Hs = 10m$ and $T = 10s$

The same is done for the wave speeds to develop a wave speed velocity over time that corresponds with the wave's irregular wave height. This is displayed in the image below where the wave speed of the last 45 seconds of the time series is plotted at 3 different depths. As can be seen in figure 2.11 the deeper depth shows smaller amplitudes of the wave speed.

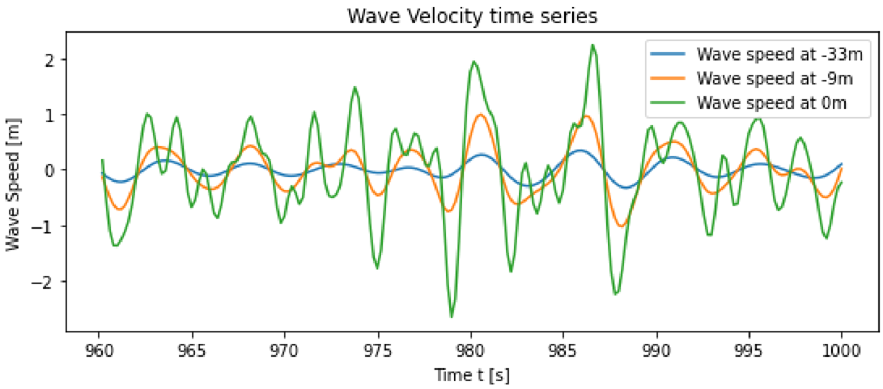
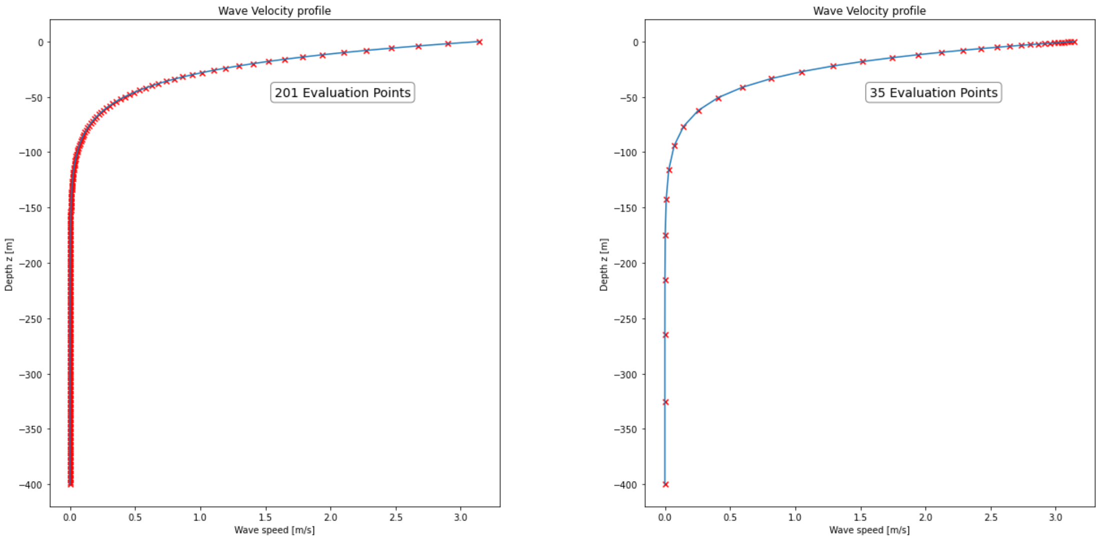


Figure 2.11: Irregular wave speeds at three depths, 33, 9 and 0 meters deep. The time series was generated using $Hs = 10m$ and $T = 10s$

Underwater Profile The response is investigated for both regular and irregular wave forcing. By using the JONSWAP spectrum and the velocity profile of irregular wave forcing along the spar can be calculated. The velocity is calculated at an array of depths underwater. The spacing of which has an impact on the computational efficiency of the optimizer. One approach would be using a linear spacing for the array describing the coordinates underwater z_{uw}. The argument could be made that when a spar is very deep that a linear spacing of the velocity profile adds unnecessary computing time. Considering the linear wave velocity 2.133 it's clear that there is an exponential relationship between depth and wave speed. To demonstrate this the velocity profile for a seabed with a depth of 400 meters wave height and a period of 10 meters and 10 seconds is displayed.



(a) Linear spacing of 2 meters.

(b) Log spacing from z_{bot} to -1 and linear spacing between 0 and 1.

Figure 2.12: Comparison between the shape of the wave velocity profile when modelled using a log spacing scale along the depth and a linear spacing along the depth. The red markings indicate each depth considered.

When the spar has a large draught, i.e. deeper than 150 meters there is not a lot of wave velocity at deeper depths. Looking at equations for wave speed 2.133, it is clear that there is an exponential relationship between the wave speed and wave speed. As seen in the comparison figures 2.12 the profile's shape can be closely estimated using log spacing instead of linear spacing. This will save computing time as more 'relevant' points are being considered. That is to say, with the log spacing, most points considered are in the part of the profile with significantly more velocity. Whereas the points near the bottom, where the velocity becomes negligible are considered less. This will make a difference when the Morison force is integrated over the depth.

As a check for validation, the acceleration and wave velocity can be plotted over crucial time stamps in the wave period. As is expressed in equation 2.2 the wave velocity is cos dependent. This is also the only term dependent on time, which implies that the acceleration will be dependent on sin. As such the two should be 90°. This is best illustrated in figure 2.13 where sin and cosine are plotted over a full cycle of 2π.

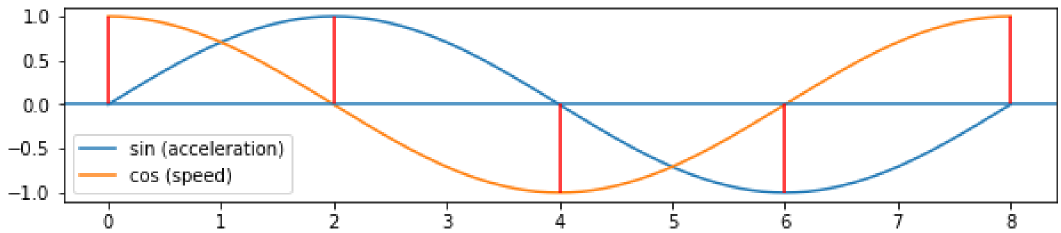


Figure 2.13: Plot of sin and cos with red lines plotted at 0,0.5,1,1.5 and 2 π

As wave velocity and acceleration are dependent on cosine and sin it is to be expected that the wave acceleration is maximum when the wave speed is zero and vice versa. This can be seen in figure 2.14, where the wave velocity and acceleration are plotted for a regular wave with a period of 8 seconds. In the right plot, the blue and purple lines that are the wave speed at the beginning and end of one wave period are at the same maximum speed. These same lines in the plot on the left (wave acceleration) are centred at zero as the wave acceleration is zero here. When the wave speed is zero, red and orange lines (2 and 6 seconds), the wave acceleration is maximum.

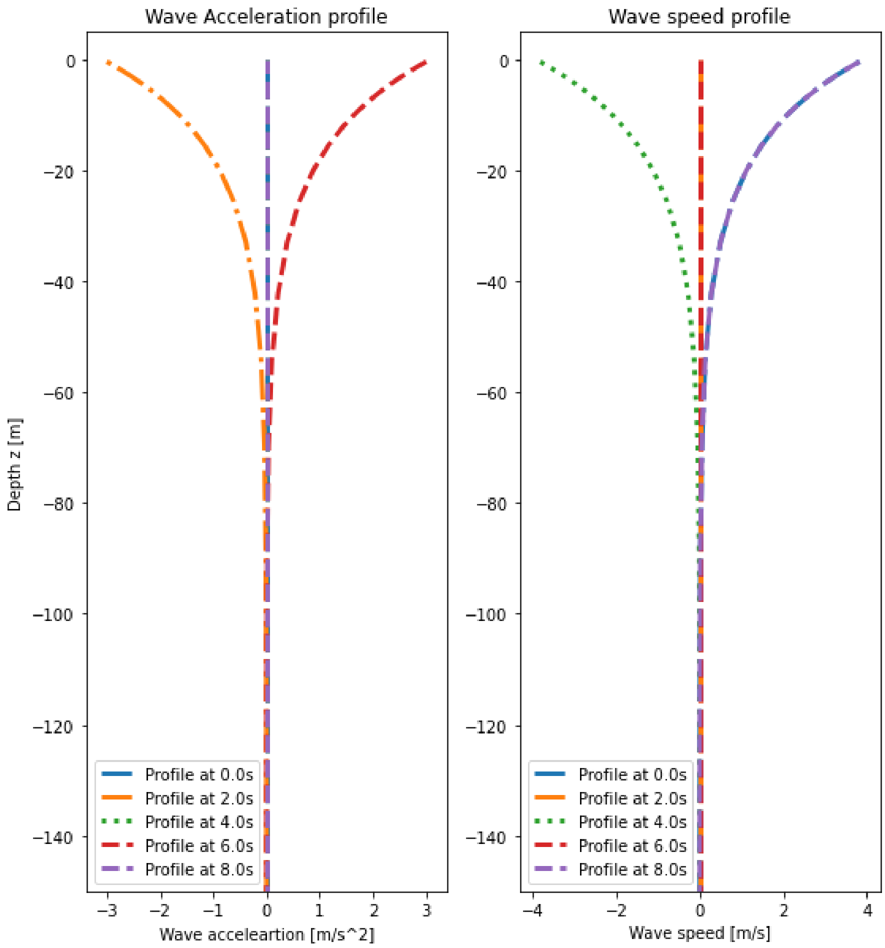


Figure 2.14: Wave speed an acceleartion profile plot for a regular wave of 8 seconds.

Hydrodynamics The Morison equations (equation 2.17) are used to calculate the forcing in the x-direction as a function of depth. To avoid using the entire water depth for calculating the force the position of the substructure within the water is considered. By taking the position z of the structure adding that to the draught, and finding the nearest index in the velocity vector at that depth. The inertial and drag forces can then be defined as:

$$dF_i(z) = (Cm + 1) \cdot \rho_{sw} \cdot \frac{\pi}{4} \cdot D_{spar}^2 \dot{u}(z) \tag{2.87}$$

$$dF_d(z) = Cd \cdot \frac{1}{2} \cdot \rho_{sw} \cdot D_{spar} \cdot (u(z) - (\dot{x} + z \cdot \dot{\theta})) \cdot |(u(z) - (\dot{x} + z \cdot \dot{\theta}))| \tag{2.88}$$

It should be noted that the Froude Krylov part of the Morison forcing equation is taken up in the added mass matrix. The force is integrated over the draught of the spar using Simpson's method where.

$$F_{hydro} = \int_{-L_{spar}}^{0} [dF_i + dF_d]\, dz = simpson((dF_i + dF_d), z_{uw}) \tag{2.89}$$

As such the overturning moment caused by hydrodynamic forcing can be calculated as the forcing at each point multiplied by its distance to the global coordinate system origin.

$$M_{hydro} = \int_{-L_{spar}}^{0} [(dF_i + dF_d) \cdot z]\, dz = simpson((dF_i + dF_d) \cdot z, z_{uw}) \tag{2.90}$$

Aerodynamics For aerodynamic forcing both steady and unsteady wind are considered, and the wind speed for all conditions is computed using the theories described in 2.1.1. Where for computational efficiency reasons and the general scope of this research the thrust is calculated using a similar formulation to 2.31. But in this case, a reduction factor is used to factor in the loss due to the difference from mean wind speed to relative wind speed.

$$F_{wind} = F_{wind_{mean}}(V_{wind}, C_{T_{wind}}) + f_{red} \cdot (F_{wind_{red}}(V_{rel}, C_{T_{rel}}) - F_{wind_{mean}}(V_{wind}, C_{T_{wind}})) \quad (2.91)$$

Where $F_{wind_{mean}}$ is the thrust force created by the average wind speed. So this is calculated using the simplified thrust formula expressed in equation 2.31, the wind speeds V_{wind} and the thurst coefficient stemming from the wind speed.

$$F_{wind_{mean}} = 0.5 \cdot \rho_{air} \cdot A_{rotor} \cdot C_{T_{wind}} \cdot V_{wind}^2 \quad (2.92)$$

$$F_{wind_{red}} = 0.5 \cdot \rho_{air} \cdot A_{rotor} \cdot C_{T_{rel}} \cdot V_{rel} \cdot |V_{rel}| \quad (2.93)$$

$$C_{T_{wind}} = \begin{cases} 0.81 & V_{wind} \leq V_{rated} \\ 0.81 \cdot exp(-a \cdot (V_{wind} - V_{rated})^b) & V_{wind} > V_{rated} \end{cases} \quad (2.94)$$

$$f_{red} = \begin{cases} 0.54 & V_{wind} \leq V_{rated} \\ 0.54 + 0.027 \cdot (V_{wind} - V_{rated}) & V_{wind} > V_{rated} \end{cases} \quad (2.95)$$

Where $C_{T_{rel}}$ is calculated in the same way as $C_{T_{wind}}$ but replacing the wind speed V_{wind} with the relative wind speed V_{rel}.

$$C_{T_{rel}} = \begin{cases} 0.81 & V_{rel} \leq V_{rated} \\ 0.81 \cdot exp(-a \cdot (V_{rel} - V_{rated})^b) & V_{rel} > V_{rated} \end{cases} \quad (2.96)$$

The parameter values a and b are set to 0.5 and 0.65 respectively. $F_{wind_{red}}$ is there to factor in the situations where the movement of the platform itself changes the inflow that the turbine experiences.

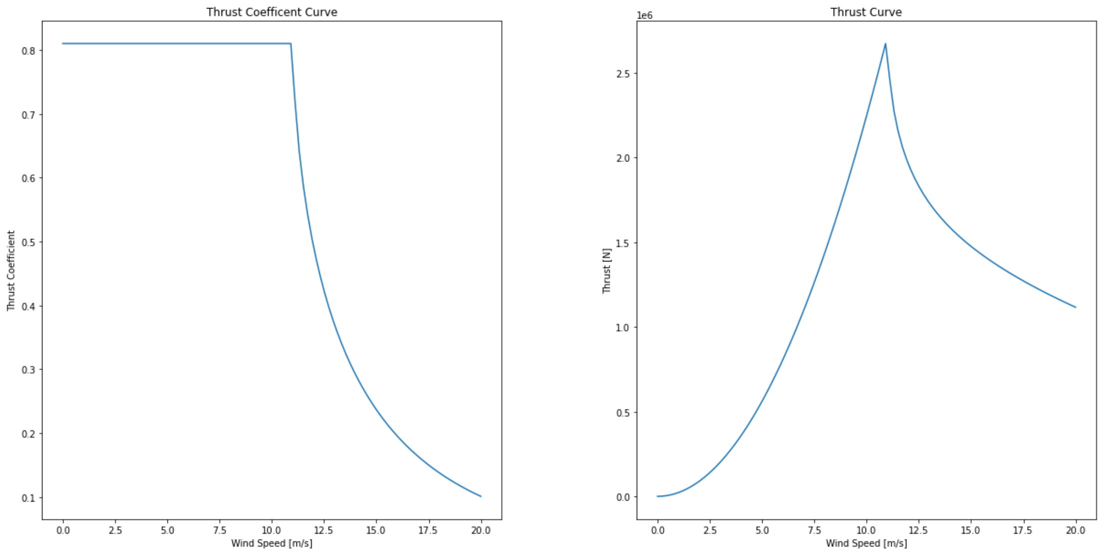
$$V_{rel} = V_{wind} - z_{hub} \cdot \theta_{dot} + x_{dot} \quad (2.97)$$

The overturning moment caused by the thrust force is a simple force multiplied by the arm equation

$$M_{wind} = F_{wind} \cdot z_{hub} \quad (2.98)$$

where z_{hub} is the turbine's hub height.

The function for thrust force F_wind and thrust coefficient C_t can be plotted as a function of wind speed. These two curves for the 15 MW reference turbine are presented in figure 2.15. It is clear that above the rated wind speed, the thrust coefficient and, therefore the thrust decrease with any further increase in wind speed. The thrust force before that point increases exponentially due to the quadratic relationship between thrust and wind speed.



(a) Thrust Coefficient curve using equation 2.95.

(b) Thrust curve using equation 2.91

Figure 2.15: Thrust and thrust-coefficient curve plotted over wind speed from 0 to 20 m/s

2.2.4. Response Calculation

This section will contain the details of the response calculations. First, the general equation of motion is set up after which each entry within the equation of motion is discussed. Finally, the equation of motion is put into an ordinary differential equation solver to get the solution to the system.

Equation of Motion

The response calculations define how the system defined in section 2.2.2 will respond when exposed to environmental forcing. The response is calculated in the time domain approach, this means that a loading time series is used to calculate the response over time of the system. These results can then be post-processed to get information on how the system is behaving. The system's behaviour can be described with the:

$$[\mathbf{M} + \mathbf{A}] \cdot [\ddot{\mathbf{X}}] = [\mathbf{B}] \cdot [\dot{\mathbf{X}}] + [\mathbf{K}] \cdot [\mathbf{X}] - [\mathbf{F}] \tag{2.99}$$

Where $[M]$ is the mass matrix, $[A]$ is the added matrix, $[X]$ is the positional vector with its time derivatives representing the movement of the system in the 3 degrees of freedom. $[K]$ and $[B]$ are the stiffness and damping matrices respectively.

Mass Matrix

The mass matrix can be given by:

$$[M] = \begin{bmatrix} m_{fs} & 0 & z_{cm} \cdot m_{fs} \\ 0 & m_{fs} & 0 \\ z_{cm} \cdot m_{fs} & 0 & I_{fs_{xx}} \end{bmatrix} \tag{2.100}$$

(2.101)

If the centre of the coordinate system is at the centre of gravity of the structure, the mass matrix $[M]$ becomes a diagonal matrix. However in this case the centre of the global coordinate system is located at the tower base. This means that there are two cross-terms introduced in the matrix. The cross terms are in positions $i, j = 5, 1$ and $i, j = 1, 5$. The term $m_{5,1}$ describes what the inertial moment in pitch would be due to surge. The term $m_{1,5}$ describes what the surge force due to pitch would be. Which is expressed by the same term.

$$m_{1,5} = m_{5,1} = m_{fs} \cdot z_{cm} \tag{2.102}$$

Added Mass

The spar is a structure moving through a fluid as such added mass needs to be taken into account. The added mass is defined as the weight of the surrounding volume that needs to be moved for the movement of the structure. This is expressed as a matrix form with entries a_{ij}. Each entry stands for mass associated with force on the body in the i^{th} direction due to a unit acceleration in the j^{th} direction. The subscripts will conform with 6 degrees of freedom systems. Where subscripts 1, 2 and 3 are translational motions: surge, sway and heave. Subscripts 4,5 and 6 represent the rotational motions pitch, roll and yaw respectively. Due to the symmetry of the structure, the added mass matrix is symmetrical in nature and as such not all entries need to be calculated. Furthermore, some of the entries will be zero-valued as the movement in that direction would not cause action on the concerning axes. An example would be the force in x direction due to heave: vertical motion will not result in horizontal forcing. The added mass in the heave is neglected

- a_{11} = Force in x-direction due to surge. Non-Zero
- a_{13} = Force in x-direction due to heave. Zero
- a_{15} = Force in x-direction due to pitch. Non-Zero
- a_{33} = Force in y-direction due to heave. Non-Zero
- a_{35} = Force in y-direction due to pitch. Zero
- a_{55} = Moment around x axis due to pitching. Non-Zero

$$[A] = \begin{bmatrix} a_{11} & 0 & a_{a51} \\ 0 & a_{33} & 0 \\ a_{15} & 0 & a_{55} \end{bmatrix} \tag{2.103}$$

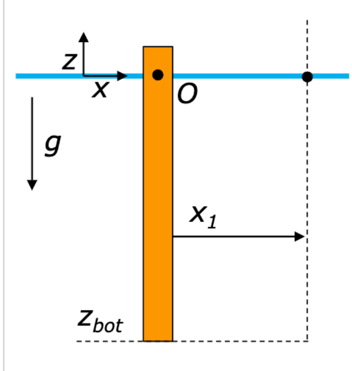


Figure 2.16: FBD surge motion

Added mass in Surge a_{11} would be the mass associated with force in the x-direction caused by the acceleration in the x-direction. Thanks to the simple nature of the spar this can be calculated by integrating the 2D hydrodynamic coefficient of a circle in surge over the length. To derive the added mass in surge the equation of motion for the surge is set up 2.104. EOM:

$$m \cdot \ddot{x}_1 + k_1 \cdot x_1 = \tilde{F}_{Hydro} \tag{2.104}$$

Local motion along the spar depth:

$$x(z) = x_1 \tag{2.105}$$

Using the full morison equation 2.106 it becomes clear that the only term on the right side of 2.104 that involves the acceleration of the body is the hydrodynamic mass force. As such this term can be added in the mass matrix.

Hydro-Force:

$$\tilde{F}_{Hydro} = \int_{z_{bot}}^{0} \left[\underbrace{\rho \frac{\pi}{4} D^2 C_m \left(\frac{\partial u}{\partial t} - \ddot{x}_1 \right)}_{\text{Hydrodynamic mass force}} + \underbrace{\rho \frac{\pi}{4} D^2 \frac{\partial u}{\partial t}}_{\text{Froude Krylov}} + \underbrace{\frac{1}{2} \rho D C_D \left(u - \dot{x}_1 \right) \left| u - \dot{x}_1 \right|}_{\text{Drag Term}} \right] dz \tag{2.106}$$

$$a_{11} = \int_{z_{bot}}^{0} \rho \frac{\pi}{4} D^2 \cdot C_m dz \tag{2.107}$$

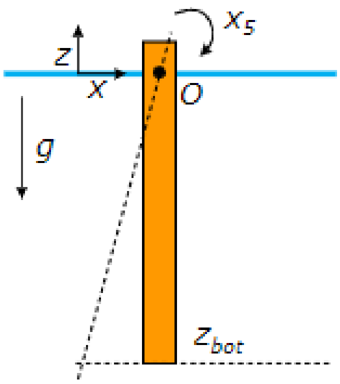


Figure 2.17: FBD pitch motion

Added mass in pitch The same approach is taken for the pitch motion. However in this case inertia and rotation are considered rather than mass and translational acceleration. The equation of motion for the hydrodynamic moment is set up as:

$$I_O \cdot \ddot{x}_5 + c_5 \cdot x_5 = \tilde{M}_{Hydro} \tag{2.108}$$

Local Motion along the spar depth:

$$x(z) = z \cdot x_5 \tag{2.109}$$

Hydro Moment:

$$\tilde{M}_{Hydro} = \int_{z_{bot}}^{0} \underbrace{\left(\rho_{sw}\frac{\pi}{4}D^2 C_m\left(\frac{\partial u}{\partial t} - z\ddot{x}_5\right)\right.}_{\text{Hydrodynamic mass moment}} + \underbrace{\rho\frac{\pi}{4}D^2\frac{\partial u}{\partial t}}_{\text{Froude Krylov}} + \underbrace{\frac{1}{2}\rho D C_D\left(u-(z)\dot{x}_5\right)\left|u - z\dot{x}_5\right|\Big)}_{\text{Drag Term}} \cdot z dz \tag{2.110}$$

Where in equation 2.110 the only term to be multiplied with the acceleration is in the hydrodynamic mass moment term. This is then moved to the added mass term

$$a_{55} = \int_{z_{bot}}^{0} \left[\rho\frac{\pi}{4}D^2 C_m\right](z)^2 dz \tag{2.111}$$

$$\left(I_O + a_{55}\right)\ddot{x}_5 + c_5 x_5 = M_{hydro} \tag{2.112}$$

(2.113)

To be clear this then influences how the forces will be calculated. Although elaborated later on the Hydrodynamic forces are calculated while ignoring the acceleration in mass:

$$M_{hydro} = \int_{z_{bot}}^{0} \left(\underbrace{\rho\frac{\pi}{4}D^2\left(C_m + 1\right)\frac{\partial u}{\partial t}}_{\text{Inertia load}} + \underbrace{\frac{1}{2}\rho D C_D\left(u - z\dot{x}_5\right)\left|u - \dot{x}_1\right|_{\text{drag load}}}\cdot z \right) dz \tag{2.114}$$

Added Mass Cross Term For the cross term in the added mass in pitch, the moment caused by a movement in the surge is considered. Which uses equation 2.106 and multiplies it by the distance z. This means that the cross term in the added mass matrix can be written as:

$$a_{15} = a_{11} \cdot z = \int_{z_{bot}}^{0} \left(\rho\frac{\pi}{4}D^2 \cdot C_m \cdot z\right) dz \tag{2.115}$$

Stiffness matrix

The stiffness matrix can be formed by considering the mooring and hydrodynamic stiffnesses in every degree of freedom. From the equation of motion described in equation 2.99, the stiffness matrix is multiplied by the displacement to get a force or moment (depending on the index in the system). Similar to the added mass matrix there are only coupling terms between pitch and surge. The stiffness can be modelled as seen in figure 2.18

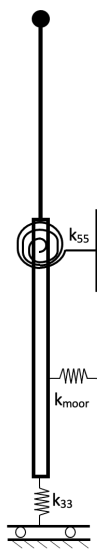


Figure 2.18: Stifness schematic

$$[K] = \begin{bmatrix} k_{11} & 0 & k_{51} \\ 0 & k_{33} & 0 \\ k_{15} & 0 & k_{55} \end{bmatrix} \tag{2.116}$$

The mooring is attached at a third of the draught of the spar. When considering the change in static forces at a displacement in surge the entry in the stiffness matrix affecting surge can be expressed as the mooring stiffness.

$$k_{11} = k_{moor} \tag{2.117}$$

The mooring stiffness is calculated based on a presumption of the natural period in surge of 60 seconds and the mass and added mass in the surge. Where the mass of the spar is considered.

$$k_{moor} = \omega_{n_{surge}}^2 \cdot [m_{spar} + a_{11}] \tag{2.118}$$

Where $\omega_{n_{surge}} = 2\pi cdot \frac{1}{60}$.

The stiffness in heave is the hydrodynamic stiffness caused by a change in buoyancy force when the structure is submerged further in water. That buoyancy force can be expressed as the weight of the increased volume when the structure moves along the z-axis. As the stiffness is multiplied by the translation along the z-axis it is the change volume of the spar divided over the translation.

$$k_{33} = \rho_{sw} \cdot g \cdot A_{wp} \tag{2.119}$$

where $A_w p$ is the waterplane area.

The pitch stiffness $k_5 5$ is based on the hydrodynamic stability properties of the spar. The spar is self-stabilizing when the centre of mass lies below the centre of buoyancy. This means that when the spar takes on some pitch angle θ the buoyancy and gravity force will create a righting moment. On top of that, there is also a hydrodynamic stiffness caused by having to rotate through a liquid. This term is based on the second moment of the waterplane area. The stiffness in pitch can thus be calculated by dividing the moment caused by a change in pitch, over the pitch. As small angles are presumed this ratio can be calculated as:

$$k_{55} = \frac{M_{hydrostatic}}{\theta} = g \left(\underbrace{\rho_{sw} \cdot I_{xx_{wp}}}_{\text{Added Hydrodynamic Stiffness}} + \underbrace{\cdot \nabla \cdot z_{cb}}_{\text{Bouyancy}} - \underbrace{m_{fs} \cdot z_{cm}}_{Gravity} \right) \tag{2.120}$$

The cross term in the stiffness matrix refers to the moment caused to pitch by surge translation or the addition to the force in x direction caused by pitching. As such it can be described by the extension of the mooring stiffness multiplied by the distance of the mooring point to the centre of the coordinate system.

$$k_{15} = k_{51} = -z_{moor} \cdot k_{moor} \tag{2.121}$$

Damping Matrix

The damping matrix describes how the system loses energy. A floating wind turbine is exposed to aero- and hydro-dynamic forcing and as such there is also aero- and hydro-dynamic damping. This problem has been approached using Meng's (2022) analytical approach to the damping terms for a spar floating wind turbine [55], ignoring the radiation damping terms.

Aerodynamic Damping Aerodynamic damping comes from the interaction between the rotor and the wind. The assumptions are made that the connections between the nacelle tower top and rotor are rigid. The blades are also assumed to be rigid, ignoring blade- and edge-wise vibrations. The vibration velocity of the blades is presumed to be much smaller than the inflow of wind speed, meaning that linearization should offer a good approximation. The vibration of each blade element can be divided into two parts, one is caused by the platform motions, and the other by the tower top. Using these assumptions Meng (2022) derives an analytical expression for the aerodynamic damping per blade element that is integrated over the blade. Where the damping force can be expressed as:

$$\mathbf{F}_{\text{Aerop}}^{\text{damp}} = \mathbf{C}_{\text{Aero}}^{\text{pp}} \dot{\mathbf{U}} + \mathbf{C}_{\text{Aero}}^{\text{pt}} \dot{\mathbf{u}} \tag{2.122}$$

Where $\mathbf{F}^{\text{damp}}_{\text{Aerop}}$ is the aerodynamic damping force, $\mathbf{C}^{\text{pp}}_{\text{Aero}}$ is the damping caused by platform motions **U**. $\mathbf{C}^{\text{pt}}_{\text{Aero}}$ is the damping caused by the tower top motions **u** For application in this research, some adjustments had to be made, as the thrust force is calculated from an average thrust coefficient, and the aerodynamic coefficients are immediately integrated over the length of the blade. The damping matrix for platform-induced damping can be expressed as:

$$\begin{aligned}\mathbf{C}^{\text{pp}}_{\text{Aero}} &= \mathbf{A}_{\text{pt}}\mathbf{C}^{\text{tp}}_{\text{Aero}} \\ &= \begin{bmatrix} c_{xU_1} & 0 & c_{xU_5} \\ 0 & c_{zU_3} & 0 \\ h_R c_{xU_1} & 0 & c_{\theta_y U_5} + h_R c_{xU_5} \end{bmatrix}.\end{aligned} \tag{2.123}$$

Where $mathbfA_{\text{pt}}$ is the transformation matrix to move damping matrix to the same space.

$$c_{xU_1} = \frac{T}{V_0} \tag{2.124}$$

$$c_{xU_5} = h_R \frac{T}{V_0} \tag{2.125}$$

$$c_{\theta_y U_5} = \frac{3}{2}\frac{T}{V_0} \tag{2.126}$$

$$c_{zU_3} = 0 \tag{2.127}$$

The damping introduced by the tower top motions are defined as:

$$\mathbf{C}^{\text{pt}}_{\text{Aero}} = \mathbf{A}_{\text{pt}}\mathbf{C}^{\text{tt}}_{\text{Aero}} = \begin{bmatrix} c_{xx} & 0 & 0 \\ 0 & 0 & 0 \\ h_R c_{xx} & 0 & c_{\theta_{\theta},\theta_y} \end{bmatrix} \tag{2.128}$$

Where after similar simplification:

$$c_{xx} = \frac{T}{V_0} \tag{2.129}$$

$$c_{\theta_\theta \theta_\theta} = \frac{3 \cdot L^3_{blade}}{6}\frac{T}{V_0} \tag{2.130}$$

After simplification, the terms in the aerodynamic damping matrix become, see C for the complete expression for the damping terms as intended by Meng(2022) [55].

Hydrodynamic Damping The hydrodynamic damping is derived from the Morison equation by integrating the viscous damping force over the draught of the spar. The viscous damping force in the x-direction and moment over the y-axis is defined as.

$$F^{\text{Viscous}}_{1,\,\text{damp}}(t, Z) = -C_D \rho_w D \,\mathrm{d}Z\, |v_1(t,Z)| \left(\dot{U}_1 + \dot{U}_5 Z\right) \tag{2.131}$$

$$F^{\text{Viscous}}_{5,\,\text{damp}}(t, Z) = -C_D \rho_w D \int_{-L_{spar}}^{0} \left(\dot{U}_1 Z + \dot{U}_5 Z^2\right) |v_1(t,Z)| \,\mathrm{d}Z \tag{2.132}$$

Using linear wave theory in deep water

$$v_1(t, Z) = H_s \sigma e^{kZ} \sin \sigma t \tag{2.133}$$

and the dispersion relation

$$\sigma^2 = gk \tag{2.134}$$

Inserting 2.133 and 2.134 into the linearized damping expressed in equations 2.131 and 2.132 that the damping forces can bee written as:

$$F^{\text{Viscous}}_{1,\,\text{damp}}(t, Z) = -C_D \rho_w D H_s \sigma |\sin \sigma t| \left[A_1 e^{k(Z-h_T)}\dot{U}_1 + A_2 e^{k(Z-h_T)}\dot{U}_5\right]^{0}_{-L_{spar}} \tag{2.135}$$

$$F_{5,\text{ damp}}^{\text{Viscous}}(t,Z) = -C_D\rho_w DH_s\sigma|\sin\sigma t|\left[A_2e^{k(Z-h_T)}\dot{U}_1 + A_3e^{k(Z-h_T)}\dot{U}_5\right]_{-L_{spar}}^{0} \tag{2.136}$$

Where:

$$\begin{aligned} A_1 &= \frac{1}{k} \\ A_2 &= \frac{Z}{k} - \frac{1}{k^2} \\ A_3 &= \frac{Z^2}{k} - \frac{2Z}{k^2} + \frac{2}{k^3} \end{aligned} \tag{2.137}$$

This leads to the damping coefficients:

$$\begin{aligned} c_{U_1U_1}^{vis} &= C_D\rho_w DH_s\sigma\tau A_1e^{kZ}\big|_{-L_{spar}}^{0} \\ c_{U_1U_5}^{vis} &= C_D\rho_w DH_s\sigma\tau A_2e^{kZ}\big|_{-L_{spar}}^{0} \\ c_{U_4U_4}^{vis} &= C_D\rho_w DH_s\sigma\tau A_3e^{kZ}\big|_{-L_{spar}}^{0} \end{aligned} \tag{2.138}$$

Where τ is defined as $\frac{2}{\pi}$. Finally this viscous damping matrix can be defined as:

$$\mathbf{C}_{\text{Vis}}^{\text{Morison}} = \begin{bmatrix} c_{U_1U_1}^{vis} & 0 & c_{U_1U_5}^{vis} \\ 0 & 0 & 0 \\ c_{U_5U_1}^{vis} & 0 & c_{U_5U_5}^{vis} \end{bmatrix} \tag{2.139}$$

Ordinary Differential Equation Solver

The system is solved using an ODE solver. all matrices and the force vector are known the ODE solver solves for what solution of acceleration, speed and position the equation 2.99 is true. For this thesis, an ODE solver is used which uses an Adams/BDF method with automatic stiffness detection. The solver is part of a larger library Scipy. The solver's manual, it refers to two papers that describe the automatic ODE solving method [69] and [32].

Test Cases

To test the time domain simulation a single simulation is run using a substructure defined by the following design variables:

$$[x] = [D, L, T]^T = [20, 170, 0.13] \tag{2.140}$$

The water depth for all the tests run in this section will be set to 400 metres.

Regular Wave response The regular wave response is tested for a wave height of 8 meters and a period of 12 seconds. Damping is considered from both the platform and the turbine motions, this is an overestimation of the damping as the turbine motion damping is supposed to mimic the damping caused by the vibration of rotating blades. The simulation is run for 1500 seconds and the first 500 are ignored to remove any transient behaviour from the system. For the power spectral density analysis only 100 seconds are considered. The response is plotted below in figure 2.19. The figure shows the response over time on the left and the power spectral density on the right. what can be seen is that the surge and pitch response fluctuate around zero, which is to be expected in regular waves. The pitch and surge motion are not caused by any thrust force at the tower top and as such, there is no offset in pitch or surge.

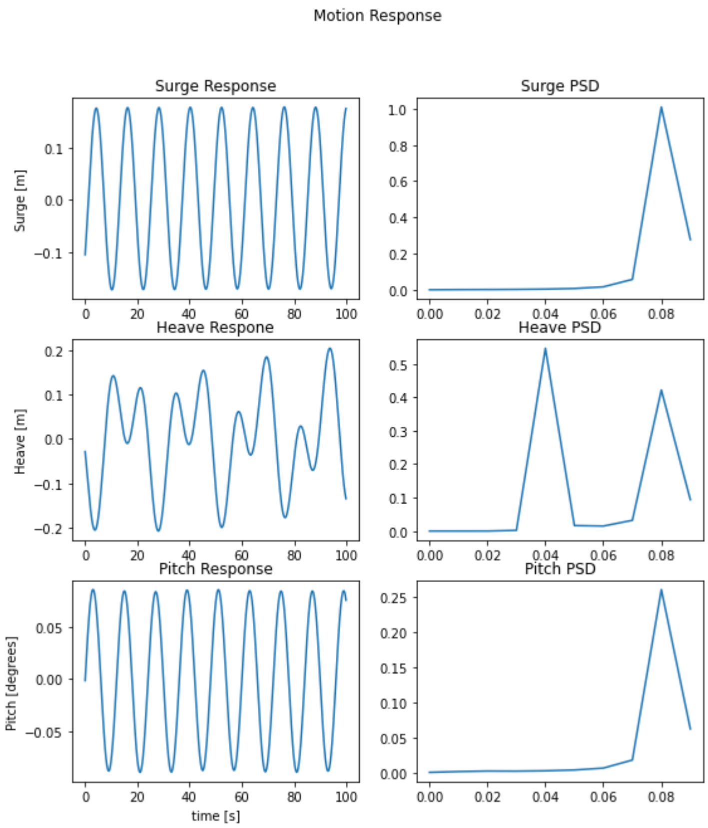


Figure 2.19: Response of the system in regular wave of 8 meters and period of 12 seconds

As the pitch is a response of the mooring force increasing and decreasing out of phase with the surge, the pitch response will be of the same frequency as the surge response. The power spectral density analysis of these 100 seconds indicates that there is a heave response at half the frequency of the regular wave and pitching, this is due to the natural frequency in heavy caused by the change in buoyancy caused by a regular wave and the inertia of the system itself.

Steady Wind response For the steady wind response first, a wind speed of 11 m/s is considered, this is the rated wind speed and as such, it should cause the largest pitching response. This is because above rated thrust force will decrease as the thrust coefficient is modelled to decrease from the rated wind speed onwards. This is to emulate the blades pitching out of the wind to keep the turbine spinning at rated RPMs. There is no response in heave from pitching as there are no coupling terms from pitch to heave. When a wind turbine is pitched by a thrust force that is presumed to be constantly horizontal there is no contribution to the heave behaviour from the thrust force. The pitch response pitches around $1°$, which if it is the largest response of this turbine could indicate that it is slightly oversized for its application.

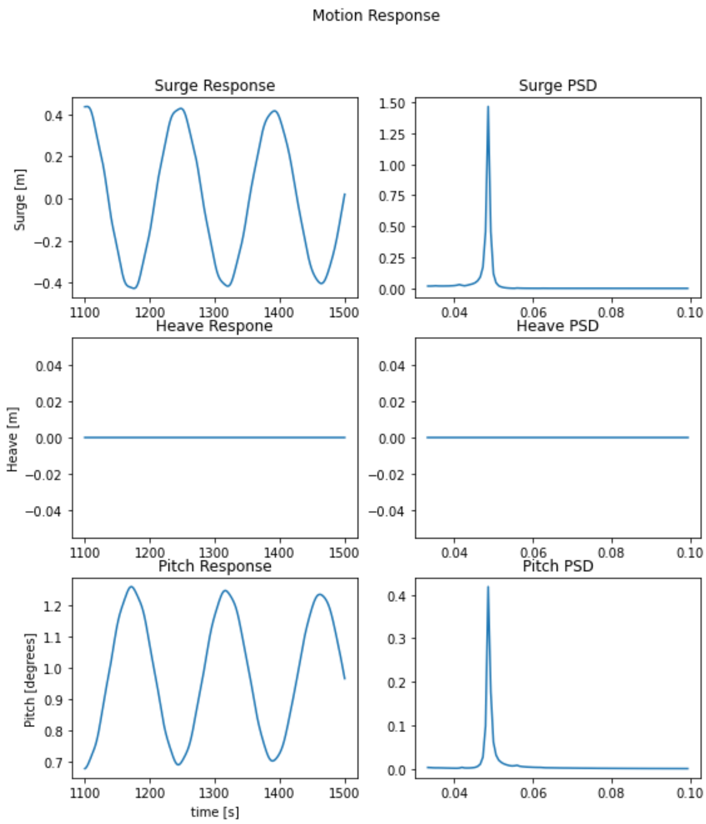


Figure 2.20: Response of the system in steady wind of 11 m/s.

