NTNU – Trondheim
Norwegian University of
Science and Technology

Simulation-Based Optimization of Lattice Support Structures for Offshore Wind Energy

Håvard Molde

Master of Science in Product Design and Manufacturing
Submission date: June 2012
Supervisor: Michael Muskulus, BAT

Norwegian University of Science and Technology
Department of Civil and Transport Engineering

Institutt for bygg, anlegg og transport

FAKULTET FOR INGENIØRVITENSKAP OG TEKNOLOGI

NTNU – Norges teknisk-naturvitenskapelige universitet

MASTEROPPGAVE 2012

for

Håvard Molde

Optimering av fagverksstrukturer for offshore vindturbiner ved hjelp av simulasjonsverktøy

Simulation-Based Optimization of Lattice Support Structures for Offshore Wind Energy

Support structures for offshore wind turbines are typically multi-member jackets with complex geometry. The standard design process is based a lot on experience and simplified analyses, since a full analysis (especially of fatigue damage) is time-consuming. In contrast to this intuitive-iterative design process, there exist relatively simple automatic optimization techniques that should lead to potentially better designs. One defines an objective function, based on the amount of steel used (as an indicator of total cost), and some constraints on the response (utilization of joint capacity), and then uses standard search algorithms to find locally optimal solutions in design space.

During this master thesis, a relatively new method for the optimization of multi-member support structures for offshore wind turbines shall be implemented and tested. The performance of the designs will be evaluated by integrated time-domain analyses with FEDEM Windpower, a new software for flexible multibody dynamics of wind turbines. Tools for postprocessing already exist that allow for performing joint checks and estimating joint fatigue lifetimes. Spall's simultaneous perturbation algorithm will be implemented to estimate a pseudo-gradient in design space and to automatically optimize the design. The method will initially be used for a full-height lattice tower, an alternative innovative support structure that is part of the NOWITECH 10 MW reference turbine. It is assumed that all sections have constant leg and brace dimensions. The optimization will be performed for different site conditions, i.e., different load cases, and the sensitivity of the design, both with regard to the site conditions and to changes of parameters, will be assessed. The main goal of the project is to understand and answer the question whether simulation-based structural optimization with Spall's algorithm is feasible for offshore wind turbine support structures.

If there is enough time available, the following optional activities can be performed to round off the master thesis:

1. optimization of other structures, e.g., the UpWind reference jacket;
2. is it possible to optimize designs if member dimensions are allowed to change (once) in each section, and how will this affect the results?
3. is it possible to optimize designs if sections can have variable heights, and by how much will this additional freedom improve designs?
4. more involved simulation-based optimization methods such as response-surface modeling can be tested

Besvarelsen organiseres i henhold til gjeldende retningslinjer.

Veileder(e): Michael Muskulus

NTNU, 12. januar , 2012

Michael Muskulus

Førsteamanuensis ved Institutt for bygg, anlegg og transport

Offshore wind turbine technology

Preface

The period I have spent working on this thesis has been both challenging and rewarding. I went into this work without any previous experience in either optimization techniques or windmill structures. The learning curve has therefore been steep. I am pleased with being able to find answers to most of the questions I was investigating, but at the same time I deeply feel that there is so much more I want to look into, and just wish I had the time and resources to do it! As usual with these kinds of projects, you wish you had the knowledge you have today when you started the work, because then you would have done so much better. But then there wouldn't really be any reason to do it, so that feeling is probably a good sign. If I could point out one thing that would have really helped me it would be access to more computational power. My simulations have pretty much been running 24/7 for three months, and it would be preferable if these results could have been available earlier. This would also have allowed me to further investigate and validate my findings. However, all in all I think I have found some interesting results and hope my work will be appreciated and used in the future. I would like to give my sincere thanks to my adviser, Michael Muskulus, and PhD-researcher Daniel Zwick.

Abstract - English

Today, design of wind energy support structures is to a large extent a manual process. It requires a lot of experience, and the design tools are often based on simplified methods. As larger structures are being developed and installations move to larger water-depths, the need for efficient and accurate design tools increases. Simulation-based design is a promising technique that can help automate this process.

In this study, Spall's simultaneous perturbation stochastic approximation (SPSA) method was implemented to automatically optimize thickness and diameter of the members in offshore lattice tower support structures. The method utilizes a pseudo-gradient based on only two function evaluations per iteration, which allows for a computationally efficient process. Each evaluation of the design consists of time-domain simulations of the complete wind turbine in FEDEM Windpower, subsequent rainflow counting and calculation of joint lifetimes with stress concentration factors. The utilization of both ultimate and fatigue limit states is reported for each joint. Tower weight was chosen as an indicator of cost, and an objective function comprising variables for weight and joint lifetimes was defined. Joint lifetime was ignored whenever its value was above the design lifetime of the tower, allowing the algorithm to search solely for the lightest design, as long as the design lifetime constraint was sustained. The method has shown promising results, and is able to successfully find viable designs, even when starting from highly unacceptable starting points.

Some of the major challenges when using SPSA for lattice support structures are to find a good objective function, as well as appropriate values for the parameters controlling perturbation and step size. Existing guidelines were followed when doing this calibration, but for an efficient search the parameters had to be adapted. Results for both appropriate parameters and the optimization itself are reported for the 10MW NOWITECH reference turbine on a full-height lattice tower. These results show that superior results can be achieved, but at a high cost in terms of computational time. Recommendation is given to use alternative methods to come up with a partially optimized staring point, from which the SPSA method can optimize further.

Samandrag – Norsk

I dag blir fundament for offshore vindturbinar stort sett utvikla ved hjelp av manuelle prosedyrar. Simuleringsverktøya er ofte basert på forenkla metodar, og mykje erfarings-kunnskap er nødvendig. Ettersom strukturene stadig blir større og blir installert på stadig djupare vatn blir behovet for effective og nøyaktige utviklingsverktøy stadig større. Simulerings-basert optimialisering er ein lovende teknikk som kan vera med på å automatisere denne prosessen.

I denne studien blei simultaneous perturbation stochastic approximation (SPSA) –metode implementert for å automatisk optimalisere tjukkelsen og diameteren av stavane i fagverkskonstruksjoner for offshore vindturbiner. Denne metoden etablerer en psaudo-gradient ved hjelp av kun to evalueringar, noko som gjer metoden effetiv i forhold til andre optimaliseringsalgoritmer. Kvar evaluering består av simulering av heile konstruksjonen i tids-domenet ved hjelp av FEDEM Windpower, etterfulgt av evaluering av lastsykluser og berekning av spenningskonsentrasjonsfaktorer. Ut i fra dette blir blir utnyttelsen av både maksimal styrke og utmattingsstyrke rapportert for kvart enkelt knutepunkt. Total vekt blei valgt som indikator for kostnaden, og ein evaluerings-funksjon med vekt og levetid for knutepunkta som variablar blei definert. Levetida for knutepunkta blei ignorert så lenge dei var over den dimensjonerte levetida. Algoritmen søkte dermed utelukkende etter laveste vekt så lenge alle krav var oppfylt. Metoden har vist veldig gode resultater og har vore i stand til å finna gode design, sjølv frå dårlige utgangspunkt.

A definere ein god evaluerings-funksjon og finne gode verdiar for parameterane i algoritmen er nokon av utfordringane ved å bruke SPSA. Eksisterande retningslinjer blei fulgt under denne kalibreringa, men for å forbedre effectiviteten var ytteligare tilpassing nødvendig. Resultat fra ei fullstendig optimalisering av NOWITECH sin 10MW referanseturbin på eit full-høgde fagverkstårn, i lag med anbefalte verdiar for dei ulike parameterane vil bli presentert. Desse resultata viser at overlegne resultat kan bli oppnådd med denne metoden, men at konstaden, i form av stort tidsforbruk, er høg. Det blir anbefalt å bruke alternative metoder for å komme opp med eit delvis optimalisert startpunkt, for så å bruke SPSA-metoden for vidare optimalisering.

Contents

Preface III
Abstract - English V
Samandrag – Norsk VII
List of Figures XI
1. Introduction 1
1.1 Motivation 1
1.2 Objectives 1
1.3 Structure of the thesis 2
2. Background 3
2.1 Background on optimization methods in general 3
2.2 SPSA method 4
2.3 Comparison to alternative optimization methods 5
2.4 Offshore wind energy support structures 5
2.5 Full-height lattice tower 7
2.6 Simulation-based optimization 8
3. Theory 9
3.1 SPSA method 9
4. Method 12
4.1 Tower 12
4.2 Analysis 12
4.3 Computer code 13
4.4 Enviromental conditions and loading 13
4.5 Measurements and noise 13
4.6 Procedure 15

5. Results 16
5.1 Implementation 16
5.2 Objective function 19
5.3 Performance 22
5.4 Accuracy 35
6. Discussion and conclusion 36
6.1 Discussion 36
6.2 Conclusion 37
6.3 Future work 38
References 39
Appendix 41

List of Figures

Figure 1: Different wind energy support structures. From left to right: monopile, tripod, jacket, gravity, tension-legged platform, spar buoy [2] 6

Figure 2: Nowitech reference turbine on full height lattice tower 7

Figure 3: Optimization cycle 8

Figure 4: Tower geometry 12

Figure 5: Superposition of stresses for tubular joints [1] 14

Figure 6: Joint dimensions [1] 17

Figure 7: Weight term of objective function 21

Figure 8: Lifetime term of objective function. 1D example 21

Figure 9: Optimization progress for varying *c* values 23

Figure 10: Optimization progress for varying gamma values 23

Figure 11: Perturbation width, ck, for varying gamma values 23

Figure 12: Optimization progress. Two identical runs with gamma = 0.1667, and three identical runs with 24

Figure 13: Optimization with, and without locking. 27

Figure 14: Minimum lifetime values for all sections, ... 28

Figure 15: Minimum lifetime values for all sections, mixed starting point 28

Figure 16: Optimization progress with different staring points 28

Figure 17: Minimum lifetime values for all sections, best result obtained with Minimization method 29

Figure 18: Optimization progress for the Minimization of Lifetime's method compared to that of the SPSA method 30

Figure 19: Optimization progress for weight on the left and objective function on the right, staring from the best result obtained by the Minimization of Lifetime's method 31

Figure 20: Minimum lifetime values for all sections, best results obtained with the SPSA method 32

Figure 21: Dimensions before (green) and after (blue) optimization..... 32
Figure 22: Optimization progress for full-length optimization. Arrows point to approved designs..... 33
Figure 23: Green curve showing the tendency of the weight (blue) 34
Figure 24: Dimensions before (green) and after (blue) optimization..... 35

1. Introduction

1.1 Motivation

The global demand for energy is increasing every year. Lately, there has been an increasing focus on the idea that the world needs to move to more environmentally friendly energy production. Offshore wind energy is gaining traction as one of the technologies that will enable the world to handle both of these demands. However, the first offshore wind farm was installed as late as 1991, and further development needs to be done to improve the economy of offshore wind energy. As turbine sizes grow, and installations move to deeper water, the cost of the support structure increases significantly [3]. To ensure efficient and economical development and production of offshore wind turbines, efficient and robust tools are essential. Support structures for offshore wind turbines on intermediate to deep water depths are typically multi-membered jacket structures with complex geometry. A critical part of the design process is thus to decide layout and dimensions of these members. An efficient, automatic tool for doing such a design would be highly desirable.