If a wind speed of just under-rated wind speed is used there is a second excitement coming from the change in thrust force. For a steady wind speed of just under rater wind speed, when the turbine surges into the wind the relative wind speed increases and actually increases over the rated wind speed. As such the thrust coefficient decreases, which causes a decrease in thrust and therefore in pitch and surge response. This behaviour is seen in the pitch response.

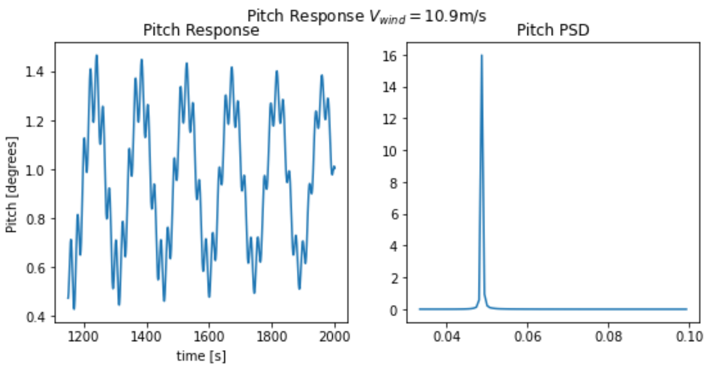


Figure 2.21: Response of the system in steady wind of 10.9 m/s.

Steady Wind and Regular Wave Response The response to a combination of regular waves and steady wind speed is presented below. A rated wind speed 11 m/s is used in combination with a wave height of 8 meters and a period of 8 seconds. What can be seen in the figure is that there is a heave frequency response at both frequencies of its own natural frequency and that of the wave velocity. Both the pitching and surge have the most response at the same frequencies. This indicates that the surge response for this simulation is governed by either wind or wave. And as the wave natural frequency is $\frac{1}{8} \approx 0.013$ it must be governed by the waves. The pitch offset is caused by the steady wind and its fluctuation is caused by the waves.

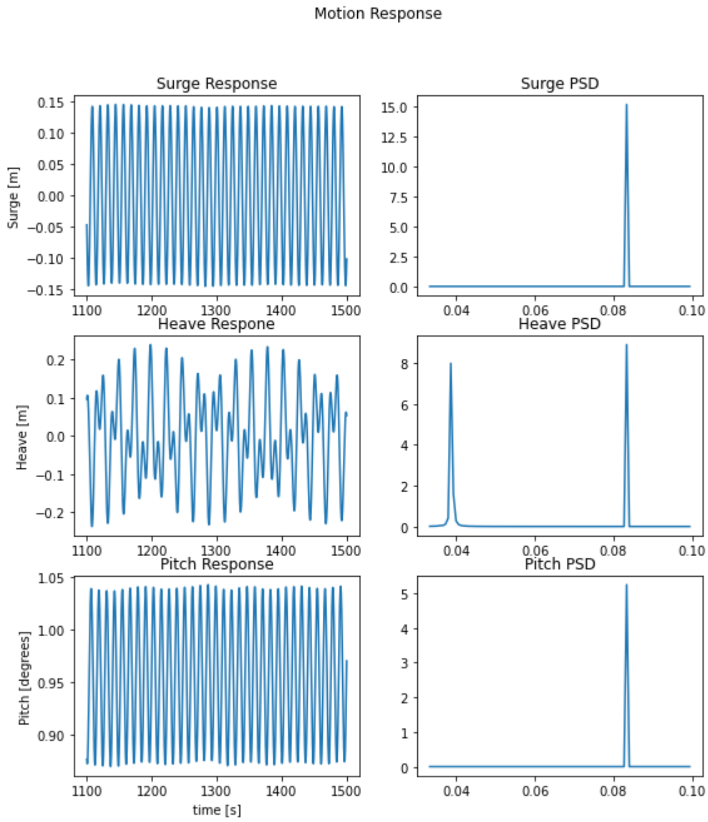


Figure 2.22: Response of the system in steady wind of 11 m/s, wave height 8 m and period 8 seconds.

Irregular Wave | Steady Wind response If the wind is left as a constant and the Jonswap spectrum is used to generate an irregular wave will cause an irregular motion response much nearer to what would happen at sea. As the spectrum is a sum of a set of regular waves the power spectral density should look like the Jonswap spectrum. This is clearly seen in the figure 2.23 where the response still fluctuates around zero, but the fluctuations are far less predictable. There is no difference in the heave response plot as the change in buoyancy has not been related to the irregular wave. The pitch response is dominated by the waves as such it mimics the wave spectrum seen in the heave response spectral density plot.

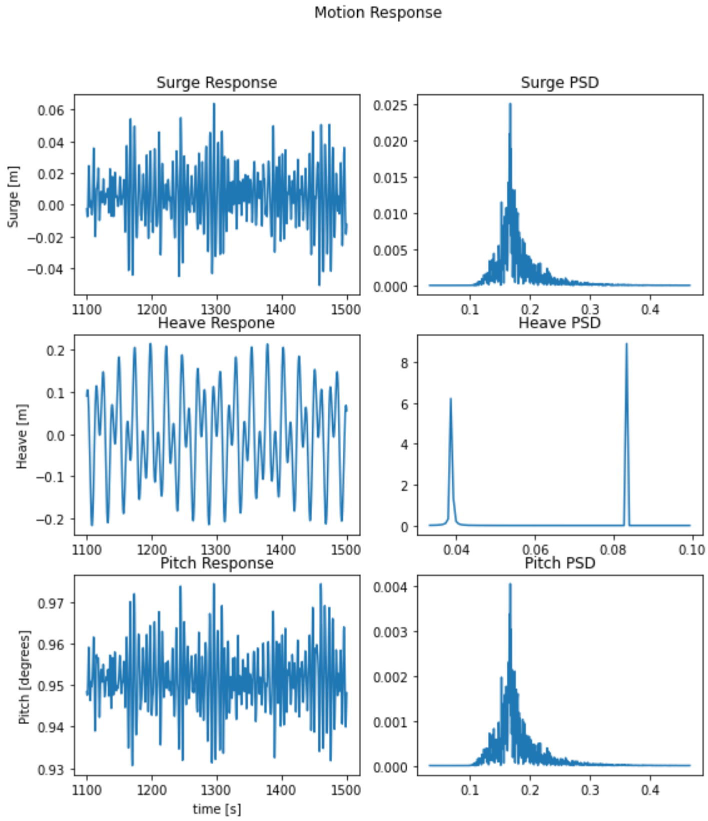


Figure 2.23: Response of the system in steady wind of 11 m/s, and an irregular wave pattern generated with significant wave height 8 m and significant period 12 seconds.

Irregular Wave | Unsteady Wind response Irregular wave and unsteady wind lead to a combination of the Kaimal and Jonswap spectra in the spectral density plots of the heave and pitch response. It's interesting to see that the surge response is about as influenced by the wind as by the waves. The surge spectral density is seen to peak at the slow wind-changing frequencies and at the quicker wave frequencies. Whereas the spectral density for the pitch shows a negligible peak at the wave frequencies (≈ 0.15 to 0.20).

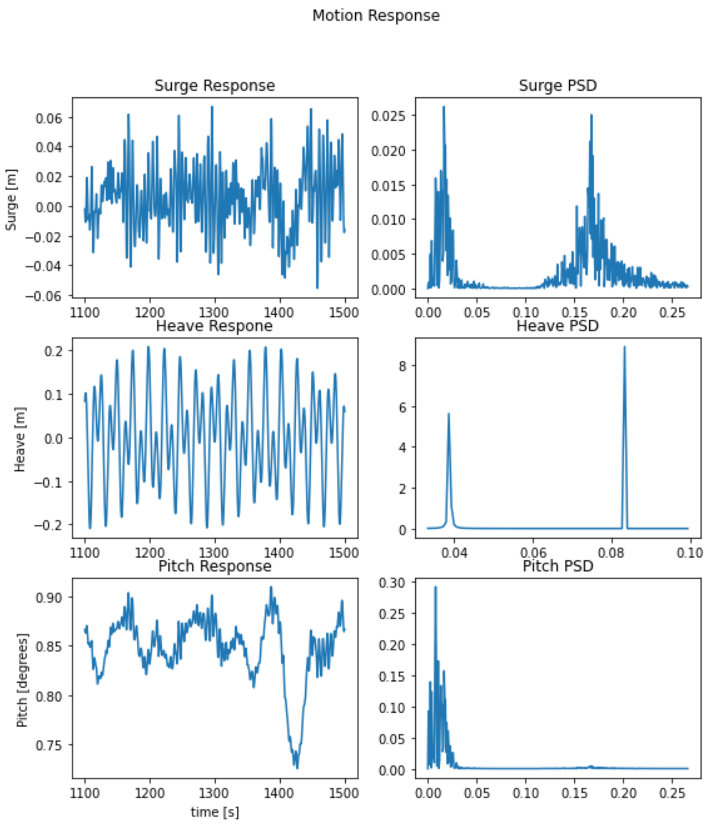


Figure 2.24: Response of the system in the unsteady wind with a ten-minute average of 11 m/s, and an irregular wave pattern generated with significant wave height 8 m and significant period 12 seconds.

5MW spar comparison case

For some validation of the response characteristics of this turbine, the natural frequencies can be compared to the research done in 2021 by Zhang Y [86]. Which investigated and the dynamic response of the 5MW reference turbine on a specified spar floater. Where the response was validated by comparing it to six other investigations of the dynamic response of that reference turbine and floater. The natural frequencies found are presented in the image of a table below taken from that research.

Table 4. Natural frequencies of SPAR-type FOWT.

	Surge (rad/s)	Heave (rad/s)	Pitch (rad/s)
Our work	0.050	0.207	0.211
FAST	0.050	0.201	0.214
Ruzzo et al. [15,43]	0.031	0.199	0.202
Salehyar [44]	0.044	0.198	0.228
Bae et al. [45]	0.05	0.20	0.22
Ma et al. [46]	0.050	0.201	0.220
Yue et al. [47]	0.050	0.200	0.223

Figure 2.25: Table from dynamic response investigation [86]

The natural period found for the test case in this response calculations for surge heave and pitch are 0.114, 0.222 and 0.244 respectively. It is thus found that the responses in heave and pitch are similar and therefore likely to be valid. However, the surge natural frequency is twice as large. This is attributed to the substructure in this test case being significantly larger that the superstructure. This can be seen when comparing ratios of the spar vs superstructure investigated by Zhang [86]. The spar of the 5MW turbine had a depth of 120 meters whereas the hub height was 90 meters. This is the same ratio as the test spar where the depth is 200 with a superstructure hub height of 150. However, the diameter of the spar in the test case for this research is twice as large as the tower diameter whereas the investigation of the 5MW turbine floater was considered to be about 1.5 times as large. This becomes a significant difference when considering the effect it has on the mass of the structure. Furthermore, the natural frequency in surge is determined by the mooring stiffness and mass of the structure so this could also be the effect of a different choice in mooring stiffness calculation. The model is considered to give realistic responses.

2.2.5. Fatigue Calculation

This section goes over the fatigue calculations. The theory behind fatigue calculation is covered in section 2.1.2, where approaches for fatigue damage calculations in the time and the frequency-domain are discussed. For the time domain simulation, the overturning moment history can be used to calculate the stress history at the tower base.

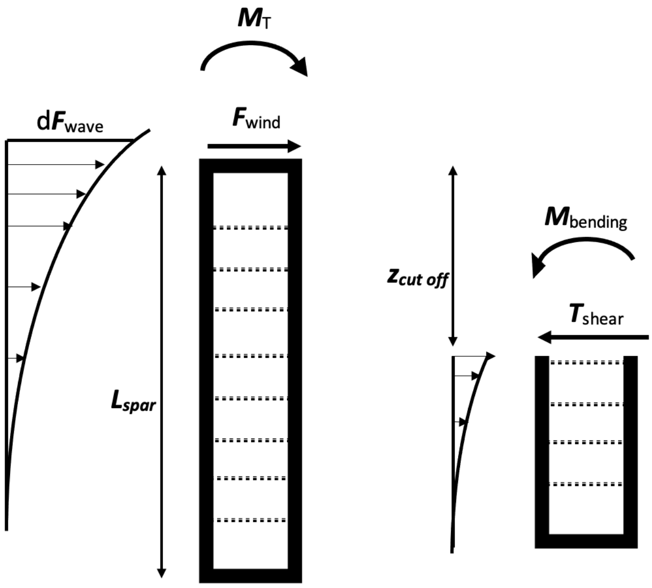


Figure 2.26: Schematic of how the spar is 'sliced' from the weld locations

The fatigue analysis inherits information from the time domain simulation. Most importantly the forcing and positional history of the floater. The spar is presumed to be made of a set of cylinders welded together at 10 meter intervals. These welds are presumed to be the quickest to fail and are therefore investigated for lifetime fatigue. The stress is calculated over the steel cross-sectional area of the spar, thus ignoring any added stiffness that the ballast would provide. Meaning that any optimum solution that is led by the fatigue calculations is expected to suggest a conservative thickness.

Stress History The first step is to calculate what the stress will be in the structure. This is done by setting up the equations of motions as described in equation 2.99. If the free body diagram at a weld is shown it's clear that Newton's first law should still apply; the sum of all forces and moments should equal the mass(or inertia) multiplied by the acceleration. From the response analysis the forces, moments and velocities are known. As such if the structure is 'sliced' open at a weld, the internal forces should still allow for that balance to be made up. Even for a small part of the spar:

$$dm \cdot \ddot{\mathbf{x}} = \sum F \tag{2.141}$$

$$dI \cdot \ddot{\theta} = \sum M \tag{2.142}$$

As the positional and velocity time history of the structure are known from the response analysis, the accelerations can be calculated by presuming linear acceleration between time steps.

$$\ddot{x}_i = \frac{\dot{x}_{i+1} - \dot{x}_i}{t_{i+1} - t_i} \tag{2.143}$$

where x can be any of the degrees of freedom.

The mass of the 'sliced' FBD can be calculated as:

$$dm = \frac{L_{spar} - z_{cutoff}}{L_spar} \cdot m_{spar} \tag{2.144}$$

The same ratio can be applied to the moment of inertia

$$dIxx = \frac{L_{spar} - z_{weld}}{L_spar} \cdot I_{xx_{spar}} \tag{2.145}$$

where z_{weld} is the coordinate from which the 'slicing' is being done. Which allows for the calculation of the shear force.

$$T_{shear} = dm \cdot \ddot{x} - simpson(F_{x_{hist}}, z_{cutoff}) \tag{2.146}$$

Which allows for the shear stress τ calculation

$$\tau = \frac{T_{shear}}{A_{crosssection}} \tag{2.147}$$

Where $A_{crosssection}$ is the cross sectional area of the spar. Calculated from a hollow cylinder cylinder.

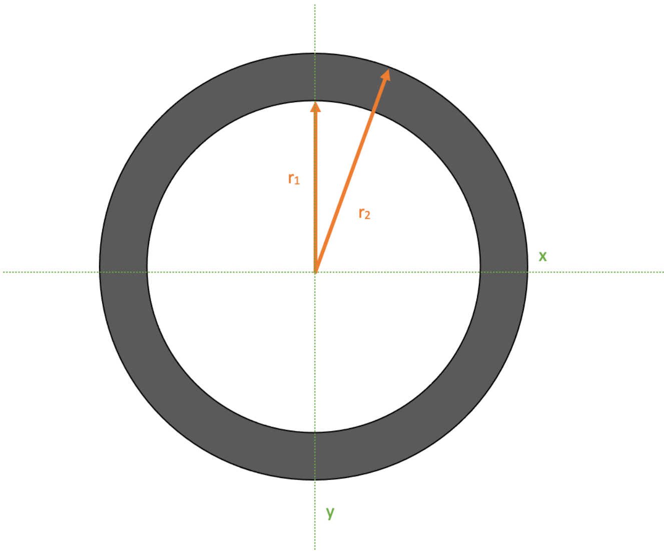


Figure 2.27: Cross section of the spar

$$A_{crosssection} = (r2^2 - r1^2) \tag{2.148}$$

The bending stress is calculated using the overturning moment at the weld locations.

$$M_{overturning} = dI_x \cdot \ddot{x} - simpson(T_{shear} \cdot z_{cutoff}, z_{cutoff}) \tag{2.149}$$

So the bending stress becomes:

$$\phi = \frac{M_{overturning}}{\frac{Ix_{area2}}{y}} \tag{2.150}$$

Where y is the distance from the neutral line, to get the biggest stresses this is considered to be the radius of the spar. $y = 0.5 \cdot D_{spar}$. Ix_{area2} is the second moment of the cross-sectional area of a hollow cylinder.

$$Ix_{area2} = \frac{\pi}{4} \cdot (r_2^4 - r_1^4) \tag{2.151}$$

where r_1 and r_2 are the inner and outer diameters of the spar.

Lifetime Stress and Cycle calculation A rainflow counter is used to get the cycles and stress amplitudes over the lifetime of the structure. With any given stress history the signal is first filtered through a hysteresis, or 'peak-valley' filter to filter out any noise in the stress history. After which the rainflow count cycle is used to get the range, and average of the stress history.

The range and average are calculated from the stress history which is only a limited amount of time. Stress is presumed to increase linearly with time. So the cycles can be scaled up in time as:

$$N_{lifetime} = n_{history} \cdot \frac{t_{lifetime}}{t_{stresshistory}} \tag{2.152}$$

From here the lifetime equivalent stress can be calculated as the following sum:

$$\phi_{eq} = \sum (N_{lifetime} \cdot (\frac{\phi^m_{range}}{n_{eq}})(\frac{1}{m}) \tag{2.153}$$

2.2.6. Lifetime fatigue and Environment

To be able to model the lifetime fatigue of a floating wind turbine it's important to be able to represent all the combinations of wind and wave speeds. The largest contributor to lifetime fatigue will be the condition that occurs most at the given sight. For this research, the decision is made to base the lifetime fatigue on the method described by Gao (2015) [49]. Where a method for determining the joint probability of wind speed, significant wave height and significant wave period is presented. In this research, the Norway 5 site is used with an adjusted depth to allow for a realistic substructure for the 15 MW reference turbine and tower [27].

The total lifetime damage is calculated by analysing the time series of bending stress at every weld on the substructure. The time series for bending stress is calculated using the response, and the response can be calculated for the set of environmental conditions listed in table 2.5. The kinematics for each environmental condition is precalculated. The Weibull distribution of the wind is based on a 10-minute average wind speed. For a realistic simulation, the wind should be simulated as changing throughout those ten minutes. As such the Kaimal spectrum (eq:2.33) is used for wind kinematics. The wave kinematics are also produced using the significant wave height from the table and use the Jonswap spectrum to generate the irregular wave time series 2.84. It can be seen in the table that by using the weighted average of the wave period there are as many wave time series as there are wave heights.

3
Optimization

This chapter regards the theory and methodology of optimization and how it relates to this research. Optimization is a discipline within itself and as such will first be broadly introduced in the theory section. Covering both gradient and non-gradient based approaches. The decision is made to use a non gradient-based optimization algorithm: simulated annealing. In the methodology section, this algorithm will be thoroughly explored and tested on a set of test functions.

3.1. Theory: Optimization

Optimization is used in many engineering problems and has become a useful tool for exploring design spaces. An optimization is a minimization of some objective function. By defining an objective function, the goal is to find the set of design variables that minimize that objective. Formally expressed as:

$$\begin{array}{ll} \text{minimize} & f \\ \text{with respect to} & x \in \mathbb{R}^n \\ \text{subject to} & \widehat{c}_j(x) = 0, \quad j \in [1, \widehat{m}] \\ & c_k(x) \geqslant 0, \quad k \in [1, m] \end{array} \tag{3.1}$$

Where f is the objective function, x is the set of design variables within the design space $\mathbb{R}^n$. $\widehat{c}_j$ and $c_k(x)$ are the equality and inequality constraints respectively. There are many approaches to minimizing the objective function f, the first clear division in approaches is gradient vs non-gradient-based optimizations.

3.1.1. Gradient Based Methods

Gradient-based methods try to find the gradient within the design space to move in a direction that leads to a smaller outcome before re-evaluating a new point. By using the design space's gradient information, the direction in which to move to lead to a smaller outcome is clear. Many algorithms perform gradient-based optimizations. Determining the direction of the search is where the key differences lie.
Before determining the search direction, it is helpful to consider how such a gradient-based optimization method moves along a given direction. The step size determining algorithm is often referred to as a line search. It determines how big of a step to take before re-evaluating whether or not a new local or global minimum has been found. At this point, the gradient would need to be re-evaluated. Typically this point is found by finding a point that adheres to the Wolfe conditions introduced in 1969 [82]. The first condition is the 'sufficient decrease' condition which checks whether the next point considers lies far enough below the initial point

$$f\left(x_k+\alpha p_k\right) \leq f\left(x_k\right)+\mu_1 \alpha g_k^T p_k \tag{3.2}$$

Here f is the objective function, x_k the initial point, α the stepsize and p_k the search direction. The tolerance μ_1 determines the slope of the steepest descent line.

The next thing to check is the curvature condition, where the line search checks whether or not the curvature at the new point is less than the previous point.

$$g\left(x_k + \alpha p_k\right)^T p_k \geq \mu_2 g_k^T p_k \tag{3.3}$$

Here g is the gradient of the objective function, and μ_2 is the tolerance. It should be noted that there is a relationship that $\mu_1 << \leq \mu_2 \leq 1$, which prevents the line search from getting stuck. Optimizations can be done with many versions of the line search adhering to strong or weak Wolfe conditions with different tolerances. The Wolfe conditions are considered strong when all tolerance has to be absolute in the curvature condition.

Suppose an objective function is defined as $f(x)$ where x is defined as a set of design variables. Presuming a smooth function, the first and second derivative of $f(x)$ can be defined as the gradient g_i and the hessian A_{ij} where i and j are the indexes of the variable being derived to.

$$g_i(x) = \frac{\partial f}{\partial x_i} \tag{3.4}$$

$$A_{ij}(x) = \frac{\partial^2 J}{\partial x_i \partial x_j} \tag{3.5}$$

One can imagine that with large amounts of design variables, the computational expense to calculate the Hessian becomes very large as such most gradient-based methods will differ in the approach to estimating gradients and Hessians.

Steepest Descent What is often considered the most intuitive of methods is the *steepest-descent* method. Reviewed by Meza in 2010 [56], the steepest descent finds its basis in the fact that for a continuous smooth function, the gradient points in the opposite direction to the steepest descent. This method doesn't store any information from previous gradients but purely looks at the gradient at the current point, in doing so it often results in a zig-zag motion towards a minimum. This method defines the search direction p_k.

$$p_k = -\frac{g_k}{\|g_k\|} \tag{3.6}$$

Conjugate Gradient The conjugate gradient method is only minimally different from the steepest descent and saves information on the search direction from previous iterations. The history and workings of which can be found in the publication of Nazareth (2009) [62]. This method uses the same calculation for the direction as the steepest descent from equation 3.6 but adds to it a β term which is calculated with information from the previous iteration. Within conjugate gradient methods, there is a further variation on how to define the β term, below, you can find one example.

$$\beta = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} \tag{3.7}$$

$$p_k = -\frac{g_k}{\|g_k\|} + \beta_k p_{k-1} \tag{3.8}$$

Quasi Newton In 2001 Schoenberger [75] published a clear explanation of the Quasi-Newton method which finds its characteristic in iteratively building an estimation for the Hessian. The widely used algorithms were all developed around in the early seventies and still hold relevance today. Some of the most well-known include Broyden's (1965) method, the SR1 formula (Davidon, 1959; Broyden, 1967), the DFP method (Davidon, 1959; Fletcher and Powell, 1963) and the BFGS method (Broyden, 1969; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970).

Derivative Calculation The calculation of derivatives is another problem to be considered. For systems that consist of a high number of design variables, calculating the derivative analytically is not a feasible solution. There are numerical approaches to getting the gradient and, if required, to calculating the Hessian. Estimating the derivative with finite differences is the simplest one. The gradient is calculated by taking making a straight line between two points within the design space. It should be noted then that numerically if the step size is too small, the difference between the two points can become impossible to calculate due to computational limitations on the fidelity of the computer. A general version for applying this to meshes was presented by Perrone in 1975 [68]. Another example of derivative calculations is the complex step, presented methodically by Martins in 2003 [54]. This method takes the derivative similarly to the finite-difference method but instead of moving within the real space, a Taylor expansion is done to estimate small steps in the complex plane. It is then found that the derivative can be estimated by taking the imaginary part of the expansion and thus no subtraction is needed. The two remaining derivative calculation methods seen in optimizations are Analytical and algorithmic. Both see any computational model as a sequence of explicit functions. Depending on how every step of the sequence is taken, it is possible to get partial derivatives at every step and in doing so calculate derivatives within the system. This requires access to the source code of any model as between sequential steps the derivatives have to be calculated. It can be concluded then that this is a very involved process. Which finally leads to the most numerically exact method, analytic differentiation. Analytic differentiation can be split up into direct and adjoint methods. Both use the traditional chain rule and set it to zero. But differ in either factorizing the residuals (direct) or the partial derivative (adjoint)

3.1.2. Gradient Free Methods

Gradient-free methods are ways to navigate the design space without calculating the gradient of the design space. Gradient-free methods are efficient in finding local minima for high-dimensional, nonlinearly constrained, convex problems. However, they are sensitive to noisy and discontinuous functions. Oftentimes, it is seen that these methods are based on heuristic logic or copy some sort of natural phenomena. Genetic algorithms, for instance, mimic evolution by introducing similar stages to the exploration of a design space as stages of population growth. Considering a set of design solutions to be the population which can procreate and share characteristics. For further reading on genetic algorithms the reader is referred to Sivandam (2008) [78]. Particle swarm is another method based on natural phenomena, using principles from bees searching for honey. Developed in 1995 by Kennedy and Eberhart [23] the method is a widely popular approach and was reviewed just six years after by Shi (2001) [24]. The general idea is to give each agent within the swarm (one particle moving along the design space) a direction and a velocity. At each iteration, the best overall position is stored together with the best ever position ever visited. Depending on where the agent is, the velocity and direction will change. As such there are many versions of gradient-free methods, like the divided rectangles which navigate a space on the basis of dividing rectangles until the optimal solution is found. The Nelder-mead method was developed in 1965 [63] and is still a popular optimization algorithm which is based on producing a simplex in the design space that navigates based on specific rules. As gradient-free methods are based on heuristics there is a nearly endless amount of possible approaches. As such to avoid an endless list of approaches; the simulated annealing and genetic algorithm approach will be further explored.

Genetic Algorithm Inspired by Darwin's theory of evolution, where the creature best suited to its environment has the highest likelihood of survival. The genetic algorithm consists of making a population of designs where the design variables can be seen as chromosomes. The genetic algorithm evaluates each design with the objective function. The initial population can be generated from any random distribution to increase diversity. The goal of the initial population is to spread uniformly around the search area in the design space, as such Gaussian random distributions are commonly used. The next phase is the selection phase, where the lowest solutions (most fit) have the highest chance of finding a mate. This is done by assigning a probability to each design in the design space. In the literature there are many operators for the selection phase; Boltzmann selection [28], fitness uniform selection [35], Tournament Selection [84]. After the selection phase is the crossover, or procreation phase. Taking two 'parent' designs and generating a new one. Two popular crossover methods are single and double-point crossovers. The single-point crossover takes the design variables of one parent and swaps them with the design variable of the other parent from a specific point. An example would be if parent 1 had design variables $[A, B, C, D, E]$ and parent 2 had $[F, G, H, I, J]$ a single point crossover from point 2 would result in children: $[A, B, H, I, J]$ and $[F, G, C, D, E]$. A two-point crossover is similar but taken between two points. There are many more versions of crossover to be found in the literature: Uniform Crossover [34], partially mapped crossover [17] and more to be found in the crossover review from Kora [43] The last phase is a mutation, in which the design variables of the child designs are adjusted. The mutation rate is typically low to avoid a random search. At high mutation rates, the selection phase is nullified. As with the other phases, there are many techniques for the mutation phase: Power mutation [18], Gaussian [8], Supervised Mutation[89]. The best solution from the last generated population is returned as the global optimum solution to the objective function.

Simulated Annealing The simulated annealing (SA) algorithm was first independently proposed by Kirkpatrick et al in 1983 [41] and Černy in 1985 [13]. As explained by Sahab M in 2013 [72], the algorithm is based on the way crystalline structures form to the optimum energy state during the slow cooling process of metals. The SA algorithm applies a similar process to finding the optimum of an objective function. Using a statistical search of the design space the SA algorithm is increasingly less likely to accept bad solutions. For the cooling analogy, the objective function can be seen as the energy state, and moving to a new set of design variables corresponds to a change in that state. A plethora of engineering-related optimizations using simulated annealing is named in Sahab M 2013 [72] showing its wide spread usage in the engineering industry.

The SA-algorithm starts with a feasible starting design. From this point, a new design is developed by adjusting the design variables to a specified degree from the starting point. The designs have to be feasible, so pass some sort of feasibility check, after which the design is tested on the objective function. If the new design results in a lower value of the objective function, the design is accepted. When this is not the case a random number is generated, and compared to a temperature PDF function, if the random number is below the temperature pdf function, the 'worse' design is accepted. The laws of thermodynamics would state that at any given temperature the probability of change in the energy state of δE is:

$$p(\Delta E) = \exp\left(-\frac{\Delta E}{k_B T}\right) \tag{3.9}$$

Where k_b is the Boltzmann constant and δE is the difference between current energy state E_i and new energy state E_j:

$$\delta E = E_j - E_i \tag{3.10}$$

As more iterations go on the temperature pdf function returns smaller values which decreases the chance of accepting worse designs. There are variations to this approach. For instance, rather than ignoring infeasible designs, a penalty function can be introduced which allows more freedom of movement in the design space as introduced by J Stern in 1992 [79]. The initial temperature and cooling schedule are important parameters in simulated annealing. Kirkpatrick's first paper suggests that the initial temperature be determined in terms of the initial probability value.

$$T_0 = -\frac{\Delta E}{\log(1 - P_0)} \tag{3.11}$$

For the cooling schedule, there is a balance to be found as well. Cool too quickly and the chance of finding the global optimum decrease. Cool too slowly and you have computational inefficiency. In the first proposed algorithm [41] used a linear cooling schedule.

$$T(k) = -\eta k + T_0 \tag{3.12}$$

Where T_0 is the initial temperature, η is the slope of the decreasing line and k the iteration count. Golden and Skiscim [29] also used a linear approach to the cooling schedule but defined it as:

$$T(k) = T_0 \frac{c - k}{c} = T_0 - \frac{k}{c} T_0 \tag{3.13}$$

Where c should be chosen through trial and error. The ratio of k over c is a decreasing factor with increased iterations k. Sekihara had an approach where after the minimum amount of iterations the cooling schedule would speed up [77].

$$T(k) = \begin{cases} \frac{T_0}{(1+k)} & \text{if } k \leq k_{\text{lim}} \\ \alpha \cdot T(k-1) & \text{if } k > k_{\text{lim}} \end{cases} \tag{3.14}$$

Johnson [37] proposed another popular cooling schedule found in the literature:

$$T(k)\frac{T_0}{(i+\beta T_0)} \tag{3.15}$$

Where β is a coefficient for the initial temperature T_0

Or the widely used adaptive cooling schedule from Atiqullah [7] that uses the variance of temperature over the iterations:

$$T(k+1) = T(k) \exp\left(-\frac{\lambda T(k)}{\sqrt{\text{Var}[T(k)]}}\right) \tag{3.16}$$

Where $0 < \lambda =< 1$.

Although the temperature could theoretically decrease to zero, this does not have to be a necessity for finding the global optimum. In a method suggested by Van Laarhoven [80] the optimization stopped when the temperature dropped below a predetermined final temperature T_M. Lundy and Mees [51] suggested this final temperature be defined as:

$$T_M \leq \frac{\varepsilon}{\ln\left[\frac{|S|-1}{P}\right]} \tag{3.17}$$

There seems to be a near endless amount of variants on the SA algorithm, Chaotic SA, Fast SA, Hybrid SA. All of which try and improve the performance of the algorithm. Chaotic SA is based on chaos which is a mathematical property of dynamical systems which shows unstable pseudo-random, non-periodic behaviour. Chaotic sequences have been adopted rather than random sequences in heuristic algorithms [85]. Mingjun and Huanwen introduce chaotic systems to the SA algorithms [58]. The first chaotic map is made using a one-dimensional logistic map defined as:

$$z_{k+1} = f(\mu, z_k) = \mu z_k (1 - z_k) \quad \text{with } k = 0, 1, \ldots \tag{3.18}$$

Where z_k is between 0 and 1 and is the chaotic variable z at the $k$þiteration. μ is the bifurcation factor of the system. This system carries the stochastic property and sensitivity dependence on the initial conditions of chaos. The second chaotic system can be produced by a new chaotic map defined as:

$$z_{k+1} = \eta z_k - 2 \tanh(\gamma z_k) \exp\left(-3z_k^2\right) \tag{3.19}$$

This mapping model was derived from a chaotic neuron, developed in chaotic genetic algorithms [83]. The chaotic mapping is the initialising step for the first feasible design. Given an initial value z_0, generate set of chaotic variables z_{k_i} using equations 3.18 and 3.19

A new solution is generated using a formula dependent on one of the chaotic variables z_{k_m}:

$$y_{m,i} = x_{m,i} + \alpha \times (b_i - a_i) \times z_{k_m}, \tag{3.20}$$

Where α is a decreasing variables adjusted with $\alpha = \alpha \cdot e^{-\beta}$

Implying that the main difference between chaotic SA and normal SA is the replacement of how the first design is generated, and how the design space is navigated. From Mingjun and Huanwen's paper [58] it is unclear whether the results from chaotic SA are significantly better than the classic SA proposed by Kirkpatrick [41].

3.2. Methodology: Optimization

This section will explore how the optimization algorithm used for this thesis has been implemented and tested. The optimization method used is a gradient-free optimizer; simulated annealing. The theoretical background is discussed in section 3.1.

3.2.1. Simulated Annealing

The heuristic logic of the simulated annealing algorithm is based on the cooling of the crystalline structures in metals. The logic is motivated by the need to allow the optimizer to move to less optimal places and in doing so explore more of the design space. The optimization chooses an arbitrary new point within the design space, checks for feasibility and based on a temperature variable either will or will not accept a 'worse' design set.

This section will go into detail about to steps within the algorithm and discuss its performance in a set of test functions. The simulated annealing algorithm can be broken down into 4 steps.

- Initialize
- Generate new design
- Evaluate new point
- accept or generate again

These steps continue until the stopping criteria are met.

Initialize The simulated annealing algorithm begins with a point within the feasible design space, so the starting point cannot be an infeasible design. When the optimization is initialized it uses the initial feasible starting point within the design space x_{start} to evaluate the objective function $f(x_{start})$. An initial temperature T_0 and a cooling rate r are parameters with which to control how quickly the optimizer settles to a solution. Further is the parameter L that determines the minimum amount of trial movements before moving on to the next point. The optimization loop can be divided into two iteration loops, one inner iteration loop counted by k and an outer iteration loop counted by K.

Generate new design Beginning at some point x_{start} a new point is generated by adjusting the design.

$$x_{new} = x_{old} + \alpha \cdot x_{old} \tag{3.21}$$

Where α is a parameter that is uniformly distributed between plus and minus Δ_{max}. Where Δ_{max} is chosen to be 0.1. This defines the max search distance at one step. This does mean that if the initial feasible design is far away from the optimum and consists of small design variables, it will take longer to converge to an optimum solution. However for the case of a spar-type floater, the system is expected to be sensitive to small changes, and as such a feasible design is not expected to be very far from the optimum.

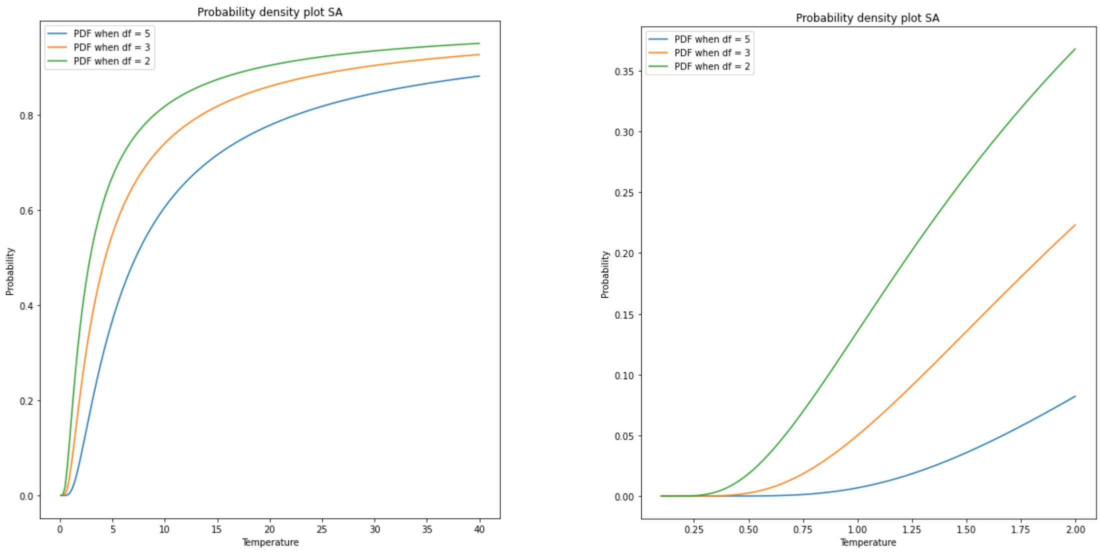
Accept or Reject Design The new point is then checked for feasibility, where a check is done to see if it crosses any of the constraint conditions. After the new point is deemed to be within the feasible space, the objective function is evaluated at the new point. If the new point results in a smaller value of the objective function the point is automatically accepted. If this is not the case the point could still be accepted. The acceptation is based on the probability density function and a random number z uniformly generated between 0 and 1. In literature, this is also referred to as the metropolis condition. The probability density function is defined as:

$$p = \exp(\frac{-df}{Tk}) \tag{3.22}$$

Where df is the difference in objective function values $f(x_{new})$ and $f(x_{old})$, where if df is positive, means that the new point is worse off than the old point x_{old}. If the randomly generated number is less than p the point is accepted.

The probability density function is plotted below to show the behaviour and working of this part of the algorithm. From figure 3.1 it becomes clear that as the difference becomes less positive the likelihood of acceptance increases as function value p converges to 1 from a lower temperature. And in figure

3.1b it is noted that when the difference df is large the probability p remains negligible until a much higher temperature.



(a) plotted from temperature range 0 to 50.

(b) plotted from temperature range 0 to 2

Figure 3.1: Probability density plot for less optimum designs

If the new point is worse off and the randomly generated number z is higher than the p a new feasible point is generated again. This continues until a new acceptable point is generated. Completion of this acceptance is considered the inner iteration. Which will go on until $k \geq L$, from which point either the global optimum is found by checking stopping criteria. Or where the best point is taken and from this best point the same inner iteration loop goes again. In which the outer loop counter K goes up by one as well.

Stopping Criteria There are three stopping criteria:

- Lack of consecutive change
- Proportional Improvement
- Outer Iteration Limit

Lack of consecutive change The lack of consecutive change checks that the best position is found after a set number of outer iterations is still improving. What happens is that for J number of inner iterations the evaluation of those iterations is compared. If the evaluations are too close together it means that the optimizer is barely moving through the design space which indicates a global optimum. By checking all the J set of accepted new points within an inner iteration if they are all near each other it could also indicate that the temperature is so low that accepting worse points becomes impossible. Which also means that the optimizer can stop running. This check is done by comparing the difference in objective function between iterations and dividing it over the initial evaluation. If in the J set of new points the improvement is all smaller than parameter γ, the stopping condition is met.