1.2 Objectives

This thesis will investigate the use of simultaneous perturbation stochastic approximation (SPSA) in the design of lattice support structures for offshore wind energy. The method will be evaluated based on its capacity of obtaining an acceptable design, the speed of the algorithm, and results compared to current best practice.

The method will be tested on a reference design that is currently being developed. The goal is not to find the optimal design for this particular structure, but use it as a basis for comparison.

Some important limitations apply:

- This study will not consider the natural frequency of the structure. As this is a crucial feature of any wind turbine, one can argue that it should be included in an optimization code. The author does very much agree on that, but given the limited time and resources allocated to this master thesis, it had to be left for future work.
- This study will not consider different load cases. Time was better spent focusing on the core implementation of the method and investigating its characteristics based on a single load case. Also, for a support structure for wind power to be verified and approved it needs to run through a large number of simulations, subject to different

load cases. How this process can be integrated and implemented as a part of the optimization process is not considered in this report.

- As there was a limited amount of time available for this study and the main objective, with its many lengthy simulations, proved to be very time-consuming, none of the four optional activities in the problem description will be covered.

1.3 Structure of the thesis

The first part of this thesis will provide background information on optimization in general, and on the SPSA method in particular. Some general background information on offshore windmill structures and lattice towers will also be given, as that is the problem that will be optimized. In chapter 3, a more detailed review of the theory behind the SPSA method will be presented. Chapter 4 gives the methodology on which this study is based upon. Results will then be given in chapter 5, before some additional discussion and conclusions will be found in chapter 6.

2. Background

2.1 Background on optimization methods in general

Mathematical optimization is the process of formulating and solving mathematically defined optimization problems. It is methods for finding the *best* solution to a problem. Optimization is a broad theoretical field, and is applied to countless problems. New applications are constantly made possible thanks to ever-increasing computational power and continuous development the techniques and algorithms.

While the very origin of optimization, which is simply a one-dimensional line search or root-finding problem, has been done for centuries, the history of multivariable optimization methods is much shorter. The important Simplex method was introduced in the late 1940s and numerous methods have later followed. Important contributions are many, but [5], on finding function extreme values for the scalar case, and [6] on finding function extreme values for multivariable cases are especially important. [7]

Stochastic optimization

Stochastic optimization refers to the minimization (or maximization) of a function when there is random noise in the measurements, and/or there is a random choice made in the search direction as the algorithm iterates towards a solution. [8]

Within stochastic optimization we separate between two different approaches. Traditionally we have many algorithms that use direct gradient evaluations in a deterministic setting. But this information is not always available, and that has propelled a growing interest in algorithms that approximate the gradient based only on measurement of an objective function. In general, algorithms that utilize direct gradient measures will use fewer iterations to converge. But even if this information is possible to attain, it can be very difficult, or it can require costly evaluations. Because of this one cannot say that one procedure is superior to the other, but normally, if information on gradients is conveniently available, gradient-based algorithms are usually preferred.

Multi-objective optimization

Multi-objective optimization is the part of the optimization theory that is concerned with optimization of problems with multiple, often conflicting, success factors. These success factors, or objectives, can be tangible features like weight, cost, production time and speed, or more abstract dimensions like customer satisfaction. For the rest of this report we will focus

on the former, as they are most relevant to the problem being addressed, and are easier to quantify. You usually have a set of alternatives, herby called Θ, which is confined within a design space, Ω. The design space is an *n*-dimensional vector space, where *n* is the number of variables in the problem. Θ is thus an *n*-dimensional vector subset to the *n*-dimensional vector space. In a multi-objective optimization there is no trivial measure of success readily available. We need to come up with a way to quantify how good an alternative Θ is, thus giving a way to choose the best solution. For this, an *objective function* is defined. The objective function is designed by the analyst and its main purpose is to weight the different objectives involved. For practical mathematical reasons it is usually the goal to minimize this function. When this function, say *f(Θ)*, is at its minimum, the most favorable combination of variables, Θ = Θ*, is found. Optimization problems are, however, rarely done in a vacuum, and you often have to deal with more constraints and conditions. These are often called constraint functions and typically look something like: $g_j(\Theta) \leq 0, j = 1, \ldots . m$ [9] .

2.2 SPSA method

The SPSA method was introduced by James C. Spall in 1987 [10] and fully analyzed by the same author in 1992 [11]. It's a stochastic optimization technique that doesn't require direct gradient measures, and instead utilizes a simple but highly effective gradient approximation based on two evaluations of the objective function. It is relatively easy to implement, and is well suited for difficult multivariable problem. The method has since its introduction attracted considerable attention, and has been used to solve a broad range of problems. Some of the attractive features of the method are:

- The key selling point for SPSA compared to other SA-algorithms is that it only requires two function evaluations per iteration, no matter how many variables there are in the problem. The more traditional SA-algorithm: finite difference stochastic approximation (FDSA) uses by comparison *2p* function evaluations per iteration, where *p* is the number of variables. The fact that the number of evaluations in SPSA is independent of the number of variables makes it very suited for large multivariable problems.
- SPSA does not need information on the gradient of the objective function. This information is not available in many real life situations, and the fact that it is not necessary for SPSA makes the method applicable for many more problems.
- The function evaluations are in many situations noisy measurements, making the search harder. SPSA accommodates noisy measurements, making it a very robust algorithm.
- Because the algorithm is only doing two function evaluations and moves towards the steepest descending of the two, it will not always move towards the actual steepest decent at that point. This, to some extent, allows it to escape local minima and search for global minima. However, to be certain the results are not in a local minimum, additional measures have to be taken. A further discussion of this topic will be presented later.
- The SPSA-method is very well documented. It is not just that there are many interesting articles available on the subject, but most of them are also very easily

available through the SPSA-website [12]. This site provides, along with a very comprehensive list of references (with direct links to PDF-files), an easy understandable introduction to the method, example MATLAB-code, as well as videos demonstrating the method.

However, the SPSA method is not suited for all problems. The fact that the method is only choosing among two function evaluations, the same mechanism that allows it to escape local minima, does also slow the method down. Since the gradient approximations are less accurate, more iterations are needed to reach a solution. If the problem is unsuited for the method, one risk the number of iterations increasing enough to make the total time increase, despite the decreased time spent per iteration.

A more detailed description of the method will be given in chapter 3.

2.3 Comparison to alternative optimization methods

SPSA was first introduced in the 1990s, but other similar algorithms have been around for much longer. Finite difference stochastic approximation (FDSA) has been the classical gradient-free stochastic optimization method, and its foundation was laid by J. Kiefer and J. Wolfowitz in 1952 [5]. The principle behind FDSA is the same as for SPSA, and the main difference lies in the way perturbations are done. As will be explained in more detail later, SPSA does simultaneous perturbations of all the variables resulting in only two function evaluations per iteration (with them being opposite directions). FDSA in comparison does individual perturbations of each variable on a one-at-a-time basis. For a p-dimensional problem this leads to $2p$ function evaluations per iteration (each variable is perturbed in positive and negative directions). This is a very intuitive approach, as it closely replicates the way analytical gradients are derived from the partial derivative with respect to all the variables. Studies has shown, however, that "under reasonably general conditions, SPSA and FDSA achieve the same level of statistical accuracy for a given number of iterations, even though SPSA uses p fewer function evaluations than FDSA (because each gradient approximation uses only $1/p$ the number of function evaluations)" [13]. In other words: "One properly chosen simultaneous random change in all the variables in a problem provides as much information **for optimization** as a full set of one-at-a-time changes of each variable." [13]. This is somewhat surprising, but it shows that for applications with expensive function evaluations, SPSA can provide large saving compared to FDSA, without losing much on convergence rate or accuracy.

2.4 Offshore wind energy support structures

Design

Offshore wind turbines can be mounted either on a bottom-fixed rigid structure, or on a floating structure. Floating structures are still uncommon, but several concepts have been investigated throughout the last 20 years [14]. Some of the more promising designs are the submerged tension-legged platform and the spar buoy [15]. Bottom-fixed structures are the dominating solution among wind farms today. Noticeable technical designs among the bottom-fixed solutions are: monopile, tripod, gravity foundation, and jackets [16]. Usually, a

tubular tower is mounted on top of the respective foundation structures, as illustrated in Figure 1, but as discussed in this report and described in the next sub-section also other solutions exist.

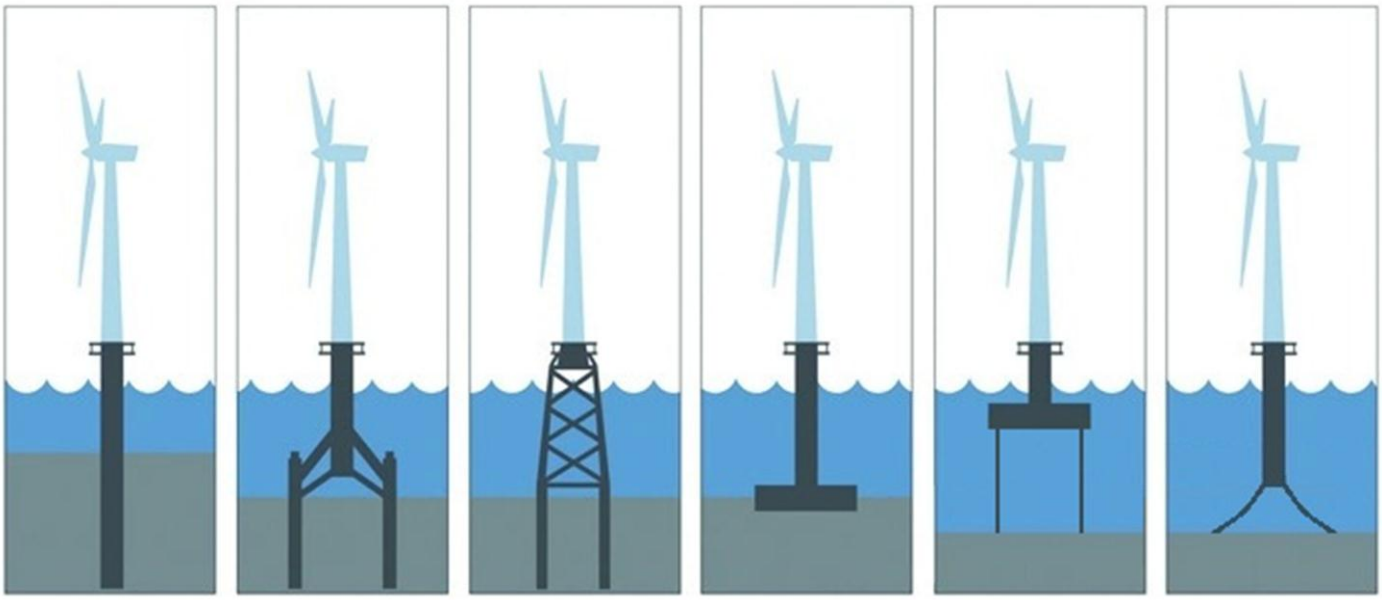

Figure 1: Different wind energy support structures. From left to right: monopile, tripod, jacket, gravity, tension-legged platform, spar buoy [2]

Loading and critical factors

Offshore wind turbines have a very unique and complex loading situation. Unlike onshore wind turbines that are only affected by wind, offshore turbines also have wave forces. These are, as everybody knows, periodic forces. Making the situation even more complex is the fact that wind and waves do not always come from the same direction, and to some extent, one of the two can be high, while the other is low. General characteristics that apply to all wind turbines are of course also present. These include the important *1p* and *3p* frequencies that are critical in wind turbine design. These are frequencies that correspond to, respectively, one and three loading cycles per revolution of the rotors. The *3p* excitations are important because a blade passes the support structure three times per revolution, causing changes to the loading situation (given a 3-bladed turbine). These critical frequencies are often at the same order of magnitude as the natural frequency of the structure, which makes it very important that the designer is aware of them, and finds a design with a non-problematic natural frequency. This is because a structure that is excited by a load with a frequency close to the natural frequency of the structure will experience resonance, which can easily result in fatal damage. The designer must also avoid the frequencies of waves hitting the structure, as even small forces can cause large oscillations of the structure. However, waves are usually less of a problem, because of the relatively low frequencies of waves.

A jacket-like structure will need some additional consideration. There will be a large number of relatively slender members with joints connecting them. These members can be excited

locally, in additional to global vibrations of the tower, giving additional contributions to the fatigue loading. The result of all of this is a structure with fatigue lifetime as the main dimensioning factor. [17]

Fatigue

Fatigue is a failure mechanism that causes structures to fail after repeated loading, even though the loading itself is well below the ultimate strength of the structure. The failure is caused by microscopic cracks that form and grow for each loading cycle. We can split the fatigue life into a crack initiation period, and a crack growth period. Research shows that the initial cracks are often formed at a very early stage of the lifetime. They are, however, very small, and can remain invisible for most of the structure's lifetime. Different conditions affect crack growth in the two periods differently. For instance, surface roughness has negligible effect on the crack growth period, but can have a large effect on the crack initiation process. Finally, these cracks will go from a micro stage to a macro stage and become large enough to cause structural failure. For a lattice structure, the fatigue cracking will almost always happen in the joints first. This is due to the higher stress concentration factor in these areas, which is the most important parameter for prediction on crack initiation. [18]

2.5 Full-height lattice tower

As a part of the large NOWITECH research program, work has been done to develop a 10 MW reference turbine. A part of this work is to develop a full-height lattice support structure for deep water (~60 m). Full-height lattice support structures have been used for small turbines onshore, and lattice jacket structures have been used for sub-surface structures for large offshore turbines, but the principle of using a lattice structure all the way from the seabed to the rotor-nacelle-assembly has never been done in a large scale for offshore turbines before. A 10MW turbine is also much larger than anything currently in production and the combination of a large turbine, with a large rotor diameter, and deep water results in a massive structure. This alternative structure has interesting advantages over the more traditional tubular tower. If a tubular tower has a monopile sub-surface foundation, the diameter required for use on deep water poses problems for fabrication and pile-driving. If a jacket structure is used as a foundation for the monopile, this requires a

Figure 2: Nowitech reference turbine on full height lattice tower

transition piece that is both heavy and expensive. A full-height lattice tower would not have any of these problems, but one would need four piles per structure, something that can complicate installation. Another advantage is that a full-height lattice tower can achieve a significantly lower weight by reducing the total consumption of steel. However, fabrication will be significantly more expensive, both due to a large number of welded joints, and due to the increased size of the structure compared to a monopile/tubular tower design. A major challenge concerning full-height lattice towers is the difficulty related to designing and optimizing such a complex structure. There are a large number of joints and members; the critical design factor will be the fatigue lifetime, and fatigue lifetime analysis is very computationally expensive. Research is currently going on to find good optimization techniques that can deal with this problem. One would need a fast code that can analyze the design, and estimate the fatigue lifetime. Then we would need an efficient optimization algorithm that can optimize the design within a reasonable number of iterations / amount of time. Daniel Zwick has developed an algorithm that does local optimization of thickness and diameter for all sections in the structure based on their respectively fatigue lifetimes. His method is able to find attractive designs within as few as 20 iterations. A more detailed comparison to Zwick's method will be given later.

The NOWITECH program runs from 2009 to 2017. There are six work packages (WP), and the total budget is NOK 320 million. [19] [20]

2.6 Simulation-based optimization

Today, design of wind energy support structures is to a large extent a manual process. It requires a lot of experience, and the design tools are often based on simplified methods. Simulation-based optimization is a technique where a sequence of different configurations is simulated to help obtain a configuration that is an optimal, or near the optimal, solution to the problem. By constantly learning from the previous simulations, the configuration used in the next simulation can be improved, and hopefully the algorithm is able to automatically find a good design. [21]

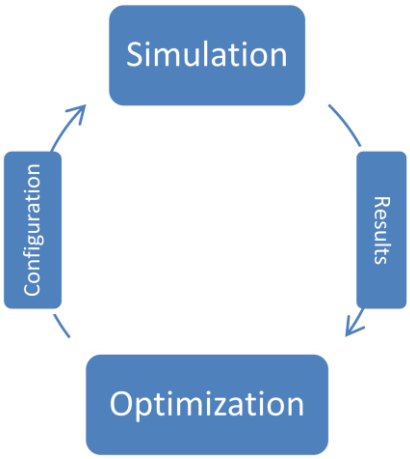

Figure 3: Optimization cycle

Some researchers have been thinking about utilizing such simulation-based optimization techniques on wind turbine support structures. Most focus on tubular tower design as the lattice tower investigated in this study is a new and novel design. Negm and Maalawi [22] show for the tubular case how the interior penalty function technique can be used to optimize with respect to both mass and stiffness. Long and Moe lays the foundation for optimization of lattice suppert structures in [23] and [24]. Their work is also closely related to the work done by Zwick [25], which is used for comparison in this report.

3. Theory

3.1 SPSA method

To understand the workings of the SPSA method, let's start with a step -by -step summary of the procedure. The SPSA algorithm consists of 5 steps. These are:

1. **Coefficients selection.** Coefficients are: A, a, c, α, and γ. All of them are used in gain sequence $a_k = \frac{a}{(1+A+k)^\alpha}$ and $c_k = \frac{c}{k^\gamma}$. The a_k parameter controls the step size between iterations, while c_k controls how large a gap there is between the function evaluations in each iteration, hereby called perturbation width. *k* represents the iteration counter, and should initially be set to 1. Often -used values for α and γ are 0.602 and 0.101 respectively. These are practically effective values, and also the lowest allowable values that satisfy conditions in [11]. Asymptotically optimal values are 1 and 1/6 respectively, and these might also be used, although the former values often give better performance since they maintain a larger step size. Anything in-between these values is of course also a legitimate choice. *A* is not always included in the algorithm, but can be useful to reduce the very large initial step sizes that would often be the result without it. *A* is typically chosen to be 10% or less of the maximum allowed or expected iterations. a and c are chosen based on the function evaluation values and how large step sizes are desirable. This will be described in more detail later.

2. **Generation of a simultaneous perturbation vector Δ_k.** This has to be a *p*-dimensional (where p equals the number of variables) vector generated by Monte Carlo satisfying conditions outlined in [11]. Among the requirements is that each component of the vector has to be independently generated from a zero-mean probability distribution with finite inverse moment. A typical choice satisfying these conditions is the Bernoulli ±1 distribution [26]. This is also the distribution that has been used throughout this study.

3. **Execution of the objective function evaluation.** This is where most of the computational effort is put down. First, based on the perturbations found in step 2, calculate new positions for the functions evaluations $(\theta_k \pm c_k\Delta_k)$. Secondly, execute the objective function evaluations. In our case, that means to run full time-domain simulations of the structure to determine loading, followed by subsequent rainflow counting and calculation of joint lifetimes with stress concentration factors, before the resulting lifetimes and structural weight is inputted into the objective function and two scalar $y(\theta_k + c_k\Delta_k)$ and $y(\theta_k - c_k\Delta_k)$ - values are obtained.

4. **Gradient approximation.** From the results in step 3, the two scalar objective-values $y(\theta_k + c_k\Delta_k)$ and $y(\theta_k - c_k\Delta_k)$ together with the Δ_k-vector and c_k makes up the gradient approximation:

$$\hat{g}_k(\theta_k) = \frac{y(\theta_k+c_k\Delta_k)-y(\theta_k-c_k\Delta_k)}{2c_k}\begin{bmatrix}\Delta_{k1}^{-1}\\ \Delta_{k2}^{-1}\\ \vdots \\ \Delta_{kp}^{-1}\end{bmatrix} \quad [27]$$

5. **Updating $\boldsymbol{\theta_k}$.** Using previous θ_k-values and the gradient from step 4, the new θ_k-estimate is calculated using standard SA form:

$$\theta_{k+1} = \theta_k - a_k\hat{g}_k(\theta_k).$$

If constraints are imposed on variables, these should be dealt with at this stage.

Iteration. Finally, if the solution is satisfactory, the algorithm can be terminated; if not, return to step 2, increase the iteration counter with one, (k+1), and iterate.

Selection of *a* and *c*

The *a* parameter controls the step size. Its value should thus be chosen based on how large a step is desired. Spall gives the following guideline for selection of *a*; "Choose *a* such that $\frac{a}{(A+1)^\alpha}$ times the magnitude of the elements in $\hat{g}_0(\theta_0)$ is approximately equal to the smallest of the desired change magnitudes among the elements of θ_k in the early iterations". To follow these guidelines, you first need to decide on all the other parameters, then run test-simulations to find out how large your $\hat{g}_0(\theta_0)$ is. You also need to assess an appropriate change in your θ_k -values.

Choosing *c* for noise-free settings is very easy. In this setting *c* can simply be chosen as some small positive number. If the measurements of the objective function are noisy, *c* can be chosen approximately equal to the standard deviation of the measurement noise.