Proportional Improvement The proportional improvement check looks at the number of points within an inner iteration that is actually better than the starting point. In a situation where the iteration limit L is set to 50, if a very small set of those 50 is actually resulting in a better solution for the objective function, chances are likely that an optimum has been found. This stopping condition is governed by a percentage of L. After acceptance of this, a new inner iteration counter goes up. This will continue until the inn K is reached.

Outer iteration Limit The outer iteration limit is the limit of K. This stopping condition prevents the optimizer from carrying on in an infinite loop. After the outer iterations have met K_lim the optimizer needs to stop. This forms a quadratic relationship with the max amount of iterations that the optimizer can achieve.

$$i_{max} = L \cdot K_{lim} \tag{3.23}$$

A schematic is presented below to give an overview of the simulated annealing optimization algorithm.

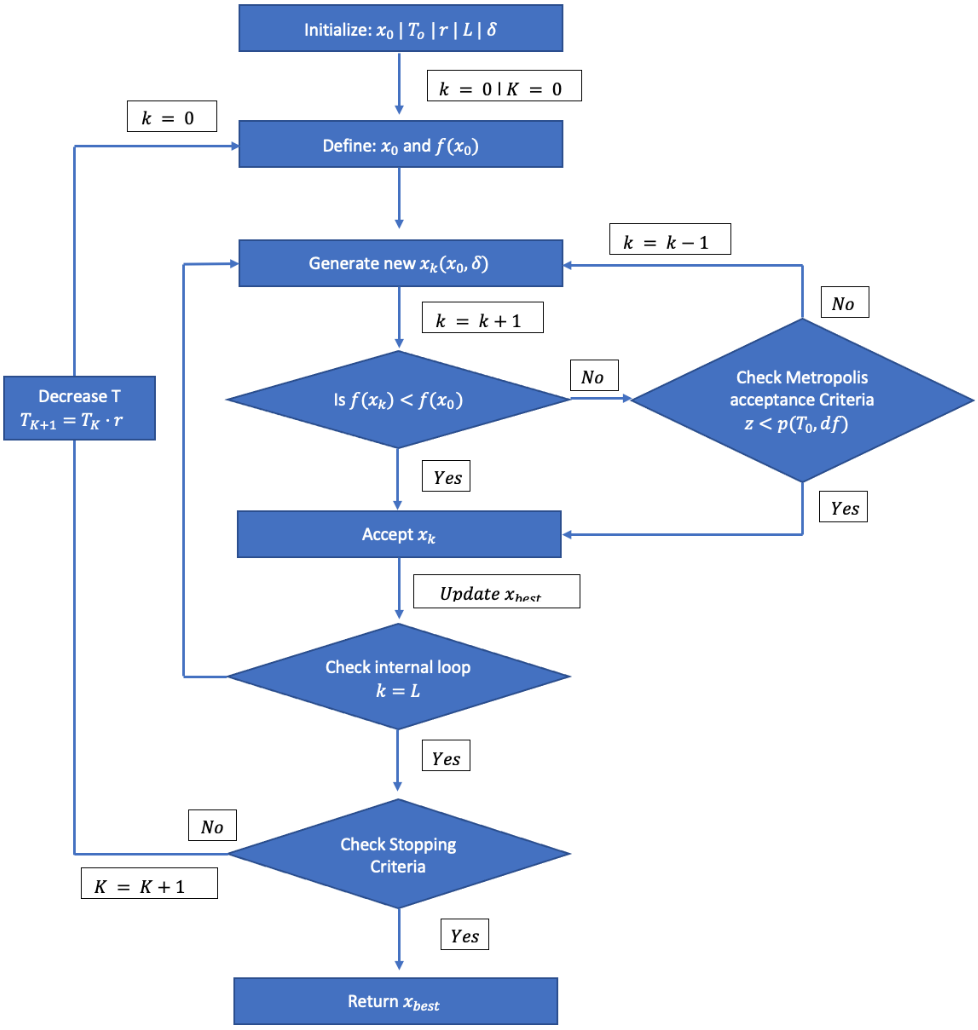


Figure 3.2: Simulated annealing schematic

Going through two cycles

For added clarity on how the optimizer works a loop will be described following the change of a design variable throughout two cycles of the simulated annealing algorithm. The optimization takes some objective function, constraint function and control parameters. Relevant to this explanation are listed below:

- Starting temperature: $T_0 = 100$
- Cooling factor: $r = 0.97$
- Adjustment parameter: $\delta = 0.1$
- Inner loop limit: $L = 2$

Starting from $i = 0$ to $i = 2$. In the first iteration, a better point is immediately found, in the second cycle the metropolis acceptance criteria will be used. The goal of the optimizer is to minimize some objective function $f(x)$. The optimizer begins with some feasible starting variable x_0, that is to say, it is within the feasible design space and therefore does not cross the constraints. The optimizer is initialized with a starting temperature T that will decrease over time with parameter r.

The first step is to evaluate the objective function at the initial feasible design point.

$$f_0 = f(x_0) \tag{3.24}$$

Next a new point is generated by adjusting every design variable in x with some random number between positive and negative $\delta \cdot x_0$ of the starting $x0$. This is always taken from the starting design to avoid favouring smaller design. If this adjustment was taken from x_i instead of x_0, the potential area that the optimizer would consider would increase and decrease as the next x_i gets bigger and smaller respectively. The design is then checked for feasibility by seeing if any of the constraints are activated with the new design, in which case a new point is generated and checked again. This carries on untill a point is found within the feasible design space.

$$x_1 = x_0 + rand(-\delta \cdot x_0, \delta \cdot x_0) \tag{3.25}$$

After a point is found within the feasible design space. The objective function is evaluated with the new design point.

$$f_1 = f(x_1) \tag{3.26}$$

Then to check if the design can be accepted or not the difference in the result of the objective function is determined.

$$df = f_1 - f0 \tag{3.27}$$

For the first cycle, df is a negative value, implying that $f(i + 1)$ is smaller than $f(i)$ and as such the new design point is immediately accepted. This marks the completion of one inner loop. As such the design resulting in the lowest value of the objective function is saved as x_{best} and a new inner loop is started.

With the new acceptable design, it needs to be checked whether enough inner loops have been completed, that is to say, have enough new points been accepted to check for convergence?
If not, a new point is generated again. Where the same steps are performed. Evaluating the difference of this new point from the starting design point x_0.

$$x_2 = x_0 + rand(-\delta \cdot x_0, \delta \cdot x_0) \tag{3.28}$$

$$f_2 = f(x_2) \tag{3.29}$$

$$df = f_2 - f_0 \tag{3.30}$$

Presuming that in this case, df is a positive number, meaning that f_2 is higher than f_0. This is to say that this design, although feasible is actually worse. In such a scenario the metropolis condition is called where a random uniform number z is generated that lies between 0 and 1. At the same time, the pdf function (eq 3.22) is evaluated, and if the randomly generated number z is lower than the evaluation of the pdf function, the design point is still accepted. If not a new feasible point is generated again until either the difference df is a negative value, or the worse design is accepted. Let's presume that df is a positive value of 2 and that the starting temperature $T_0 = 100$. As such the pdf can be calculated as:

$$p = e^{-\frac{df}{T_0}} = e^{-\frac{2}{100}} = 0.98 \tag{3.31}$$

As the evaluation of the pdf function results in a number close to 1 it is likely that the randomly generated number z will be lower and thus that the point will be accepted regardless. In this case, a random number of 0.6 is generated and as such the new design is accepted.
As there has again been an accepted design the check is done to see if enough inner iterations have been done. As parameter $L = 2$ that is the case. This means that a check for stopping criteria can be done. As it turns out the newly generated point does not suffice any of the stopping criteria. As such the temperature is adjusted. And the best of the explored design points is now used as x_0

$$x_0 = min(x_k) \tag{3.32}$$

$$T_{i+1} = r \cdot T_0 \tag{3.33}$$

Where x_k are all the accepted new points, in this case, x_1 and x_2. This marks the completion of the outer loop. In the image below figure 3.3the route can be seen from the schematic presented before in 3.2.

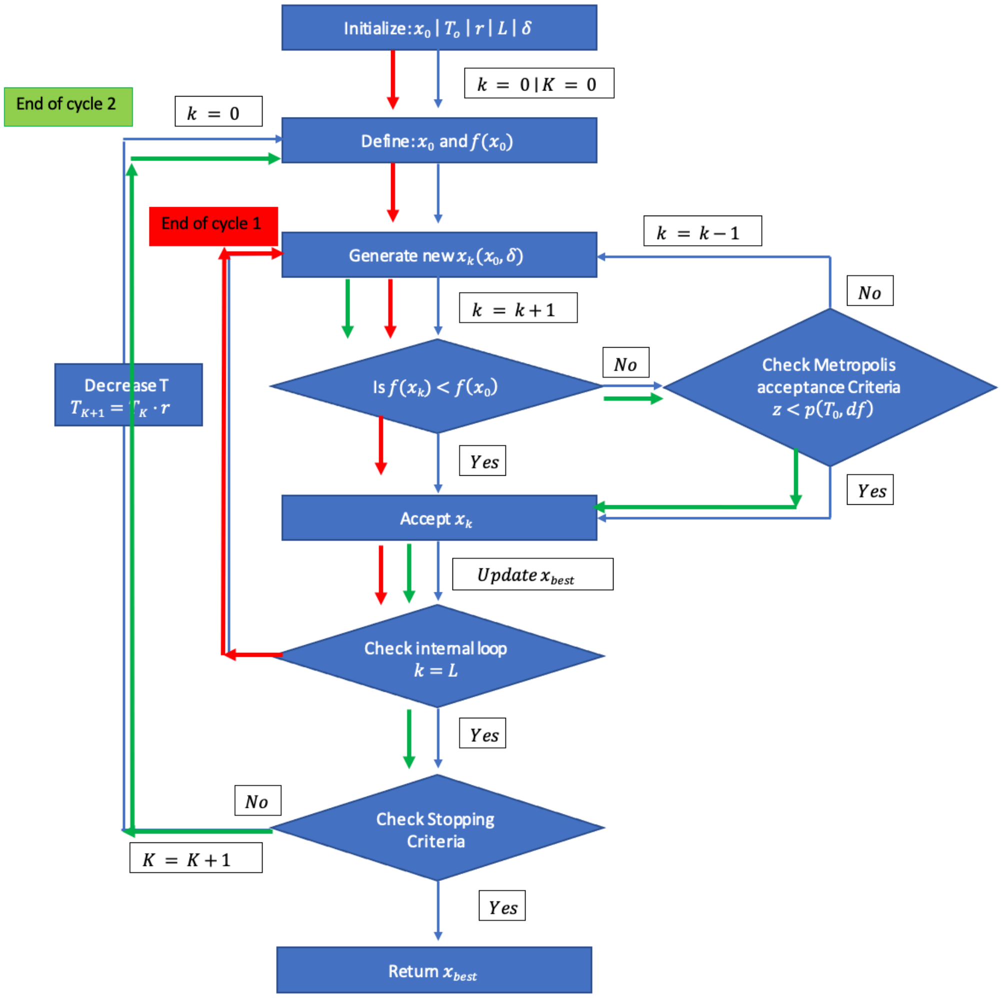


Figure 3.3: Cycles explained in the simulated annealing schematic

3.2.2. Test Functions

To test the functionality of the optimizer a set of test functions are used. These functions come with specific challenges and will help to illustrate how to optimizer is functioning.

Ackley

The first function to be tested is the Ackley function. The Ackley function is characterized by a nearly flat outer edge and a sudden large drop in the centre surrounded by smaller divots. And is defined as:

$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1) \quad (3.34)$$

With the global minimum:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0) \quad (3.35)$$

Where a, b and c are parameters that can be adjusted to control the shape of the function and d is the amount of dimensions considered. The function is illustrated in figure 3.4 for a case where $d = 2$

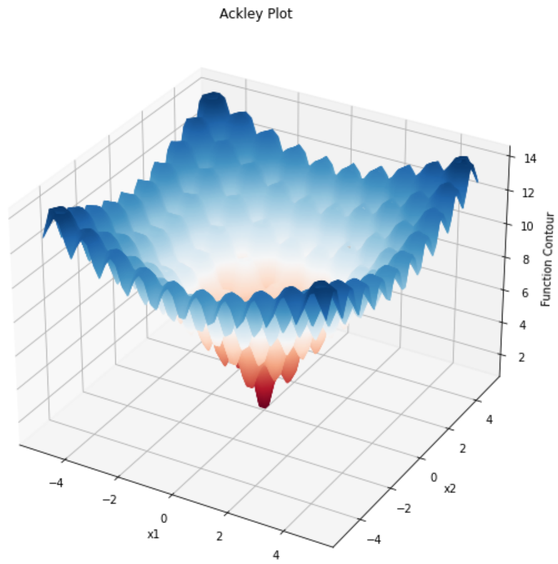
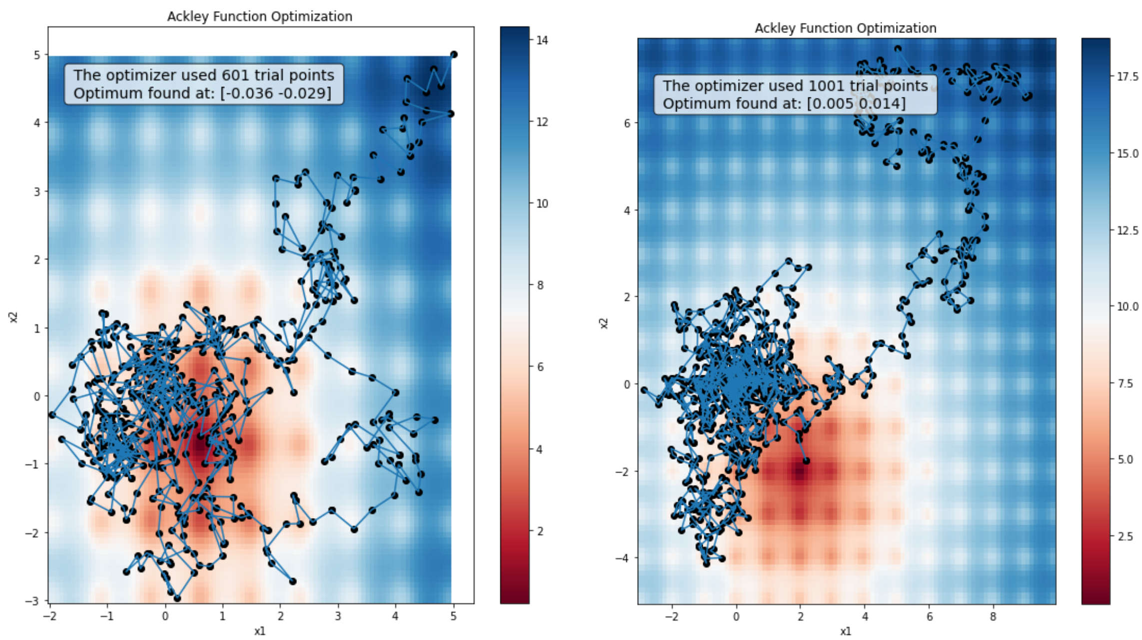


Figure 3.4: 3D plot of the Ackley function in two dimensions with parameters set as $a = 20$, $b = 0.2$ and $c = 2\pi$

Two single tests are run with this function and the SA algorithm at starting point $x_0 = [5, 5]$, but two different inner loop limitations. As the exact optimum is situated at $x = [0, 0]$ the figure shows that with more iterations a better point is found. However, depending on the function it might not be worth doubling the computational cost for a decimal point improvement.



(a) SA optimization using $L = 200$

(b) SA optimization using $L = 1000$

Figure 3.5: SA optimiation of the Ackley Function

To further investigate the iteration limit necessary another test can be run with this optimizer. The goal is to see how sensitive convergence is to its starting point. To test this the Ackley function is tested for 1000 randomly generated starting points between -10 and 10. Using an optimization limit of $L = 200$ results in 200 of the 1000 starting points converging to a solution smaller than 0.1 and 720 converging to a solution smaller than 1. Whereas if $L = 1000$ the number of converges solution goes up to 400 for solutions smaller than 0.1 and 920 for solutions smaller than 1.

There is no pattern to be recognized in the starting points that have not converged. Below the figure shows the starting point for an iteration limit of 200 that did not converge to a solution smaller than 0.1.

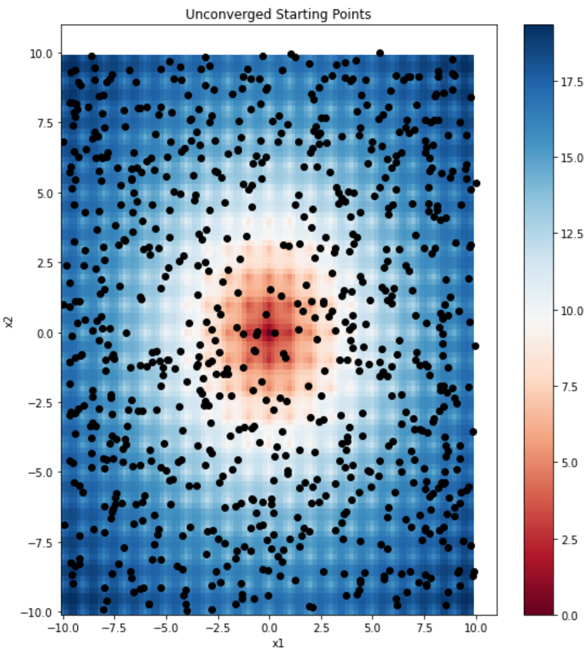


Figure 3.6: Contour plot of starting points that did not lead to a convergence of a solution smaller than 0.1

In a design space where there are a lot of local optima the iteration limit, L has to be set much higher to avoid getting stuck in local optimum solutions. This problem is enhanced when the optimum lies around the zero point. Because the Simulated annealing algorithm will only allow movement based on a percentage of the design point.

Booth

The booth function is defined as:

$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \tag{3.36}$$

with a global optimum at

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (1, 3) \tag{3.37}$$

This function is characterized by a long flat middle, on the $x = -y$ axis, as such it can be a challenge for optimizers to still get to the global optimum. A common starting point for testing the booth function is $x_0 = [10, -10]$.

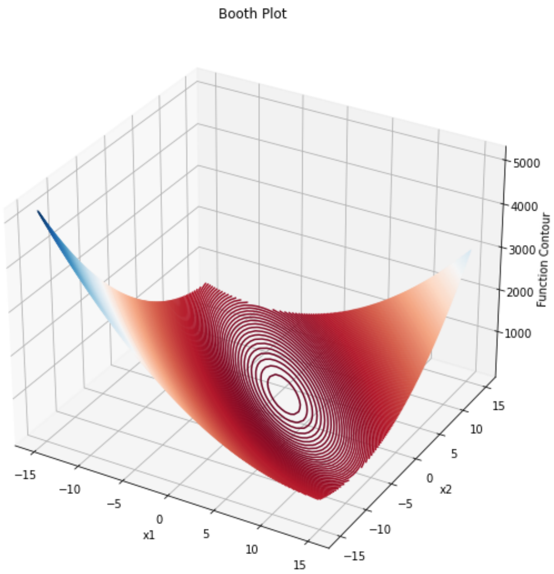
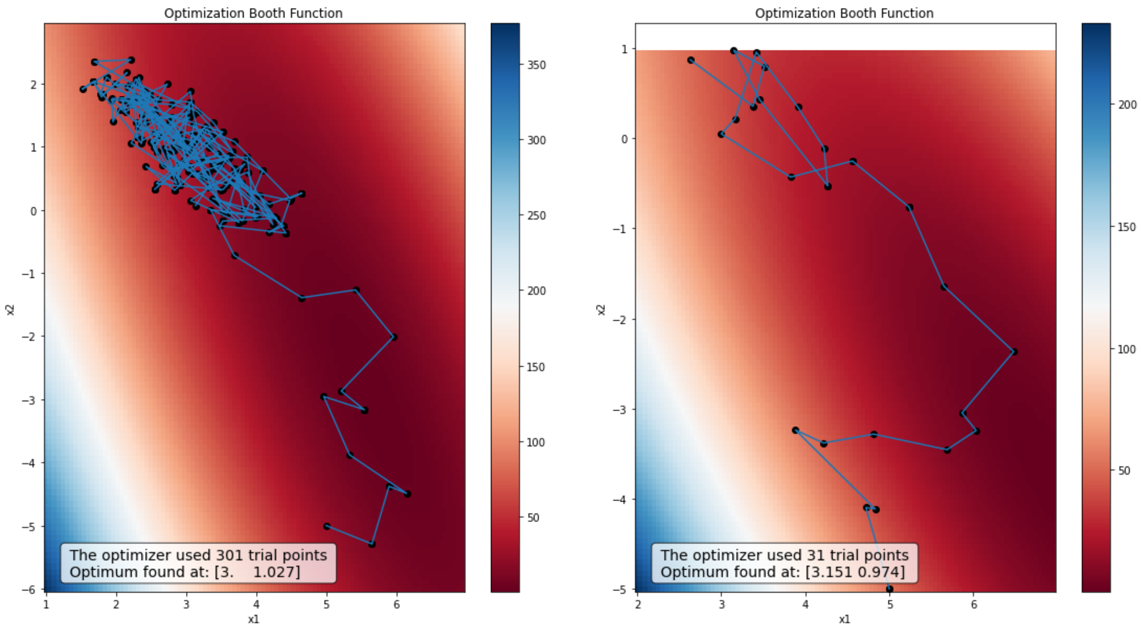


Figure 3.7: 3D plot of the booth function

When this function is optimized with the simulated annealing algorithm it becomes clear that the iteration limit can be set too far smaller values than the Ackley function and still achieve good results.



(a) SA Booth function optimization using $L = 300$

(b) SA booth function optimization using $L = 15$

Figure 3.8: SA optimization of the Ackley Function

What can be seen through these figures is that in a design space that does not have many local minima but rather a slow-changing minimum that the iteration limit can be set to a far smaller value and still get reasonable results.

3.2.3. Optimization Formalism

The goal of the optimizer is to minimize some objective function f with respect to design variables $x \in \mathbb{R}^n$. while subject to equality and in-equality constraints $\widehat{c}_j$ and c_k. The optimizer developed in this thesis optimizes for the mass of the substructure, mass can be correlated to cost.

$$f = m_{spar} = \rho_{steel} \cdot t \cdot 2\pi \cdot \frac{D}{2} \cdot L + 2 \cdot \rho_{steel} \cdot t \cdot (\frac{D}{2})^2 \tag{3.38}$$

Where ρ_{steel} is the density of steel, t is the thickness of the steel plating of the spar, assumed to be equal along the entire length L. D is the diameter of the spar buoy. The objective function contains all three design variables. As such x is defined as

$$x = \begin{bmatrix} D \\ L \\ t \end{bmatrix} \tag{3.39}$$

Although the design space has 4 dimensions, function 3.38 can be divided over the thickness to create a design space that can be illustrated in a three-dimensional plot. From figure 3.9 it is seen that the design space at any given thickness is an angled bowed plane. There are no obvious local minima. It can be theorised that if this objective function was left unconstrained that the global optimum would lie at $-\infty$. Which would lead to either a constraint that the design variables must be larger than 0 or a change in the objective function to take the absolute result of equation 3.38. What is clear however is that the optimum will be governed by the constraints put on the objective function.

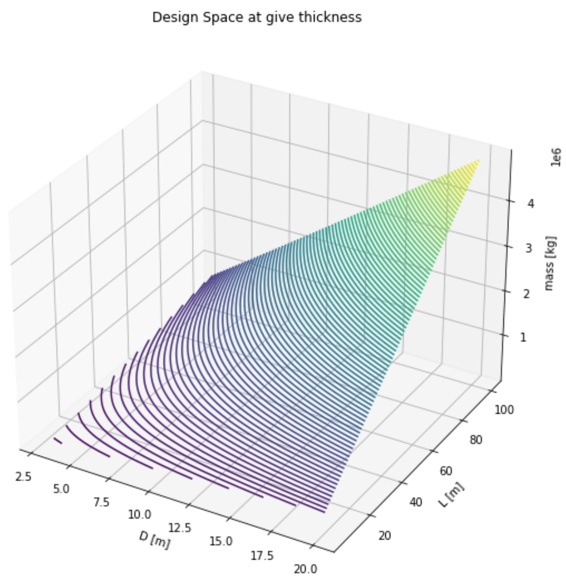


Figure 3.9: 3D plot of the spar mass objective function with a steel thickness set to $t = 0.1$

3.2.4. Constraints

The constraint function is expected to govern the optimum result within the design space. As such this research will investigate and compare what the optimum solution looks like when using a varied set of constraints. Specifically, the difference in global optimum design when the optimizer constraints include response and fatigue calculations. The constraint function used in the optimizer will return an infeasible design as soon as one of the constraints is met. As such it makes sense to present the constraints in order of computational complexity.

Logical Design check

The first constraint check on the design consists of some logical ratios of the design variables. All the design variables should be above 0, there is no such thing as a negative length, this also ensures that only positive values of the objective function will be considered. Furthermore, the draught of the floater should be greater than the diameter, it would not be considered a spar otherwise. The thickness has to be less than half the diameter, if not the diameter would be determined by $2 \cdot t$ rather than D. More realistically the thickness should be within an acceptable range of between 0.01 and 0.1.

$$[\mathbf{x}] > 0 \tag{3.40}$$
$$D < L \tag{3.41}$$
$$t < 0.5D \tag{3.42}$$
$$\frac{D}{L} > 0.01 \tag{3.43}$$
$$0.01 < t < 0.2 \tag{3.44}$$

Geometry vs Environment

With the sizing of the structure accepted the spar can now be placed in the environment. This includes information about the average wave height and water depth. The hydrodynamic forcing calculations are based on Morison equations. These equations are applicable to slender cylinders where the wavelength is longer than approximately five times the diameter of the cylinder. As such a constraint is made based on the average wavelength and the diameter of the cylinder. Where the significant wave height of the considered area is taken for the calculation of λ

$$\lambda > 5 \cdot D \tag{3.45}$$

The floating structure should also float freely. As such, a limit is introduced to the length of the spar versus the depth of the water.

$$L < 0.8 \cdot h \tag{3.46}$$

Where h is the depth of the water.

Full System Constraints

When considering the full system. The diameter of the substructure cannot be smaller than the diamter of the tower base.

$$D_{spar} \geq D_{towerbase} \tag{3.47}$$

With an acceptable spar design, checks can be done on the combination of floater and superstructure. These constraints are based on DNV-OS-J103 [65] and DNVGL-ST-0119 [21]. According to J103 (sec:4 2.1.2), the natural period of floaters in surge sway and yaw is typically above 100 seconds to allow for some room; the constraint is formed as follows:

$$T_{surgenatural} > 90s \tag{3.48}$$

Depending on the location and sea state being considered, there is substantial energy in the spectral period range of 5 to 25 seconds. That is why a natural period for a spar-type floater is advised to be above 25 seconds. (J103 sec 4: 2.1.3)

$$T_{pitchnatural} > 25s \tag{3.49}$$

The intact stability constraint defined by DNVGL-ST-0119 Section 10.2.4.2 states that the GM of the spar structure must be greater than 1 metre. The GM is represented s the difference between the vertical level of the metacentre and the critical level of the centre of gravity and shall be calculated based on the maximum vertical centre of gravity VCG. For deep draught floaters it is required that GM be larger than 1m (ST-0119 sec 10: 2.4.2)

$$GM > 1 \tag{3.50}$$

Furthermore the design has to have a positive value for ballast

$$m_{ballast} > 0 \tag{3.51}$$

The next constraint is for the Mathieu instability; this occurs when the natural period in heave is some multiple of the natural period in pitch. It occurs at large heave motions causing harmonic pitching. Although the standard only includes the case where the ratio of the natural period in heave over the pitch is half. Haslum 1999 ([30]) shows instability regions at a ratio of 0.5, 1, 1.5 or 2. As such, the constraints are defined as follows.

$$0.45 > \frac{T_{surgenatural}}{T_{pitchnatural}} > 0.55 \tag{3.52}$$

$$0.95 > \frac{T_{surgenatural}}{T_{pitchnatural}} > 1.05 \tag{3.53}$$

$$1.45 > \frac{T_{surgenatural}}{T_{pitchnatural}} > 1.55 \tag{3.54}$$

$$1.95 > \frac{T_{surgenatural}}{T_{pitchnatural}} > 2.05 \tag{3.55}$$

Furthermore, natural period constraints are imposed for the 1P and 3P frequency ranges. At these frequencies, an excitation from tower shadowing could cause resonance if any of the natural periods are close.

$$T_{1P_{low}} > T_{natural} > T_{1P_{high}} \tag{3.56}$$

$$T_{3P_{low}} > T_{natural} > T_{3P_{high}} \tag{3.57}$$

Response Constraints

For optimizations of floating wind turbine systems that include aerodynamic loading, it is common to see a constraint on the pitch response under specified wind loading, as wind loading is the most significant contributor to pitching behaviour. For this simplified model, a pitch constraint of $theta < 8$ °is imposed. This is tested by running a time domain simulation at rated wind speed where the largest pitch response is to be expected. This expectation comes from the fact that at rated wind speed, the thrust force is largest as the blades do not pitch out of the wind.

$$\theta < 8^\circ \tag{3.58}$$

The surge motion is constrained to a maximum of 30 metres to safeguard the electrical wiring. This is tested by running a time domain simulation with a sizeable significant wave period. Large wave periods result in large surge responses.

$$x < 30 \tag{3.59}$$

The response is tested using a time domain a 50-minute time domain simulation (excluding the transient time of 10 minutes)

3.2.5. Fatigue Constraints

The lifetime fatigue can be calculated for every design; however, every design is checked for feasibility in the simulated annealing algorithm. As such, any design that passes through the time domain constraint check will also be limited for its fatigue lifetime. This is computationally expensive, as the randomly designed new feasible points can endlessly lie outside the feasible region. Nonetheless, it is essential to formulate what the fatigue constraints would look like. Which in this case would be a fatigue damage of less than 80% of the structure's life cycles at 50 years.

$$\sigma(N_{lifetime}) < 0.8 \cdot \sigma(N_{C203:lifetime}) \tag{3.60}$$

Where $f_S(N_{lifetime})$ is the function for the SN-curve of offshore steel given by DNV standard C203 [20] where the sn curve for the base material of high-strength steel is used.

$$\log N = 17.446 - 4.70 \log \Delta\sigma \tag{3.61}$$

4

Results

This Chapter will discuss the results from the optimization run according to the methodology from chapter 2.2. To research the effect and usefulness of response constraints on the optimization function of the design of a spar-type floating wind turbine. A set of experiments is run on two test cases after which the optimum designs are compared. To offer insight into how the constraints are affecting the optimum designs the most active constraints will be discussed for each case. In the first test case, logic constraints are used to run the optimization algorithm. This optimization is then run 100 times to investigate what the optimum design area looks like. The next experiment includes the response of the system as a constraint in the optimizer. Finally, the fatigue damage on designs from each experiment is compared.

4.1. Constraining by Geometrics

In this section results from running the simulated annealing algorithm using constraints from 3.44 to 3.57 are presented. This excludes any fatigue and response calculations. It also highlights the role that the mooring stiffness has on what the optimum design looks like. As in the first section of this test case the mooring stiffness is considered to be a set value.

4.1.1. Set Mooring Stiffness

For a single optimization the figure below shows the trial points used to find the optimum, in other words it shows the explored design space by plotting all the potential acceptable designs.

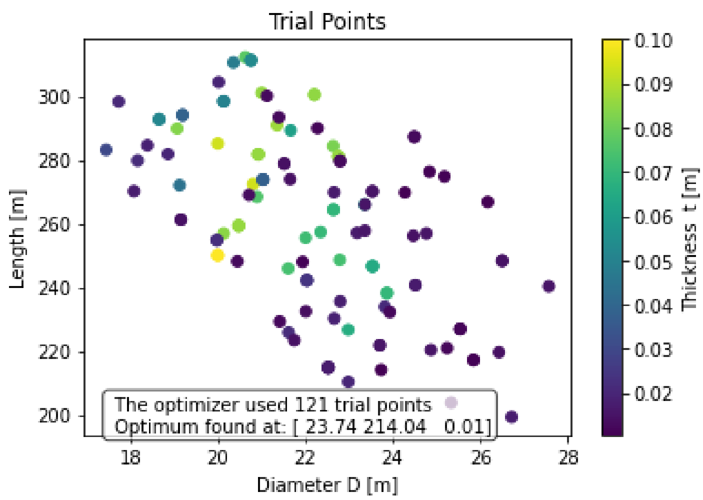


Figure 4.1: Distribtution of trial points for a single iteration with $L = 5E^5$,$L = 30$,$\gamma = 0.01$, $K_{lim} = 50$, $r = 0.97$)

As the mass volume is linearly affected by the length and quadratically by diameter. The expectation is that the optimizer will favour a very thin design long and slender. The exploration of the design space is random, with a limiting amount of iterations there is a chance that this also limits the exploration of the design space. As such it is good to investigate what happens when the optimization is run multiple times from the same starting point $x = [20, 250, 0.1]^T$, the parameters are shown in the first table presented below under the experiment. What was interesting to see is that with the current set up there were not many temperature changes. The optimizer had never reached the K_{lim}, rather it would find a solution that met one of the stopping criteria within two Outer loop iterations. As the starting point was always the same but the search is random it is surprising to see that the standard deviation in improvement is only 0.75% This means that the solutions found all have similar weights to each other. The average improvement is 89% which suggests that the starting point is far too heavy for these constraints.

Experiment	Iteration	L	K_lim	γ	Unsolved	Obj.reduc [%]	σ[%]	t_{avg} [s]	T
1	100	30	10	0.01	1	89.9	0.75	49.23	6
2	300	40	10	0.005	1	90.23	0.696	62.14	6
3	200	30	10	0.005	2	90	0.81	56.10	6
4	50	10	10	0.0025	9	87	8.31	4.23	6

Table 4.1: Results from experimenting with different parameters and a set starting point that is on the heavy side, (expected to be far from global optimum). Thie optimization does not take response or fatigue into account

In the third experiment, a counter was introduced to see what stopping criteria are being met the most. It turns out that in 200 iterations the consecutive change was called upon 118 times and the proportional improvement 82 times. The temperature never went down more than 7 times. This adds up with what was discussed in the methodology and test functions 3.2.2. As the design space can be considered to be similar to the booth function, where the design space does not show many local optima, the amount of inner iterations required to find an optimum drastically decreases.

The fourth experiment included lowering the inner iteration limit $L = 10$ and halving the γ parameter that controls the stopping criteria for consecutive change. This drastically decreased the average time it took to find a better point but came at the cost of 20% of the solutions not being any better than the starting point. This happens because of the proportional improvement-stopping conditions. With such a small inner iteration limit the chance becomes likely at high temperatures that all accepted new designs are worse designs, allowing the optimizer to stop. As discussed in the methodology the proportional improvement-stopping criteria only look at improvement within an iteration.

To solve this problem it is proposed to adjust the proportional improvement stopping criteria to include some measurement of the PDF value at that stage. This is done by considering the temperature of the system before checking the stopping criteria. By introducing a new parameter β that determines when the proportional improvement stopping condition is called upon.

$$Stop = \begin{cases} K = \beta K_{lim} \wedge \frac{I}{L} \leq 0.01 & True \\ r^K < \beta \wedge \frac{I}{L} \leq 0.01 & True \\ else & False \end{cases} \tag{4.1}$$

$K = \frac{1}{2}K_{lim}$ or if where r is the cooling parameter in the cooling schedule $T_{i+1} = T_i \cdot r$.

With this adjusted stopping criteria, and β set to half gives the following results when experiment 4 is rerun. Where now no point do not offer convergence. However the average time has gone up significantly, $t_{avg} = 88.5$. Upon further inspection, it is found that the reason why it is taking longer is that the optimizer is not able to find new feasible points. When generating a new space in the design space the metropolis condition is never met. As such the random adjustment of the initial design point carries on until a better design is found. This is caused by the metropolis condition's pdf function always returning $p = 0$, thus forcing a near-endless loop of generating a new point where the difference in the objective function is negative. This behaviour was due to a very low starting temperature of 6. For the optimization of the mass function of a spar that has a starting $x = [20, 250, 0.1]^T$, the first differences found by the optimizer are of the order of magnitude 10^4. Using the metropolis condition 3.22 results in a constant $p = 0$.

When a single iteration is run this results in far more trial points as shown below. It's obvious that there are far more trial points used, and with that a lot more of the feasible design, space is covered.

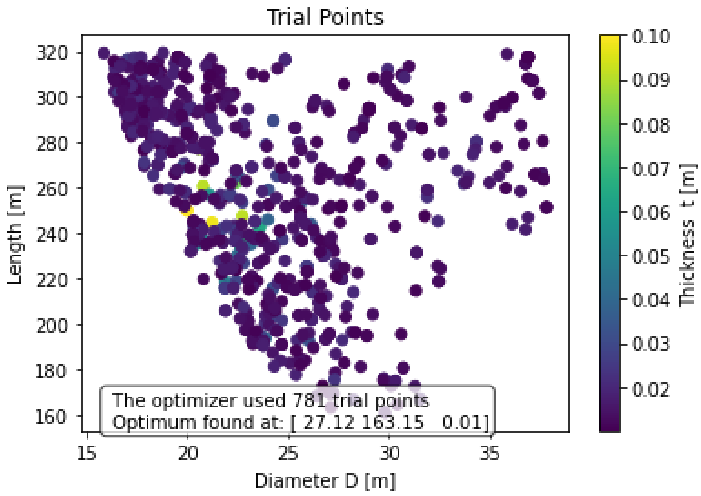
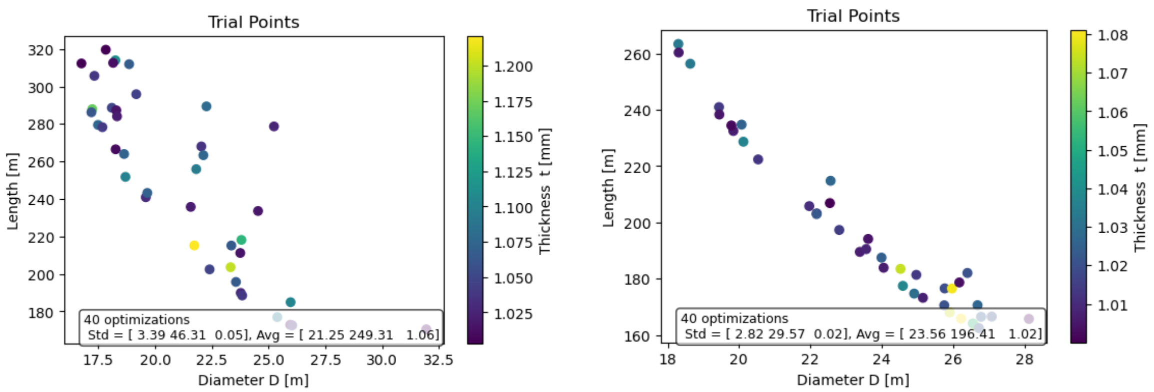


Figure 4.2: Distribution of trial points when using the adjusting proportional improvement condition for a single iteration. Using parameters: $L = 5E^5$,$L = 30$,$\gamma = 0.01$, $K_{lim} = 50$, $r = 0.97$)

After this is adjusted the optimizer shows that it cools down a lot more, caused by the new acceptance of 'worse' designs. As such more inner loop iterations are completed with a proportion of worse designs. When the inner loop limits L is left to a low number this still causes the proportional improvement criteria to be called on quickly. Beneath is a comparison of two identical simulations that only differ in having the 'proportional improvement' criteria adjusted or not.



(a) Global optima found without using the adjust stop criteria, standard deviation and the average of the design variables are presented at the bottom. The iteration had 120 temperature changes, never reached K_{lim} and always used the 'proportional improvement' stopping criterion.

(b) Global optima found using the adjust stop criteria, standard deviation and the average of the design variables are presented at the bottom. The iteration had 946 temperature changes, never reached K_{lim} and always used the 'proportional improvement' stopping criterion.