Convergence, local and global minimum

Spall provides a detailed discussion on the theory behind SPSA in the original publication on the method [11]. Given that his conditions A1 to A5 and Lemma 1 hold, he shows that as $k \to \infty$

$$\Theta_k \to \Theta^* \text{ for almost all } \omega \in \Omega$$

meaning as the number of iterations increase, the algorithm will for almost any case converge to the Θ-value that minimizes the objective function. ($\Omega = \{\omega\}$ denotes the sample space generating Θ_k)

Several papers on SPSA as a global optimization/minimization technique have been published. In [28] they show that by injecting noise into the new variable estimate:

$$\theta_{k+1} = \theta_k - a_k\hat{g}_k(\theta_k) + q_k\omega_k$$

where ω_k is independent identically distributed *N(*0,*I)* injected noise (*I* = identity matrix), and $q_k^2 = \frac{q}{k}\log\log(k),\ q > 0$, and satisfying hypothesis H1 through H8 in [28], the solution will converge in probability to the set of global minima of the objective function. The injected noise does however make the algorithm more complicated, and it can slow down the convergence rate significantly. It is therefore very interesting that they, in the same paper, prove that given a different but similar set of assumptions J1 through J12, basic SPSA without injected noise does also in probability converge to the set of global minima of the objective function. These conditions are, however, not necessarily met without slowing down the convergence considerably, and is therefore not of particular interest for our specific problem. Regardless of the previous results, there is generally not a risk of converging to a saddle point, or to a maximum instead of a minimum, ensuring that the solution obtained is in fact a minimum, either a global or a local.

Extensions, and related methods

SPSA has great advantages in ease of use and generality [29], but the widespread adoption has also resulted in several extensions to the original algorithm. Spall discusses a second-order SPSA-algorithm in [30], which uses five function evaluations per iteration to estimate both the objective function gradient and an inverse Hessian matrix. In [31] a variant of the SPSA-method that only requires one function evaluation per iteration is presented. An implementation of the SPSA-method for global minimization is discussed by D.C. Chin in [32], and in more detail in [28]. Several publications discuss various methods for smoothing or averaging of the objective function gradient. This can be either smoothing based on measurements from previous iterations [33], or averaging between several measurements per iteration [11]. In addition, there are a large number of publications on the theoretical and practical use of SPSA method and its subsets, on various specific problems.

4. Method

4.1 Tower

The tower optimized in this study is shown in Figure 4. It is a full-height lattice tower with a total height of 151 m, of whom 60 are under water. There are 12 sections with X-bracings on each side, all with constant brace angles. There are four legs with 24 m distance at the seabed, and 4 m distance at the top. There are a total of 240 members in the structure. Joints are welded together in K- and X-joints.

This work has considered optimization with respect to member diameter and thickness. Thickness and diameter of the members was individually adjusted for each section, with different dimensions for legs and bracing. This gives a total of 48 variables (12 sections x 2 (legs and bracings) x 2 (thickness and diameter)). Other parameters that could also be optimized, but were not considered in this study, include: number of legs, number of sections, section design (constant brace angle, constant section height or variable section height), bottom leg distance, and member dimensions changing once or more within each section.

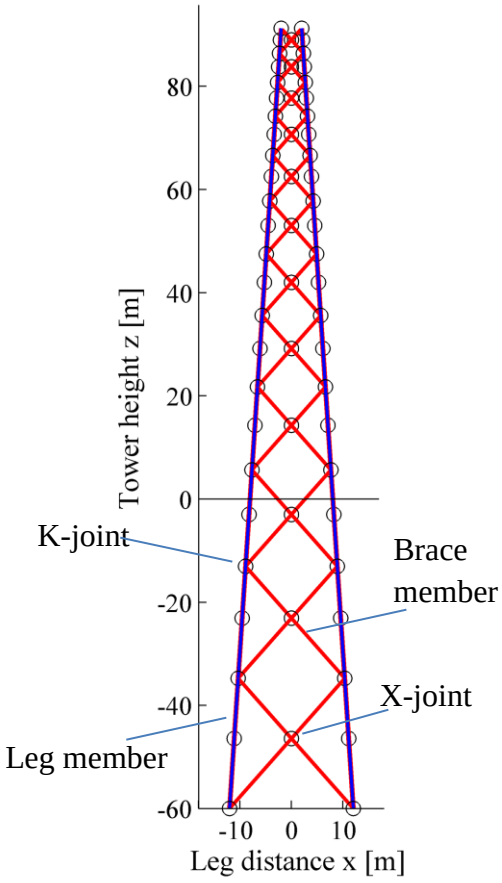

Figure 4: Tower geometry

4.2 Analysis

A complete finite element model of the whole structure, including blades, tower and soil properties, built using FEDEM Windpower, was used to execute integrated time domain simulations of the tower. This model was adjusted by the optimization code as the dimensions were updated.

Stress concentration factors (SCF) for eight hot spots around the circumference of each member intersection were calculated. These, together with forces and moments from the integrated analysis, were used to determine the hot spot stress (HSS) at the same locations.

The HSS's, result from rainflow counting of each time series, and S-N curves for tubular joints were used to estimate joint lifetimes. [25]

4.3 Computer code

The computer code necessary to implement the SPSA- method was written on top of a program developed by Daniel Zwick during his PhD-studies on full-height lattice support structures for offshore wind [25]. The optimization algorithm itself, along with post-processing tools, was written in Matlab, while the time-domain simulations were performed in FEDEM Windpower. For higher efficiency, Matlab parallel computing toolbox was used to run the two simulations, and their post-processing in parallel. The construction of the program allowed implementation of the new optimization-algorithms with limited modifications to the rest of the program.

4.4 Enviromental conditions and loading

For simplicity, one single load case was used to demonstrate this method. This load case represents a typical condition for the structure, with the turbine operating at rated speed. An irregular sea state was constructed using a JONSWAP spectrum with significant wave height $H_s = 4$m and mean wave period $T_p = 9$s. Wind and wave directions were aligned, with the wind being a 13.5m/s turbulent wind field (16% turbulence intensity). The wind and wave fields were identical for all simulations and iterations, but the response of the structure did of course change as the structure changed.

4.5 Measurements and noise

An important feature of the SPSA method that is comprehensively discussed in the literature is its ability to cope with noisy measurements. In this study, the simulations were performed using identical predefined wind and wave fields. Also, the weight measurements and rainflow-counting are without noise and were repeated without change for every run. This resulted in noise-free objective function measurements: you could run two simulations with the same configuration and get exactly the same result. However, this does not mean that one actually gets identical result when running multiple similar optimization runs. For that to happen the random numbers that generate the perturbation vector most be reproduced, and that is generally not the case (unless that is what is desired). The results will therefor vary for identical configurations, but that is due to the random generation of the perturbation vector, not noisy measurements.

Normalization of lifetimes values

The design lifetime of the structure can change between projects. For generality and simplicity, all functions utilizing the lifetime of joints, members or structure used a normalized lifetime, where the normalization was with respect to the design lifetime. Because of this, from now on we will refer to a lifetime equal to one as the design lifetime. If the lifetime is equal to 0.5, the structure has an estimated lifetime that is half that of the design lifetime; similarly, if the lifetime is equal to 3.0, the structure has an estimated lifetime that is

three times longer than the design lifetime. The actual design lifetime used in this report was 20 years.

Measurements

The two-sided SPSA method used in this study does two measurements, $y(\theta_k + c_k\Delta_k)$ and $y(\theta_k - c_k\Delta_k)$, to estimate the gradient at θ_k. However, the algorithm does not require simulation of the actual performance at θ_k. That means that even though you run the simulation long enough for the solution to converge, you have never tested the actual solution. It is of course trivial to implement an additional run at θ_k, either for each iteration, or only the last, but it would require some extra computational resources. Also, as the optimization runs, the two perturbations $(\theta_k + c_k\Delta_k)$ and $(\theta_k - c_k\Delta_k)$ keep coming closer and closer to θ_k as c_k decrease. Based on this, the testing at θ_k was not done in this study; instead, the best of the two perturbations, $(\theta_k + c_k\Delta_k)$ and $(\theta_k - c_k\Delta_k)$, was used to evaluate the solutions.

Rainflow counting

Rainflow counting is a technique for counting and analyzing cycles resulting from time domain simulations such that the results can be used for lifetime prediction. The technique utilizes the successive extremes of the loading sequence, and is well suited for situations where the amplitude of the loading is varying [34]. Rainflow counting was used throughout this study when estimated fatigue lifetimes were determined.

Stress concentration factors

A component, subject to internal stress, that has some kind of disturbance to its shape, like a hole or a constriction, will experience an increased stress around these disturbances [35]. The stress concentration factor can be defined as the ratio between stresses at certain hot spots, relative to the nominal stress range [1]. Stress concentration factors and stresses were calculated for a total of eight hot spots around the circumference at the intersection between connected members. These were calculated based on guidelines in DNV-RP-C203 [1]. Stresses are derived by summation of single stress components from axial, in-plane and out-of-plane action.

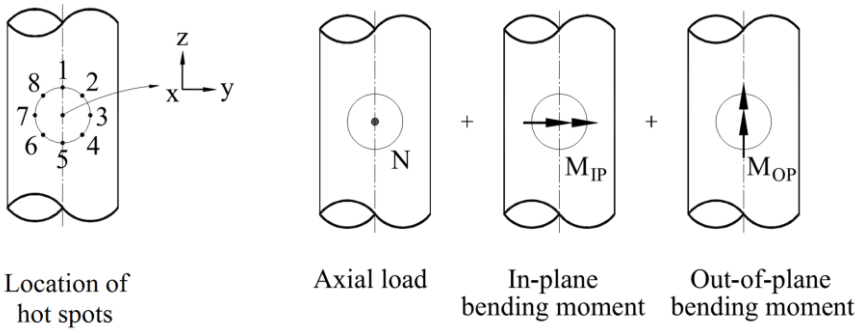

Figure 5: Superposition of stresses for tubular joints [1]

Time domain simulations

The time domain simulations were all simulating 120 seconds of real world performance. The IEC design standard 61400-3 recommends at least six 600-second simulations, or one one-hour simulation to ensure "statistical reliability of the estimate of the characteristic load effect" [36]. There is therefore an increased uncertainty in the simulation results obtained, but since the main goal was to investigate the performance of the SPSA method, not to obtain a validated optimal design, this uncertainty was accepted to achieve a faster optimization process.

4.6 Procedure

This work was initiated by a literature study on the SPSA method, along with work to get familiarized with the already existing code for optimization of lattice support structures. Next, a substantial amount of work was laid down to implement the optimization algorithm and all relevant constraints, and incorporate this into the existing pre- and post-processing. This phase did of course also include quite a lot of debugging. Once a functioning and stable code was ready, the process of finding a good objective function, along with all the corresponding parameters, was started. This was a highly time-consuming process, as it included a lot of trial and error. For every single run it takes at least 24 hours before one can say anything about its performance. And if the run looks promising initially, one often have to let it run for 48-72 hours before one can say anything conclusive about its performance. In addition, for every single objective function that was evaluated, several runs were performed to investigate whether other parameters could improve its performance. Halfway through the study a better computer was made available, allowing parallelization of the two concurrent function evaluations. This allowed more efficient calculation, which made it possible to run more iterations in the same amount of time. However, changing the code from serial-runs to parallel runs was not entirely trivial and required some work, along with subsequent debugging. When a promising objective function was finally identified, a more systematical study on the effect of different parameters on that particular function was performed. Also, its tolerance for varying staring configurations was investigated by running several different optimization runs with different initial guesses for the member dimensions. A substantial effort has of course also been laid down to evaluate all the results, devise new improvements and write this report.

5. Results

5.1 Implementation

One of the main advantages of SPSA relative to other optimization techniques is its relatively easy implementation. Still, there is quite a variety of adjustments that can be made to accommodate different problems. When implementing SPSA it is only natural to follow the steps of the algorithm outlined in section 3.1.

Steps one and two are completely trivial to implement once parameters and perturbation vector is decided on. Step three is a bit more cumbersome. Once the new positions where function evaluations shall be performed have been calculated, taking into account constraints, the program needs to write an input file that contains all relevant dimensions and parameters needed by FEDEM to run the simulation. That input file is sent to FEDEM, who is called to execute the simulations. FEDEM then writes its result to a series of files containing the time history for all joints in the structure. These files then need to be read by MATLAB, and then used by a rainflow-counting algorithm to calculate fatigue lifetime for all joints. The dimensions must also be used to calculate the weight, before both the weight and all joint lifetimes are used to calculate the objective function value.

In step 4, an estimated gradient vector is calculated based on the two function evaluations in the previous step. In step 5, the new estimate for the variables is calculated, and constraints are enforced before the algorithm jumps back to the top and start the next iteration.

Post processing was implemented by creating various plots and text-files containing critical performance measures.

Constraints

When working on structural optimization there can be many constraints that need to be complied with. In our case, we have two different types of constraints that need fundamentally different approaches in their treatment. First, we have constraints that place limitations on the variables of the problem. These constraints are relatively easy to comply with, as they are well defined, and we have full control over all variables. They are what we can call hard explicit constraints. Hard, because no variable can be taken outside the constraints, not even during the optimization process, and explicit because they are specified directly on the variables [8]. Secondly, we have constraints that place conditions on the *result* of the simulations. These are not as easily complied with, as we don't have a linear relationship between the variables (our input), and the results of the simulations (the output).

That means that we don't know whether the constraints have been breached until after the simulations are complete. These are what we can call soft implicit constraints. Soft, because the constraints can be breached during the optimization, as long as the final solution is ok. Implicit, because they are not placed directly on the variables. A way to deal with such constraints is to include them in the objective function. That way they will influence the direction the solution goes, and gradually as the iterations goes, the constraints will hopefully no longer be breached. In terms of offshore lattice support structures the lifetime of the structure is such a soft, implicit constraint. A more detailed description of how the lifetime is included in the objective function will be given in the next subsection.

Let's go back to simple constraints placed directly upon the variables. These might come from a designer setting some upper and lower limits for de dimensions, based on experience, production limitations or other factors. Others might come from more trivial reasons, like that the wall thickness of a member can't be larger than the radius of the member. This might happened if the upper limit of the thickness is larger than half the lower limit of the diameter.

The fatigue lifetime of the joint is calculated by a rainflow-counting algorithm that takes the stress concentration (SCF) factors in the joints as one of its inputs. These SCFs are calculated using formulas given in [1], and have some additional constraints. Figure 6 shows all the relevant dimensions, and these have to satisfy the following constraints:

$$0.2 \leq \beta \leq 1.0$$
$$0.2 \leq \tau \leq 1.0$$
$$8 \leq \gamma \leq 32$$
$$4 \leq \alpha \leq 40$$
$$20° \leq \theta \leq 90°$$
$$\frac{-0.6 * \beta}{\sin\theta} \leq \zeta \leq 1.0$$

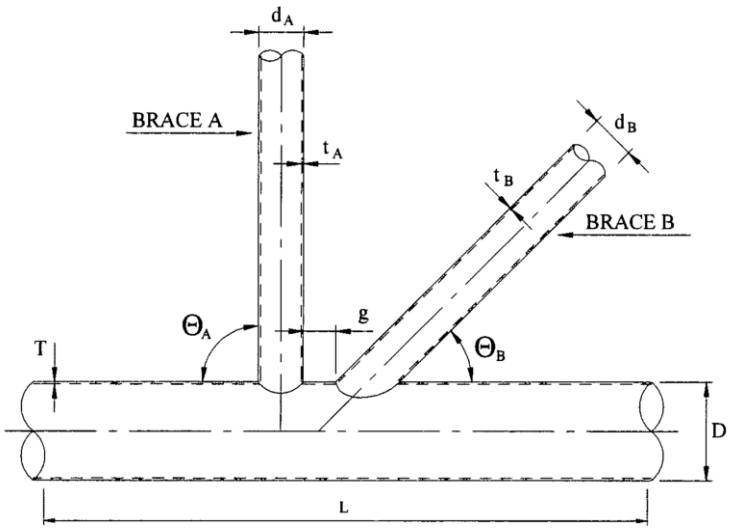

Figure 6: Joint dimensions [1]

where

$$\beta_A = \frac{d_A}{D}, \quad \beta_B = \frac{d_B}{D}, \quad \tau_A = \frac{t_A}{T}, \quad \tau_B = \frac{t_B}{T}, \quad \gamma = \frac{D}{2T}, \quad \zeta = \frac{g}{D} \quad \alpha = \frac{2L}{D}$$

When trying to comply with all of these constraints, you run into some challenges. If a step takes one of the variables outside the design space, different constraints yield different responses. If the breached constraint is a simple upper or lower limit, the variable can just be projected back into the design space. For our design, the minimum allowed member wall thickness is 0.005m. If the algorithm should try to use a lower value, say 0.004m, then the constraint would kick in and "project" the thickness back to 0.005m. If the breached constraint is that the thickness has become larger than radius, one has to decide which variable to change. During this study it was chosen to increase the diameter if this constraint was breached. If the bracing dimensions were getting bigger than the leg dimensions, it was

chosen randomly whether to decrease the bracing, or increase the leg. In both cases they would be increased or decreased such that the dimensions became equal.

So far we have discussed constraints breached when the algorithm moves one step forward between each iteration. Constraints may, however, also be breached by the small perturbations within each iteration. If this happens, one also has to consider what to do with the corresponding Δ-value. If a positive Δ-value causes the variable θ_i to breach a constraint, the situation can be temporarily saved by moving the variable to within the acceptable design space. However, if the algorithm finds that perturbation to be the most favorable, it will move in the direction indicated by the Δ-value, meaning it will try to breach the constraint again. Fixing this by changing both the Δ-value and the variable itself is not an option, as you will run into the same problem if the opposite perturbation turns out to be the best, and if you set the Δ-value to zero you will probably be stuck at that dimension. The result of all of this is that if the algorithm is trying to move outside your design space you just have to project the variable back in, leave the Δ-value unchanged, and hope it will move in the right direction next iteration.

P. Sadegh discusses constrained optimization with SPSA in [37]. Similarly to what was implemented in this study, he suggests simply projecting variables that move outside the design space to within the design space. However, to avoid the situations where the perturbations breach constraints, he suggest projecting such that $\Theta_k \pm c_k\Delta_k$ is within the design space, not just Θ_k. This allows the perturbations, who are $c_k\Delta_k$ long, to be executed without interference. Although this was not implemented in this study, it is not believed to affect the end results, as the most of the constraints in this study are relatively wide and are therefore seldom breached (with the exception that bracing dimensions have to be less than leg dimensions).

Constraining step size

When optimizing lattice support structures, there are two variables influencing each objective measure: thickness and diameter of the members. If both of these increase or decrease at the same time it will make a much larger impact on both the weight of the member, and the lifetime of its joint, than if the variables move in opposite directions. This is reflected in the objective function, which sometimes can have a quite large difference between the two corresponding function evaluations in each iteration. Naturally, this is not the only factor contributing to the varying differences of the objective function. The objective function includes lifetime measures for all the sections in the structure, in addition to the weight-term. Since the perturbations are random they will sometimes move in directions that cancel each other out, and sometimes move in directions that maximize the difference. Because of all of this, the step size between iterations can vary significantly and sometimes become inappropriately large. To avoid this it is possible to simply reduce a, but this will slow down the entire process. A better way was found to be to set a maximum step-size and thus filter out the extremes, while not affecting the majority of the iterations. This maximum value was tried to keep as high as possible, to not "disturb" the optimization process more than necessary, and was found effective in the 10-15% range of the magnitude of the variables. Other designs might have a different sensitivity and require a different maximum step size.

Dealing with variables with large differences in magnitude

When optimizing simultaneously for diameter and thickness of the members, some measures have to be taken to account for the very different magnitude of the diameter compared to the thickness. Spall suggests in [13] to use matrix scaling of the gain a_k if information on the relative magnitudes is available. This, however, will result in objective function evaluations that are heavily dominated by either diameter or thickness, since the perturbations are not scaled. The probable reason for this recommendation is that the perturbations should imitate an infinitesimal change of the parameters, like they would if calculating gradients based on other numerical methods, and would therefore not need scaling. To solely scale a_k would therefore not be wrong, but a scaling of both a_k, and c_k seams more intuitive. In this study it was therefore chosen to scale both the gain a_k, and the perturbation-controlling parameter c_k. In this way it is ensured that all variables have approximately the same relative changes, giving approximately the same contribution to changes in the objective function. Since the objective function returns scalar values, it is, as already mentioned, also necessary to scale the gain a_k. This is so the actual step is in accordance with the measurements taken using a scaled c_k. The amount of scaling was chosen to be 20 for the diameter, compared to 1 for the thickness. These values were chosen based on the initial guess used for most of the simulations, where the diameter was initially chosen to be 20 times larger than the thickness.

5.2 Objective function

The objective function is perhaps the most important factor for a successful implementation of SPSA. With respect to lattice structure optimization, it needs to serve two purposes. First, it needs to ensure that the result satisfies the minimum-lifetime constraint. Secondly, it needs to search for the most economical (lightest) solution satisfying that constraint. The fact that we have two objectives that are conflicting makes this a multi-objective optimization problem [9]. One cannot find one single optimum that satisfies both conditions simultaneously, leading to an optimization process that is more dependent on decisions made by the designer. He needs to design the objective functions in a way that balances the two objectives.

The choice of objective function is highly dependent on the problem it is intended to solve, and there is little published on how to choose it. During this work several different objective functions with several different configurations for the parameters have been investigated. Since we have two distinct objectives that should be handled by the objective function, a two term function was chosen, one term representing the lifetime constraint and one term representing the weight. Some of the considerations that were found to be important during this work were:

- How many lifetime values should be included per joint? For every joint there are eight different hot spots, all with lifetimes calculated. All of them can be summed up, or averaged, or just the smallest could be used.
- How many joints should be included? All joints can be included, or just those with lifetimes less than a given value.

Once decided on the two previous points, one needs to decide on which mathematical function to be used. This study has investigated function based on average, root mean square, root mean square deviation, curve fitted functions and sums. For those functions that don't naturally have the right sign of the slope, the inverse or the negative can be used.