What can clearly be seen from the subfigures 4.3a and 4.3b is that the adjusted stopping criteria lead to a smaller standard deviation in all three design variables. However, the amount of iterations is tenfold. This could perhaps still be adjusted by increasing parameter β. In both cases, the only stopping criterion used was proportional improvement which means that there is never a lack of consecutive change. This criterion is governed by parameter γ and can be relaxed for better involvement. In this experiment, it was set to $\gamma = 0.0025$, which means in an inner iteration with L trials, if none of the trials has improved by 0.25% a global optima is presumed.
Another interesting point from the optimum solutions is found in figure 4.3b are the extremes on the left and on the right. Where an optimum spar has either a large depth $L \approx 260$ and slender diameter $D \approx 18$. Or vice versa $L \approx 160$ and $D \approx 28$. Similar extreme cases were found in figure4.3a, however due to the larger exploration of the design space figure 4.3b is considered more reliable. figure 4.3b also shows a clearer Pareto front.

4.1.2. Mooring stiffness based on surge natural period

The results from the test case run with a set mooring stiffness resulted in optimum designs that were on the larger side of what was expected. It was found that the constraint most governing the search for feasible design were the natural period in surge. This lead to the decision to redefine the mooring stiffness as a variable dependent on mass and added mass. Figure 4.4 shows the coverage of the design space when one optimization is run. Using the starting point of $x = [12, 200, 0.2]^T$ (meters). What can be seen is that the optimizer no longer considers designs of sizes comparable to what was seen in figure 4.2. Instead, a fall smaller range of spar lengths and diameters is seen to populate the feasible design space.

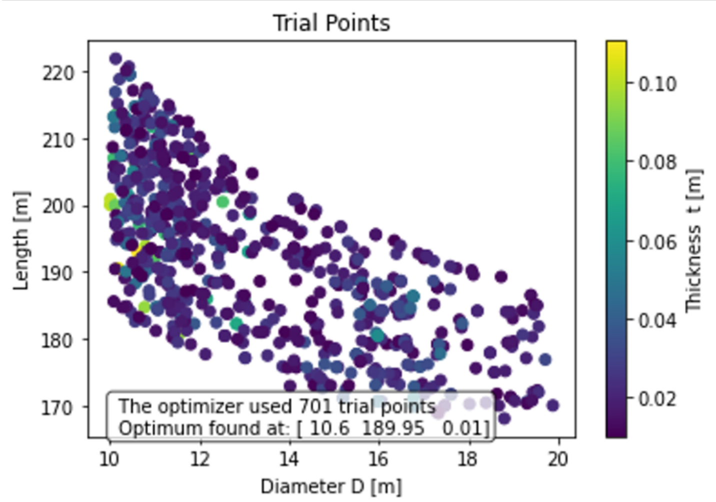
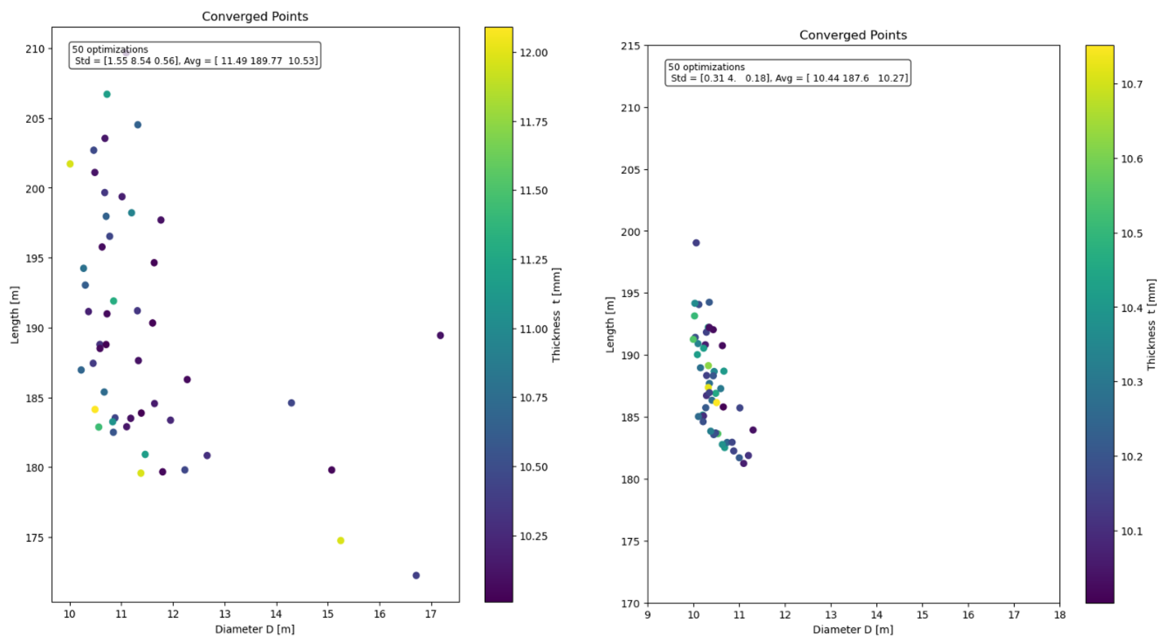


Figure 4.4: Coverage of the design space when one optimization is run using a mass dependent stiffness

Just as with experiments using a set mooring stiffness, the optimization using the adjusted stiffness is also computationally inexpensive. As such it can be run over an iteration loop. Presented below is the comparison of two versions of the optimization loop, using the standard stopping criteria on the image on the left, and the adjusted criteria proposed in equation 4.1 on the right.



(a) Global optima found using 50 separate optimizations. Did not utilize the adjust stop criteria. The stiffness of the system is dependent on the mass of the structure. The standard deviation and the average of the design variables are presented at the top of the figure. The iteration never reached K_{lim} and always used the 'consecutive change' stopping criterion.

(b) Global optima found using 50 separate optimizations. Utilizes the adjusted stop criteria. The stiffness of the system is dependent on the mass of the structure. The standard deviation and the average of the design variables are presented at the top of the figure. The iteration reached K_{lim} and always used 'consecutive change' and 'proportional improvement stopping criterion

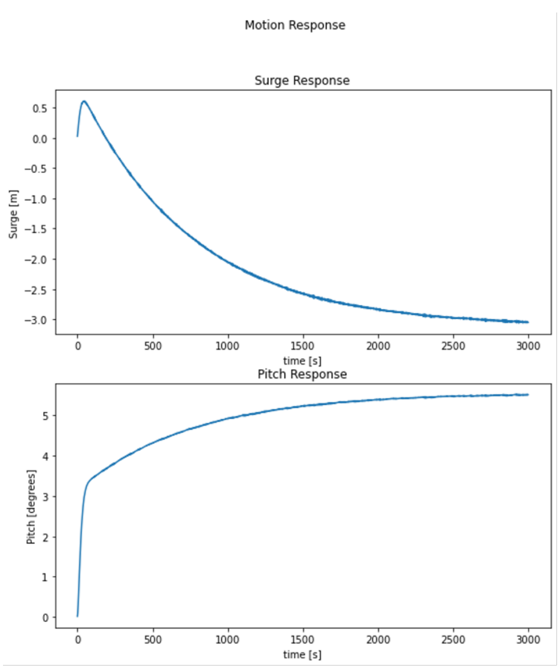
The two figures are plotted on similar-sized scales. It can be seen in the figure to the right (4.5b) that the global optima converge to a much smaller area. Forming a clearer Pareto front, the diameter on the front runs within a range of 10 to 11.38 meters. The spar length has a range of 181.23 to 199.04 meters and a thickness range of 100 to 108 millimetres. Which for all three design parameters is far smaller than the range presented in the figure on the left. As the figure on the right presents a mapping of global optima using more trial points and presents a smaller standard deviation of all the optima, the two designs with the most extreme diameters are compared for motion response and fatigue analysis. Acceptable motion response is defined as what will be used as a constraint in the optimization using motion response, where the pitch can not ever exceed 8° and surge must remain under 10 meters.

The optimal designs that differ the most in diameter are presented in the equation set below:

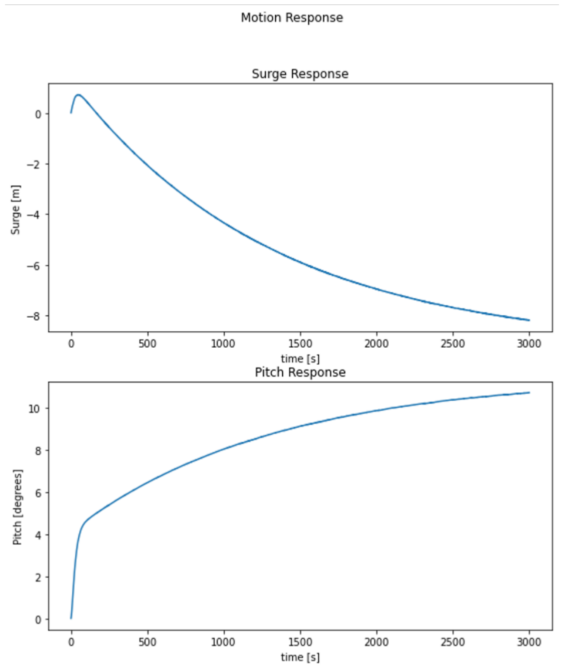
$$x_{1|D_{max}} = [11.38m, 183.98m, 10.0mm]^T 511 \cdot 10^3 \tag{4.2}$$

$$x_{2|D_{min}} = [10.00m, 191.24m, 10.5mm]^T 489 \cdot 10^3 \tag{4.3}$$

The response is investigated in the condition where the largest response is expected to occur. This happens at the highest aero- and hydrodynamic loading which occurs at rated wind speed and the largest slowest significant wave height and period: $U_{mean} = 11m/s$, $Hs = 10$, $Tp = 14s$. From the figure 4.6b it can be seen that the response of the global optima with the smallest diameter ($x_{2|D_{min}}$) does not have an acceptable response. The pitch response settles at over 10°. However the design with the largest diameter ($x_{1|D_{max}}$) falls well within the acceptable range, settling the pitch at just over 5°. Fatigue calculations are done on both designs, it is found that the max damage on the substructure is at the tower base. After a lifetime of 25 years the damage is far within the limit, $D < 0.1$, which is far within the limit on both designs. This indicates that for these designs the global fatigue damage is not a driving factor. However, if the response were to be considered a constraint designs within this Pareto front could still be considered infeasible.



(a) Motion Response of a spar using design $x_{1|D_{max}}$. The environmental conditions consist of rated wind speed (11m/s), $Hs = 10, Tp = 14s$



(b) Motion Response of a spar using design $x_{2|D_{min}}$. The environmental conditions consist of rated wind speed (11m/s), $Hs = 10, Tp = 14s$

Looped over 1000 iterations the Pareto front begins to get a more defined shape. The constraint stopping designs from being smaller than the tower base can clearly be seen by the vertical cut-off at diameters of 10 meters. The thickness range is near the minimum thickness of 10 mm. The standard deviations and averages of this set of global optima are similar to that presented for 50 iterations.

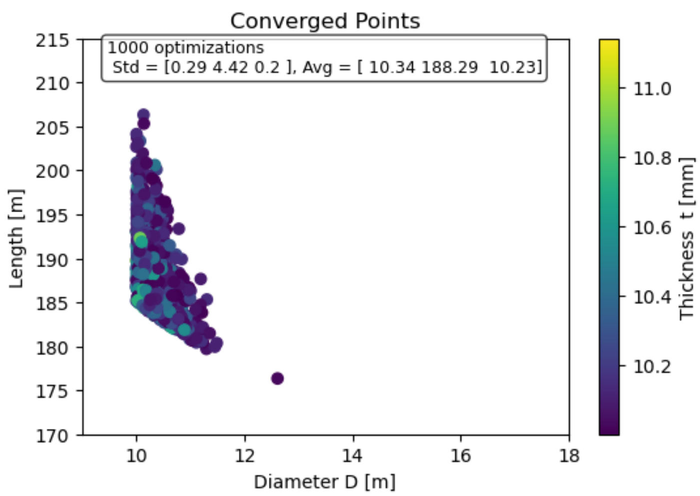
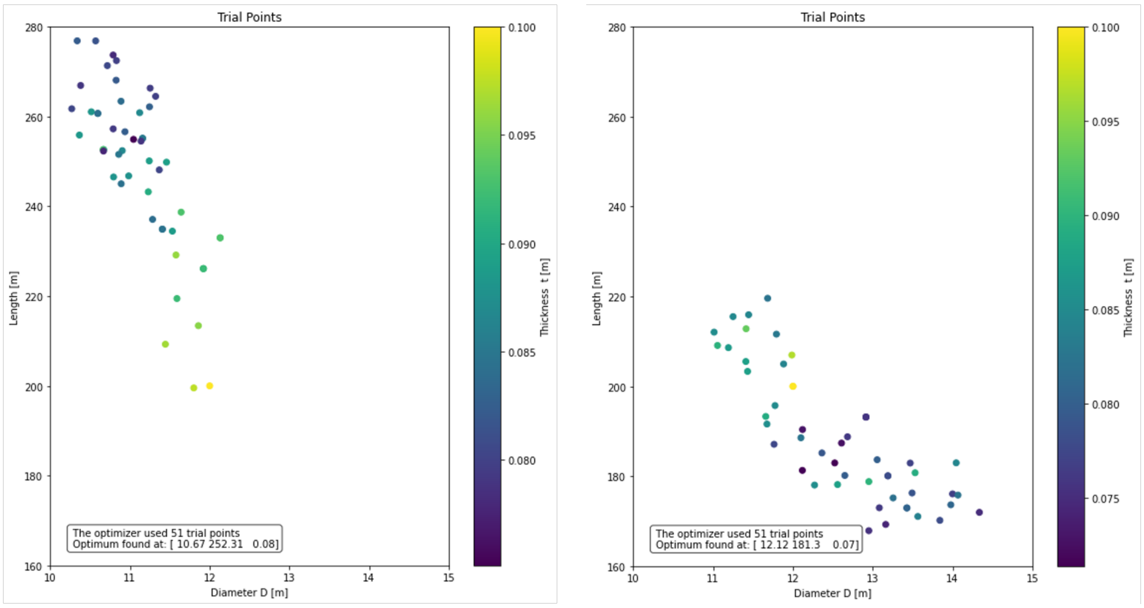


Figure 4.7: Converged points when optimizing 1000 times. Using parameters: $L = 5E^5$,$L = 50$,$\gamma = 0.01$, $K_{lim} = 50$, $r = 0.97$)

4.2. Constraining with Time Domain

For the next set of constraints, the time domain response is included in the design evaluation of the optimizer. This includes a constraint on both pitch and surge response of 8° and 10 meters respectively. This means that in the evaluation of a new feasible point a time domain simulation is run to get the most extreme response. Where $U_{mean} = 11m/s$, $Hs = 10$ and $Tp = 14s$. This has a significant slowing effect on the time it takes to run the optimizer. The exact slowing effect is computer-dependent and in this study, it lead to an evaluation time that was 40 times slower per design compared to the logic constraint evaluation. Due to the expectations of a slower optimization the parameters of the optimizers were adjusted to quicker converging values using an inner loop minimum L of 10 and the convergence parameter $\gamma = 0.1$. When using the adjusted stopping criteria the optimizer used its maximum amount of iterations possible before finding an optimum. The same is found with a traditional more relaxed stopping constraint. The exploration of both optimizations is presented in the two subfigures below. As both the optimizations used the same amount of iterations the adjusted stopping criteria have not had any effect on the reliability of the optimum that was found. Instead, they can be combined as one exploration of the design space. In both cases, the optimizer converged because less than 2 out of 10 trial points resulted in a lower mass.



(a) Using the adjusted stopping criteria proposed in equation 4.1. Optimum found as design: $x = [10.67m, 252.31m, 0.08mm]^T$ and mass:$524 \cdot 10^4$kg

(b) Using traditional Simulated Annealing stopping criteria. Optimum found as design: $x = [12.12m, 181.3m, 0.07mm]^T$ and mass:$376 \cdot 10^4$kg

Figure 4.8: Design space exploration using time domain response as a constraint in the optimizer. Starting point: $x = [12m, 200m, 0.1m]^T$

The two optimal masses found have a significant difference of more than 35%. This leads to the interpretation that the design space has not been sufficiently explored. To counter the limited exploration of these two optimizations, a third optimization is performed where L is set to 50, implying that there will be 50 design evaluations before the temperature is adjusted and stopping criteria are checked. This ensures a broader exploration that will increase the likelihood of finding a better design.

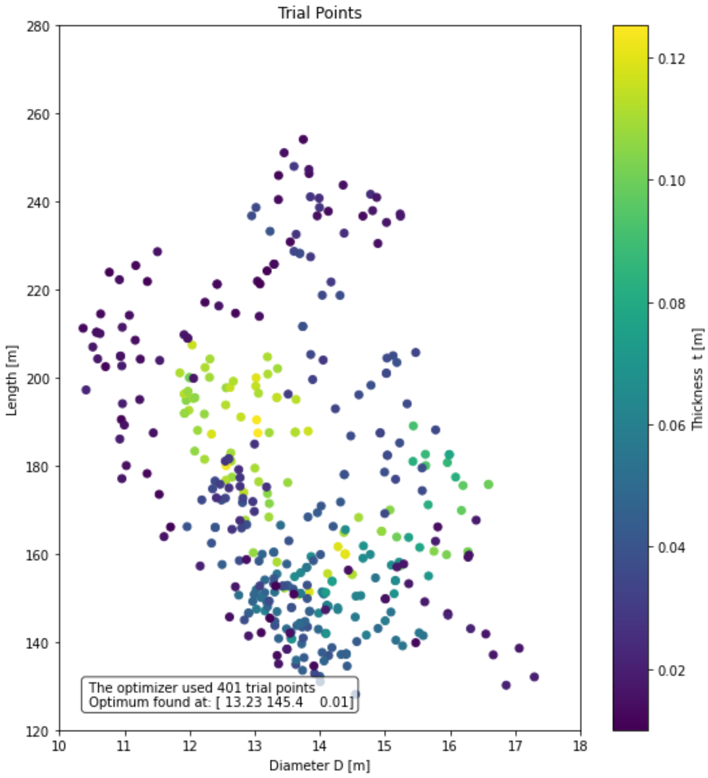


Figure 4.9: Distribtution of trial points for a single iteration with $T_0 = 5E^5$,$L = 50$,$\gamma = 0.01$, $K_{lim} = 50$, $r = 0.97$

Figure 4.9 shows that for this optimization run there is a good investigation of the feasible design space. The length, diameter and thickness range investigated ranges in a far wider area than what was done before and presented in figure 4.8. This maps a part of the feasible design space. It is peculiar that in the space where the length of the spar is between 180 and 200 meters, the diameter between 12 and 14 the feasible points found all have the thicker steel walls $t > 0.10$ meter. The optimum found within this optimization run is: $x = [13.12, 145.5, 0.01]^T$ meters. Which sizing wise is far smaller when compared to the optima found before. The optimum mass found is also far less than in the other two runs. The optimal mass found in this optimization run is $472 \cdot 10^3$kg, which is an order of magnitude lighter than the optimal designs found in the other two optimization runs.

This leads to the conclusion that running the optimization with parameter L set to 10 is insufficient to result in a reliable optimum. Furthermore, when compared to the results of the optimzation done using only logical constraints, the optimizer has found lighter designs that are still within the acceptable motion response range.

The fatigue performance of the optima found in this test case, using design $x = [13.12, 145.5, 0.01]^T$ meters. Results in the maximum motion response are presented in figure 4.10, where as expected the response is within an acceptable range. This can be expected because if it were to exceed those values the design would be considered infeasible. It is good to note that the lightest design does not necessarily reach the limit of the motion response, as such it can be stated that the optimum design that lies within the feasible space does not go to the extremes of response. In other words, the current constraints could be made stricter without negatively impacting the global optimum.

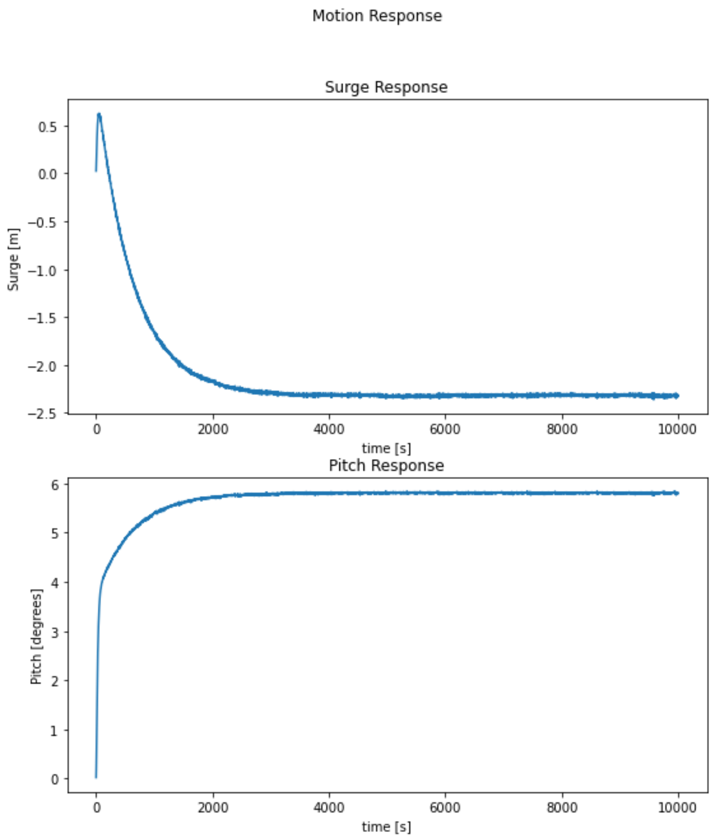


Figure 4.10: Response of optimal design in extreme conditions. $Hs = 10$, $Tp = 14s$

When running this global optimum through the fatigue analysis it results in the highest fatigue being found at the tower base. Where with a lifetime of 25 years the accumulated damage is 0.04766. This means that global fatigue will not be a governing factor in the optimum design of the spar.

Increasing design space exploration

As figure 4.10 suggests more of the design space had been explored when compared to the figures in 4.8. This came at a large computational cost which brings into question the practicality of this optimizer as a design tool for the clientele of DNV. However to get an idea of what the feasible design space looks like the parameters are adjusted once again to try and cover even more space. The result is shown

K_{lim}	L	T_0	r	δ	J	γ	Δ
50	100	5^5E	0.97	0.05	50	0.01	0.05

in the figure below. Where the optimum found is at: $x = [13.12, 145.5, 0.01]^T$ meters. And resulting in a steel mass of $429 \cdot E^3$kg. This is the lightest feasible design found so far. It is interesting to see that there is a feasible design space when using the designs with a thickness of less than 0.02 meters, which lies within a L/D ratio of approximately $200/16 \approx 11$ to $275/21 \approx 13$. However, while following this feasible axis that lies between those ratios designs become increasingly heavy, so although interesting does not have to be further investigated for optimal design research. It can also be seen that when the optimizer considers designs that are on the thicker side (green in the figure) feasible designs are found in a smaller range of diameter and length. The designs with a larger thickness are found at lengths of 100 to 200 meters however at a sloped relationship. Where the length-to-diameter ratio actually varies quite some going from an approximate range of L/D of 15 down to 5.55. In this design set the shorter the length of the spar the wider the diameter has to be to allow for enough ballast for stability and enough mass for a feasible response.

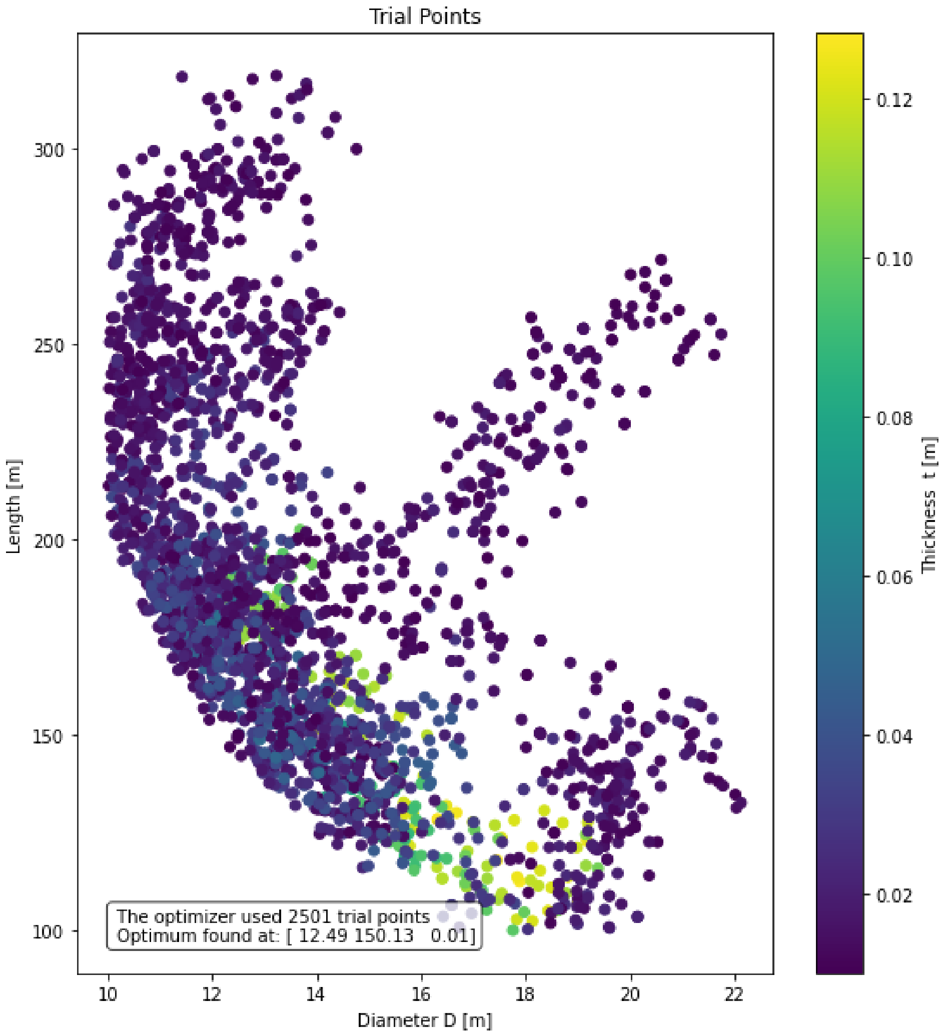


Figure 4.11: Distribtution of trial points for a single iteration with $L = 5E^5$,$L = 100$,$\gamma = 0.01$, $K_{lim} = 50$, $r = 0.97$

When comparing the feasible design space shown in figure 4.11 to the Pareto space found using the geometric constraints (figure: 4.7) it seems that the feasible design space using time domain constraint is far broader than the optimum found in the Pareto front from the geometric constraints. As such considering the extreme response as a constraint is seen to be computationally heavy and results in considering designs that can be thrown away beforehand. They are already known to be too heavy and fall outside the before-found Pareto front.

4.3. Constraining with Fatigue Damage

When the optimizer includes fatigue constraints the optimizer fails to run within a reasonable time. This is due to the large number of simulations necessary to acquire a realistic damage lifetime of the structure. The fatigue analysis consists of running 21-hour long simulations. If this needs to be done for every potential design it would result in an analysis that is too long to be considered feasible. As such this has not been done for this research, the decision was made instead to use some of the optima found using both the time domain constraints and the geometric constraints and investigate the lifetime fatigue damage. In both cases, this has led to very small damage to overall global fatigue. This is attributed to the fact that due to the floating nature of the substructure the bending stress taken up in the structure is actually far less when compared to a structure that is bottom-fixed. This reduction is attributed to the fact that bending moments on a floating structure cause a motion response rather than solely causing internal bending stress.

5

Conclusion and Discussion

5.1. Conclusion

This research has been an investigation of constraint sets on the optimization of the design of a spar-type substructure. In this chapter, the findings from this research are presented after which some crucial critiques are expressed about the study, followed by recommendations for future research on the topic.

This research has been a review of a time domain simulation and non-gradient-based optimization for the design of a spar-type substructure of a floating wind turbine. Two constraint sets have been investigated, the first considering only the geometry of the substructure and the next considering the geometry and time domain response. The third constraint set, using fatigue life damage as a constraint was not investigated as this would not govern the design space and would be computationally infeasible.

Geometric constraint conclusions

The first constraint set only considered geometric constraints. This resulted in two main findings: mooring stiffness is a critical consideration and the stopping criteria need adjusting for proper exploration of the design space.

The geometric constraints are first used with a set mooring stiffness, that is to say, the mooring design is considered as a set input that does not depend on the design of the substructure. This is first used with a single optimization run, the exploration of the feasible design space shows a favour of deep and thin, or shallow and wide designs. The set mooring stiffness is used while running 4 experiments to investigate optimization parameters. In the experiments, the optimization is run for a set amount of iterations and varying parameters. This showed that using a parameter L of 10 results in varying optima of nearly 8% and increased the number of unsolved optimizations. From the other experiments, it is concluded that using L of 30 results in reliable optima. Furthermore reducing γ from 0.01 to 0.005 did not result in any significant increase in optimum. When using the generic stopping constraints the optimizer was found to converge rather quickly and as such the stopping criteria are adjusted to only be called upon at a set amount of trial points or change in temperature. This results in a much broader exploration of the design space. The optimization is run for 40 times using the normal stopping constraint and adjusted one which greatly improves the view on the Pareto front. The Pareto front is found to be on an axis that would seem unrealistic for the design of a substructure. And it is found that this Pareto front is governed by the surge natural period constraint which is mainly dependent on the mooring stiffness. As such it is concluded that a set mooring stiffness is not appropriate for the investigation of multiple designs. This is a significant finding as mooring design, due to its low cost, is considered to be an afterthought in the design of floating structures. Where often times the mooring design is a result of the structure rather than the other way around. This led to the use of a mooring stiffness that is based on a natural frequency in surge of 60 seconds. Where a similar investigation is done using both the traditional and adjusted stopping criteria, this time using 50 iterations. Using the adjusted stopping criteria resulted in a far narrower area of optimal solutions. From the 50 optimizations run with the adjusted stopping criteria, two optimal designs are used, chosen to be the most extreme diameters found in the Pareto front. When the response is investigated it showed one to be within and one to be outside of acceptable response behaviour. This indicates that if anything outside the response constraint is deemed to be unacceptable, using time domain response as a constraint would be useful. Furthermore, the fatigue analysis on both these designs resulted in damage that was less than 10%, which indicates that global fatigue is not a governing factor in determining the optimal design. Rendering any investigation of using the fatigue damage as a constraint useless, this was the biggest motivation to not consider fatigue analysis as a constraint in one of the constraint sets. This would only slow down the optimizer without adding any new information to the system. Due to the computationally feasible nature of the first constraint set, it has been run 1000 times. This showed a clearer view of the Pareto front consisting of a broad length of spar ($180 - 208m$), limited thickness ($10.0 - 11.2mm$) and a limited diameter range of ($10 - 12.8m$). The lowest diameter comes from the set constraint that the diameter of the spar can not be smaller than that of the tower, in this case, that is $10m$. Most optimizations result in a small thickness and it does not seem as though there is any reason to have a thicker than necessary thickness. As such it can be concluded that for the preliminary design of a spar-type substructure of a floating wind turbine where only the global fatigue is considered. The steel thickness can be chosen to be the thinnest allowable thickness rather than a design variable. As while using the thinnest steel the lifetime fatigue remains far from a critical value.

Time domain constraint conclusions

Using the extreme response of the system as a constraint results in a significant increase in computational cost. So much so that it gives another reason why using global fatigue would not be a suitable constraint. When using the extreme response as a constraint a one-hour simulation is run. For the global fatigue investigation, this is done for 21 environmental conditions which would render an optimization run computationally infeasible for any worthwhile results. To decrease the computational cost the extreme response is used as a constraint in two optimizations using parameter $L = 10$. Although from the first constraint set it was concluded that this would result in less reliable results it does allow for a quick exploration of the design space and at least two potential optima. Not only did this result in two very different explorations, but also in two designs that resulted in an objective function difference of more than 35%. This further verified the conclusion that $L = 10$ does not result in a sufficient exploration of the design space. Given the large computational cost, the choice was made to use $L = 50$ which resulted in a far broader exploration of the design space, after which $L = 100$ was also run (costing multiple days to run) to really get a good idea of what the feasible design space looks like. When using thicknesses that are more than the minimum value the feasible design space is pulled to smaller lengths and thinner diameters. Whereas when using the thinnest steel thickness a far broader area is considered to be feasible. This leads to the conclusion that due to the large nature of the feasible design space, it is not computationally efficient to immediately consider the time domain response.

5.1.1. Overal conclusions

This research has shown that both response analysis and geometrical constraints are relevant to be considered in the optimization of a spar type floating substructure. However global fatigue analysis is nor relevant, nor computationally viable to use as a constraint in such optimizations. The mooring stiffness governs the natural period in surge, and in doing so plays a defining role in determining the optimal solution. As when determining the optimal design a governing constraint becomes the natural period.

5.2. Discussion and Reccomendations

Aerodynamic damping When calculating the aerodynamic forcing the thrust force is calculated with a simplified thrust coefficient that is based on the relative wind speed. Where a reduction factor is used to account for ”situations where the movement of the platform changes the inflow that the turbine experiences”, this can otherwise be translated to aerodynamic damping. Or at least the aerodynamic damping caused by the movement of the platform. On top of that the damping matrix also has an analytical expression for aerodynamic damping. This results in the finding that actually the damping is considered twice and as such the aerodynamic damping effects have been ’over’ considered. It would be worth investigating what the effects are on the thrust force when this reduction factor is not used, or when the aerodynamic damping is not considered in the damping matrix.

Coupling thrust force and Heave motion This model presumes that the thrust force is always working in the x-plane of the global coordinate system. There is no coupling between pitch motion and heave motion. In contrast, it could be theorized that at any pitch angle ϕ, the thrust force is angled as well and therefore has a contribution to the z-plane of $\sin\theta \cdot F_{thrust}$ which in turn should affect the heave motion of the structure. At small angles, this can be reduced to $\theta \cdot F_{thrust}$. The calculation of the heave response does not seem to serve any real purpose in this research as without it coupling to bending to pitch or heave, and it will not add to bending stress and, therefore, not influence the fatigue. There are also no constraints used for heave.

Response The response testing of the turbine showed that the largest response would be around 1 degree (for the given test design). Compared to the Pareto front from the first experiment, this is an infeasible, too-small, too-light design. As such, it is to be expected that the designs from this Pareto front would have even less response. It would be worth considering either changing the constraints on response or finding a more governing way to test the system for optimization. Using inactive constraints is an unnecessary calculation.

Design Variables The optimizer considers the steel thickness to be a continuous function. It could be a point of discussion whether this is a good design decision, as plate thickness is typically predetermined and can be considered a discrete variable. It might even lead to far less computational expense to run the optimizer for 4 set thicknesses, using only the diameter and length of the spar as design variables. Furthermore, it would be interesting to increase the complexity of the design of the spar by allowing a variation in diameter and steel thickness between the welded parts of the spar.

Fatigue Calculations The fatigue calculations are based on a set of environmental conditions with a combination of the most probable wind speed and wave height. At higher wind speeds, it is said to be more important to consider a set of wave heights and therefore wave periods. However, if the frequency of occurrence is also considered the question is how much these conditions contribute to the lifetime damage. Most of the damage over the lifetime of the turbine will be done when the response is at its largest, which is the case at lower wind speeds which come in combination with a smaller amount of relevant possible wave heights. It could be investigated whether, by limiting the conditions used for the fatigue calculation analysis, the lifetime damage constraint would become a feasible constraint for the simulated annealing algorithm.
Furthermore, the fatigue calculations are done using a global analysis of the structure. For floating structures this damage is found to be very low, this is attributed to the fact that compared to a bottom-founded offshore structure the bending stress in the structure is far less. However, what has not been investigated is the local fatigue at the mooring location. This cyclic loading could offer a more relevant fatigue life damage than the global bending stress.

Two Optimzations For further development of this tool, it would be recommendable to use the Pareto front found using many iterations with the geometric constraint set as design constraints in the response constraint set. It has been seen that when considering the extreme response as a constraint the feasible design space remains very large. As such many designs are considered that are already too large. For any other given design, this could be achieved by running two separate runs of optimizations. One where the geometric constraints are considered for many optimzations to develop the Pareto front and therefore define the boundaries that should be considered for the time domain extreme response. This would in some way eliminate the heavy computational nature of the optimizer when also considering the extreme response.

References

[1] Irena – International Renewable Energy Agency. *FUTURE OF WIND Deployment, investment, technology, grid integration and socio-economic aspects A Global Energy Transformation paper Citation About IRENA*. 2019. ISBN: 978-92-9260-155-3. URL: www.irena.org/publications..

[2] Nana O. Abankwa et al. “Ship motion measurement using an inertial measurement unit”. In: *IEEE World Forum on Internet of Things, WF-IoT 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2015, pp. 375–380. ISBN: 9781509003655. DOI: 10.1109/WF-IoT.2015.7389083.

[3] A. R. Abdulghany. “Generalization of parallel axis theorem for rotational inertia”. In: *American Journal of Physics* 85.10 (Oct. 2017), pp. 791–795. ISSN: 0002-9505. DOI: 10.1119/1.4994835.

[4] Ahmed Abrous, Rene Wamkeue, and El Madjid Berkouk. “Modeling and simulation of a wind model using a spectral representation method”. In: *Proceedings of 2015 IEEE International Renewable and Sustainable Energy Conference, IRSEC 2015*. Institute of Electrical and Electronics Engineers Inc., Apr. 2016. ISBN: 9781467378949. DOI: 10.1109/IRSEC.2015.7455141.

[5] Alesund University Press. *HywindScotland2*. URL: https://tinyurl.com/HywindImage.

[6] C. Amzallag et al. “Standardization of the rainflow counting method for fatigue analysis”. In: *International Journal of Fatigue* 16.4 (June 1994), pp. 287–293. ISSN: 0142-1123. DOI: 10.1016/0142-1123(94)90343-3. URL: https://reader.elsevier.com/reader/sd/pii/0142112394903433?token=7D3E1A5346731D33522C3D05D92862E9A8504789C033DA336B355EA8D04D63AB9525F7A728FDC15452EB1629DC45F18B&originRegion=eu-west-1&originCreation=20221024173242.

[7] Mir M. Atiqullah. “An Efficient Simple Cooling Schedule for Simulated Annealing”. In: 2004, pp. 396–404. DOI: 10.1007/978-3-540-24767-8{_}41.

[8] Okezue Bell. “Applications of Gaussian Mutation for Self Adaptation in Evolutionary Genetic Algorithms”. In: (Jan. 2022). URL: http://arxiv.org/abs/2201.00285.

[9] D. Benasciutti and R. Tovo. “Spectral methods for lifetime prediction under wide-band stationary random processes”. In: *International Journal of Fatigue* 27.8 (Aug. 2005), pp. 867–877. ISSN: 0142-1123. DOI: 10.1016/J.IJFATIGUE.2004.10.007.

[10] Jeffrey Bierman and Eric Kincanon. “Reconsidering Archimedes’ Principle”. In: *The Physics Teacher* 41.6 (Sept. 2003), pp. 340–344. ISSN: 0031-921X. DOI: 10.1119/1.1607804.

[11] Matthias Brommundt et al. “Mooring System Optimization for Floating Wind Turbines using Frequency Domain Analysis”. In: *Energy Procedia* 24 (Jan. 2012), pp. 289–296. ISSN: 1876-6102. DOI: 10.1016/J.EGYPRO.2012.06.111.

[12] Bernard H Bulder, Andrew Henderson, and Rene Huijsmans. *Study to feasibility of and boundary conditions for floating offshore wind turbines Float Wind (Drijfwind) View project Offshore Loading and Discharge View project*. Tech. rep. 2002. URL: https://www.researchgate.net/publication/260432811.

[13] V. Cerny. “Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm”. In: *Journal of Optimization Theory and Applications 1985 45:1* 45.1 (1985), pp. 41–51. ISSN: 1573-2878. DOI: 10.1007/BF00940812. URL: https://link.springer.com/article/10.1007/BF00940812.

[14] Etienne Cheynet, Jasna Bogunović Jakobsen, and Charlotte Obhrai. “Spectral characteristics of surface-layer turbulence in the North Sea”. In: *Energy Procedia* 137 (Oct. 2017), pp. 414–427. ISSN: 1876-6102. DOI: 10.1016/J.EGYPRO.2017.10.366.