Although a steep objective function for lifetimes below design lifetime is preferable to efficiently reach the allowed design space, it might increase the time it takes for the algorithm to find an economical design. This is because a steep objective function will require a small a, the parameter that governs the step size, to ensure that the step size is reasonable. This is of particular relevance if the starting point is highly under-dimensioned. An under-dimensioned design with a steep objective function will result in large differences in the objective function value for the two evaluations performed at each iteration. This again result in a large $\hat{g}$ which then requires a small a for the step size to be reasonable large. This problem can be circumvented by changing a during the optimization. The disadvantage of this measure is that it will require more work by the user, and the parameters will be more dependent on the initial starting position. A better objective function will be one that is just slightly steeper in the low-lifetime area than it is for sufficient lifetimes. Of course, the sign of the slopes will have to be opposite, making a minimum where the constraint meets the wish for low weight.

The term representing the weight in the objective function does not have that many obvious functions to choose from. It would either be a linear function, so that all changes to the weight are equally important, or a convex function to speed up improvement when the weight is high relative to the expected outcome.

Generally, it is important that the function, and especially the lifetime term can only reach its minimum if all its inputs are also at its minimum. This means that if some sort of average or root mean square value is to be used, care needs to be taken to ensure that the minimum is unique. One cannot just average the lifetime values, because a given positive target value can be reached even though several members have to low lifetime, as long as there are some that outweigh them with longer lifetimes. To circumvent this problem one can e.g. simply average over all values less than the target value (If one are averaging values between 0 and 1, an average of either 0 or 1 can only be achieved if all inputs are at the same value), or one can use the root mean square deviation instead of regular root mean square.

When trying out a new term in the objective function, one might upset the balance with the other terms. This means that for every new lifetime term that should be investigated, several different slopes for the weight term might be tried out. The same goes for the other parameters in the algorithm. When trying out a new objective function it most likely is required to adjust the gain sequence, etc.

To find a good objective function is absolutely critical for a successful and efficient implementation of SPSA. Although it is time-consuming to find a good function, there is reason to believe that once a function is obtained, it can be reused for other, similar structures. If applying to a structure of significantly different size it might be necessary to change the slope of the weight-term, but that can be done relatively easily. The lifetime term, on the other hand, is only dependent on the normalized lifetime of the members, and is thus independent of the size of the structure.

Recommended objective function

After investigating a large variety of objective functions (see appendix A, and appendix B page: B13–B20), the one that proved most successful was:

$$\left(\frac{weight - 1200}{50}\right) + \sum_{NLT_i<1} ((NLT_i - 1.25)^2 + (NLT_i - 1.1)^{20})$$

$$NLT = Normalized\ member\ LifeTime$$
$$weight\ in\ metric\ tons$$

where the first term represents the weight, and the second the joint-lifetime constraint. The slope of the weight term was adjusted to the lifetime-term, and was found to be suitable at $\frac{1}{50} ton^{-1}$. The subtraction of 1200 in the weight-term has no other purpose than to shift the objective function towards zero at optimum weight. The lifetime-term sums over all members with a lifetime less than design lifetime. It takes the lowest lifetime within each particular member and inserts that value in the function above. (The code does lifetime checks for several critical positions for all members.) Using this function, the lifetime term dominates the weight term for all insufficient lifetimes, while it is simply a linear function of the weight whenever all lifetimes are sufficient. The lifetime term consists of two individual terms. The first of the two is the most dominating for the majority of NLT values, while the second ensures an increased slope should the NLT values become very small. It does not, however, go to infinity if the NLT value goes to zero, and this is because there is nothing to gain on that. The algorithm can only handle a certain limited step-size before it does more damage than good, thus making it more favorable with a bounded objective function that ensures controlled reasonably sized steps. Figure 7 shows the contribution on the objective function, with respect to weight, for a single-variable problem, and Figure 8 shows the contribution on the objective function from the lifetime term as a function of lifetime, for a single-variable problem. How the objective function looks with respect to member size, which is our main variable, has no easy visualization.

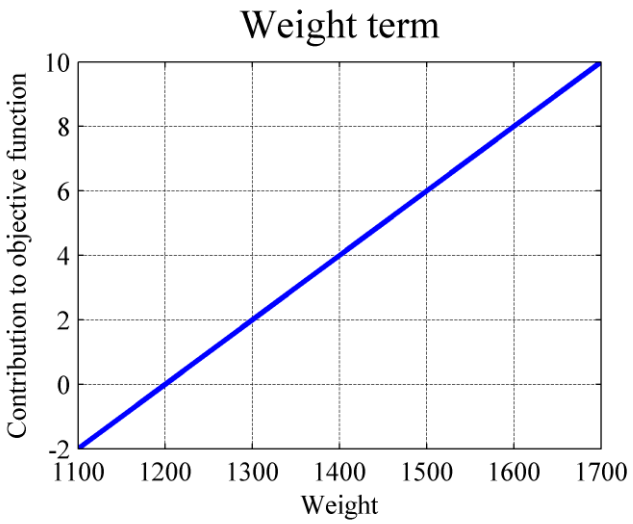

Figure 7: Weight term of objective function

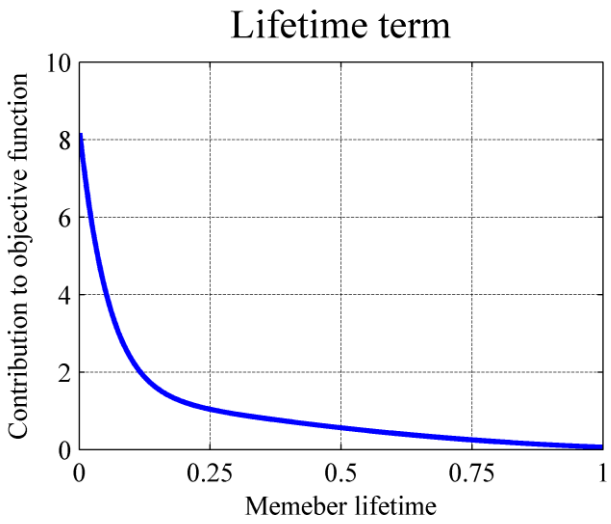

Figure 8: Lifetime term of objective function. 1D example

A weakness with this function is that there is no theoretical justification behind its shape. It has been developed based on experiments and trial and error and chosen based on the author's understanding of a good performing function. Its origin and inspiration is that of the calculation of root mean square deviations (RMSD). It started out as a standard RMSD, but a wish for a steeper function resulted in the removal of the square root that normally surrounds the sum. Also, to put some additional pressure on the lowest lifetime values, the second term of the sum was added. This term has negligible effect as long as the normalized lifetime is above 0.25. The most controversial aspect might, however, be that we are summing over all normalized lifetime values less than one, but the function takes the square of the normalized lifetime minus *1.25* (instead of just 1.0). This results in a small jump in the objective function every time a member moves from sufficient lifetime to insufficient, or the other way around. The jump is quite marginal at 0.0625, and was included to give members that are almost strong enough an extra push to the right side. Whether it improves speed is, however, uncertain, as it might cause undesirable interruption to the process. Others might find better functions, but this one was found to perform satisfactorily, and was therefore found to be suitable for the study in question.

5.3 Performance

The effect of the *c* and γ parameters

Once a potentially good configuration was identified, several runs were performed with varying values for *c* to determine what perturbation width is the most favorable. This is of course also highly dependent on what other choices have been made throughout the implementation, but for the objective function described on the previous page noticeably performance gains was observed with $c = 0.001$, compared to the runs with *c* equal to respectively 0.0005 and 0.002. Figure 9 shows how the weight decreased using the three different c-values: 0.0005, 0.001 and 0.002. The general behavior is similar, but we clearly see that the weight decreased faster using $c = 0.001$. This value did also show the best performance with respect to acceptable joint lifetime. These tests were performed under completely identical conditions. Also the perturbation vectors were identical, something that gives us a reasonably fair comparison. However, as we will see next, repeated simulations with identical configurations (but with varying perturbation vectors) are necessary to say anything conclusive on the performance and effect of various parameters.

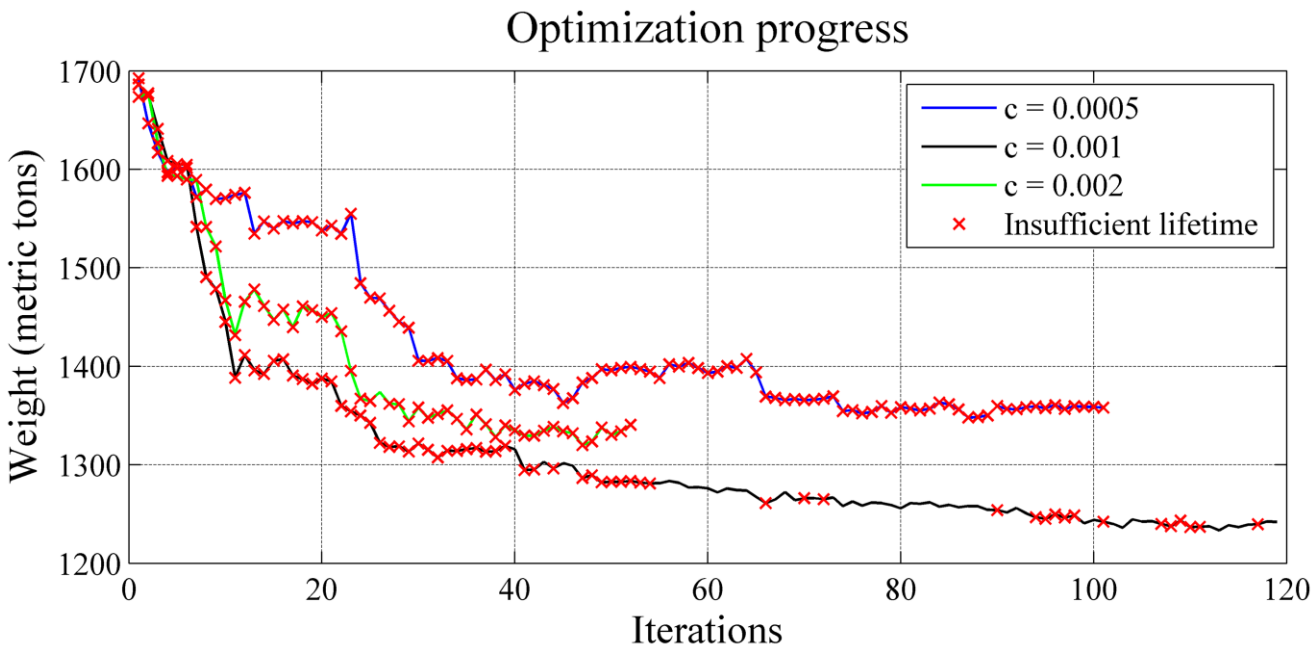

Figure 9: Optimization progress for varying *c* values

For the optimization performed with starting point at a low weight (see below), we observed some significant performance differences when adjusting the γ-parameter. γ controls how fast the perturbation width c_k decreases as the iterations increase, and was not expected to have large influence on performance. When using the asymptotically optimal γ, γ = 1/6, we observe a steady decrease of the structural weight throughout the process, while Spalls recommended γ value, γ = 0.101, levels out already after 20 iterations. Figure 11 shows c_k values for the 45 first iterations with different gamma-values. Figure 10 shows how the development of the weight when the only change was the gamma-value.

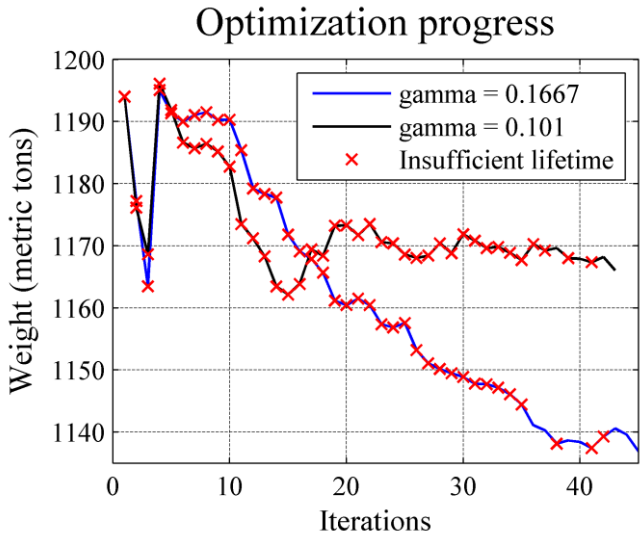

Figure 10: Optimization progress for varying gamma values

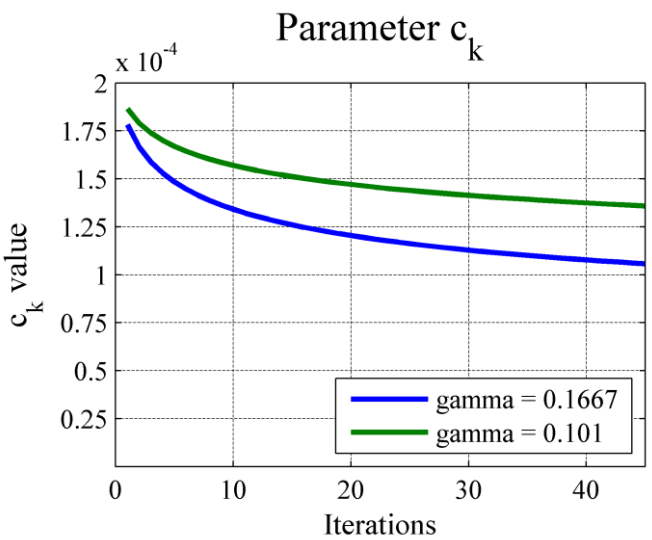

Figure 11: Perturbation width, c_k, for varying gamma values

There is of course a significant difference between the two γ-values evaluated, but the steps are in both situations usually much bigger than the perturbations, and the ĝ-value is almost unchanged because the c_k in the denominator compensates for the differences in perturbation width. As for the investigation of the effect of *c*, everything except the γ-parameter was identical, including the perturbations, so the reason for the big difference in behavior was therefore somewhat unclear. Further investigation was therefore performed to determine whether this result was due to pure bad luck, or if it could be reproduced for other perturbation vectors. Normally, the perturbation vector will be unique for every single optimization run. In this study the perturbation vector was controlled by setting a known seed in the random number generator creating the perturbations, so that parameter could be compared under identical conditions. Two more optimizations was performed using γ = 0.101 and otherwise equal to the one that leveled out above, but now with different perturbation vectors. These optimization runs did not experience the same problems that were just identified. Figure 12 shows that these new optimizations behave in a similar manner to the first optimization that used γ = 1/6. A second optimization using γ = 1/6 was also performed. That one showed a similar unusual behavior, but in the opposite way; it performed drastically better than the others. From this we can learn several things. First, repeated identical optimizations must be performed for all configurations to be able to say anything meaningful about the relative performance. Secondly, even a well calibrated and functioning optimization configuration risk drawing unfortunate perturbation vectors, drastically reducing its performance. Though this might increase the time spent before a good solution is obtained, there is reason to believe that such an unfortunate series of perturbation vectors will not exist for very long, thus allowing the optimization to finally reach a comparable solution sooner or later.

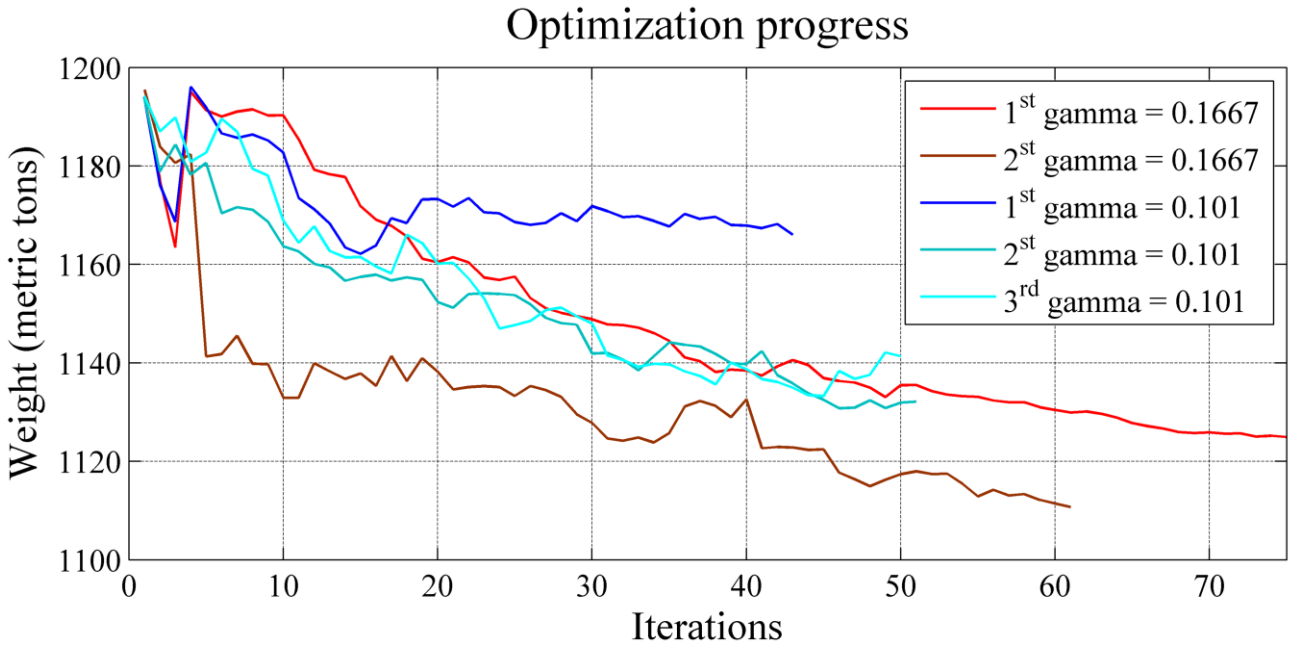

Figure 12: Optimization progress. Two identical runs with gamma = 0.1667, and three identical runs with gamma = 0.101

The effect of the *a* and α parameters

a was the parameter that was most frequently adjusted during this work. *a*, α and *A* are the parameters controlling step size. For our particular structure, an appropriate step size was found to be maximum 0.002m (remember, this is the step for the thickness. The step for the diameter will be 20 times larger after scaling). With the selected objective function and parameters *A* and α set to 15 and 0.602 respectively, this called for *a* to be approximately 0.000025. The effect of changing *a* is completely linear, and therefore straightforward; by doubling *a* the step size doubles as well. As the step size had a tendency to drop quicker than desired, the α parameter was usually kept at 0.602, the lowest value recommended by Spall. A higher α would cause the step size to decrease more rapidly.

The effect of the *A* parameter

The *A* parameter found in the gain sequence $a_k = \frac{a}{(1+A+k)^\alpha}$ is often even left out of the algorithm. During this study it was found to be crucial to the performance of the optimization. During the initial optimization runs a value between 2 and 4 was often used. This was done partly because it was not clear how many iterations were actually needed to find a solution, and partly because it was not clear how important the parameter was, and it was thus not given any particular attention. As it became clear that a full optimization would require a three-digit number of iterations, the *A* parameter was increased to 15 (Spall recommend setting A to 10% or less of the expected number of iterations). The reason why this *A* parameter is so critical is simply that without it, a_k becomes unmanageably large during the first couple of iterations, and then to avoid instability, *a* has to be reduced. This results in very large movements initially, before it rapidly calms down such that the movements are actually too small. By introducing *A* the step size will decrease at a slower and steadier pace, increasing performance at late iterations, while not risking instability in the initial phase.

«Locking»

With a total number of variables of 48, and low-noise function evaluations, convergence should be possible within a reasonable number of iterations. Each iteration is, however, very computationally expensive, resulting in a high time consumption before a converged solution is reached. Even though a short time domain of 120 seconds was used, each iteration took approximately 35 minutes on a regular desktop (Intel Xenon X5550 at 2.67GHz). We have seen most of the optimization runs run for more than a hundred iterations without even being close to convergence, which means that a full optimization can easily take up to a week to finish. The speed could of course be improved somewhat on a state-of-the-art computer, but as long as the simulation software, in this case FEDEM Windpower, does not benefit from multicore-processors the optimization is limited to parallelization of the two concurrent function evaluations, resulting in long simulation times. This has led to the investigation of techniques for improved convergence-rate.

Motivated by the desire for improved optimizing speed, small modifications to the algorithm have been investigated. Key behind these investigations is the assumption that low, but

acceptable joint lifetime is a sign of close to optimal utilization of the structure. Assuming such an optimal utilization means that the weight is minimized with the current loading, we can let the members in question skip the part of the algorithm that takes one step forward, thus avoiding they step out of their assumed favorable dimensions. More and more members are "trapped" by this locking mechanism as the iterations go, making it easier for the rest of the members to improve their utilization, without ruining the "good" result already obtained.

This method is, however, not without consequences. One of the motivations behind this study was to find a mathematically well-documented method for optimization with respect to weight. When implementing these restrictions on the free search, you get closer to the algorithms already in use, thus taking away some of the arguments for this new method. However, it is possible to relax these restrictions during the optimization, allowing "free search" once it closes in on the assumed optimum.

Some results from this investigation are shown in Figure 13. They indicate that some increased performance can be achieved in the early iterations. However, the comparison in Figure 13 is not entirely fair, as some of the parameters were different. Especially the A parameter, that were 4 for the simulations with locking, and 15 for the one without, might have had a big influence on the behavior for the early iterations. One can also observe that the optimization runs with locking are faster to reach solutions with acceptable lifetimes. This is natural, as with the current implementation, the penalty on lifetime only goes into effect at the moment the lifetime actually becomes insufficient. This means that all members with sufficient lifetimes, even the one who are practically at the border, will move towards the insufficient region to try to reduce the weight until they actually breach the lifetime requirement and are pushed back. The locking feature will restrict those members from moving. For implementation of the method without locking, it might therefore be smart to punish lifetimes, $NLT < NLT_{cr}$ where $NLT_{cr} > 1$. This way members right at the border would have no, or less, incentive to reduce the lifetime further. Enabling locking shows some interesting results. Unfortunately, time did not permit redoing the optimization under equal conditions, and running enough simulations to give conclusive statements about the performance. It would be especially interesting to see if one can maintain the good performance one see initially, so that there is a real performance gain all the way till convergence (maybe with higher a and A).