[15] Tzu Ching Chuang, Wen Hsuan Yang, and Ray Yeng Yang. “Experimental and numerical study of a barge-type FOWT platform under wind and wave load”. In: *Ocean Engineering* 230 (June 2021). ISSN: 00298018. DOI: 10.1016/j.oceaneng.2021.109015.

[16] Alex D.D. Craik. “The origins of water wave theory”. In: *Annual Review of Fluid Mechanics* 36 (2004), pp. 1–28. ISSN: 00664189. DOI: 10.1146/annurev.fluid.36.050802.122118.

[17] Kusum Deep and Hadush Mebrahtu. “Variant of partially mapped crossover for the Travelling Salesman problems”. In: *México © International Journal of Combinatorial Optimization Problems and Informatics* 3.1 (2012), pp. 2007–1558. ISSN: 2007-1558. URL: http://www.redalyc.org/articulo.oa?id=265224466006.

[18] Kusum Deep and Manoj Thakur. “A new mutation operator for real coded genetic algorithms”. In: *Applied Mathematics and Computation* 193.1 (Oct. 2007), pp. 211–230. ISSN: 0096-3003. DOI: 10.1016/J.AMC.2007.03.046.

[19] Turan Dirlik. *Application of computer in fatigue analysis*. Tech. rep. 1985. URL: http://go.warwick.ac.uk/wrap/2949.

[20] DNV GL. *RECOMMENDED PRACTICE DNV GL AS RP-C203: Fatigue design of offshore steel structures*. Tech. rep. 2014. URL: www.dnvgl.com..

[21] DNV GL. *STANDARD DNV GL AS Support structures for wind turbines*. Tech. rep. 2016. URL: www.dnvgl.com..

[22] Suguang Dou et al. “Optimization of floating wind turbine support structures using frequency-domain analysis and analytical gradients”. In: *Journal of Physics: Conference Series*. Vol. 1618. 4. IOP Publishing Ltd, Sept. 2020. DOI: 10.1088/1742-6596/1618/4/042028.

[23] R. Eberhart and J. Kennedy. “A new optimizer using particle swarm theory”. In: *MHS’95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*. IEEE, 1995, pp. 39–43. ISBN: 0-7803-2676-8. DOI: 10.1109/MHS.1995.494215.

[24] Eberhart and Yuhui Shi. “Particle swarm optimization: developments, applications and resources”. In: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*. IEEE, 2001, pp. 81–86. ISBN: 0-7803-6657-3. DOI: 10.1109/CEC.2001.934374.

[25] M. Dolores Esteban, José Santos López-Gutiérrez, and Vicente Negro. *Gravity-based foundations in the offshore wind sector*. 2019. DOI: 10.3390/jmse7030064.

[26] Giuseppe Failla and Felice Arena. “New perspectives in offshore wind energy”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373.2035 (Feb. 2015). ISSN: 1364503X. DOI: 10.1098/rsta.2014.0228.

[27] Evan Gaertner et al. *Definition of the IEA Wind 15-Megawatt Offshore Reference Wind Turbine Technical Report*. Tech. rep. 2020. URL: www.nrel.gov/publications..

[28] David E Goldberg. *A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing*. Tech. rep. 1990, pp. 445–460.

[29] Bruce L Golden and Christopher C Skiscim. *Using Simulated Annealing to Solve Routing and Location Problems*. Tech. rep. 1986.

[30] H.A. Haslum. “MathieuInstability”. In: *Aleternative Shape Of Spar Platforms or Use in Hostile Areas*. 1999.

[31] John Marius Hegseth, Erin E. Bachynski, and Joaquim R.R.A. Martins. “Integrated design optimization of spar floating wind turbines”. In: *Marine Structures* 72 (July 2020), p. 102771. ISSN: 0951-8339. DOI: 10.1016/J.MARSTRUC.2020.102771.

[32] C Hindermarsh. “ODEPACK, a systemized pack of ODE solvers”. In: *Scientific Computing* (1982).

[33] R J Howe. *OTC 5354 Evolution of Offshore Drilling and Production Technology*. Tech. rep. 1986. URL: http://onepetro.org/OTCONF/proceedings-pdf/86OTC/All-86OTC/OTC-5354-MS/2031553/otc-5354-ms.pdf/1.

[34] Xiao Bing Hu and Ezequiel Di Paolo. “An efficient genetic algorithm with uniform crossover for air traffic control”. In: *Computers & Operations Research* 36.1 (Jan. 2009), pp. 245–259. ISSN: 0305-0548. DOI: 10.1016/J.COR.2007.09.005.

[35] Marcus Hutter. “Fitness uniform selection to preserve genetic diversity”. In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC’02 (Cat. No. 02TH86002002*. 2002.

[36] A. Ioannou et al. "A preliminary parametric techno-economic study of offshore wind floater concepts". In: *Ocean Engineering* 197 (Feb. 2020), p. 106937. ISSN: 0029-8018. DOI: 10.1016/J.OCEANENG.2020.106937.

[37] David Jhonson et al. "OPTIMIZATION BY SIMULATED ANNEALING: AN EXPERIMENTAL EVALUATION; PART II, GRAPH COLORING AND NUMBER PARTITIONING." In: (1990).

[38] J. M. Jonkman and D. Matha. "Dynamics of offshore floating wind turbines-analysis of three concepts". In: *Wind Energy* 14.4 (2011), pp. 557–569. ISSN: 10991824. DOI: 10.1002/we.442.

[39] J. C. Kaimal et al. "Spectral characteristics of surface□layer turbulence". In: *Quarterly Journal of the Royal Meteorological Society* 98.417 (1972), pp. 563–589. ISSN: 1477870X. DOI: 10.1002/qj.49709841707.

[40] Dan Kallehave et al. "Optimization of monopiles for offshore wind turbines". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373.2035 (Feb. 2015). ISSN: 1364503X. DOI: 10.1098/rsta.2014.0100.

[41] S Kirkpatrick, C D Gelatt, and M P Vecchi. *Optimization by Simulated Annealing*. Tech. rep. 4598. 1983, pp. 671–680.

[42] Ivan Komusanac et al. *Wind Energy in Europe: 2020 statistics and the outlook for 2021-2025*. Tech. rep. Wind Europe, 2019.

[43] Padmavathi Kora and Priyanka Yadlapalli. "Crossover Operators in Genetic Algorithms: A Review". In: *International Journal of Computer Applications* 162.10 (Mar. 2017), pp. 34–36. DOI: 10.5120/ijca2017913370.

[44] Martin O L Hansen. *Aerodynamics of Wind Turbines*. 2015.

[45] Simon Lefebvre and Maurizio Collu. "Preliminary design of a floating support structure for a 5 MW offshore wind turbine". In: *Ocean Engineering* 40 (Feb. 2012), pp. 15–26. ISSN: 0029-8018. DOI: 10.1016/J.OCEANENG.2011.12.009.

[46] Mareike Leimeister, Maurizio Collu, and Athanasios Kolios. "A fully integrated optimization framework for designing a complex geometry offshore wind turbine spar-type floating support structure". In: *Wind Energy Science* 7.1 (Feb. 2022), pp. 259–281. ISSN: 2366-7451. DOI: 10.5194/wes-7-259-2022.

[47] Mareike Leimeister et al. "Design optimization of the OC3 phase IV floating spar-buoy, based on global limit states". In: *Ocean Engineering* 202 (Apr. 2020), p. 107186. ISSN: 0029-8018. DOI: 10.1016/J.OCEANENG.2020.107186.

[48] Mareike Leimeister et al. "Design optimization of the OC3 phase IV floating spar-buoy, based on global limit states". In: *Ocean Engineering* 202 (Apr. 2020). ISSN: 00298018. DOI: 10.1016/j.oceaneng.2020.107186.

[49] Lin Li, Zhen Gao, and Torgeir Moan. "Joint Distribution of Environmental Condition at Five European Offshore Sites for Design of Combined Wind and Wave Energy Devices". In: *Journal of Offshore Mechanics and Arctic Engineering* 137.3 (2015). ISSN: 1528896X. DOI: 10.1115/1.4029842.

[50] Yichao Liu et al. "Developments in semi-submersible floating foundations supporting wind turbines: A comprehensive review". In: *Renewable and Sustainable Energy Reviews* 60 (July 2016), pp. 433–449. ISSN: 1364-0321. DOI: 10.1016/J.RSER.2016.01.109.

[51] M Lundy and A Mees. *CONVERGENCE OF AN ANNEALING ALGORITHM*. Tech. rep. 1986, pp. 111–124.

[52] RC Maccamy and R Aam Fuchs. *Wave forces on piles: a diffraction theory*. 69th ed. US Beach Erosion Board, 1954.

[53] Sanjeev Malhotra. "Design and construction considerations for offshore wind turbine foundations". In: *Proceedings of the International Conference on Offshore Mechanics and Arctic Engineering - OMAE*. Vol. 5. 2007, pp. 635–647. ISBN: 0791842711. DOI: 10.1115/OMAE2007-29761.

[54] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. "The complex-step derivative approximation". In: *ACM Transactions on Mathematical Software* 29.3 (Sept. 2003), pp. 245–262. ISSN: 0098-3500. DOI: 10.1145/838250.838251.

[55] Qingshen Meng et al. “Analytical study on the aerodynamic and hydrodynamic damping of the platform in an operating spar-type floating offshore wind turbine”. In: *Renewable Energy* 198 (Oct. 2022), pp. 772–788. ISSN: 0960-1481. DOI: 10.1016/J.RENENE.2022.07.126.

[56] Juan C. Meza. “Steepest descent”. In: *WIREs Computational Statistics* 2.6 (Nov. 2010), pp. 719–722. ISSN: 1939-5108. DOI: 10.1002/wics.117.

[57] Miner and A Milton. “Cumalative damage in fatigue ”. In: (1945).

[58] Ji Mingjun and Tang Huanwen. “Application of chaos in simulated annealing”. In: *Chaos, Solitons & Fractals* 21.4 (Aug. 2004), pp. 933–941. ISSN: 0960-0779. DOI: 10.1016/J.CHAOS.2003.12.032.

[59] JR Morison, JW Johnson, and SA Schaaf. “The force exerted by surface waves on piles”. In: *Journal of Petroleum Technology* (1950).

[60] Matjaž Mršnik, Janko Slavič, and Miha Boltežar. “Frequency-domain methods for a vibration-fatigue-life estimation – Application to real data”. In: *International Journal of Fatigue* 47 (Feb. 2013), pp. 8–17. ISSN: 0142-1123. DOI: 10.1016/J.IJFATIGUE.2012.07.005.

[61] Michael Muskulus. *Design optimization of wind turbine support Structures-A Review European Academy of Wind Energy View project AWESOME-Advanced Wind Energy Systems Operation and Maintenance Expertise View project*. Tech. rep. 2014. URL: http://www.isope.org/publications.

[62] J. L. Nazareth. “Conjugate gradient method”. In: *WIREs Computational Statistics* 1.3 (Nov. 2009), pp. 348–353. ISSN: 1939-5108. DOI: 10.1002/wics.13.

[63] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (Jan. 1965), pp. 308–313. ISSN: 0010-4620. DOI: 10.1093/comjnl/7.4.308.

[64] *Non-hazardous material safety data sheet for*. 2018. URL: https://www.lkabminerals.com/wp-content/uploads/2019/02/MagnaDense-SDS-12-06INT-19-03.pdf.

[65] Det Norske Veritas. *OFFSHORE STANDARD DET NORSKE VERITAS AS Design of Floating Wind Turbine Structures*. Tech. rep. 2013. URL: http://www.dnv.com.

[66] NS Energy. *A sea change for HYWIND*. URL: https://www.nsenergybusiness.com/features/featurea-sea-change-for-hywind-4303964/attachment/4-hywind/.

[67] A Palmgren. “Life Length of Roller Bearings or Durability of Ball Bearings”. In: *ZVDI* 68.14 (1924), pp. 339–341.

[68] Nicholas Perrone and Robert Kao. “A general finite difference method for arbitrary meshes”. In: *Computers & Structures* 5.1 (Apr. 1975), pp. 45–57. ISSN: 0045-7949. DOI: 10.1016/0045-7949(75)90018-8.

[69] Linda Petzold. *AUTOMATIC SELECTION OF METHODS FOR SOLVING STIFF AND NONSTIFF SYSTEMS OF ORDINARY DIFFERENTIAL EQUATIONS* LINDA PETZOLDt*. Tech. rep. 1983. URL: https://epubs.siam.org/terms-privacy.

[70] Nicolò Pollini et al. “Gradient-based optimization of a 15 MW wind turbine spar floater”. In: *Journal of Physics: Conference Series*. Vol. 2018. 1. IOP Publishing Ltd, Sept. 2021. DOI: 10.1088/1742-6596/2018/1/012032.

[71] Guido van Rossum. *The python language reference*. Amsterdam: Python Software Foundation, 2010.

[72] Mohammed Ghasem Sahab, Vassili V. Toropov, and Amir Hossein Gandomi. “A Review on Traditional and Modern Structural Optimization: Problems and Techniques”. In: *Metaheuristic Applications in Structures and Infrastructures* (2013), pp. 25–47. DOI: 10.1016/B978-0-12-398364-0.00002-4.

[73] Ameya Sathe and Wim Bierbooms. “Influence of different wind profiles due to varying atmospheric stability on the fatigue life of wind turbines”. In: *Journal of Physics: Conference Series*. Vol. 75. 1. Institute of Physics Publishing, June 2007. DOI: 10.1088/1742-6596/75/1/012056.

[74] Matti Scheu et al. “Human exposure to motion during maintenance on floating offshore wind turbines”. In: *Ocean Engineering* 165 (Oct. 2018), pp. 293–306. ISSN: 0029-8018. DOI: 10.1016/J.OCEANENG.2018.07.016.

[75] Ronald Schoenberg. *Optimization with the Quasi-Newton Method*. Tech. rep. 2001.

[76] Peter Schubel and Richard Crossley. "Wind Turbine Blade Design". In: *Wind Turbine Technology*. Apple Academic Press, Mar. 2014, pp. 1–34. DOI: 10.1201/b16587-3.

[77] Kensuke Sekihara, Hideaki Haneishi, and Nagaaki Ohyama. *Details of Simulated Annealing Algorithm to Estimate Parameters of Multiple Current Dipoles Using Biomagnetic Data*. Tech. rep. 2. 1992.

[78] S.N. Sivanandam and S.N. Deepa. "Genetic Algorithm Optimization Problems". In: *Introduction to Genetic Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 165–209. DOI: 10.1007/978-3-540-73190-0{_}7.

[79] Julio M. Stern. "Simulated annealing with a temperature dependent penalty function". In: *ORSA journal on computing* 4.3 (1992), pp. 311–319. ISSN: 08991499. DOI: 10.1287/ijoc.4.3.311.

[80] Peter J.M. Van Laarhoven, Emile H.L. Aarts, and Jan Karel Lenstra. "Job shop scheduling by simulated annealing". In: *Operations Research* 40.1 (1992), pp. 113–125. ISSN: 0030364X. DOI: 10.1287/opre.40.1.113.

[81] Elizabeth Wayman. *Coupled Dynamics and Economic Analysis of Floating Wind Turbine Systems*. Tech. rep. 2004.

[82] Philip Wolfe. "Convergence Conditions for Ascent Methods". In: *SIAM Review* 11.2 (Apr. 1969), pp. 226–235. ISSN: 0036-1445. DOI: 10.1137/1011036.

[83] Li-Jiang Yang and Chen Tian-Lun. *Application of Chaos in Genetic Algorithms*. Tech. rep. 2. 2002.

[84] Jiaping Yang and Chee Kiong Soh. "STRUCTURAL OPTIMIZATION BY GENETIC ALGORITHMS WITH TOURNAMENT SELECTION". In: *Journal of computing in civil engineering* 11.3 (1997), pp. 195–200.

[85] Yanli Zhang, Weidong Zhou, and Shasha Yuan. "Multifractal analysis and relevance vector machine-based automatic seizure detection in intracranial EEG". In: *International Journal of Neural Systems* 25.6 (Sept. 2015). ISSN: 01290657. DOI: 10.1142/S0129065715500203.

[86] Yichi Zhang et al. "Dynamic responses analysis of a 5 MW spar-type floating wind turbine under accidental ship-impact scenario". In: *Marine Structures* 75 (Jan. 2021), p. 102885. ISSN: 0951-8339. DOI: 10.1016/J.MARSTRUC.2020.102885.

[87] Dan Zhao et al. "Analysis codes for floating offshore wind turbines". In: *Wind Turbines and Aerodynamics Energy Harvesters* (2019), pp. 401–430. DOI: 10.1016/B978-0-12-817135-6.00006-5.

[88] Wangwen Zhao and Michael J. Baker. "On the probability density function of rainflow stress range for stationary Gaussian processes". In: *International Journal of Fatigue* 14.2 (Mar. 1992), pp. 121–135. ISSN: 0142-1123. DOI: 10.1016/0142-1123(92)90088-T.

[89] Yu Zhou et al. "A problem-specific non-dominated sorting genetic algorithm for supervised feature selection". In: *Information Sciences* 547 (Feb. 2021), pp. 841–859. ISSN: 0020-0255. DOI: 10.1016/J.INS.2020.08.083.

A
Coding Basics

The research for this thesis has been done in Python, a widely used programming language in the engineering industry. Classes, objects and functions will be explained in this chapter for purposes of legibility of the thesis. The reader is referred to the 'Python Reference' [71] for more details on the language itself. Although the appendix will contain the code, the thesis report will not go into detail on every function made.
A function in python is much like a mathematical equation, there is an input and an output. The main difference is that in python the type of output can be a set of data types, a value, a print off, virtually anything could be returned by a function. When modeling a system multiple functions can be relevant, it is useful to find efficient ways to group these together.
The code for this thesis was developed using object oriented programming or (OOP). This implies a specific code structure where an 'object' is made which can contain both data and functions. These objects can be made multiple times and carry varieties of the same information. The set of common information and functions is contained within a class. An object is an instance of a class. For example, imagine the goal of a piece of code is to model the manufacturing of three types of cars: SUV, coupé and a sedan. Each type of car will have information relevant to that car, and with it comes specific functions about that car i.e: cost and weight. AS every car will have some of the same information a class 'car' can be made containing information like what type of engine, how many doors, car shape and infotainment system. On the basis of that information the class can then contain functions like;'calculate cost' and 'calculate weight'. Which means that with the class 'car' an instance could be made of the SUV, coupe and sedan that can than be recalled in every other function (or class) that would require an instance of the 'car class' as input.
Throughout the thesis classes will be mentioned. For the optimization of the spar floater it makes sense that the substructure is a separate class containing the design variables. Allowing for the optimizer to make multiple instances of the spar to test and evaluate.

B

Tables

Name	X_{TT} [m]	Z_{TT} [m]	Mass [t]	I_{xx} [kg m^2]	I_{yy} [kg m^2]	I_{zz} [kg m^2]
Yaw system	0.000	−0.190	100.0	490,266	490,266	978,125
Turret nose	5.786	4.956	11.394	12,571	10,890	10,909
Inner generator stator	5.545	4.913	226.629	3,777,313	2,012,788	2,042,312
Outer generator rotor	6.544	5.033	144.963	3,173,003	1,673,269	1,691,864
Shaft	6.208	5.000	15.734	33,009	22,906	23,018
Hub	10.604	5.462	190.0	1,382,171	2,169,261	2,160,637
Bedplate	0.812	2.697	70.329	398,973	515,880	535,055
Flange	4.593	4.831	3.946	4,081	2,065	2087
Misc. equipment	0.000	0.500	50.0	16,667	16,667	25,000
TDO shaft bearing	6.582	5.040	2.230	3,515	1,784	1,803
SRB shaft bearing	5.388	4.914	5.664	8,930	4,593	4,641
Nacelle total	5.486	3.978	820.888	12,607,277	21,433,958	18,682,468
Nacelle total minus hub	3.945	3.352	630.888	10,680,747	122,447,810	10,046,187
kg m^2 kilogram square meters	m	meters			t	metric tons
TDO tapered double outer	SRB	spherical roller bearing				

Table B.1: Lumped Masses and Moments of Inertia for the Nacelle Assembly (The coordinate system has its origin at the tower top, with x pointed downwind [parallel to ground/water and not the shaft] and z pointed up) [27] (Table5-1)

Height [m]	Outer Diameter [m]	Thickness [mm]
0	10	45.517
0.001	10	43.537
5	10	43.527
5.001	10	42.242
10	10	42.242
10.001	10	41.058
15	10	41.058
15.001	10	39.496
28	10	39.496
28.001	10	36.456
41	9.926	36.456
41.001	9.926	33.779
54	9.443	33.779
54.001	9.443	32.192
67	8.833	32.192
67.001	8.833	30.708
80	8.151	30.708
80.001	8.151	29.101
93	7.39	29.101
93.001	7.39	27.213
106	6.909	27.213
106.001	6.909	24.009
119	6.748	24.009
119.001	6.748	20.826
132	6.572	20.826
132.001	6.572	23.998
144.582	6.5	23.998

Table B.2: Height, Outer Diameter and thickness of tower at 5 meter intervals. Taken from [27](Tabel: 4-2)

Wind speed u [m/s]	f_uw	Wave height h [m]	f_hs|uw	Wave Period$t$ [m/s]	f_hs|tp
3	0.05767	1	0.8033	25	0.064
5	0.09329	1	0.858	25	0.064
		2	0.131	5	0.218
				6	0.298
				7	0.2199
7	0.10609	1	0.513	25	0.064
		2	0.47	5	0.218
				6	0.298
				7	0.2199
9	0.09395	1	0.2252	25	0.064
		2	0.605	5	0.218
				6	0.298
				7	0.2199
		3	0.169	6	0.157
				7	0.342
				8	0.287
				9	0.137
11	0.06703	2	0.432	5	0.218
				6	0.298
				7	0.2199
		3	0.428	6	0.157
				7	0.342
				8	0.287
				9	0.137
13	0.03902	2	0.2264	5	0.218
				6	0.298
				7	0.2199
		3	0.4725	6	0.157
				7	0.342
				8	0.287
				9	0.137
		4	0.2541	7	0.134
				8	0.3679
				9	0.322
				10	0.132
15	0.01861	2	0.1004	5	0.218
				6	0.298
				7	0.2199
		3	0.3271	6	0.157
				7	0.342
				8	0.287
				9	0.137
		4	0.4144	7	0.134
				8	0.3679
				9	0.322
				10	0.132
		5	0.142	8	0.141
				9	0.3839
				10	0.3265
				11	0.1151
17	0.00728	3	0.1758	6	0.157
				7	0.342
				8	0.287
				9	0.137
		4	0.369	7	0.134
				8	0.3679
				9	0.322
				10	0.132
		5	0.3291	8	0.141
				9	0.3839
				10	0.3265
				11	0.1151
		6	0.0799	9	0.1643
				10	0.3905
				11	0.3064
				12	0.101

Table B.3: Table with all the relevant environmental conditions

C

Original Formulas

C.1. Aerodynamic Damping

The entries in the matrix can be defined as.

$$c_{xU_1} = N_b \int_0^R \frac{\partial(\mathsf{d}T)}{\partial V_0} dr \tag{C.1}$$

$$c_{xU_5} = N_b h_R \int_0^R \frac{\partial(\mathsf{d}T)}{\partial V_0} dr \tag{C.2}$$

$$c_{\theta_y U_5} = \frac{N_b}{2} \int_0^R r^2 \frac{\partial(\mathsf{d}T)}{\partial V_0} dr \tag{C.3}$$

where N_b is the number of blades, R is the blade length, r the position along the blade and V_0 is the steady inflow of wind. dT is the thrust force at each blade element and h_R is the distance from rotor centre to the centre of the global coordinate system.

$$c_{\theta_y \theta_y} = \frac{N_b}{2} \int_0^R r^2 \frac{\partial(\mathsf{d}T)}{\partial V_0} \tag{C.4}$$

$$c_{xx} = N_b \int_0^R \frac{\partial(\mathsf{d}T)}{\partial V_0} \tag{C.5}$$

D

Source Code

D.1. Experiment 1:Logical Constraint Optimization

Experiment 2 is almost the exact same with a change in $constrainttype$ in line 43 from 'logical' to 'time domain'. And some change in the optimization parameters that are discussed in the relevant chapters

```
1     #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Mon Oct 17 11:32:47 2022
5
6 @author: Giles
7 """
8 import time
9 start_time = time.time()
10
11 import numpy as np
12 from OptimizationAlgorithms.SimulatedAnnealingClass import Simulated_Annealing_Class
13 from objective import calc_steel_mass
14 from Constraintclasses.ConstraintClass import constraints_class
15 from Substructure.SparClass import Spar_Class
16 from SuperStructure.IEA15MWTowerClass import IEA15MW_Tower_Class
17 from SuperStructure.IEA15MWTurbineClass import IEA15MW_Turbine_Class
18 from FullSystem.FullSystemClass import Full_System_Class
19 from Utilities.Windfunctions import weibull_pdf,movewindup
20 from TimeDomain_Response.EnviromentClass import Enviroment_Class
21 from main_utility import optimizeloop,didnotconverge,statistics,design_variable_stats
22 from OptiloopClass import Spar_Optimize_Loop_class
23 from plotfile import plot_single_run
24 # load Turbine & Tower
25 turbine    = IEA15MW_Turbine_Class()
26 tower      = IEA15MW_Tower_Class()
27
28 # set up environment for site 14
29 h   = 400                   # water depth
30 H   = 10.96                 # 50 year wave Significant Wave height
31 T   = 11.06                 # 50 Year wave Period
32 U   = movewindup(33.49,130) # 50 Year Wind (U10 from Join Distirbution site 14)
33 TI  = 0.14                  # Turbulence Intensity
34 environment = Enviroment_Class(h,H,T,U,TI)
35
36 # Set up Spar
37 D0 = 10   #m
38 L0 = 200   #m
39 t0 = 0.1 #m
40
41 x0 = np.array([D0,L0,t0])
42 T0 = 500000
43 constraintclass = constraints_class(turbine,tower,environment,printswitch =0,constrainttype =
     'logical')
44 initial_check = constraintclass.constraint_function(x0)
```

```
print(initial_check)

optimize = Simulated_Annealing_Class(calc_steel_mass,constraintclass.constraint_function,T0,
    x0,L=100,J=50,delta = 0.1,gamma = 0.01, K_lim = 10, r = 0.97)
print(optimize.xbest)

print("---%s seconds ---" % (time.time()-start_time))
plotsingle =1
if plotsingle:
    plot_single_run(optimize.x0_list,optimize.xbest,calc_steel_mass(optimize.xbest))
xbest1 = optimize.xbest
optiloop =1
if optiloop:
    iterations  = 1000
    optclass    = Simulated_Annealing_Class
    objective   = calc_steel_mass
    constraint  = constraintclass.constraint_function
    L           = 50
    optiloop    = Spar_Optimize_Loop_class(iterations,optclass,objective,constraint,x0,L,T0=1
        E5)


    print("---%s seconds ---" % (time.time()-start_time))
```

D.2. Constraint classes

D.2.1. Main Class

```
 #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Nov  6 19:51:10 2022

@author: Giles
"""
from Constraintclasses.LogicConstraintClass import logical_constraint_class
from Constraintclasses.Time_Constraint_Class import time_constraint_class

class constraints_class():
    def __init__(self,turbine,tower,environment,printswitch =1,constrainttype = 'logical'):
        self.turbine,self.tower,self.environment,self.printswitch  = turbine,tower,
            environment,printswitch

        self.constraint_function = self.choose_constraint_function(constrainttype)

        return


    def choose_constraint_function(self,constrainttype):
      if constrainttype == 'logical':
        constraintclass= logical_constraint_class(self.turbine,self.tower,self.environment,
            self.printswitch)
        constraint_function = constraintclass.constraint_function
        return constraint_function
      if constrainttype == 'timedomain':
        constraintclass = time_constraint_class(self.turbine,self.tower,self.environment,self
            .printswitch)
        constraint_function = constraintclass.constraint_function
        return constraint_function
```

D.2.2. Geomtric Constraint Class

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 17 10:54:36 2022

@author: Giles
```

```
"""
import numpy as np

from main_utility import unpack_x
from Substructure.SparClass import Spar_Class
from FullSystem.FullSystemClass import Full_System_Class
from TimeDomain_Response.TimeDomainSimulationClass import Time_Domain_Simulation_Class
from TimeDomain_Response.EnviromentClass import Enviroment_Class
class logical_constraint_class():
    def __init__(self,turbine,tower,environment,printswitch =1):
        self.tower   = tower
        self.turbine = turbine
        self.environment = environment

        #self.D,self.L,self.t = unpack_x(x[0])

        self.printswitch = printswitch
    def constraint_function(self,x):
        #print(x)
        D,L,t = unpack_x(x)

        #check logical feasibilities
        constraint = []

        #check spar sizing
        check_x = self.check_x(D,L,t)
        constraint.append(check_x)
        if any(constraint):
            return np.array(constraint)

        #generate spar
        spar = Spar_Class(D,L,t,self.turbine,self.tower)

        #Check Faulty ballast
        faultyballast = self.check_spar(spar)
        constraint.append(faultyballast)
        if any(constraint):
            return np.array(constraint)

        #check environment
        environment_check = self.check_geometry_v_environment(spar,self.environment)
        constraint.append(environment_check)
        if any(constraint):
            return np.array(constraint)

        #Generate Full System
        fullsystem = Full_System_Class(spar,self.tower,self.turbine)

        full_systemcheck = self.check_full_system(fullsystem)
        constraint.append(full_systemcheck)
        if any(constraint):
            return np.array(constraint)

        #Storm Stability check


        return np.array(constraint)



    def check_x(self,D,L,t):
        '''
        Function to check geometry of just the spar

        Returns
        -------
        bool
            if True, infeasible.

        '''
        if D < 0 or L <0 or t<0:
```

```
            if self.printswitch:
                print('constraint: D<0,L<0 or t<0')
            return True
        if D>L:
            if self.printswitch:
                print('constraint: D>L')
            return True
        if t>D:
            if self.printswitch:
                print('constranint: t>D')
            return True
        if D/L < 0.01 :
            if self.printswitch:
                print('constraint: D/L<0.01')
            return True
        if t > 0.2 or t< 0.010:
            if self.printswitch:
                print('constraint: t out of bounds')
            return True

        else: return False


    def check_geometry_v_environment(self,spar,environment):
        if spar.L> 0.8*environment.h :
            if self.printswitch:
                print('constraint: L>0.8 h')
            return True
        if environment.lamda < 5*spar.D:
            if self.printswitch:
                print('constraint: lamda < 0.5 D (morison invalid)')
            return True
        else:
            return False

    def check_spar(self,spar):
        if spar.faulty:
            if self.prinswitch:
                print('constraint: negative ballast')
            return True
        if spar.GM <1.0 :
            if self.printswitch:
                print('constraint: GM spar<1')
            return True
        else:
            return False

    def check_full_system(self,full_system):
        #logical checks
        if   full_system.D_spar < full_system.tower.tower_base_diameter:
            if self.printswitch:
                print('constraint: D_spar<D_tower')
            return True

        #2.1.2 DNV-OS-J103
        if full_system.spar.T_period[0]<100:
            if self.printswitch:
                print("Design has a natural period of" + str(full_system.spar.T_period[0]))
            return True
        #2.1.3 DNV-OS-J103
        if full_system.spar.T_period[2] < 25:
            if self.printswitch:
                print('constraint: natural period in heave too small')
            return True
        #2.4.2 DNV-OS-J103
        if full_system.GM <1 :
            if self.printswitch:
                print('constraint: GM full system <1')
            return True
        #3.2.18 DNV-OS-J103 (mathieu instability)
        ratio = full_system.T_period[1]/full_system.T_period[2]
```

```
        if ratio >0.45 and ratio < 0.55 :
            if self.printswitch:
                print('constraint:Mathieu instabillity 0.45,0.55')
            return True
        if ratio >0.95 and ratio < 1.05 :
            if self.printswitch:
                print('constraint: Mathieu Instability 0.95,1.05')
            return True
        if ratio >1.45 and ratio < 1.55 :
            if self.printswitch:
                print('constraint:Mathieu Instability 1.45,1.55')
            return True
        if ratio >1.95 and ratio < 2.05 :
            if self.printswitch:
                print('constraint:Mathieu Instability 1.95,2.05')
            return True
        #--blade passing Frequencies
        if self.check_bladepass(full_system.T_period[0],full_system.turbine):
            if self.printswitch:
                print('constraint:Surge is within blade passing frequency')
            return True
        if self.check_bladepass(full_system.T_period[1],full_system.turbine):
            if self.printswitch:
                print('constraint: Heave is within blade passing frequency')
            return True
        if self.check_bladepass(full_system.T_period[2],full_system.turbine):
            if self.printswitch:
                print('constraint:Pitch  is within blade passing frequency')
            return True

        return False

    def check_storm_stability(self,full_system):
        storm = self.environment
        storm.U_mean  = 36
        #2.1.3 DNV-OS-J103 (storm stability (no yaw fautl)
        seed    = 1
        wind    = 1
        waves   = 0
        t_eval  = 3600
        dt      = 0.5
        q0      = np.array([0,0,0,0,0,0])
        TD_sim = Time_Domain_Simulation_Class(q0,full_system,storm,dt,t_eval,0.5,seed,wind,
            waves)
        if any(TD_sim.y[2]>8):
            return True
        return False

    def check_bladepass(self,naturalperiod,turbine):
        if naturalperiod > turbine.P1low and naturalperiod < turbine.P1high:
            return True
        if naturalperiod > turbine.P3low and naturalperiod < turbine.P3high:
            return True
```

D.2.3. Time Domain Constraint Class

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Nov  6 20:14:35 2022

@author: Giles
"""

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 17 10:54:36 2022

@author: Giles
```

```
"""
import numpy as np

from main_utility import unpack_x
from Substructure.SparClass import Spar_Class
from FullSystem.FullSystemClass import Full_System_Class
from TimeDomain_Response.TimeDomainSimulationClass import Time_Domain_Simulation_Class
from TimeDomain_Response.EnviromentClass import Enviroment_Class
class time_constraint_class():
    def __init__(self,turbine,tower,environment,printswitch =1):
        self.tower   = tower
        self.turbine = turbine
        self.environment = environment

        #self.D,self.L,self.t = unpack_x(x[0])

        self.printswitch = printswitch
    def constraint_function(self,x):
        #print(x)
        D,L,t = unpack_x(x)

        #check logical feasibilities
        constraint = []

        #check spar sizing
        check_x = self.check_x(D,L,t)
        constraint.append(check_x)
        if any(constraint):
            return np.array(constraint)

        #generate spar
        spar = Spar_Class(D,L,t,self.turbine,self.tower)

        #Check Faulty ballast
        faultyballast = self.check_spar(spar)
        constraint.append(faultyballast)
        if any(constraint):
            return np.array(constraint)

        #check environment
        environment_check = self.check_geometry_v_environment(spar,self.environment)
        constraint.append(environment_check)
        if any(constraint):
            return np.array(constraint)

        #Generate Full System
        fullsystem = Full_System_Class(spar,self.tower,self.turbine)

        full_system_check = self.check_full_system(fullsystem)
        constraint.append(full_system_check)
        if any(constraint):
            return np.array(constraint)

        response_check = self.check_max_response(fullsystem)
        constraint.append(response_check)
        if any(constraint):
            return np.array(constraint)

        #Storm Stability check
        if not all(constraint):
            if self.printswitch:
                print('design is feasible')
            return np.array(constraint)



    def check_x(self,D,L,t):
        '''
        Function to check geometry of just the spar

        Returns
```

```
        -------
        bool
            if True, infeasible.

        '''
        if self.printswitch:
            print('D =' + str(D),'L ='+str(L),'t ='+str(t))
        if D < 0 or L <0 or t<0:
            if self.printswitch:
                print('constraint: D<0,L<0 or t<0')
            return True
        if D>L:
            if self.printswitch:
                print('constraint: D>L')
            return True
        if t>D:
            if self.printswitch:
                print('constranint: t>D')
            return True
        if D/L < 0.01 :
            if self.printswitch:
                print('constraint: D/L<0.01')
            return True
        if t > 0.15 or t< 0.010:
            if self.printswitch:
                print('constraint: t out of bounds')
            return True
        #if L<65 :
          # if self.printswitch:
            #    print('constraint: L<65')
            #return True


        else: return False


    def check_geometry_v_environment(self,spar,environment):
        if spar.L> 0.8*environment.h :
            if self.printswitch:
                print('constraint: L>0.8 h')
            return True
        if environment.lamda < 5*spar.D:
            if self.printswitch:
                print('constraint: lamda < 0.5 D (morison invalid)')
            return True
        else:
            return False

    def check_spar(self,spar):
        if spar.faulty:
            if self.printswitch:
                print('constraint: negative ballast')
            return True
        if spar.GM <1.0 :
            if self.printswitch:
                print('constraint: GM spar<1')
            return True
        else:
            return False

    def check_full_system(self,full_system):
        if full_system.D_spar < full_system.tower.tower_base_diameter:
            if self.printswitch:
                print('constraint: D_spar < Tower Base')
            return True
        if full_system.L_spar < 2/3*full_system.z_hub:
            if self.printswitch:
                print('constraint: L_spar < 2/3 z_hub')
            return True
        else:
            return False
```

```
    def check_max_response(self,full_system):

        waves = 1
        wind =  1
        Tdur = 1000
        T_transient = 250
        dt = 1
        f_highcut = 0.5

        q0 = np.array([0,0,0,0,0,0])
        H = 10
        T = 14
        U = self.turbine.V_rated
        TI = 0.14
        seed = 1

        environment = Enviroment_Class(self.environment.h,H,T,U,TI)
        TD = Time_Domain_Simulation_Class(q0,full_system,environment,dt,Tdur,f_highcut,seed,
            waves,wind,T_transient)
        if any(abs(TD.x_response)>10):
           if self.printswitch:
               print('Constraint: surge response too high')
           return True
        if any(abs(TD.phi_response)>8):
           if self.printswitch:
               print('Constraint: Pitch Response too high')
           return True
        else:
           return False


    def check_bladepass(self,naturalperiod,turbine):
        if naturalperiod > turbine.P1low and naturalperiod < turbine.P1high:
            if self.printswitch:
                print('constraint: Naturalperiod within 1P')
            return True
        if naturalperiod > turbine.P3low and naturalperiod < turbine.P3high:
            if self.printswitch:
                print('constraint: Naturalperiod within 3P')
            return True
```

D.3. Simulated annealing class

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 30 10:06:01 2022

@author: Giles
"""
from Math.Significance import round_it
from math import exp
import random
import numpy as np

class Simulated_Annealing_Class():
    def __init__(self,objective,constraints,T0,x0,L=10,J=50,delta = 0.1,gamma = 0.01, K_lim =
         10, r = 0.97,printswitch = 0):
        '''
        Initialization of the simulated annealing optimization class

        Parameters
        ----------
        objective : Function
            Objective function to minimize.
        constraints : Function
            constraint function (must return array of boolean, and take the
                                 design variables) (false = within feasible).
        T0 : Value
```

```
            Initial Temprature (often the expected global minumum).
        x0 : Array or List
            Design Variables.

        L : Value, optional
            Minimum trials for each iteration. The default is 10.
        J: Value optional
            Consecutive iterations for stopping. The default is 50


        Returns
        -------
        None.

        '''
        #---- Stop conters
        self.consec_change_count=0
        self.howmanydown_count  =0
        self.K_lim_count        =0
        self.Tempchange_count   =0

        self.T0 = T0                    #expected global minimum
        self.x0 = x0                    #feasible trial point
        self.objective = objective      #Objective function
        self.constraints = constraints  #constraint function
        self.L = L                      #minimum trials in iteration

        self.r=r


        #Parameters for final check
        self.K_lim = K_lim              #Max iterations
        self.J = J                      #consecutive iterations for stopping
        self.delta = delta              #how nearby new points can lie
        self.gamma = gamma              #how small can stopping change be
        #printing
        self.print = printswitch
        #include tempchange in stop criteria
        self.tempchange =1


        self.xbest = x0
        self.xbest, self.x0_list, self.f0_list = self.basic_version()




        return

    def basic_version(self):


        #------step1 (Set up  initial Values )
        Tk = self.T0 #expected global minimum/first temperature
        x0 = self.x0 #feasible trial point
        r  = self.r
        objective = self.objective

        fx0= objective(x0)

        L = self.L #Minimum trials inside loop

        #List for stop criteria
        x0_list = []
        f0_list = []

        x0_list.append(x0)
        f0_list.append(fx0)
        #iteration counter
        K=0             #Outside loop counter
        k=1             #Inside  loop counter
```

```
        I= 0            #Counts the amount of better trial points in an iteration

        while K<self.K_lim:
            #------step 2 (Start to Iterate)
            xk = self.gen_xk_feas(x0)

            fxk = objective(xk)
            df = fxk - fx0

            #------step 3
            if df>0:
                mc = 0
                while df>0:                           #metropolis condition
                    mc  = mc+1
                    pdf = exp(df/-Tk)
                    z = random.uniform(0,1)
                    if  self.print:
                        #print(str(mc))
                        moreinfo = self.print_metropolic_counter(mc)
                        if moreinfo:
                            print('df ='+ str(df), 'pdf ='+ str(pdf),'z='+ str(z),Tk)
                    if z<pdf:
                        x0 = xk
                        fx0 = objective(x0)
                        if I > 0:
                            I=I-1
                        break

                    else:
                        xk = self.gen_xk_feas(x0)
                        fxk = objective(xk)
                        df= fxk-fx0
            else:
                x0 = xk
                fx0= objective(x0)
                I=I+1
            #----- Evaluate the new point

            if k<L:
                k=k+1
                x0_list.append(x0)
                f0_list.append(fx0)
                self.xbest,self.f0_best = self.updatelowest(x0,self.xbest)

            elif self.checkanystop(f0_list, I,K):
                x0_list.append(x0)
                f0_list.append(fx0)
                self.xbest,self.f0_best = self.updatelowest(x0,self.xbest)

                if self.print:
                    print("Optimum found",self.f0_best)
                return self.xbest,x0_list,f0_list
                break
            else:
                K=K+1
                print("K =", K)
                k = 1
                Tk = r*Tk
                x0_list.append(x0)
                f0_list.append(objective(x0))
                I=0

        if self.print:
            print("Iteration Limit reached", objective(self.xbest), 'was the best point' )

        return x0,x0_list,f0_list



    def gen_xk_feas(self,x0):
        '''
```

```
        Function that generates a new feasible set of design variables

        Parameters
        ----------
        x0 : List or Array
            Design Variables.

        Returns
        -------
        xk : List or Array
            New set.

        '''
        xk = self.gen_xk(x0) #randomly generated point in neibourhood of last point

        infeasible = self.checkinfeasible(xk)
        while infeasible:
            xk = self.gen_xk(x0)
            infeasible = self.checkinfeasible(xk)
        return xk

    def gen_xk(self,x0):
        '''
        Function used to generate a new set of variables. Will stay within
        20% of the given trial point

        Parameters
        ----------
        x0 : array
            array of design variables (inputs to objective function).

        Returns
        -------
        array
            Newly randomly generated set of design variables.

        '''
        adjustment = np.zeros(len(x0))
        for i in range(len(x0)):
            adjustment[i] = round_it(random.uniform(-self.delta*self.x0[i],self.delta*self.x0
                [i]),5)


        return x0+adjustment

    def checkanystop(self,f0_list,I,K):
        '''
        Checks all the stopping criteria

        Parameters
        ----------
        f0_list : List
            List of all accepted new points in outer iterations.
        I : Integer/Value
            Counter for the amount of inner iterations that have df<0.
        K : Integer/Value
            Outer iteration counter.

        Returns
        -------
        bool
            Any stop criteria met.

        '''
        Consecutive_change_check    = self.consecutive_change(f0_list)
        notenoughnew                = self.howmanydownstop(I,K,self.tempchange)
        K_check                     = K==self.K_lim

        if Consecutive_change_check:
            if self.print:
                print('ConsecutiveChange')
```

```
            self.consec_change_count = self.consec_change_count +1
            return True
        elif notenoughnew:
            if self.print:
                print('notenoughnew')
            self.howmanydown_count = self.howmanydown_count +1
            return True
        elif K_check:
            if self.print:
                print('K_limit reached')
            self.K_lim_count = self.K_lim_count +1
            return True
        else:
            self.Tempchange_count = self.Tempchange_count +1
            return False



    def consecutive_change(self,f0_list):
        '''
        (1) The algorithm stops if change in the best function value is less
        than some specified factor gamma for the last J consecutive iterations.

        Parameters
        ----------
        f0_list : List
            List of accepted new best points.

        Returns
        -------
        bool
            Stop criteria met.

        '''
        #We need to have atleast a minum amount of outer loop iterations
        if len(f0_list)<self.J:
            return False
        #Then we take the last relevant ones
        else:
            lastlist = f0_list[-self.J:]

        change_list = []
        for i in range(len(lastlist)-1):
            df = abs(lastlist[i]-lastlist[i+1])
            change = df/f0_list[0]
            change_list.append(change)

        if all(i <self.gamma for i in change_list):
            return True
        else:
            return False

    def howmanydownstop(self,I,K,tempchange=0,Delta=0.05):
        '''
        The program stops if I1L < 6, where L is a limit on the number of
        trials (or number of feasible points generated) within one iteration,
        and I is the number of trials that satisfy df < 0 .
        Basically the optimiser should stop if in one iteration only small
        portion of the new points is actually in a better position

        Parameters
        ----------
        I : Value
            Counter of new good positions within iteration.

        Returns
        -------
        bool
            Stop Criteria met.

        '''
```

```
        if tempchange:
            if self.r**K < 0.5 or K > 0.5*self.K_lim:
                if I/self.L<Delta :
                    print(I ,'/', self.L, "<",str(Delta))
                    return True
                else:
                    return False
            else:
                return False
        else:
            if I/self.L<Delta :
                print(I ,'/', self.L, "<",str(Delta))
                return True
            else:
                return False




    def checkinfeasible(self,x0):
        '''
        Function that checks infeasibility. So if the trial point is infeasible
        the function will return True. Please be sure that the constraints function
        returns only a set of Boolean values or integers between 0 and 1.


        Parameters
        ----------
        x0 : List or Array
            Trial Design variables.

        Returns
        -------
        bool
            True    = Infeasuble.
            False   = Feasible Point

        '''
        array_2_check = self.constraints(x0)
        if any(array_2_check):
            return True
        else:
            return False

    def updatelowest(self,x0,xbest):
        if self.objective(x0)<self.objective(xbest):
            #print("bestpoint so far", self.objective(x0))
            xbest = x0
            f0_best = self.objective(x0)
        else:
            xbest = xbest
            f0_best = self.objective(xbest)
        return xbest,f0_best

    def print_metropolic_counter(self,mc):
        if mc == 500:
            return print('Generated 500 infeasible trial points'),1
        if mc == 750:
            return print('Generated 750 infeasible trial points'),1
        if mc == 1000:
            return print('Generated 1000 infeasible trial points'),1
        if mc == 1500:
            return print('Generated 1500 infeasible trial points'),1
```

D.4. Structure

D.4.1. Superstructure

Tower

```python
    # -*- coding: utf-8 -*-
"""
Created on Tue May 10 00:25:34 2022

@author: TORSPI
"""
import pandas as pd
from math import pi, sqrt
import sys
sys.path.append('/Users/Giles/Desktop/Desktop - 'Torstens MacBook Pro/Code2.0/Code/Structure')


class IEA15MW_Tower_Class():
    """ CLass instance to make the tower based on the IEA15 MW reference turbine
    Initially this tower was put on a monopile, therefore it has only been considered
    from 0.00 meters onwards (SWL) """
    def __init__(self) :
        #Tower Properties
        self.tower_steel_E      = 2.00e11 #Youngs modulus in Pa
        self.tower_steel_G      = 7.93e10 #Sheer modulus in Pa
        self.tower_steel_rho    = 7.85e3 #kg/m^3

        #import tower data
        data = pd.read_excel(r'/Users/Giles/Desktop/Desktop - 'Torstens MacBook Pro/Code2.0/ThesisPackages/Structure/SuperStructure/TowerData.xlsx')
        self.tower_height             = pd.DataFrame(data,columns=['Height'])        #heights (transition piece and tower)
        self.tower_diameter     = pd.DataFrame(data,columns=['Outer Diameter'])      #outer Diameter
        self.tower_thickness    = pd.DataFrame(data,columns=['Thickness'])*1e-3      #tower thickness

        #masses
        self.m_tower    = 860e3 #Tower mass

        #Heights
        self.z_hub           = 150   #hub height in meters (above SWL)
        self.z_transition    = 15 #transition piece height
        self.m_list          = self.calc_mass_per_section()
        self.m_tower_IEA     = self.calc_tower_mass()
        self.m_tran          = self.calc_transition_mass()
        self.m_total         = self.m_tran + self.m_tower_IEA
        self.z_cm            = self.calc_centerofmass()

        #Diameters
        self.tower_base_diameter = 10 #m
        self.tower_top_diameter  = 6.5#m


        # Distances from SWL
        self.Ix      = self.calc_moment_of_inertia()                          #Moment of inertia around swl
        self.I_tower = 4.2168*10**8 + self.m_tower*self.z_cm**2
        self.I_test  = 4.2168*10**8 +self.m_total*self.z_cm**2



    def calc_mass_per_section(self):
        self.m_list = []
        for i in range(0,26):
            D     = self.tower_diameter.iat[i,0]
            t     = self.tower_thickness.iat[i,0]
            a_sec = pi *(((D/2)**2)-(D/2-t)**2)
            m_pl = a_sec *self.tower_steel_rho
            delta_h = self.tower_height.iat[i+1,0]-self.tower_height.iat[i,0]
            m_sec = delta_h * m_pl
            self.m_list.append(m_sec)
        return self.m_list

    def calc_tower_mass(self):
```

```
        self.m_tower = 0
        for i in range(7,26):
            m_sec = self.m_list[i]
            self.m_tower += m_sec
        return self.m_tower

    def calc_transition_mass(self):
        self.m_tran = 0
        for i in range(0,7):
            m_sec = self.m_list[i]
            self.m_tran += m_sec
        return self.m_tran

    def calc_centerofmass(self):
        self.dcm_list = []
        dcmm_list = []
        #making a list of masses and their respective individual COM
        for i in range(0,26):
            dcm = (self.tower_height.iat[i,0]+self.tower_height.iat[i+1,0])/2
            self.dcm_list.append(dcm)
            dcmm_list.append(dcm*self.m_list[i])

        z_cm =   sum(dcmm_list)/(self.m_tower_IEA+self.m_tran)
        return z_cm

    def calc_moment_of_inertia(self):
        """
        Function that returns the moment of inertia of the tower.
        The coordinate system is centered at the SWL


        Returns
        -------
        Value
            Moment of inertia.

        """
        Ix_list = []

        for i in range (0,26):
            D = self.tower_diameter.iat[9,0]
            t = self.tower_thickness.iat[i,0]
            h =  self.tower_height.iat[i+1,0]-self.tower_height.iat[i,0]

            Ix_sec = 1/12 * self.m_list[i]*(3*((D/2)**2+(D/2-t)**2)+h**2)
            Ix_list.append(Ix_sec + self.m_list[i]*self.dcm_list[i]**2)

        self.Ix_tower = sum(Ix_list)
        return self.Ix_tower
```

Turbine

```
# -*- coding: utf-8 -*-
"""
Created on Wed May 11 23:09:02 2022

@author: TORSPI
"""
from math import sin, cos, radians
import numpy as np
class IEA15MW_Turbine_Class():
    def __init__(self):
        """
        Initializes the turbine class which are modeled \ atop the tower
        Can be further detailed at later stage, see table 5-1 in the defenition of
        the 15MW reference turbine. CUrrently the Ixx moment of inertia is taken
        and the blades weights are neglected (although mentioned in the INIT function)

        """
```

```
        self.m_rna       = 820.888e3                                    #mass of rotor nacelle assembly
        self.m_blade     = 65e3                                         #mass of individual blade
        self.m_total     = self.m_rna + 3*self.m_blade
        self.z_hub       = 150                                          #meters above sea level

        self.z_cm        = self.z_hub + 3.97

        self.D_rotor     = 240                                          #m
        self.V_rated     = 11                                           #m/s
        self.L_blade     = 120                                          #m given in IEA
        self.blade_cm    = 26.8                                         #m given in IEA
        self.y_cm_blade,self.z_cm_blade = self.calc_cm_blades()         #--> should be 0  (center of gravity of three blades)

        #----- moments  of inertia
        self.Ixx_rna_IEA     = 12602277      #kg m^2
        self.Ixx_rna     = self.Ixx_rna_IEA + 3*self.m_blade*self.blade_cm**2    #moment of inertia included blades as point masses
        self.Ixx_rna_SWL   = self.Ixx_rna + self.m_total*self.z_cm**2            #moment of inertia wrt SWL

        minrotorspeed = 5/60
        maxrotorspeed = 7.56/60
        self.P1low   = minrotorspeed
        self.P1high  = maxrotorspeed
        self.P3low   = 3* minrotorspeed
        self.P3high  = 3* maxrotorspeed

        self.cutin   = 3
        self.cutout  = 25

    def calc_cm_blades(self):
        '''
        Function to check the center of gravity of the three blades. Due to
        symmetry it should end up in the middle of the three blades-> middle of
        the hub.

        Returns
        -------
        y_cm : Value
            center of mass coordinate on y plane (across blades).
        z_cm : Value
            center of mass coordinate on z plane (height).

        '''

        n_blades = 3
        angle_between_blades = 360/n_blades
        angle2use = angle_between_blades - 90
        rad2use  = radians(angle2use)
        Blade1 = np.array([0,self.L_blade])
        Blade2 = np.array([-cos(rad2use),-sin(rad2use)])*self.L_blade
        Blade3 = np.array([ cos(rad2use),-sin(rad2use)])*self.L_blade

        cm = (self.m_blade*Blade1 + self.m_blade*Blade2 + self.m_blade*Blade3)/(3*self.m_blade)
        y_cm = cm[0]
        z_cm = cm[1]

        return y_cm,z_cm
```

D.4.2. Substructure

Spar class

```python
    # -*- coding: utf-8 -*-
"""
Created on Wed Mar 23 12:42:18 2022

@author: TORSPI
"""
import sys
import numpy as np


from math import pi, sqrt
from Math.BooleanFunctions import checkifeven

from Substructure.SparIntactStability import Intact_Stability
from Substructure.SparGeometrics import Spar_Geometrics
from Substructure.SparStructuralArrayClass import Spar_structural_arrays


class Spar_Class():
    def __init__ (self,D,L,t,turbine,tower,cylinder_length = 10 ):
        """
        When initialized this class constructs the spar
        calls on the Spar_Geometrics class to determine masses and inertia which are
        all considered dependent on the geometry.
        Then it calls the IntactStability class to determine the intact stability criteria.


        Parameters
        ----------
        D : Value
            Diameter of Spar.
        L : Value
            Length of Spar.
        t : Value
            steel thickness (considered to be the same over the entire spar) .

        Returns
        -------
        None.

        """
        self.D = D
        self.L = L
        self.t = t
        self.cylinder_length = cylinder_length
        self.D_vec = self.calc_D_vec()
        self.A_vec = self.calc_A_spar_vec()
        self.A_wp = pi*(D/2)**2         #Wateplane area (when pitch =0)
        self.L11  = pi/4*(D/2)**4     #Waterplane Moment of inertia FOR SOLID CYLINDER
        self.L12  = self.L11; self.L22 = self.L11
        #constants
        self.rho_seawater = 1025 # kg/m^3
        self.rho_steel    = 7700 # kg/m^3
        self.rho_ballast  = 5100 # kg/m^3  --> magnadense (dense cement = 2400)
        self.z_freeboard    = 5 #metres
        self.g              = 9.81 #m/s^2
        #mooring
        #self.k_moor = 667000*3
        self.z_moor = -1/3 * self.L #The mooring is thirdway down the spar
        self.I_11a  = pi/4 *(self.D/2)**4   #Used for the rotational stiffness (comes from area moment of inertia solid cylinder)

        #geometrics
        geometrics = Spar_Geometrics(self,turbine,tower)
        self.m_spar,self.m_maincollumn, self.m_topplate,self.m_bottomplate = Spar_Geometrics.calc_steel_mass(geometrics,self) #m_spar is the steel weight of the spar


        self.m_ballast  = Spar_Geometrics.calc_m_ballast(geometrics,self,turbine,tower)
        #print("fault is", self.faulty)
        self.m_total    = Spar_Geometrics.calc_m_total(geometrics,self)
```

```
        self.z_ballast  = Spar_Geometrics.calc_z_ballast(geometrics,self)
        self.ballast_cm = Spar_Geometrics.calc_ballast_cm(geometrics,self)
        self.z_cm       = Spar_Geometrics.calc_centerofmass(geometrics,self)
        self.z_cm_full  = Spar_Geometrics.calc_full_centerofmass(geometrics,self)

        self.Ix         = Spar_Geometrics.calc_Ix(geometrics,self)
        self.Ix_area2   = Spar_Geometrics.calc_Ix_momentofarea(geometrics,self)  #Waterplane
            Moment of inertia FOR spar!
        self.nabla      = Spar_Geometrics.calc_displacement(geometrics,self)


        self.A_cross_section = Spar_Geometrics.calc_cross_sectional_area(geometrics,self)
        self.weld_locations = Spar_Geometrics.calc_weld_locations(geometrics,self)

        #stability
        stability = Intact_Stability(self)
        self.CG = Intact_Stability.calc_CG(stability,self)
        self.CB = Intact_Stability.calc_CB(stability,self)
        self.GM = Intact_Stability.calc_GM(stability,self)
        #Structural Arrays
        structuralarrays = Spar_structural_arrays(self)
        self.Mass_Array = Spar_structural_arrays.Mass_array(structuralarrays,self)
        self.AddedMass_Array = Spar_structural_arrays.Added_mass_array(structuralarrays,self)

        self.k_moor = (2*pi*1/60)**2*(self.Mass_Array[0][0]+self.AddedMass_Array[0][0])
        self.Stiffness_Array = Spar_structural_arrays.K_array(structuralarrays,self)
        self.Natural_Period = Spar_structural_arrays.Natural_Period(structuralarrays,self)
        self.f_natural       = 1/(2*pi)*self.Natural_Period
        self.T_period        = 1/self.f_natural

        #print (pi* (self.D/2)**2 *(self.L-self.z_freeboard))

        return

    def calc_D_vec(self):
        """
        Returns
        -------
        array
            Makes an array of positions along the diameter of the spar. These
            steps are taken at 10% of the diameter.

        """
        self.D_vec = np.arange(-0.5*self.D,\
                          +0.5*self.D+0.05*self.D,\
                              0.1*self.D)
        return self.D_vec

    def calc_A_spar_vec(self):
        """
        Function that calculates the area of a circle using a 'strip' method.
        If the amount of slices is even an extra slice is taken from the zero
        point too the sides.

        Then using pytagoras the area of a strip inside a circle can be estimated
        taking the rectangular area and then removing the difference between
        y1 and y2.


        Returns
        -------
        array
            Array of all the areas inside the circle.

        """
        dx = abs(self.D_vec[0]-self.D_vec[1])
        self.Ai = []

        #Determening even amount of slices
        if checkifeven(self.D_vec):
            D_half  = np.array_split(self.D_vec,2)[0]
```

```
            D_half  = np.append(D_half,0)
        else:
            D_half  = np.array_split(self.D_vec,2)[0]


        #loop to create the area
        for i in reversed(range(len(D_half)-1)):
            dx = abs(D_half[i]-D_half[i+1])
            x1 = D_half[i+1]
            x2 = D_half[i]

            y1 = sqrt((0.5*self.D)**2 - x1**2)
            y2 = sqrt((0.5*self.D)**2 - x2**2)

            self.Ai.append(2*y1*dx - dx*(y1-y2))

        Ai_1 = self.Ai[::-1]
        self.A_vec = np.array([*Ai_1, *self.Ai])
        #if len(self.A_vec) != len(self.D_vec):
            #print('houston, problem len(A_vec)-len(D_vec)='+str(abs(len(self.A_vec)-len(self.D_vec))))
        return self.A_vec
```

Spar geometrics

```
    # -*- coding: utf-8 -*-
"""
Created on Mon Mar 28 15:28:31 2022

@author: TORSPI
"""
from math import pi
import numpy as np

class Spar_Geometrics():
    def __init__(self,spar,turbine, tower, stiffeners = 8):
        spar.Cm = 0.50
        spar.L_dis  =np.linspace(0, spar.L,100)
        return

    def calc_steel_mass(self,spar):
        self.m_maincollumn = spar.rho_steel * spar.t * 2*pi * spar.D/2*spar.L
        self.m_topplate    = spar.rho_steel * spar.t * (spar.D/2)**2
        self.m_bottomplate = self.m_topplate

        self.m_spar = (self.m_maincollumn+self.m_topplate+self.m_bottomplate)
        return self.m_spar,self.m_maincollumn, self.m_topplate,self.m_bottomplate

    def calc_m_ballast(self,spar,turbine,tower):
        """
        Takes spar and superstructure to detemine how much ballast is needed

        Parameters
        ----------
        spar : Class Instance
        superstructure: Class Instance.

        Returns
        -------
        Value
            Ballast weight.

        """
        spar.faulty = 0
        self.m_ballast = spar.rho_seawater*(spar.L-spar.z_freeboard) *(spar.D/2)**2*pi\
            -(self.m_spar + turbine.m_total + tower.m_total)
        if self.m_ballast <0:
            #print('negative ballast, faulty design')
            spar.faulty = 1
            return 0
```

```
            return self.m_ballast

    def check_faulty(self):
        if self.faulty:
            return True
        else:
            return False

    def calc_m_total(self,spar):
        '''
        Calculates the total mass of  the spar. i.e. the mass of the steel
        and ballast.

        Parameters

        '''
        self.m_total = self.m_spar + self.m_ballast
        return self.m_total

    def calc_z_ballast(self,spar):
        '''
        Calculates the height from the bottom of the spar  thats filled with ballast


        '''
        ballast_cube = self.m_ballast/spar.rho_ballast
        self.z_ballast = ballast_cube/(pi*(spar.D/2)**2)
        return self.z_ballast

    def calc_ballast_cm(self,spar):
        return -spar.L + 0.5 * spar.z_ballast

    def calc_Ix(self,spar):
        '''
        Calculates the moment inertia of a the spar

        '''


        Ix_steel    = 1/12 * spar.m_spar*(3*((spar.D/2)**2+(spar.D/2-spar.t)**2)+spar.L**2)
        Ix_ballast  = 1/2  * spar.m_ballast * (spar.D/2-spar.t)**2

        self.Ix = Ix_steel+spar.m_spar * spar.z_cm**2 \
            + Ix_ballast+spar.m_ballast*spar.ballast_cm**2
        return   self.Ix

    def calc_Ix_momentofarea(self,spar):
        r2 = spar.D/2
        r1 = spar.D/2 -spar.t
        self.Ix_area2 = pi/4 * (r2**4-r1**4)
        return self.Ix_area2

    def calc_displacement(self,spar):
        self.nabla = spar.rho_seawater*(spar.L-spar.z_freeboard) *(spar.D/2)**2*pi
        return self.nabla

    def calc_centerofmass(self,spar):
        """
        Calculates the distance from SWL to the centre of mass of the spar
        only considers the steel mass

        Parameters
        ----------
        spar : Class instance.

        Returns
        -------
        Value .

        """
        self.z_cm = (self.m_maincollumn*-0.5*spar.L+self.m_topplate*-0.5*spar.t+\
```

```
                    self.m_bottomplate*(-spar.L+0.5*spar.t))/self.m_spar
        return self.z_cm

    def calc_full_centerofmass(self,spar):
        self.z_cm_full = (self.m_spar*self.z_cm + self.m_ballast *(-spar.L+self.z_ballast))/(self.m_total)
        return self.z_cm_full

    def calc_weld_locations(self,spar):
        """
        Function that returns array of locations of welds. Z-axis is arranged such
        that the zero point is at the top of the spar. The array entries begin
        at the frist weld from the bottom of the spar

        Parameters
        ----------
        spar : Class Instance

        Returns
        -------
        weld_locations : Array
            Array containing locations of welds.

        """
        amount_of_welds= int(np.ceil(spar.L/spar.cylinder_length)-1)
        #weld locations
        weld_locations = np.zeros(amount_of_welds)
        for i in range(amount_of_welds):
            weld_locations[i] = -spar.L+spar.cylinder_length*(i+1)

        return weld_locations

    def calc_cross_sectional_area(self,spar):
        A1 = spar.A_wp
        A2 = pi*((spar.D-2*spar.t)/2)**2

        return A1-A2
```

Spar structural arrays

```
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 25 13:02:30 2022

@author: TORSPI
"""
import numpy as np
import scipy.linalg as la
#from Spar_3D.SparClass import Spar_Class
from Math import IntegrateClass
from math import pi

class Spar_structural_arrays():
    def __init__(self,spar):
        return


    def Mass_array(self,spar):
        self.M = np.array([[spar.m_total,0,0],
                           [0,spar.m_total,0],
                           [0,0,spar.Ix]])
        return self.M

    def Added_mass_array(self,spar):
        def a11func(z):
            return spar.rho_steel * pi/4 * spar.D**2*spar.Cm

        a11_integrated = spar.rho_steel * pi/4 * spar.D**2 *spar.Cm * (spar.L)
        self.a11 = a11_integrated

```

```
        self.a33 = 0

        def a55func(z):
            return (spar.rho_steel* pi/4*spar.D**2 * spar.Cm) * (z)**2
        a55_integral = IntegrateClass.Integrate(a55func)
        self.a55 =a55_integral.integral(-spar.L,0,1)

        def a15func(z):
            return (spar.rho_steel* pi/4*spar.D**2 * spar.Cm) * (z)

        a15_integral = IntegrateClass.Integrate(a15func)
        self.a15    =a15_integral.integral(-spar.L,0,1)

        self.A = np.array([[self.a11,   0,          self.a15],
                     [0,                self.a33,   0       ],
                     [self.a15,         0,          self.a55]])
        return self.A


    def K_array(self,spar):
        C11 = spar.k_moor; C33 = spar.rho_seawater*spar.g*spar.A_wp;\
            C15 = -spar.z_moor*spar.k_moor

        C55 = spar.g*((spar.rho_seawater*spar.L11) +(spar.nabla*spar.CB) -spar.m_total*spar.
            z_cm)

        self.K_matrix =  np.array([[C11,0,C15],
                          [0,C33,0],
                          [C15,0,C55]])
        return self.K_matrix


    def Natural_Period(self,spar):

       D, V = la.eigh((np.linalg.inv(self.M+self.A))*self.K_matrix)
       self.omega_natural    = np.sqrt(D)
       return self.omega_natural
       #D, V = la.eigh((np.linalg.inv(self.M+self.A))*self.K_matrix)
       #Omega_natural    = np.sqrt(D)
       #return Omega_natural
```

Spar intact stability

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 28 13:39:24 2022

@author: TORSPI
"""

class Intact_Stability():
    def __init__(self,spar):
        return


    def calc_CG(self,spar):
        self.CG = (spar.m_maincollumn*0.5*spar.L+spar.m_bottomplate*0.5*spar.t+spar.
            m_topplate*spar.L\
              +spar.m_ballast*spar.z_ballast)/spar.m_total

        self.CG = spar.z_cm
        return self.CG

    def calc_CB(self,spar):
        self.CB = -(spar.L-spar.z_freeboard)/2
        return self.CB

    def calc_GM(self,spar):
        self.BM = spar.Ix/spar.nabla

```

```
        GM =self.BM - self.CG-self.CB
        return GM
```

D.4.3. Full system

Full system class

```
    # -*- coding: utf-8 -*-
"""
Created on Mon May  9 21:20:22 2022

@author: TORSPI
"""
import sys
import scipy.linalg as la
import scipy.sparse.linalg as sla
import numpy as np
from math import sqrt, pi
from FullSystem.FullIntactStability import Full_Intact_Stability

class Full_System_Class():
    def __init__ (self,spar,tower,turbine):
        self.spar    = spar
        self.turbine = turbine
        self.tower = tower

        self.D_spar          = spar.D
        self.D_spar_vec      = spar.D_vec
        self.A_spar_vec      = spar.A_vec
        self.L_spar          = spar.L
        self.t_spar          = spar.t
        self.freeboard_spar = spar.z_freeboard
        self.nabla           = spar.nabla
        self.Ixx             = self.calc_inertia(spar,tower,turbine)
        self.g               = 9.81
        self.rho_seawater    = spar.rho_seawater # kg/m^3

        self.m_total         = self.calc_mass(spar,tower,turbine)
        self.z_cm            = self.calc_center_of_mass(spar,tower,turbine)

        #Stability
        stability = Full_Intact_Stability(self)
        self.CG = Full_Intact_Stability.calc_CG(stability,self)
        self.CB = Full_Intact_Stability.calc_CB(stability,self)
        self.GM = Full_Intact_Stability.calc_GM(stability,self)




        self.M_matrix        = self.gen_M_matrix()
        self.K_matrix        = self.gen_K_matrix(spar)


        self.A_matrix        = spar.AddedMass_Array
        self.omega_natural   = self.calc_omega_natural()



        self.f_natural        = 1/(2*pi)*self.omega_natural
        self.T_period         = 1/self.f_natural




        self.D_rotor         = turbine.D_rotor
        self.V_rated         = turbine.V_rated
        self.z_hub           = tower.z_hub

```

```
    def calc_mass(self,spar,tower,turbine):
        self.m_total = spar.m_total + tower.m_total + turbine.m_rna
        return self.m_total

    def calc_center_of_mass(self,spar,tower,turbine):
        self.z_cm = (spar.m_total*spar.z_cm_full +tower.m_total* tower.z_cm + turbine.m_total*turbine.z_cm)\
            /self.m_total
        return self.z_cm

    def calc_inertia(self,spar,tower,turbine):
        self.I_xx = spar.Ix + tower.Ix + turbine.Ixx_rna_SWL
        return self.I_xx

    def gen_M_matrix(self):
        self.M_matrix = np.array([[self.m_total,0,self.m_total*self.z_cm],
                                  [0,self.m_total,0],
                                  [self.m_total*self.z_cm,0,self.Ixx]])
        return self.M_matrix

    def gen_K_matrix(self,spar):
        C11 = spar.k_moor; C33 = spar.rho_seawater*spar.g*spar.A_wp;\
            C15 = -spar.z_moor*spar.k_moor
        x_cb= 0
        C53 = -spar.rho_seawater*spar.g*spar.A_wp*x_cb

        C55 = spar.g*((spar.rho_seawater*spar.L11) +(spar.nabla*self.CB) -self.m_total*self.z_cm)



        self.K_matrix =  np.array([[C11,0,C15],
                                  [0,C33,C53],
                                  [C15,C53,C55]])
        return self.K_matrix


    def calc_omega_natural(self):
        D, V = la.eigh((np.linalg.inv(self.M_matrix+self.A_matrix))*self.K_matrix)
        self.omega_natural   = np.sqrt(D)
        return self.omega_natural

    def calc_z_uw(self,spar):
        L = spar.L

    #def dqdt
```

Full system Intact stability

```
    class Full_Intact_Stability():
    def __init__(self,fullsystem):
        return


    def calc_CG(self,fullsystem):
        self.CG = fullsystem.z_cm
        return self.CG

    def calc_CB(self,fullsystem):
        self.CB = - (fullsystem.L_spar-fullsystem.freeboard_spar)/2
        return self.CB

    def calc_GM(self,fullsystem):
        self.BM = fullsystem.Ixx/fullsystem.nabla


        GM =self.BM - self.CG-abs(self.CB)
```

```
        return GM
```

D.5. Time domain simulation class

```
# -*- coding: utf-8 -*-
"""
Created on Thu May 12 20:57:48 2022

@author: TORSPI
"""
import numpy as np
from math import pi
from scipy.integrate import solve_ivp, simps
from scipy.optimize import fsolve
from scipy.linalg import inv
from math import pi, tanh, cos, sin, cosh, sinh, sqrt
from Kinematics.WaveKinematicsClass import Wave_Kinematics_Class
from TimeDomain_Response.StateClass import State_Class
from Kinematics.WindKinematicsClass import Wind_Kinematics_Class
from TimeDomain_Response.SiteWindClass import Site_Wind_Class

from numbalsoda import lsoda_sig, lsoda

class Time_Domain_Simulation_Class():
    def __init__(self,q0,full_system,environment,dt,Tdur,f_highcut,seed=1,waves=2,wind=2,
        T_transient = 600):
        #inherit matrices + constants from full sytem
        self.environment = environment
        self.spar       = full_system.spar
        self.full_system  = full_system
        self.MA_matrix  = full_system.M_matrix#+full_system.A_matrix
        self.K_matrix   = full_system.K_matrix


        self.g          = full_system.g
        self.D_spar     = full_system.D_spar
        self.D_spar_vec = full_system.D_spar_vec

        df = 1/(2*Tdur)
        self.f_matrix = np.arange(df,f_highcut+df,df)
        #damping matrix guessed

        #generate time matrices
        self.dt = dt;
        self.Tdur = Tdur;
        self.t      = []
        self.t_eval = (np.arange(0,self.Tdur+self.dt,self.dt)).tolist()

        T_transient = T_transient ;
        index_notrans = np.where(np.array(self.t_eval)>T_transient)[0][0]


        self.A_spar_vec     = self.calc_A_spar_vec()
        self.L_spar         = full_system.L_spar
        self.t_spar         = full_system.t_spar
        self.rho_seawater   = full_system.rho_seawater # kg/m^3
        self.D_rotor        = full_system.D_rotor
        self.V_rated        = full_system.V_rated

        self.H    = environment.H   #significant Wave Heigh
        self.T    = environment.T   #significant Wave Period
        self.h    = environment.h
        self.z_uw = environment.z_uw
        self.U_mean = environment.U_mean
        self.TI     = environment.TI
        #passing kinematics class
        np.random.seed(seed)
        self.phi_water = 2*pi * np.random.rand(len(self.f_matrix))
        np.random.seed(seed+1)
```

```
        self.phi_wind  = 2*pi * np.random.rand(len(self.f_matrix))

        self.sitewind       = Site_Wind_Class(self.t_eval,self.f_matrix,self.TI)

        self.wavekinematics = Wave_Kinematics_Class(self.f_matrix,self.H,self.T,self.h,self.
            z_uw,self.t_eval,self.dt,self.phi_water)
        self.windkinematics = Wind_Kinematics_Class(self.f_matrix,self.t_eval,self.U_mean,
            self.TI,self.phi_wind,l=340.2)


        self.A_vec = self.calc_A_spar_vec()

        #Setting up stateclass
        self.stateclass = State_Class(self.full_system,self.wavekinematics,\
                                      self.windkinematics,environment,dt,Tdur)

         #--- Hyrdodynamic Loading
        self.Hydro = self.stateclass.Hydro
        self.waves = waves
        self.wind  = wind

        self.dqdt = self.stateclass.choose_dqdt(self.waves,self.wind)

        self.sol = self.response(q0)

        self.y_notransient = self.remove_transient_response(self.sol.y,index_notrans)

        self.t_response = self.t_eval[index_notrans:]
        self.x_response = self.y_notransient[0]
        self.z_response = self.y_notransient[1]
        self.phi_response = self.y_notransient[2]*180/pi

        return

    def response(self,q0):
        """
        Function that returns the response of the system when subject to wind
        and wave loading. The solution is given at the water line, and can be
        found in the result whichis an OdeResult Object called 'y'.

        The order of the result is: x,z,phi,x_dot,z_dot,phi_dot



        Parameters
        ----------
        q0 : Array or list
            Starting state, (Containing the initial position and velocities of the
                             system?).

        Returns
        -------
        sol : OdeResult Object
            Object containing the soltion to the ODE problem. Contains the states
            at every given timestep.

        """
        sol =solve_ivp(self.dqdt,[0,self.Tdur+self.dt],q0,method='LSODA', t_eval=self.t_eval)
        #sol,succes = lsoda(self.dqdt,q0,self.t_eval)
        return sol

    def calc_A_spar_vec(self):
        """
        Function that calculates the area of a circle using a 'strip' method.
        If the amount of slices is even an extra slice is taken from the zero
        point too the sides.

        Then using pytagoras the area of a strip inside a circle can be estimated
        taking the rectangular area and then removing the difference between
        y1 and y2.

```

```
        Returns
        -------
        array
            Array of all the areas inside the circle.

        """
        dx = abs(self.D_spar_vec[0]-self.D_spar_vec[1])
        D  = self.D_spar
        self.Ai = []

        #Determening even amount of slices
        if self.checkifeven(self.D_spar_vec):
            D_half  = np.array_split(self.D_spar_vec,2)[0]
            D_half  = np.append(D_half,0)
        else:
            D_half  = np.array_split(self.D_spar_vec,2)[0]


        #loop to create the area
        for i in reversed(range(len(D_half)-1)):
            dx = abs(D_half[i]-D_half[i+1])
            x1 = D_half[i+1]
            x2 = D_half[i]

            y1 = sqrt((0.5*D)**2 - x1**2)
            y2 = sqrt((0.5*D)**2 - x2**2)

            self.Ai.append(2*y1*dx - dx*(y1-y2))

        Ai_1 = self.Ai[::-1]
        self.A_vec = [*Ai_1, *self.Ai]

        return np.array(self.A_vec)

    def checkifeven(self,anyvector):
        """
        Function that checks if list or array has even amounts of entries

        Parameters
        ----------
        anyvector : Array or List
            The array or list that you want to consider

        Returns
        -------
        bool
            truth for even, false for uneven.

        """
        if (len(anyvector) % 2) ==0 :
            return True
        else: return False

    def remove_transient_response(self,y,notransientindex):
        y_notransient = np.zeros((6,len(y[0])-notransientindex))
        for i in range(len(y)):
            y_notransient[i] = y[i][notransientindex:]
        return y_notransient
```

D.6. Kinematics

D.6.1. Wave kinematics class

```
    # -*- coding: utf-8 -*-
"""
Created on Tue May 24 12:59:59 2022

@author: TORSPI
"""
```

```python
import numpy as np
from scipy.optimize import fsolve
from math import pi, tanh, cos, sin, sinh, log, exp, sqrt
import random

class Wave_Kinematics_Class():
    def __init__(self,f_matrix,H,T,h,z_uw,t_eval,dt,phi):
        self.f_matrix = f_matrix
        self.H = H      #Significant Wave Height
        self.T = T      #Wave Period
        self.h = h      #Water Depth
        self.z_uw = z_uw        #vector to the water depth
        self.t_eval = t_eval    #time vector
        self.g = 9.81           #gravity constants
        self.dt = dt            #time step
        self.Tdur = t_eval[-1]

        #calculating the kinematics with internal functions of this class
        self.uwave_x, self.uwave_x_dot, self.uwave_z,self.uwave_z_dot,\
            = self.calc_wave_motion(self.H,self.T)
        #wavenumber1wave
        self.k_reg = self.solve_k(1/T,self.h)[0]


        self.Sjs, self.Ajs = self.JonSwap(self.H,self.T,self.f_matrix)
        self.phi = phi
        self.zeta_irr_fft, self.u_irr_matrix, self.du_irr_matrix = \
            self.calc_zeta_u_du_fft(self.Ajs, self.f_matrix)


    def calc_wave_motion(self,H,T):
        x = 0
        f = 1/T
        k,L = self.solve_k(f,self.h)
        omega = f * 2 * pi

        #u_w = np.empty((len(self.z_uw),len(self.t_eval)))                       #empty
            matrix rows for water depth, collums for time
        #du_w= np.empty((len(self.z_uw),len(self.t_eval)))
        u_wx     = []
        du_wx    = []

        u_wz    = []
        du_wz   = []
        for j in range(0,len(self.t_eval)):

                u_wx.append(0.5*omega*H*np.cosh(k*(self.z_uw+self.h))*\
                    cos(omega*self.t_eval[j]-k*x)/sinh(k*self.h))
                du_wx.append(-0.5*omega**2*H*np.cosh(k*(self.z_uw+self.h))*\
                    sin(omega*self.t_eval[j]-k*x)/sinh(k*self.h))

                u_wz.append(-0.5*omega*H*np.sinh(k*(self.z_uw+self.h))*\
                    sin(omega*self.t_eval[j]-k*x)/sinh(k*self.h))
                du_wz.append(-0.5*omega*H*np.sinh(k*(self.z_uw+self.h))*\
                    cos(omega*self.t_eval[j]-k*x)/sinh(k*self.h))

        return np.array(u_wx), np.array(du_wx), np.array(u_wz),np.array(du_wz)

    def calc_zeta_matrix(self,H,T,D_spar_vec):
        """
        Calculates a matrix of waveheights at either side of the spar
        By first aranging the spar in pieces from left to right
        then using airy wave theory to calculate the wave height over time
        at that part of the spar

        The wave height is taken over the middle line of the spar.
        And is taken inbetween the points defined in self.D_spar_vec

        Returns
        -------
        Array
```

```
            wave height at everymoment at every point of the spar. Taken over the
            spars diameter.

        """
        f = 1/T
        omega = f * 2 * pi
        k,L = self.solve_k(f,self.h)
        #getting to the middle of the diameter boundaries of the spar:
        #distance between points on spar
        dx = D_spar_vec[1]-D_spar_vec[0]
        D_spar_between = D_spar_vec + dx/2
        D_spar_between = D_spar_between[0:-1]

        self.zeta_matrix = H/2 * np.cos(omega*np.array(self.t_eval) -k*D_spar_between[np.
            newaxis].T)
        return self.zeta_matrix

    def solve_k(self,f,h):
        """
        Finds the wavenumber k and calculates wavelength using the dispersion
        relationship

        Parameters
        ----------
        f : float
            wavefrequency.
        h : float
            water depth.

        Returns
        -------
        k: float
            Wave Number
        L: float
            Wave Length

        """
        omega = f*2*pi
        fun = lambda k: omega**2-self.g*k*tanh(k*h)
        k = fsolve(fun,4)
        L = (2*pi)/k
        return k.astype(np.float),L.astype(np.float)

    def JonSwap (self,Hs,Tp,f_matrix,Gamma=3.33) :
        fp = 1/Tp
        df = f_matrix[1]-f_matrix[0]
        Sjs = np.zeros(len(f_matrix))
        a   = np.zeros(len(f_matrix))

        for i in range(len(f_matrix)):
            if f_matrix[i] <= fp:
                sigma = 0.07
            else:
                sigma = 0.09

            Sjs[i] = 0.3125*Hs**2 *Tp\
                *((f_matrix[i]/fp)**-5)\
                * exp(-1.25*((f_matrix[i]/fp)**-4))\
                    *(1-0.287*log(Gamma))\
                    * Gamma**exp(-0.5*(((f_matrix[i]/fp)-1)/sigma)**2)


            a[i] = sqrt(2*Sjs[i]*df)

        return Sjs,a


    def calc_zeta_u_du_fft(self,a_wave,f_matrix):
        omega = f_matrix * 2 * pi
        M = len(self.t_eval)
        k_irr =self.calc_k_irr(f_matrix)
```

```

        #---Fast Fourier
        scaling = 1
        amplitude     = a_wave*scaling
        zeta_hat = amplitude * np.exp(1j*self.phi)
        zeta_irr_fft = M * (np.fft.ifft(self.a2sizeM(zeta_hat,M))).real

        #--going from SWL (turning the coordinates around)
        z_coordinates = self.z_uw + self.h

        u_matrix_hat  =  np.zeros([len(z_coordinates),len(self.f_matrix)],dtype = 'complex')
        du_matrix_hat =  np.zeros([len(z_coordinates),len(self.f_matrix)],dtype = 'complex')

        u_matrix  = np.zeros([len(z_coordinates),M])
        du_matrix = np.zeros([len(z_coordinates),M])

        for i in range(len(z_coordinates)):
            u_matrix_hat[i,:] = 0.5*amplitude*np.exp(1j*self.phi)*omega\
                *np.cosh(k_irr*(z_coordinates[i]))/np.sinh(k_irr*self.h)

            du_matrix_hat[i,:] =  0.5*amplitude*np.exp(1j*self.phi)*1j*omega*omega\
                *np.cosh(k_irr*(z_coordinates[i]))/np.sinh(k_irr*self.h)

            u_matrix[i,:] = M*np.fft.ifft(self.a2sizeM(u_matrix_hat[i,:],M)).real
            du_matrix[i,:]= M*np.fft.ifft(self.a2sizeM(du_matrix_hat[i,:],M)).real


        return zeta_irr_fft,u_matrix,du_matrix


    def a2sizeM(self,a,M):

        if len(a) +1 > M:
            print('M already smaller than a, must be bigger to size up vector')

        asized = np.zeros(M,dtype = 'complex')
        for i in range(len(a)):
            asized[i+1] = a[i]

        return asized

    def calc_k_irr(self,f_matrix):
        k_irr = np.zeros(len(f_matrix))
        for i in range(len(k_irr)):
            f = f_matrix[i]
            k_irr[i] = self.solve_k(f,self.h)[0]

        return k_irr

    def gen_k_irr_timeseries(self,u_irr_matrix):
        k_irr_timeseries = np.zeros(len(u_irr_matrix[0]))
        for i in range(len(u_irr_matrix[-1])):
            omega = u_irr_matrix[-1][i]
            fun = lambda k: omega**2-self.g*k*tanh(k*self.h)
            k = fsolve(fun,0.2)
            L = 2*pi/k
            k_irr_timeseries[i] = k
            k_irr_timeseries = np.clip(k_irr_timeseries,0.0001,1)
        return k_irr_timeseries
       # index = np.random.randint(0,len(self.k_irr),(len(self.t_eval)))
       # k_irr_timeseries = self.k_irr[index]
       # k_irr_timeseries = np.clip(k_irr_timeseries,0.3449203,1)
       # return k_irr_timeseries
```

D.6.2. Wind kinematics class

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 16 15:46:15 2022

```

```
@author: TORSPI
"""
import numpy as np
from math import exp
from Utilities.Windfunctions import weibull_pdf
class Wind_Kinematics_Class():
    def __init__(self,f_matrix,t_eval,U_mean,I,phi,l=340.2):
        self.f_matrix = f_matrix
        self.t_eval = t_eval    #time vector
        self.dt = t_eval[1]-t_eval[0]
        self.Tdur = t_eval[-1]

        self.phi = phi

        self.S_wind, self.a_wind = self.skaimal(U_mean,I,l)

        self.V_wind_irr = self.V_wind_fft(self.a_wind,U_mean)
        pass

    def skaimal(self,u,I,l):
        """
        Using the IEC defenition of the kamail turbulence model (International
        Electrotechnical Commission, 2015) This function returns the spectral
        densitiy


        Parameters
        ----------
        f_highcut : TYPE
            DESCRIPTION.
        df : TYPE
            DESCRIPTION.
        u : Value
            Mean Wind Speed.
        I : TYPE
            Turbulence Intensity.
        l : Turbulence Length, set to 340.2 in class. (standard DS742 2007 say 150m)
            DESCRIPTION.

        Returns
        -------
        None.

        """
        #figure out standard
       # sigma  = (TI /100)*u
       # S_f = (4 * sigma * L /u)/(1+6*f*L/u)**(5/3)


        #Past used example
        f_matrix = self.f_matrix
        df = f_matrix[1]-f_matrix[0]
        S_wind   = np.zeros(len(f_matrix))
        a_wind   = np.zeros(len(f_matrix))

        S_wind  = 4*I**2*u*l    *  ((1+6*(f_matrix)*l/u))**(-5/3)
        a_wind  = np.sqrt(2*S_wind*df)

        return S_wind, a_wind

    def V_wind_fft(self,a_wind,U_10_min):
        M = len(self.t_eval)
        V_dynamic_hat   = a_wind*np.exp(1j*self.phi)
        V_dynamic       = M*np.fft.ifft(self.a2sizeM(V_dynamic_hat,M)).real
        V_time_series   = U_10_min + V_dynamic
        return V_time_series




    def a2sizeM(self,a,M):
```

```python

        if len(a) +1 > M:
            print('M already smaller than a, must be bigger to size up vector')

        asized = np.zeros(M,dtype = 'complex')
        for i in range(len(a)):
            asized[i+1] = a[i]

        return asized
```

D.7. Dynamics

D.7.1. Hydrodynamics class

```python
        # -*- coding: utf-8 -*-
"""
Created on Tue May 24 14:11:53 2022

@author: TORSPI
"""
import numpy as np
from Math.Significance import round_it
from math import pi,tanh,sqrt,e
from scipy.optimize import fsolve
import collections.abc

class Hydrodynamics_Class():
    def __init__(self,h,z_uw,full_system):
        self.full_system = full_system
        self.h = h
        self.z_uw = z_uw


        self.g = 9.81

        self.Ca = 1.0
        self.Cd = 0.6

        self.rho_seawater = full_system.rho_seawater
        self.D_spar     = full_system.D_spar
        self.D_spar_vec = full_system.D_spar_vec
        self.A_vec      = full_system.A_spar_vec
        self.L_spar     = full_system.L_spar
        self.z_cm       = full_system.z_cm


    def solve_k(self,f,h):
        """
        Finds the wavenumber k and calculates wavelength using the dispersion
        relationship

        Parameters
        ----------
        f : float
            wavefrequency.
        h : float
            water depth.

        Returns
        -------
        k: float
            Wave Number
        L: float
            Wave Length

        """
        omega = f*2*pi
        fun = lambda k: omega**2-self.g*k*tanh(k*h)
        k = fsolve(fun,200)
        L = 2*pi/k
```

```
        return k.astype(np.float),L.astype(np.float)

    def calc_hydrodynamic_forcing_x(self,u_wave_x,u_wave_dot_x,x_dot,theta_dot,z_uw):
        """
        Function for determining Hydrodynamic forcing for one specific time against
        spar.
        Forcing is determined using the Morison equations.
        considereing unilateral waves along the x-axis

        Parameters
        ----------
        u_wave : array
            wave speed along depth. (1 or 2 dimensional)
        u_wave_dot : array
            wave acceleration along depth. (1 or 2 dimensional)

        Returns
        -------
        dF_hydro: array
            forcing along depth of spar.

        """
        self.z_down_bound,self.z_up_bound = self.gen_zbounds(z_uw)
        if u_wave_x.ndim > 1:

            dF_i = np.zeros((len(u_wave_x[0]),len(u_wave_x)))
            dF_d = np.zeros((len(u_wave_x[0]),len(u_wave_x)))
            for i in range(len(u_wave_x)):
                dF_i[:,i] = (self.Ca+1) * self.rho_seawater * pi * (self.D_spar**2)/4 * (u_wave_dot_x[i]+x_dot[i])
                dF_d[:,i] = self.Cd*0.5*self.rho_seawater*self.D_spar* \
                (u_wave_x[i]-(x_dot[i]+ z_uw*theta_dot[i]))*abs(u_wave_x[i]-(x_dot[i]+z_uw*theta_dot[i]))



        else: #abs(x_dot) < 0.01:
            dF_i = (self.Ca+1) * self.rho_seawater * pi/4 * self.D_spar**2 * \
                 (u_wave_dot_x) #-(x_dotdot-z_uw*theta_dotdot)) #-(x_dot-z_uw*theta_dot))

            dF_d = self.Cd * 0.5 * self.rho_seawater * (self.D_spar)* \
            (u_wave_x -( x_dot+ z_uw*theta_dot)) * abs(u_wave_x -( x_dot + z_uw*theta_dot))

        #else:
         #   V_dz = pi/4*self.D_spar**2
         #
         #   Froude_Krylov   = self.rho_seawater * V_dz * u_wave_dot_x
         #   Hydromassforce  = self.rho_seawater*self.Ca*V_dz*(u_wave_dot_x) #-(x_dotdot-z_uw*theta_dotdot)) #-(x_dot-z_uw*theta_dot))
         #   dF_i            = Froude_Krylov + Hydromassforce

         #   dF_d       = self.Cd * 0.5 * self.rho_seawater * pi * self.D_spar *\
         #   (u_wave_x-(x_dot- z_uw*theta_dot))*abs(u_wave_x-(x_dot-z_uw*theta_dot))

        dF_hydro =dF_i +dF_d

        return np.around(dF_hydro,6)

    def calc_hydrodynamic_forcing_z(self,zeta_array):

        if len(zeta_array) != len(self.A_vec):
            print('houston,problem')

        Fb_stat = self.rho_seawater*self.A_vec*(self.L_spar)
        Fb_dyn = self.rho_seawater *self.A_vec*(self.L_spar+zeta_array)

        Fb_vec = Fb_dyn - Fb_stat
        return Fb_vec

    def calc_A_spar_vec(self):
        """
```

```
        Function that calculates the area of a circle using a 'strip' method.
        If the amount of slices is even an extra slice is taken from the zero
        point too the sides.

        Then using pytagoras the area of a strip inside a circle can be estimated
        taking the rectangular area and then removing the difference between
        y1 and y2.


        Returns
        -------
        array
            Array of all the areas inside the circle.

        """
        dx = abs(self.D_spar_vec[0]-self.D_spar_vec[1])
        D  = self.D_spar
        self.Ai = []

        #Determening even amount of slices
        if self.checkifeven(self.D_spar_vec):
            D_half  = np.array_split(self.D_spar_vec)[0]
            D_half  = np.append(D_half,0)
        else:
            D_half  = np.array_split(self.D_spar_vec,2)[0]


        #loop to create the area
        for i in reversed(range(len(D_half)-1)):
            dx = abs(D_half[i]-D_half[i+1])
            x1 = D_half[i+1]
            x2 = D_half[i]

            y1 = sqrt((0.5*D)**2 - x1**2)
            y2 = sqrt((0.5*D)**2 - x2**2)

            self.Ai.append(2*y1*dx - dx*(y1-y2))

        Ai_1 = self.Ai[::-1]
        self.A_vec = [*Ai_1, *self.Ai]

        return np.array(self.A_vec)

    def checkifeven(self,anyvector):
        """
        Function that checks if list or array has even amounts of entries

        Parameters
        ----------
        anyvector : Array or List
            The array or list that you want to consider

        Returns
        -------
        bool
            truth for even, false for uneven.

        """
        if (len(anyvector) % 2) ==0 :
            return True
        else: return False

    def gen_C_visc_morisons(self,Hs,k):
        '''
        Function based off of a analytical expression for damping coefficients
        not funcitonal, as it introduces negative damping in current state


        Parameters
        ----------
        Hs : Value
```

```
            Waveheihgt.
        k : Value
            Wave number.

        Returns
        -------
        array
            Damping matrix .

        '''
        hb = abs(self.full_system.spar.z_cm_full - self.L_spar)
        hT = abs(self.full_system.spar.z_cm_full)
        tau = 2/pi

        def A1(Z,k):
            return  1/k
        def A2(Z,k):
            return Z/k-1/(k**2)
        def A3(Z,k):
            return Z**2/2 - 2*Z/k**2 + 2/k**3

        def gen_Cxx(Z,k,Afunc):
            sigma = sqrt(self.g*k)
            C_xx = self.Cd*self.rho_seawater*self.D_spar*Hs*sigma*tau*Afunc(Z,k)*e**(k*Z)
            return C_xx

        C11 = abs(gen_Cxx(hT,k,A1)-gen_Cxx(-hb,k,A1))
        C51 = abs(gen_Cxx(hT,k,A2)-gen_Cxx(-hb,k,A2))
        C55 = abs(gen_Cxx(hT,k,A3)-gen_Cxx(-hb,k,A3))

        C_matrix = np.array([[C11,0,C51],
                                [0,0,0],
                                [C51,0,C55]])
        return C_matrix

    def gen_zbounds(self,z_uw):
        halfstep = round_it(((z_uw[2]-z_uw[1])/2),5)
        z_up_bound = z_uw+halfstep
        z_up_bound[-1] = 0
        z_down_bound =z_uw - halfstep
        z_down_bound[0] = z_uw[0]

        return z_down_bound,z_up_bound
```

D.7.2. Aerodynamics class

```
# -*- coding: utf-8 -*-
"""
Created on Mon May 30 20:10:52 2022

@author: TORSPI
"""
from math import exp, pi
import numpy as np
from Math.Significance import round_it
class Aerodynamics_Class():
    def __init__(self,V_rated,D_rotor,V_wind):
        self.V_rated = V_rated
        self.D_rotor = D_rotor
        self.A_rotor = pi*(0.5*D_rotor)**2
        self.C_T0     = 0.81
        self.rho_air = 1.225
        self.a = 0.5
        self.b = 0.65

        self.plot = 0
        if self.plot ==1 :
            V_array = np.linspace(0,20,100)
            self.plot_CT_curve(V_array)
            self.plot_T_curve(V_array)
```

```
    def calc_CT(self, V_rel) :
        """
        Function that determines CT (thrust coefficient) for the relative wind
        speed. Is used in combination with a reduction factor to account for
        spatial variation of turbulence across the rotor

        Parameters
        ----------
        V_rel : Value
            Relative windspeed of the hub vs the 10 minute average.

        Returns
        -------
        C_T : Value
            Thrust Coefficient.

        """
        if V_rel <= self.V_rated :
            C_T = self.C_T0
        else: C_T = self.C_T0*exp(-self.a*(V_rel-self.V_rated)**self.b)

        return C_T

    def calc_C_T_10(self,V_10_min):
        """
        Function that determines CT from the 10 minute average.
        This makes up the biggest portion of the wind forcing


        Parameters
        ----------
        V_10_min : Value
            10 minute average wind speed .

        Returns
        -------
        C_T_10 : Value
            Thrust coefficient .
        f_red : Value
            Reduction factor .

        """
        if V_10_min <= self.V_rated :
            C_T_10 = self.C_T0
            f_red  = 0.54
        else:
            C_T_10 = self.C_T0 * exp(-self.a*(V_10_min-self.V_rated)**self.b)
            f_red  = 0.54+ 0.027*(V_10_min - self.V_rated)

        return C_T_10, f_red

    def calc_F_wind(self,V_rel,V_10_min):
        C_T = self.calc_CT(V_rel)
        C_T_10, f_red = self.calc_C_T_10(V_10_min)

        F_wind_mean = 0.5*self.rho_air*self.A_rotor*C_T_10*V_10_min**2
        F_wind_red  = 0.5*self.rho_air*self.A_rotor*C_T*V_rel*abs(V_rel)
        F_wind      = F_wind_mean +f_red*(F_wind_red-F_wind_mean)

        return F_wind

    def calc_dF_wind(self,V_rel,V_10_min):
        C_T = self.calc_CT(V_rel)
        C_T_10, f_red = self.calc_C_T_10(V_10_min)

        F_wind_mean = 0.5*self.rho_air*self.A_rotor*C_T_10*V_10_min**2
        F_wind_red  = 0.5*self.rho_air*self.A_rotor*C_T*V_rel*abs(V_rel)
        return f_red*(F_wind_red-F_wind_mean)

    def calc_F_trust_J103(self,V_wind):
        C_T,_ = self.calc_C_T_10(V_wind)
```

```
        return 0.5*self.rho_air*C_T*self.A_rotor*V_wind**2


    def genC_aero_pt(self,V_rel,V_10_min,full_system):
        '''
        Calculates aerodynanic damping array
        based off of a analytical expression for damping coefficients
        not funcitonal, as it introduces negative damping in current state
        Has been adjusted to work off on thrusst force. (original is an integral
                                                        over the blade)

        Parameters
        ----------
        V_rel : Value
            Relative Wind Speed.
        V_10_min : Value
            Wind speed.
        full_system : instance
            turbine, tower, spar.

        Returns
        -------
        array
            Damping Matrix.

        '''
        #length from rotor to spar mass centre
        hR = round_it(abs(full_system.spar.z_cm_full),5)

        dT = self.calc_dF_wind(V_rel,V_10_min)
        cxx = abs(dT/V_10_min)

        c00 = 3/2 * abs(dT/V_10_min) * 1/3 *(0.5*self.D_rotor)**3

        return np.array([[cxx,0,0],
                          [0,0,0],
                          [hR * cxx,0,c00]],dtype=float)

    def genC_aero_pp(self,V_rel,V_10_min,full_system):
        '''
        Calculates aerodynanic damping array
        based off of a analytical expression for damping coefficients
        not funcitonal, as it introduces negative damping in current state
        Has been adjusted to work off on thrusst force. (original is an integral
                                                        over the blade)

        Parameters
        ----------
        V_rel : Value
            Relative Wind Speed.
        V_10_min : Value
            Wind speed.
        full_system : instance
            turbine, tower, spar.

        Returns
        -------
        array
            Damping Matrix.

        '''
        #length from rotor to spar mass centre
        hR = round_it(abs(full_system.z_hub),5)
        hT = round_it(abs(full_system.z_hub),5)

        dT = self.calc_dF_wind(V_rel,V_10_min)
        cxU1 = abs(dT/V_10_min)
        cxU5 = hR*abs(dT/V_10_min)
        c0U5 = 1/2*abs(dT/V_10_min)*(full_system.turbine.L_blade**3)/3


```

```
        C_matrix_pp =  np.array([[cxU1,          0,  cxU5],
                                 [0,             0,  0],
                                 [hR * cxU1,     0,  c0U5+ hR*cxU5]],dtype=float)
        return C_matrix_pp

    def genC_matrix(self,V_rel,V_10_min,full_system):
        C_matrix_pt = self.genC_aero_pt(V_rel,V_10_min,full_system)
        C_matrix_pp = self.genC_aero_pp(V_rel,V_10_min,full_system)
       # print(C_matrix_pt +C_matrix_pp)
        #print('')
        return C_matrix_pt + C_matrix_pp

    def plot_CT_curve(self,V_array):

        CT_list = np.zeros(len(V_array))
        for i in range(len(V_array)):
            CT_list[i] = self.calc_CT(V_array[i])
        import matplotlib.pyplot as plt
        fig, ax = plt.subplots()
        plt.plot(V_array,CT_list)
        plt.title("Thrust Coefficent Curve")
        plt.xlabel('Wind Speed [m/s]')
        plt.ylabel('Thrust Coefficient')

    def plot_T_curve(self,V_array):
        F_list = np.zeros(len(V_array))
        for i in range(len(V_array)):
            F_list[i] = self.calc_F_wind(V_array[i],V_array[i])
        import matplotlib.pyplot as plt
        fig,ax = plt.subplots()
        plt.plot(V_array,F_list)
        plt.title("Thrust Curve")
        plt.xlabel('Wind Speed [m/s]')
        plt.ylabel('Thrust [N]')
```

D.8. State class

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 24 12:19:29 2022

@author: TORSPI
"""
import numpy as np
from math import e, sqrt, pi , log10
from scipy.integrate import solve_ivp, simps
from scipy.linalg import inv
from TimeDomain_Response.HydrodynamicsClass import Hydrodynamics_Class
from TimeDomain_Response.AerodynamicsClass import Aerodynamics_Class
from Utilities.Utilities import find_nearest_index, cut_profile2size
from Math.Significance import round_it
class State_Class():
    '''
    Class containing all states that can be considered in the ODE function.
    Depending on how the wind/waves are considered it makes sense to use different states
    (steady wind = constant wind ) -> (unsteady wind = irregular wind)
    (steady wave = regular waves ) -> (unsteady wind = irregular waves)

    '''
    def __init__(self,full_system,wavekinematics,windkinematics,environment,dt,Tdur):
        self.xtopspeed = 100
        self.thetatopspeed = 20

        self.full_system = full_system
        self.MA_matrix  = full_system.M_matrix+full_system.A_matrix
        self.K_matrix   = full_system.K_matrix
        self.F_matrix   = np.array([[0],[0],[0]])
        #self.B_matrix   = np.array([[565855,0,68042.4],
                               #   [0,0,0],
```

```
                              #    [68042.4,0,8e10]])

        self.H          = environment.H   #significant Wave Heigh
        self.T          = environment.T   #significant Wave Period
        self.h          = environment.h
        self.z_uw       = environment.z_uw
        self.V_wind_10  = environment.U_mean
        self.TI         = environment.TI

        self.rho_seawater = full_system.rho_seawater


        self.dt = dt
        self.Tdur = Tdur
        self.t      = []
        self.t_eval = (np.arange(0,self.Tdur+self.dt,self.dt)).tolist()


        self.g      = 9.81


        self.D_spar         = full_system.D_spar
        self.D_spar_vec     = full_system.spar.D_vec
        self.A_spar_vec     = full_system.A_spar_vec
        self.L_spar         = full_system.L_spar
        self.t_spar         = full_system.t_spar
        self.rho_seawater   = full_system.rho_seawater # kg/m^3
        self.z_hub          = full_system.z_hub
        self.D_rotor        = full_system.D_rotor
        self.V_rated        = full_system.V_rated

        #-- Regular Wave Speeds
        self.wavekinematics = wavekinematics
        self.uwave_x    = wavekinematics.uwave_x
        self.uwave_x_dot= wavekinematics.uwave_x_dot
        self.uwave_z    = wavekinematics.uwave_z_dot

        self.zeta_matrix = wavekinematics.calc_zeta_matrix(self.H,self.T,self.D_spar_vec)

        #-- Irregular Wave Speed
        self.uwave_x_irr     = wavekinematics.u_irr_matrix.T
        self.uwave_x_dot_irr = wavekinematics.du_irr_matrix.T
        self.zeta_matrix_irr = wavekinematics.zeta_irr_fft

        #-- Irregular Wind Speed
        self.uwind_x_irr = windkinematics.V_wind_irr
        #--- Hyrdodynamic Loading
        self.Hydro = Hydrodynamics_Class(self.h,self.z_uw,full_system)


        #- regular Aerodynamics
        self.Aero = Aerodynamics_Class(self.V_rated,self.D_rotor,self.V_wind_10)


        self.printswitch = 0
        self.limittopspeed = 0
        self.i = 0
        # Damping Matrix
        uwave_x_use,z_uw    = cut_profile2size(self.L_spar,self.uwave_x[0],self.z_uw,0)
        self.B_matrix    = self.gen_C_matrix_visc(wavekinematics.k_reg,z_uw)
        # irregular

        #---- Temporarily stored values
        self.x_dotdot = 0
        self.theta_dotdot=0

    def choose_dqdt(self,waves=0 ,wind=0):
        """
        Function that chooses what state function will be used.
        Switches: 0 - off
                  1 - steady (regular)
```

```
                    2 - unsteady (irregular)

        Parameters
        ----------
        waves : Value, optional
            Wave Switch. The default is 0.
        wind : Value, optional
            Wind Switch. The default is 0.

        Returns
        -------
        function
            state function dqdt for the ODE solver.

        """


        if waves == 0 and wind == 0 :
            return self.dqdt_noforcing
        if waves == 0 and wind == 1:
            return self.dqdt_steady_wind_response
        if waves == 1 and wind == 0 :
            return self.dqdt_regular_wave_response
        if waves == 1 and wind ==2:
            return self.dqdt_unsteady_Wi_steady_Wa
        if waves ==1 and wind ==1 :
            return self.dqdt_steady_WiWA
        if waves == 2 and wind ==1 :
            return self.dqdt_unsteady_Wa_steady_Wi
        if waves ==2 and wind ==2:
            return self.dqdt_unsteady_WaWi
        else:
            print('no such state function has been made using no wind and wave')
            return self.dqdt_noforcing

    def dqdt_noforcing(self,t,q):
        """
        state where there is no hydrodynamic or aerodynamic forcing.

        Parameters
        ----------
        t : float
            time .
        q : array
            contains all of the states for the ODE function.

        Returns
        -------
        dqdt : array
            Derived state.

        """
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)
        self.F_matrix = np.array([[0],[0],[0]])
        C_matrix = self.B_matrix
        #Damping Matrix
        V_rel = -self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_aero_pp(V_rel,0.01,self.full_system) #only platform!
        C_matrix = C_aero
        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)
        return dqdt

    def dqdt_regular_wave_response(self,t,q):

        """
        state where there is only regular waves

        Parameters
        ----------
        t : float
            time .
```

```
        q : array
            contains all of the states for the ODE function.

        Returns
        -------
        dqdt : array
            Derived state.
            """
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)

        #-- index into wavespeeds --
        indexT = np.round((t-self.t_eval[0])/self.dt)-1
        uwave_x_use          = self.uwave_x[indexT.astype(int)]
        uwave_x_dot_use      = self.uwave_x_dot[indexT.astype(int)]
        #-- cut off wavespeed under the spar
        uwave_x_use,z_uw     = cut_profile2size(self.L_spar,uwave_x_use,self.z_uw,z)
        if len(z_uw) < 3:
            print('z_uw too small')
        uwave_x_dot_use, _   = cut_profile2size(self.L_spar,uwave_x_dot_use,self.z_uw,z)
        #-- index into waveheight
        zeta_array_use   = self.zeta_matrix[:,indexT.astype(int)]

        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot,z_uw)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
        #----------------Pitch force (moment)
        M_hydro   = simps(dF_hydro*z_uw,z_uw)
        #print(M_hydro)

        #Damping Matrix
        V_rel = -self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel,0.01,self.full_system)
        C_matrix = self.B_matrix+C_aero


        if self.printswitch == 1:
            print('x_dot =',x_dot, 'theta_dot = ', theta_dot)
            print('C_matrix = ', C_matrix)
            print('V_rel = ',V_rel)
           # print('t = ', t,'x = ', x,'z = ', z,'theta = ', theta )
           # print('wave force = ', simps(dF_hydro,z_uw))


        self.F_matrix[0] = round_it(simps(dF_hydro,z_uw),6)
        self.F_matrix[1] = sum(dF_heave)
        self.F_matrix[2] = M_hydro

        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)

        return dqdt

    def dqdt_steady_wind_response(self,t,q):
        ''' Used specifically for storm heeling calculation.
        Note: caclulation of F_wind_x is different here, as the turbine
        is suposedly not spinning.
        writen in accordance with OS-J103 (1.1.11)
        '''
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)

        #--Wind Forcing
        F_wind_x = self.Aero.calc_F_trust_J103(self.V_wind_10)
        #print(F_wind_x)
        M_wind_x = F_wind_x*self.z_hub

        #damping matrix
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel, self.V_wind_10,self.full_system)
        C_matrix = C_aero
```

```


        self.F_matrix[0] = round_it(F_wind_x,6)
        self.F_matrix[1] = 0
        self.F_matrix[2] = M_wind_x


        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)
        return dqdt


    def dqdt_steady_WiWA(self,t,q):
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)
        if self.printswitch:
            if t >105:
                print('time to check')
                pass
        if self.limittopspeed:
            x_dot = self.top_speed(x_dot,self.xtopspeed)
            theta_dot = self.top_speed(theta_dot,self.thetatopspeed)

        #-- index into wavespeeds --
        indexT = np.round((t-self.t_eval[0])/self.dt)-1
        uwave_x_use         = self.uwave_x[indexT.astype(int)]
        uwave_x_dot_use     = self.uwave_x_dot[indexT.astype(int)]
        #-- cut off wavespeed under the spar
        uwave_x_use,z_uw    = cut_profile2size(self.L_spar,uwave_x_use,self.z_uw,z)
        uwave_x_dot_use, _  = cut_profile2size(self.L_spar,uwave_x_dot_use,self.z_uw,z)

        #-- index into waveheight
        zeta_array_use  = self.zeta_matrix[:,indexT.astype(int)]

        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot,z_uw)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
        #----------------Pitch force (moment)
        M_hydro  = simps(dF_hydro*z_uw,z_uw)

        #--Wind Forcing
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        F_wind_x = self.Aero.calc_F_wind(V_rel,self.V_wind_10)
        M_wind_x = F_wind_x*self.z_hub


        #damping matrix
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel, self.V_wind_10,self.full_system)
        C_matrix = abs(self.B_matrix+C_aero)
        if self.printswitch:
            if any(np.linalg.eigh(C_matrix)[0]<0):
                print('unstable damping')
                print(C_matrix)
            else:
                print('FINE')

        self.F_matrix[0] = simps(dF_hydro,z_uw)+F_wind_x
        self.F_matrix[1] = sum(dF_heave)
        self.F_matrix[2] = M_hydro + M_wind_x

        if self.printswitch:
            print('t = ', t,'x = ', x,'z = ', z,'theta = ', theta )
            print('wave force = ', simps(dF_hydro,z_uw))
            print('Wind Force = ', F_wind_x)
            print('wind moment = ', M_wind_x, 'wave moment = ', M_hydro)
            print('t = ', t,'x_dot = ', x_dot,'theta_dot = ', theta_dot )

        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)
        return dqdt
```

```
    def dqdt_unsteady_Wa_steady_Wi(self,t,q):
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)

        #-- index into wavespeeds --
        indexT = np.round((t-self.t_eval[0])/self.dt)-1
        uwave_x_use          = self.uwave_x_irr[indexT.astype(int)]
        uwave_x_dot_use      = self.uwave_x_dot_irr[indexT.astype(int)]
        #-- cut off wavespeed under the spar
        uwave_x_use,z_uw     = cut_profile2size(self.L_spar,uwave_x_use,self.z_uw,z)
        uwave_x_dot_use, _   = cut_profile2size(self.L_spar,uwave_x_dot_use,self.z_uw,z)

        #-- index into waveheight
        zeta_array_use  = self.zeta_matrix[:,indexT.astype(int)]


        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot,z_uw)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
        #----------------Pitch force (moment)
        M_hydro  = simps(dF_hydro*z_uw,z_uw)

        #--Wind Forcing
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        F_wind_x = self.Aero.calc_F_wind(V_rel,self.V_wind_10)
        M_wind_x = F_wind_x*self.z_hub
        #damping matrix

        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel, self.V_wind_10,self.full_system)
        C_matrix = self.B_matrix+C_aero

        if self.printswitch:
            print('Hydro momen is', M_hydro)
            self.i = self.i+1
            print(self.i)

        self.F_matrix[0] = round_it(simps(dF_hydro,z_uw)+F_wind_x,6)
        self.F_matrix[1] = sum(dF_heave)
        self.F_matrix[2] = M_hydro + M_wind_x


        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)



        return dqdt

    def dqdt_unsteady_Wi_steady_Wa(self,t,q):
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)

        #-- index into wavespeeds --
        indexT = np.round((t-self.t_eval[0])/self.dt)-1
        uwave_x_use          = self.uwave_x[indexT.astype(int)]
        uwave_x_dot_use      = self.uwave_x_dot[indexT.astype(int)]
        #-- cut off wavespeed under the spar
        uwave_x_use,z_uw     = cut_profile2size(self.L_spar,uwave_x_use,self.z_uw,z)
        uwave_x_dot_use, _   = cut_profile2size(self.L_spar,uwave_x_dot_use,self.z_uw,z)

        #-- index into waveheight
        zeta_array_use  = self.zeta_matrix[:,indexT.astype(int)]

        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot,z_uw)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
```

```
        #----------------Pitch force (moment)
        M_hydro  = simps(dF_hydro*z_uw,z_uw)

        #--Wind Forcing
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        F_wind_x = self.Aero.calc_F_wind(V_rel,self.V_wind_10)
        M_wind_x = F_wind_x*self.z_hub
        #damping matrix

        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel, self.V_wind_10,self.full_system)
        C_matrix = self.B_matrix+C_aero

        if self.printswitch:
            print('Hydro momen is', M_hydro)
            self.i = self.i+1
            print(self.i)

        self.F_matrix[0] = round_it(simps(dF_hydro,z_uw)+F_wind_x,6)
        self.F_matrix[1] = sum(dF_heave)
        self.F_matrix[2] = M_hydro + M_wind_x


        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)


    def dqdt_unsteady_WaWi(self,t,q):
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(q)


        #-- index into wavespeeds --
        indexT = np.round((t-self.t_eval[1])/self.dt)-1
        uwave_x_use         = self.uwave_x_irr[indexT.astype(int)]
        uwave_x_dot_use     = self.uwave_x_dot_irr[indexT.astype(int)]
        #-- cut off wavespeed under the spar
        uwave_x_use,z_uw    = cut_profile2size(self.L_spar,uwave_x_use,self.z_uw,z)
        uwave_x_dot_use, _  = cut_profile2size(self.L_spar,uwave_x_dot_use,self.z_uw,z)

        #-- index into waveheight
        zeta_array_use  = self.zeta_matrix[:,indexT.astype(int)]

        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot,z_uw)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
        #----------------Pitch force (moment)
        M_hydro  = simps(dF_hydro*z_uw,z_uw)

        #--Wind Forcing
        V_wind = self.uwind_x_irr[indexT.astype(int)]
        V_rel =  V_wind-self.z_hub*theta_dot+x_dot
        F_wind_x = round_it(self.Aero.calc_F_wind(V_rel,self.V_wind_10),5)
        M_wind_x = F_wind_x*self.z_hub

        #damping matrix
        V_rel = self.V_wind_10-self.z_hub*theta_dot+x_dot
        C_aero = self.Aero.genC_matrix(V_rel, self.V_wind_10,self.full_system)
        C_matrix = self.B_matrix+C_aero


        self.F_matrix[0] = round_it(simps(dF_hydro,z_uw)+F_wind_x,6)
        self.F_matrix[1] = sum(dF_heave) #+F_wind_x*theta
        self.F_matrix[2] = M_hydro + M_wind_x


        dqdt = self.setup_dqdt(x,z,theta,x_dot,z_dot,theta_dot,self.F_matrix,C_matrix)

        if self.printswitch:
            print('t = ', t,'x = ', x,'z = ', z,'theta = ', theta )
```

```python
            print('wave force = ', simps(dF_hydro,z_uw))
            print('Wind Force = ', F_wind_x)
            print('wind moment = ', M_wind_x, 'wave moment = ', M_hydro)
            print('t = ', t,'x_dot = ', x_dot,'theta_dot = ', theta_dot )
            #print(self.F_matrix)
            #print('')

        return dqdt


    def unpack_q(self,q):
        x           = q[0]
        z           = q[1]
        theta       = q[2]

        x_dot       = q[3]
        z_dot       = q[4]
        theta_dot   = q[5]
        return x,z,theta, x_dot,z_dot,theta_dot

    def setup_dqdt(self,x,z,theta,x_dot,z_dot,theta_dot,F_matrix,C_matrix):
        dqdt = [0]*6
        dqdt[0] = x_dot
        dqdt[1] = z_dot
        dqdt[2] = theta_dot
        eqsolve = inv(self.MA_matrix).dot\
            (F_matrix- (C_matrix.dot(np.array([[x_dot],[z_dot],[theta_dot]]))+self.K_matrix.
                dot(np.array([[x],[z],[theta]]))))

        dqdt[3] = eqsolve[0]
        dqdt[4] = eqsolve[1]
        dqdt[5] = eqsolve[2]

        return dqdt

    def make_F_matrix(self,keyword='whatforcingyouuse'):
        keyword=keyword.upper()
        keyword=keyword.replace(' ','')

        if keyword == 'NOFORCING':
            F_matrix = np.array([[0],[0],[0]])
            return F_matrix
        elif keyword == 'WAVEREGULAR':
            return F_matrix




    def choose_wind_wave_protocol(self,waves,wind,wavekinematics,windkinematics):
        if waves ==1 :
            #-- Regular Wave Speeds
            self.uwave_x     = wavekinematics.uwave_x
            self.uwave_x_dot= wavekinematics.uwave_x_dot
            self.uwave_z     = wavekinematics.uwave_z_dot
            return self.uwave_x,self.uwave_x_dot,self.uwave_z
        if waves == 2:
            #-- Irregular Wave Speed
            self.uwave_x_irr     = wavekinematics.u_irr_matrix.T
            self.uwave_x_dot_irr = wavekinematics.du_irr_matrix.T

    def gen_C_matrix_visc(self,k,z_uw):
        Cd  = self.Hydro.Cd
        rho = self.full_system.rho_seawater
        D   = self.full_system.D_spar
        H   = self.H
        sigma = sqrt(self.g*k)
        tau = 2/pi

        def A1(k,Z) :
            return 1/k
        def A2(k,Z):
```

```
            return Z/k - 1/(k**2)
        def A3(k,Z):
            return (Z**2)/k - (2*Z)/k**2 + 2*(k**3)

        def gen_Cxx(k,Z,A):
            return Cd*rho*D*H*sigma*tau*A(k,Z)*e**(k*Z)

        dC11 = np.zeros(len(z_uw))
        dC15 = np.zeros(len(z_uw))
        dC55= np.zeros(len(z_uw))
        for i in range(len(z_uw)) :
            dC11[i] = gen_Cxx(k,z_uw[i],A1)
            dC15[i] = gen_Cxx(k,z_uw[i],A2)
            dC55[i] = gen_Cxx(k,z_uw[i],A3)

        C11 = sum(dC11)
        C15 = sum(dC15)
        C55 = sum(dC55)
        C_matrix = np.array([[C11*10, 0, C15],
                             [0,    0,    0],
                             [C15, 0,    C55]],dtype=float)

        return C_matrix

    def top_speed(self,x_dot,topspeed) :
        if x_dot > topspeed :
            if self.printswitch:
                print('over top speed')
            return topspeed
        elif x_dot<-topspeed:
            if self.printswitch:
                print('under topspeed')
            return -topspeed
        else:return x_dot
```

D.9. Fatigue calculations

D.9.1. All environments import

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov  8 12:16:36 2022

@author: Giles
"""
from threading import Thread
import pandas as pd
import numpy as np
from TimeDomain_Response.EnviromentClass import Enviroment_Class
from TimeDomain_Response.TimeDomainSimulationClass import Time_Domain_Simulation_Class
from Utilities.ThreadWithReturnValueClass import ThreadWithReturnValue
from FatigueCalculations.FatigueClass import Fatigue_Class


class fatigue_environmental_import_class():
    def __init__(self,environment,full_system,SN_func,space = 'lin'):
         self.full_system = full_system
         self.SN_func = SN_func
         #import tower data
         data = pd.read_excel('/Users/Giles/Desktop/Desktop - 'Torstens MacBook Pro/Code2.0/
             ThesisPackages/FatigueCalculations/Fatigue_Cond.xlsx')
         self.U_mean_list             = pd.DataFrame(data,columns=['Uw'])        #heights(
             transition piece and tower)
         self.T_list                  = pd.DataFrame(data,columns=['T'])         #outer
             Diameter
         self.Hs_list                 = pd.DataFrame(data,columns=['Hs'])        #tower
             thickness
         self.freqoccur               = pd.DataFrame(data,columns=['freqoccur'])

```

```
         self.h = environment.h
         self.TI = environment.TI
         self.space = space
         self.logstep = 50
         self.environmentlist = self.gen_environment_list()
         self.q0 = [0,0,0,0,0,0]

         self.eq_stress_list,self.eq_cycles_list,self.TD_sim_list \
             = self.calc_fatigue_per_condition()

         self.eq_stress_list_scaled, self.eq_cycles_list_scaled = self.scale_condition_damage()

         self.damage,self.stress_life,self.cycle_life = self.calc_damage_life(SN_func)


    def calc_damage_life2(self,SN_func):
        stress_scaled_array = np.array(self.eq_stress_list_scaled)
        cycles_scaled_array = np.array(self.eq_cycles_list_scaled)

        damage_array = np.zeros(len(stress_scaled_array))
        for i in range(len(stress_scaled_array)):
            stress_range_i = stress_scaled_array[i]
            n_i            = cycles_scaled_array[i]
            N_i            = SN_func(stress_range_i)
            D_i            = n_i/N_i
            damage_array[i]    = D_i

        return damage_array[i]


    def calc_damage_life(self,SN_func):
        '''
        Function takes weighted average of the stress over the cycles to get the
        lifetime stress and cycle

        Parameters
        ----------
        SN_func : Function
            SN curve funciton rewritten to take cycles as input and output the
            amount of stress at that number of cycles level.

        Returns
        -------
        stress_life : TYPE
            DESCRIPTION.
        cycle_life : TYPE
            DESCRIPTION.

        '''
        stress_scaled_array = np.array(self.eq_stress_list_scaled)
        cycles_scaled_array = np.array(self.eq_cycles_list_scaled)
        #check if it was the right idea

        stress_life = np.sum(stress_scaled_array*cycles_scaled_array,0)/np.sum(cycles_scaled_array,0)
        cycle_life = np.sum(cycles_scaled_array,0)

        stress_SN = SN_func(cycle_life)
        damage = stress_life/stress_SN

        #defo the right idea


        return damage,stress_life,cycle_life



    def scale_condition_damage(self):
        eq_stress_list_scaled  = []
        eq_cycles_list_scaled = []
```

```
        for i in range(len(self.environmentlist)):
            eq_stress_list_scaled.append(self.freqoccur.iat[i,0]*self.eq_stress_list[i])
            eq_cycles_list_scaled.append(self.freqoccur.iat[i,0]*self.eq_cycles_list[i])

        return eq_stress_list_scaled, eq_cycles_list_scaled



    def calc_fatigue_per_condition(self):
        '''
        Function that returns the lifetime fatigue of each condition.
        Result should be modified by the frequency of occurence. (miners rule
                                                                  means its a
                                                                  linear relationship.)

        Returns
        -------
        eq_stress_list : TYPE
            DESCRIPTION.
        eq_cycles_list : TYPE
            DESCRIPTION.
        TD_sim_list : TYPE
            DESCRIPTION.

        '''
        TD_sim_list = []
        eq_stress_list = []
        eq_cycles_list = []
        for i in range(len(self.environmentlist)):
            TD = Time_Domain_Simulation_Class(self.q0,self.full_system,self.environmentlist[i
                ],1,200,0.5,1,2,2,0)
            TD_sim_list.append(TD)

            Fatigue_i = Fatigue_Class(TD)
            eq_stress_list.append(Fatigue_i.eq_stress)
            eq_cycles_list.append(Fatigue_i.eq_cycles)


        return eq_stress_list,eq_cycles_list,TD_sim_list


    def gen_environment_list(self):
        '''
        Function that initializes all environemntal conditions needed for a fatigue analysis
        based off the environmental condition specified in excel file

        Returns
        -------
        environment_list : list
            list of isntances of the environmental conditions.

        '''
        environment_list = []
        for i in range(len(self.U_mean_list)):
            Hs = self.Hs_list.iat[i,0]
            T  = self.T_list.iat[i,0]
            U  = self.U_mean_list.iat[i,0]

            environment_i = Enviroment_Class(self.h,Hs,T,U,self.TI,self.space,self.logstep)

            environment_list.append(environment_i)

        return environment_list

    def get_TD_sim(self,environment):
        TD_sim = Time_Domain_Simulation_Class(self.q0,self.full_system,environment
            ,1,3600,0.5,1,2,2,600)
        return TD_sim
```

D.9.2. Fatigue Class

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np

from Utilities.Utilities import indices,JonSwap, Kaimal
#from Kinematics.WaveKinematicsClass import Wave_Kinematics_Class
#from Kinematics.AerodynamicsClass import Aerodynamics_Class
from scipy.integrate import solve_ivp, simps
from FatigueCalculations.FatigueLoadingClass import Fatigue_Loading_Class
from FatigueCalculations.FatigueRainflowClass import Fatigue_Rainflow_Class

class Fatigue_Class():
    """" This class runs the fatigue calculations for a single environmental
    condition, as such it requires a TD_simulation run as an input


    """


    def __init__(self,TD_simulation,m=4,n_eq=1*10**6,years = 50):
        #import scatter DNV
        #data_dnv = pd.read_csv(r'FatigueCalculations/scatter_moderate.csv')
        #data_dnv = data_dnv.set_index('Unnamed: 0')
        #data_dnv = data_dnv/data_dnv.values.sum()

        #import lumped scatter DTU
        #LumpScat = pd.read_excel("FatigueCalculations/Scatter_Diagram_ass3.xlsx")
        #LumpScat.columns = [c.replace(' ','_') for c in LumpScat.columns]
        #LumpScat.TI_normal = LumpScat.TI_normal/100
        #LumpScat.TI_extreem = LumpScat.TI_extreem/100

        #self.WindSpeed  = LumpScat.Speed
        #self.TI_normal  = LumpScat.TI_normal
        #self.TI_extreem = LumpScat.TI_extreem
        #self.Hs         = LumpScat.Hs
        #self.Tp         = LumpScat.Tp_

        self.m = m
        self.n_eq = n_eq
        self.years = years
        Hsmj  = self.calc_Hs_eq

        #---Inherit from TD
        self.spar       = TD_simulation.spar
        self.D_rotor    = TD_simulation.D_rotor
        self.D_spar     = TD_simulation.D_spar
        self.f_matrix   = TD_simulation.f_matrix
        self.dt         = TD_simulation.dt
        self.t_eval     = TD_simulation.t_eval
        self.z_uw       = TD_simulation.z_uw
        self.sol        = TD_simulation.sol

        self.Tdur           = TD_simulation.Tdur
        self.t_60_index     = indices(self.t_eval, lambda x: x==60)[0]
        self.t_nontrans     = self.t_eval[self.t_60_index:]
        #--- Wave kinematics
        self.wavekinematics = TD_simulation.wavekinematics
        self.windkinematics = TD_simulation.windkinematics
        self.Hydro          = TD_simulation.Hydro
        self.stateclass     = TD_simulation.stateclass
        self.dqdt           = TD_simulation.dqdt

        self.fatigueloads = Fatigue_Loading_Class(self)
        phi               = self.fatigueloads.phi #bending stress
        tau               = self.fatigueloads.tau #shear stress

        #--- lifetime stress
        self.eq_stress,self.eq_cycles = self.calc_life_stress(phi)

```

```python
        return

    def eq_moment_UPWIND(self,scatter):
        for i in range(len(scatter)):
            Hs = self.Hs[i], Tp = self.Tp[i]
            windspeed = self.WindSpeed[i]
            I           = self.TI_normal[i]
            [Mdynamic] = self.fftmoment(self.t_eval,Hs,Tp,windspeed,I,340.2) #CREATE THIS
                FUNCITON

    def fftmoment(self,t_total,Hs,Tp,windspeed,I,l):
        Sjs,ajs = JonSwap(Hs,Tp,self.f_matrix)
        Skm,akm = Kaimal(windspeed,I,self.f_matrix,l)

        u_wave = self.wavekinmatics.u_irr_matrix[0]
        du_wave = self.wavekinematics.du_irr_matrix[0]
        V_time_series = self.windkinematics.V_wind_irr

        q0 = [0,0,0,0,0,0]

        sol = solve_ivp(self.dqdt,[0,self.Tdur+self.dt],q0,t_eval=self.t_eval)



    def calc_Hs_eq(self,scatterdiagram):
        """
        This function is a simplified damage equivalent approach taken from a
        TU delft repository "REDUCTION OF FATIGUE COMPUTATIONAL TIME FOR OFFSHORE
        WIND TURBINE JACKET FOUNDATIONS, it can lump scatter diagrams or a more
        holistic approach to environmental data.

        The function lumps the significant wave height into an equivelant wave height
        by taking the average of multiplied wave height and probabilities over
        the probability itself.

        This approach aims at the preseration of the wave period distribution
        while establishing an associated distributino of wave heights for all wave period
            classes.


        Parameters
        ----------
        scatterdiagram : Dataframe
            A scatter diagram that is read in. Where Collums represent Tp
            and the rows represent Hs

        Returns
        -------
        Hsmj : TYPE
            DESCRIPTION.

        """
        Hs = scatterdiagram.index.values
        scatterdiagram = scatterdiagram.to_numpy()

        Hsmj      = []
        pj = np.sum(scatterdiagram,axis=0)

        for j in range(len(scatterdiagram[0])): #Every Tp
            numerator = []
            for i in range(len(scatterdiagram)):#EVery HS
                pij = scatterdiagram[i,j]
                numerator.append(pij*Hs[i]**self.m)
            Hsmj.append((sum(numerator)/pj[j])**1/self.m)
        Hsmj = np.array(Hsmj)
        Hsmj[np.isnan(Hsmj)] = 0

        return Hsmj

    def calc_Tz_eq(self,scatterdiagram):
```

```
        Tp = scatterdiagram.columns.values
        Tp = Tp.astype(float)
        array = scatterdiagram.to_numpy()

        Tzni = []
        pi = np.sum(array,axis=1)

        for i in range(len(array)):
            numerator =[]
            for j in range(len(array[0])):
                pij = array[i,j]
                numerator.append((pij/Tp[j]))
            if pi[i] ==0:
                Tzni.append(0)
            else:
                Tzni.append((sum(numerator/pi[i]))**-1)
        Tzni =np.array(Tzni)
        Tzni[np.isnan(Tzni)]=0
        return Tzni

    def Calculate_Forcing_History(self):
        x,z,theta, x_dot,z_dot,theta_dot = self.unpack_q(self.sol.y)

        uwave_x_use     = self.wavekinematics.uwave_x
        uwave_x_dot_use = self.wavekinematics.uwave_x_dot
        #-- index into waveheight
        zeta_array_use  = self.wavekinematics.zeta_matrix

        #--Wave Forcing
        #---------------in x_direction
        dF_hydro =self.Hydro.calc_hydrodynamic_forcing_x(uwave_x_use,uwave_x_dot_use,x_dot,
            theta_dot)
        #----------------in z_direction
        dF_heave = self.Hydro.calc_hydrodynamic_forcing_z(zeta_array_use)
        #----------------Pitch force (moment)
        M_hydro  = simps(dF_hydro*self.z_uw,self.z_uw)

        #--Wind Forcing
        V_rel = self.V_wind_10-self.z_hub*theta_dot-x_dot
        F_wind_x = self.Aero.calc_F_wind(V_rel,self.V_wind_10)
        M_wind_x = F_wind_x*self.z_hub

        self.F_matrix[0] = simps(dF_hydro,self.z_uw)+F_wind_x
        self.F_matrix[1] = sum(dF_heave)
        self.F_matrix[2] = M_hydro + M_wind_x

        return self.F_matrix

    def calc_life_stress(self,signal):
        """
        Function that returns the lifetime equivelant stress at every weld location

        Parameters
        ----------
        signal : array
            Stress time history.

        Returns
        -------
        eq_life_stress : Array(1D)
            Array of all the lifetime stresses (for each weld).

        """
        eq_life_stress = np.zeros(len(signal))
        eq_cycles  = np.zeros(len(signal))

        for i in range(len(signal)):
            rainflow            = Fatigue_Rainflow_Class(self,signal[i],self.t_eval)
            rf_array            = rainflow.rf
            eq_life_stress[i],eq_cycles[i]  = Fatigue_Rainflow_Class.calc_eq_life_stress(
                rainflow,rf_array,self.years)
```

```
        return eq_life_stress, eq_cycles
```

D.9.3. Fatigue loading class

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Sep  7 10:06:49 2022

@author: Giles
"""
import numpy as np
from Utilities.Utilities import cut_profile2size, unpack_q, calc_ddx, find_nearest_index,
    find_length
from scipy.integrate import simps

class Fatigue_Loading_Class():
    """" This class is meant to be used to keep all the fatigue related loading
    calculations in one spot.

    """


    def __init__(self,fatigue_instance):
    #--- Importing the spar instance
        self.spar = fatigue_instance.spar
    #--- Importing the time domain solution
        self.sol = fatigue_instance.sol
        self.x,self.z,self.theta, self.x_dot,self.z_dot,self.theta_dot = unpack_q(
            fatigue_instance.sol.y)
    #--- Wave kinematics
        self.wavekinematics = fatigue_instance.wavekinematics
        self.windkinematics = fatigue_instance.windkinematics
    #--- Hyrdodynamic Loading
        self.z_uw  = fatigue_instance.z_uw
        self.Hydro = fatigue_instance.Hydro


        self.stateclass = fatigue_instance.stateclass
        self.dqdt = fatigue_instance.dqdt

        self.cut_hydro_force, self.lowest_depth, self.z_uw_cut = self.forcing4fatigue()
        self.lowest_idx = find_nearest_index(self.lowest_depth,self.z_uw)

        self.tau, self.phi = self.calc_max_stress_per_weld()

        return

    def calc_max_stress_per_weld(self):
        '''
        Calculates the max stress on the welds.
        (last entry is highest up (closest to the tower base)

        Returns
        -------
        tau_max : List
            Max shear Stress per weld.
        phi_max : TYPE
            Max bending stress per weld.

        '''
        phi_max = np.zeros((len(self.spar.weld_locations),find_length(self.cut_hydro_force)))
        tau_max = np.zeros((len(self.spar.weld_locations),find_length(self.cut_hydro_force)))

        for i in range(len(phi_max)):
            phi_iteration,tau_iteration = self.calc_internal_stresses(self.spar.
                weld_locations[i],self.spar.D/2)
            phi_max[i] = phi_iteration
            tau_max[i] = tau_iteration
```

```
        return tau_max,phi_max


    def calc_internal_stresses(self,weld_location,y):

        #getting accelerations -> F = m*ddx
        dd_x, z_double_dot, dd_theta = self.calc_acceleration_from_solution()
        #getting forcing
        forcing_x = self.cut_hydro_force

        #Calc partial mass
        #Ignores the ballast, sees spar as a hollow cylinder.
        dm = (self.spar.L - weld_location)/self.spar.L * self.spar.m_spar
        dIx = (self.spar.L - weld_location)/self.spar.L * self.spar.Ix

        #use weld location to recut hydro forcing
        cut_off = find_nearest_index(weld_location,self.z_uw_cut)
        forcing_x = self.cut_hydro_force[:cut_off]
        z_use     = self.z_uw_cut[:cut_off]


        #Forces in X  -> shear stress
        T = dm*dd_x - simps(forcing_x.T,z_use)
        tau = T/self.spar.A_cross_section

        #Moment around weld
        M = dIx*dd_x - simps(forcing_x.T*z_use,z_use)
        phi = M/(self.spar.Ix_area2/y)  #FIX dIx should instead be second moment of the area

        return tau, phi

        return

    def forcing4fatigue(self):
        L_spar = self.spar.L
        z_spar = self.sol.y[1]


        Hydro_Forcing = self.Hydro.calc_hydrodynamic_forcing_x(self.wavekinematics.uwave_x,
            self.wavekinematics.uwave_x_dot,self.sol.y[3],self.sol.y[5],self.z_uw)
        Lowest_Depth = np.amin(z_spar)-L_spar

        Hydro_Forcing_cut, z_uw_cut = cut_profile2size(L_spar,Hydro_Forcing,self.z_uw,np.amin
            (z_spar))

        return Hydro_Forcing_cut, Lowest_Depth ,z_uw_cut

    def calc_acceleration_from_solution(self):
        t = self.sol.t
        #addding a zero collumn to the solution
        position_velocity = np.array(self.sol.y)
        x,z,theta, x_dot,z_dot,theta_dot = unpack_q(position_velocity)

        x_dot       = np.insert(x_dot,0,0)
        z_dot       = np.insert(z_dot,0,0)
        theta_dot   = np.insert(theta_dot,0,0)
        t           = np.insert(t,0,0)

        x_double_dot        = calc_ddx(x_dot,t)
        z_double_dot        = calc_ddx(z_dot,t)
        theta_double_dot    = calc_ddx(theta_dot,t)

        return x_double_dot, z_double_dot, theta_double_dot
```

D.9.4. Fatigue rainflow class

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 14 16:33:38 2022
```

```

@author: Giles
"""
import numpy as np
import matplotlib.pyplot as plt
import rainflow
from Utilities.Utilities import find_nearest_index

class Fatigue_Rainflow_Class():
    def __init__(self,fatigue_class_instance,signal,time) :
        #--- Inherit from head instance
        self.m = fatigue_class_instance.m
        self.n_eq = fatigue_class_instance.n_eq

        self.time = time
        self.extremes, self.extreme_times = self.sig2ext(signal,time,0)
        self.rf = self.rainflow_extract(self.extremes)




    def sig2ext(self,sig,t=[1], plot=0 ):
        #finding the turning point
        sig = np.array(sig)
        turningpoint_list = self.turningpoints(sig)
        #-- Cutting off everything but the turning points
        extremes = sig[turningpoint_list]

        #--- The Same should be done with time
        if len(t) == 1 : #if no time was given, 1 second is presumed
            dt = np.array(range(0,len(sig)))
            extreme_times = dt[turningpoint_list]
        elif len(sig) == len(t): #if the time was given the same indexing applies
                dt = np.array(t)
                extreme_times = dt[turningpoint_list]
        else: raise ValueError("The time and signal are of different sizes")

        #-removing tripples
        triple_free_idx = self.remove_triples(extremes)
        extremes        = extremes[triple_free_idx]
        extreme_times   = extreme_times[triple_free_idx]

        #removing doubles
        double_free_idx = self.remove_doubles(extremes)
        extremes        = extremes[double_free_idx]
        extreme_times   = extreme_times[double_free_idx]

        #- re-check for turning points (we may have removed some)
        turningpoint_list = self.turningpoints(extremes)
        #-- Cutting off everything but the turning points
        extremes = extremes[turningpoint_list]
        extreme_times = extreme_times[turningpoint_list]

        if plot:
            self.plot_first50(dt,sig,extreme_times,extremes,20)

        return extremes, extreme_times


    def turningpoints(self,signal):
        """
        Function returns array of turning points.
        To maintain the same size array it is presumed that the first and last
        entry in the signal are also turningpoints.

        Parameters
        ----------
        signal : array
            series of signals (in this case loading or stress makes sense).

        Returns
```

```
        -------
        TYPE: Array
            Binary array to indicate if there is or isnt a turning point at that point.

        """
        dx = np.diff(signal)
        point_list = np.less_equal((dx[1:] * dx[:-1]), 0).astype(int)
        point_list = np.insert(point_list,0,1)
        point_list = np.insert(point_list,len(point_list),1)
        return point_list.astype(bool)


    def remove_triples(self,signal):
        dx = np.diff(signal)
        check1 = np.not_equal([dx[1:],dx[:-1]], 0)

        point_list = np.logical_and(check1[0],check1[1]).astype(int)
        point_list = np.insert(point_list,0,1)
        point_list = np.insert(point_list,len(point_list),1)
        return point_list.astype(bool)

    def remove_doubles(self,signal):
        point_list = np.not_equal(signal[1:],signal[:-1])
        point_list = np.insert(point_list,len(point_list),1)
        return point_list.astype(bool)

    def plot_first50(self,dt,sig,extreme_times,extremes,point=50):
        """
        This function plots the first 50 points of the 'cleaned' signal
        and also plots the same points on the unclean version. Matching up
        in final time stamp. This allows to visualize the effect of SIG2EXT

        The 50 is standar but can be adjusted to whichever number the user
        wants by changing the 'point' value

        Parameters
        ----------
        dt : Array
            timeseries.
        sig : Array
            unfiltered signal.
        extreme_times : Array
            Time stamps matching the filtered signal.
        extremes : Array
            Filtered signal from SIG2EXT .
        point : Value, optional
            How many points do you want plot?. The default is 50.

        Returns
        -------
        Plot
            2 plots of the same time span. Top is unfitered bottom is filtered.

        """
        idx = find_nearest_index(extreme_times[point],dt)
        plt.figure
        plt.subplot(211)
        plt.plot(dt[0:idx],sig[0:idx])
        plt.subplot(212)
        plt.plot(extreme_times[0:point],extremes[0:point])
        return plt.show()

    def rainflow_extract(self,signal):
        """
        Function that extracts the rainflow counter data and stores it in one
        numpy array. The rainflow counter works with lazy iterator which means
        the results normally would not be stored. Hence the need for this
        function

        Parameters
        ----------
```

```
        signal : 1D Array
            signal that goes through the rainflow counter.

        Returns
        -------
        rf : Array of 1D arrays
            Array containing range, mean, count, start and finish iterations
            for each cycle. ( in that order).

        """
        #Making empty arrays
        rng, mean, count  = np.empty(0), np.empty(0), np.empty(0)
        i_start, i_end   = np.empty(0), np.empty(0)

        for rng_i, mean_i, count_i ,i_start_i, i_end_i in rainflow.extract_cycles(signal):
            rng     = np.append(rng,rng_i)
            mean    = np.append(mean,mean_i)
            count   = np.append(count,count_i)
            i_start = np.append(i_start,i_start_i)
            i_end   = np.append(i_end,i_end_i)

        rf = np.array([rng,mean,count,i_start,i_end])
        return rf

    def scale_cycles2lifetime(self,cycles,cycle_time,years):
        years2seconds   = years*365*24*60*60
        scaling_factor  =  years2seconds/cycle_time
        scaled_cycle    = cycles * scaling_factor
        return scaled_cycle

    def calc_eq_life_stress(self,rf,years):
        cycles = rf[2]
        cycle_time = self.time[len(self.time)-1]
        scaled_cycles = self.scale_cycles2lifetime(cycles,cycle_time,years)


        equivelant_stress = np.sum(scaled_cycles*(rf[0]**self.m)/self.n_eq)**(1/self.m)
        eq_stress_test    = self.calc_eq_loadrange(rf)

        return equivelant_stress,np.sum(scaled_cycles)

    def calc_eq_loadrange(self,rf):
        '''
        Usually, a further simplification of the fatigue loading quantification
        is performed by converting the fatigue load spectrum into a damage
        equivalent load range ΔRe. This is a constant load range which, in an
        equivalent number of cycles Ne, will produce the same amount of damage
        as the actual fatigue load spectrum with different load ranges ΔRi and
        cycles Ni. The equivalent load range ΔRe is defined as (Hansen, 2008):

        Parameters
        ----------
        rf : List of arrays
            Rainflow output.

        Returns
        -------
        eq_loadrange : Value
            Equivelant load range.

        '''
        eq_loadrange = np.sum(((rf[0]**self.m)/self.n_eq))**(1/self.m)
        return eq_loadrange
```

D.10. Utility Functions

D.10.1. Utilities for Main

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
"""
Created on Mon Oct 17 12:20:33 2022

@author: Giles
"""
import time
import numpy as np
from Utilities.Utilities import find_nearest_index
def unpack_x(x):
    D = x[0]
    L = x[1]
    t = x[2]
    return D,L,t

def optimizeloop(iterations,x0,objective,constraints,T0,optimizationclass,L=10):

    solution_list = np.zeros(iterations)
    x_best        = np.zeros([iterations,len(x0)])
    x0_list       = np.zeros([iterations,len(x0)])
    x_start_list  = np.zeros([iterations,len(x0)])
    time_list     = np.zeros([iterations,len(x0)])
    consec_change_count_lst=[]
    howmanydown_count_lst  =[]
    K_lim_count_lst        =[]
    Tempchange_count_lst   =[]
    for i in range(iterations):
        #x0[0] = np.random.uniform(5,15)
        #x0[1] = np.random.uniform(150,300)
        #x0[2] = np.random.uniform(0.01,0.5)
        print(i)
        x_start_list[i]     = x0

        start_time = time.time()
        optimize             =  optimizationclass(objective,constraints,T0,x0,L,J=50,delta =
            0.1,gamma = 0.0025,K_lim = 50)
        x0_list[i]           = optimize.x0
        x_best[i]            = optimize.xbest
        solution_list[i]     = objective(x_best[i])
        consec_change_count_lst.append(optimize.consec_change_count)
        howmanydown_count_lst.append(optimize.howmanydown_count)
        K_lim_count_lst.append(optimize.K_lim_count)
        Tempchange_count_lst.append(optimize.Tempchange_count)
        print("---%s seconds ---" % (time.time()-start_time))
        time_list[i] = (time.time()-start_time)
    return solution_list,x_best,x_start_list,time_list, consec_change_count_lst,
        howmanydown_count_lst, K_lim_count_lst, Tempchange_count_lst


def didnotconverge (x_best_list,x0):
    x_noconverge = x_best_list[x_best_list==x0]
    return x_noconverge

def max_improve(solution_list,f0):
    diff = solution_list - f0
    return ((diff)/f0)*100

def statistics(solution_list,f0):
    '''
    Generates the average and standard deviation of a looped optimization.

    Parameters
    ----------
    solution_list : TYPE
        Array of all the found optima.
    f0 : TYPE
        Starting solution.

    Returns
    -------
    TYPE
        Average optima and standard deviation.
```

```
    '''
    A = clean_x0_list(solution_list,f0)
    A = (solution_list-f0)/f0
    A = A*100
    A=A[A!=0]
    return np.average(A), np.std(A)


def clean_x0_list(x0_list,x0):
    index = np.where(x0_list==x0)

    x0_clean = np.delete(x0_list,index,0)
    return x0_clean

def design_variable_stats(x0_list,x0):
    x0_list = clean_x0_list(x0_list,x0)
    D_list = np.zeros(len(x0_list))
    L_list = np.zeros(len(x0_list))
    t_list = np.zeros(len(x0_list))
    comb_list = np.zeros(len(x0_list))


    for i in range(len(x0_list)):
        D_list[i] = x0_list[i][0]
        L_list[i] = x0_list[i][1]
        t_list[i] = x0_list[i][2]
        comb_list[i] = D_list[i]*L_list[i]*t_list[i]
        print(D_list[i]*L_list[i]*t_list[i])
    Avg_D = np.average(D_list)
    Avg_L = np.average(L_list)
    Avg_t = np.average(t_list)
    Avg_comb = np.average(comb_list)

    Avg_x = np.array([Avg_D,Avg_L,Avg_t])
    std_x = np.array([np.std(D_list),np.std(L_list),np.std(t_list)])
    std_comb = np.std(comb_list)




    return Avg_x, std_x, [Avg_comb,std_comb]

def find_extreme_designs(xbest_list):
    D_max,L_max,t_max = np.amax(xbest_list,0)
    D_min,L_min,t_min = np.amin(xbest_list,0)

    D_max_idx = np.where(xbest_list == D_max)[0]
    L_max_idx = np.where(xbest_list == L_max)[0]
    t_max_idx = np.where(xbest_list == t_max)[0]

    D_min_idx = np.where(xbest_list == D_min)[0]
    L_min_idx = np.where(xbest_list == L_min)[0]
    t_min_idx = np.where(xbest_list == t_min)[0]

    x_Dmax = xbest_list[D_max_idx]
    x_Lmax = xbest_list[L_max_idx]
    x_tmax = xbest_list[t_max_idx]

    x_Dmin = xbest_list[D_min_idx]
    x_Lmin = xbest_list[L_min_idx]
    x_tmin = xbest_list[t_min_idx]

    x_max = np.array([x_Dmax,x_Lmax,x_tmax])
    x_min = np.array([x_Dmin,x_Lmin,x_tmin])

    return x_max, x_min
```

D.10.2. Utilities for Packages

```
    # -*- coding: utf-8 -*-
"""
Created on Mon Aug  1 12:04:34 2022

@author: TORSPI
"""
import numpy as np
from math import exp, log, sqrt,pi,tanh
from scipy.optimize import fsolve
import pickle
import math

def save_obj(obj,filename,path = 'Objects/'):
    with open(path+filename, 'wb') as outp:
        pickle.dump(obj, outp, pickle.HIGHEST_PROTOCOL)

def load_obj(filename,path = 'Objects/'):
    with open(path+filename, 'rb') as picklefile :
        content = pickle.load(picklefile)
    return content


def indices(a, func):
    return [i for (i, val) in enumerate(a) if func(val)]

def JonSwap (Hs,Tp,f_matrix,Gamma=3.33):
    """
    Function that generates the spectral densities according to the JonSwap
    Spectrum. Using a standard paramater of 3.33, this can also be adjusted when
    the function is called.

    Parameters
    ----------
    Hs : Value
        Significant Wave Height.
    Tp : Value
        Significant Wave Period.
    f_matrix : Array
        Frequence matrix.
    Gamma : Value, optional
        Jonswap Parameter. The default is 3.33.

    Returns
    -------
    Sjs : TYPE
        Spectral Density.
    a : Array
        Spectral Amplitudes.

    """
    fp = 1/Tp
    df = f_matrix[1]-f_matrix[0]
    Sjs = np.zeros(len(f_matrix))
    a   = np.zeros(len(f_matrix))

    for i in range(len(f_matrix)):
        if f_matrix[i] <= fp:
            sigma = 0.07
        else:
            sigma = 0.09

        Sjs[i] = 0.3125*Hs**2 *Tp*((f_matrix[i]/fp)**-5)\
            * exp(-1.25*((f_matrix[i]/fp)**-4))*(1-0.287*log(Gamma))\
                * Gamma**exp(-0.5*(((f_matrix[i]/fp)-1)/sigma)**2)

        a[i] = sqrt(2*Sjs[i]*df)

    return Sjs,a

def Kaimal(u,I,f_matrix,l=340.2):
    """
```

```
    Using the IEC defenition of he kaimal turbulence model (International
    Electrotechnical Commission, 2015) This function returns the spectral
    densitiy

    Expression for Kaimal spectral density found in:

    Operational Vibration-Based Response Estimation for Offshore Wind Lattice Structures
    February 2015
    DOI:10.1007/978-3-319-15230-1_9

    Parameters
    ----------
    f_matrix : Array
        Array containing all frequencies.
    u : Value
        Mean Wind Speed.
    I : Value
        Turbulence intensity. (standard deviation of the wind speed)
    l : Value
         Length .Optional: default = 340.2

    Returns
    -------
    S_wind : Array
            Spectral Density
    a_wind : Spectral Amplitudes

    """
    #figure out standard
   # sigma  = (TI /100)*u
   # S_f = (4 * sigma * L /u)/(1+6*f*L/u)**(5/3)


    #Past used example
    df = f_matrix[1]-f_matrix[0]
    S_wind   = np.zeros(len(f_matrix))
    a_wind   = np.zeros(len(f_matrix))

    S_wind  = 4*I**2*u*l    *  ((1+6*(f_matrix)*l/u))**(-5/3)
    a_wind  = np.sqrt(2*S_wind*df)

    return S_wind, a_wind

def unpack_q(q):
    x           = q[0]
    z           = q[1]
    theta       = q[2]

    x_dot       = q[3]
    z_dot       = q[4]
    theta_dot   = q[5]
    return x,z,theta, x_dot,z_dot,theta_dot

def find_nearest_index(value,array):
    array = np.asarray(array)
    idx = np.abs(array-value).argmin()
    return idx

def cut_profile2size(L_spar,profile,z_uw,spar_position):
    #-- find the lowest position of spar in water
    lowest_point = spar_position-L_spar
    lowest_profile_index = find_nearest_index(lowest_point,z_uw)
    if z_uw[lowest_profile_index] < L_spar:
        lowest_profile_index = lowest_profile_index-1

    useful_profile = profile[lowest_profile_index:]

    return useful_profile, z_uw[lowest_profile_index:]

def calc_ddx(dx,t):
    """
```

```
    This function calculates the time derivative based on linear assumpution.
    Assuming that the change in time derivative at two steps has happened linearly


    Parameters
    ----------
    dx : Array
        Thing you want to take the derivative of (at given time steps).
    t : Array
        Array of time values correspondiing with the to be derived array.

    Raises
    ------
    ValueError
        The two arrays must be the same size.

    Returns
    -------
    ddx : Array
        Time Derivative.

    """
    #check lengths
    if len(dx) == len(t):
        ddx = np.zeros(len(dx)-1)
        for i in range(len(dx)-1):
            ddx[i] = (dx[i+1]-dx[i])/(t[i+1]-t[i])
            if math.isnan(ddx[i]):
                ddx[i]=0
        return ddx
    else:
        raise ValueError("Two different size arrays can not be used")

def find_length(array):
    """
    This function returns the longest length in a multiple dimension array.
    The built in function len will only return the length of the first dimension


    Parameters
    ----------
    array : Array
        Multiple Dimension array.

    Raises
    ------
    ValueError
        This function works up to 5 dimensions.

    Returns
    -------
    Value
        Length of the longest dimension.

    """
    dimensions = np.ndim(array)
    if dimensions == 1 :
        return len(array)
    if dimensions == 2:
        len1 = [len(array),len(array[0])]
        return max(len1)
    if dimensions == 3:
        len1 = [len(array),len(array[0]),len(array[0][0])]
        return max(len1)
    if dimensions == 4:
        len1 = [len(array),len(array[0]),len(array[0][0]),len(array[0][0][0])]
        return max(len1)
    if dimensions == 5:
        len1 = [len(array),len(array[0]),len(array[0][0]),len(array[0][0][0]),len(array
            [0][0][0][0])]
        return max(len1)
```

```
    if dimensions > 5:
        raise ValueError("This function cannot return the length of an array with so many
            dimensions")
```

D.10.3. Utilities for Wind Functions

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 16 20:54:11 2022

@author: Giles
"""
from math import exp

def movewindup(U10,z,alpha=0.1):
    '''
    Function returns wind speed at any height using the power law.

    Parameters
    ----------
    U10 : Value
        Wind Speed at 10 meters.
    z : Value
        Hub Height.
    alpha : Value, optional
        Power law coefficient. The default is 0.1.

    Returns
    -------
    Value
        Wind speed at given height.

    '''
    return U10*(z/10)**alpha

def weibull_pdf(u,alpha = 2.029,beta = 9.409):
    '''
    Returns weibull pdf function given a specific wind speed.
    if not specified the parameter values are taken from Norway Site 5
    Found in:
        "Joint Distribution of Environmental Condition at Five European
    Offshore Sites for Design of Combined Wind and Wave Energy Device"s


    Parameters
    ----------
    u : Value
        DESCRIPTION.
    alpha : Value, optional
        Parameter. The default is 2.029.
    beta : Value, optional
        Parameter. The default is 9.409.

    Returns
    -------
    TYPE
        DESCRIPTION.

    '''
    return   alpha/beta*(u/beta)**(alpha-1)*exp(-(u/beta)**alpha)
```

D.11. Math Functions

D.11.1. PSD

```
    # -*- coding: utf-8 -*-
"""
```

```
Created on Fri Jun 17 11:01:32 2022

@author: TORSPI
"""
import numpy as np

def PSD(time,signal):
    """
    Return Power Spectral Density of any signal over time

    Parameters
    ----------
    time : array
        vector of time (1D array).
    signal : Array
        Response in time.

    Returns
    -------
    psd : Array
        Power spectral density over frequency.

    """
    df = 1/(time[-1]-time[0])
    f_vec = df*np.linspace(0,len(time)-1,len(time))
    ai = np.fft.fft(signal)/len(time)
    ai[0] = 0
    ai[range(round(len(time)/2),len(time))]=0
    ai = 2*ai
    psd = abs(ai)**2/2/df

    return psd,f_vec
```

D.11.2. Integrate Class

```
    # -*- coding: utf-8 -*-
"""
Created on Sun May  8 17:20:02 2022

@author: TORSPI
"""
import numpy as np

class  Integrate:
    """
    Class used to integrate functions
    """
    def __init__(self,function):
        self.function = function
        self.error = 0
        self.sign  = 1

    def integral(self,lower,upper,precision=100):
        """
        Integral using the trapezoidal method

        Parameters
        ----------
        lower : Value
            Lower limit of the integral.
        upper : Value
            Upper limit of the integral.
        precision : Value, optional
            Defines precision of integral for a hundreth precision fill in a
            hundred . The default is 1000.

        Returns
        -------
        Value
            Integral of passed function .
```

```

        """

        if lower>upper:
            lower,upper = upper,lower
            self.sign = -1
        numberofpoints = (upper-lower)*precision
        dx_list = np.linspace(lower,upper,int(numberofpoints+1))
        integral = 0
        super_sum = 0
        sub_sum = 0

        for i in range(len(dx_list)-1):
            delta = dx_list[i+1]-dx_list[i]
            try:
                y1 = self.function(dx_list[i])
                y2 = self.function(dx_list[i+1])
                sub_area = y1*delta
                super_area = y2*delta

                area = (y2+y1)/2 * delta
                integral+= area
                sub_sum += sub_area
                super_sum += super_area
            except ZeroDivisionError:
                print(f"\nAvoided pole")
        self.error = super_sum - sub_sum
        return self.sign*integral
```

D.11.3. Boolean Functions

```
    # -*- coding: utf-8 -*-
"""
Created on Tue May 24 19:45:12 2022

@author: TORSPI


Contains random mathmatical functions that return boolean values
"""

def checkifeven(anyvector):
    """
    Function that checks if list or array has even amounts of entries

    Parameters
    ----------
    anyvector : Array or List
        The array or list that you want to consider

    Returns
    -------
    bool
        truth for even, false for uneven.

    """
    if (len(anyvector) % 2) ==0 :
        return True
    else: return False
```