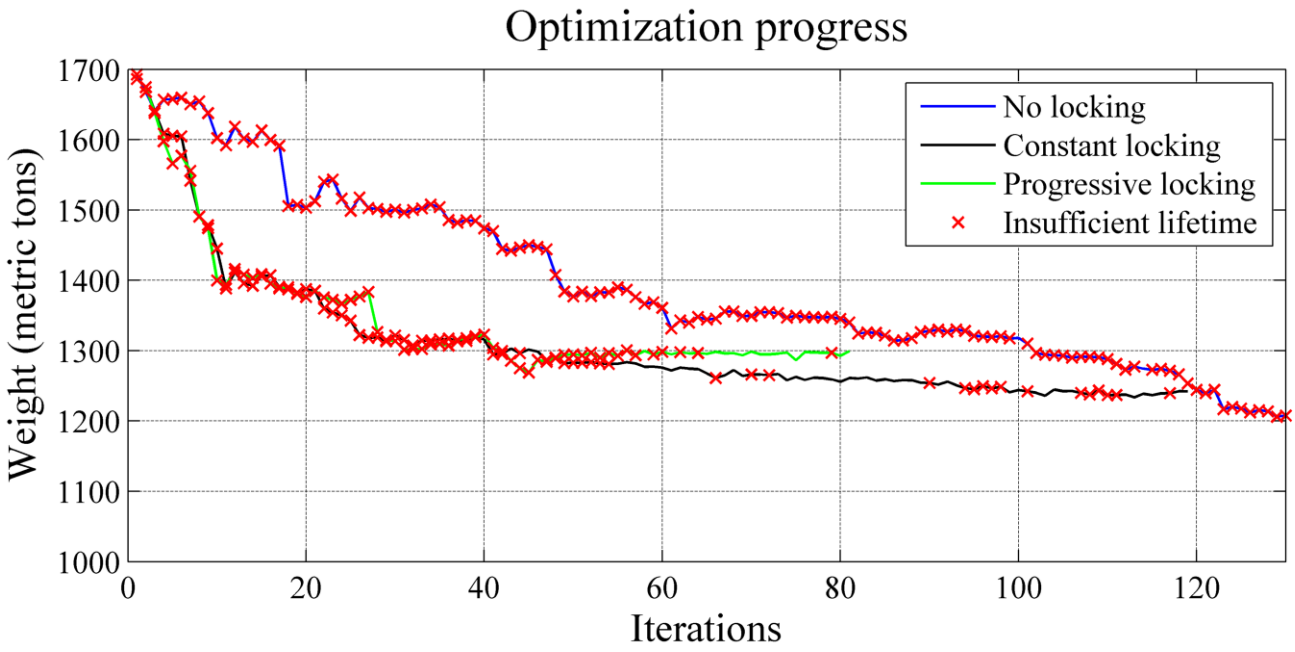

Figure 13: Optimization with, and without locking.

Different starting points

If the SPSA method should be utilized on a large scale in commercial software, it would need to be relatively robust with respect to the initial guess. Since it is in fact a guess, it would not be desirable to put too many restrictions on it. A brief study was performed to investigate how the SPSA-implementation reacts to different starting points, and how it affects the time it takes to find a solution. The initial guess used in most of the simulations in this study had equal dimensions for all legs, and equal dimensions for all bracing. This led to highly under-dimensioned members at the lower half of the structure, while the upper half was over-dimensioned. To investigate the effect of starting point, two additional optimization runs were performed with different starting positions: one where almost all members were highly over-dimensioned, and one starting from a configuration where all members had an estimated lifetime between 1 and 1.5. The highly over-dimensioned guess had an initial weight of 3250 tons; the initial guess with some over-dimensioning and some under-dimensioning (mixed starting point) had an initial weight of 1700 tons; and the guess where all members had a lifetime between 1 and 1.5 had an initial weight of 1100. The three starting points can be seen in Figure 14, Figure 15 and Figure 17 respectively. As we can see from Figure 16 there are no surprises in the different behaviors. When there is a lot of excess weight, it allows for some increased speed, especially in the beginning when step sizes are large, but it is not enough to counterbalance the fact that there is a much larger potential for optimization that requires more iterations to reach a good design. A very large number of iterations would therefore be needed to optimize the over-dimensioned staring point.

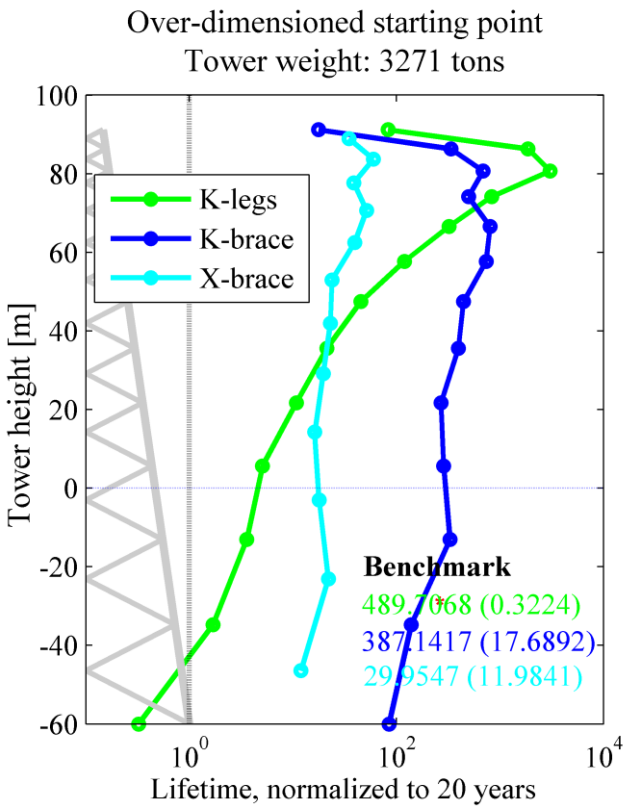

Figure 14: Minimum lifetime values for all sections, over-dimensioned starting point

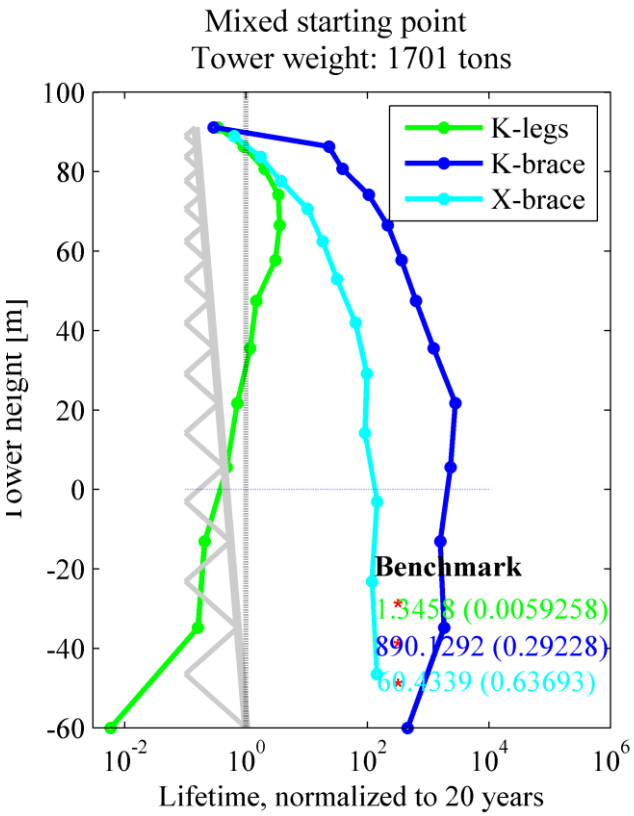

Figure 15: Minimum lifetime values for all sections, mixed starting point

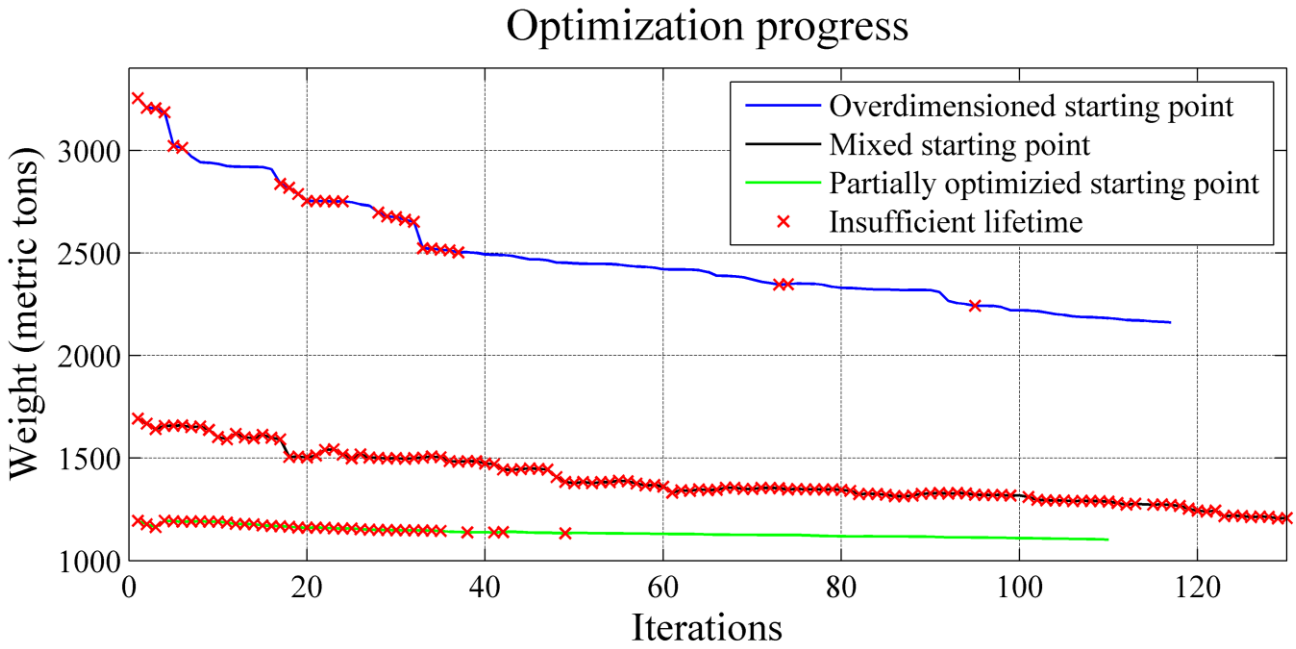

Figure 16: Optimization progress with different staring points

The Minimization of Lifetimes method

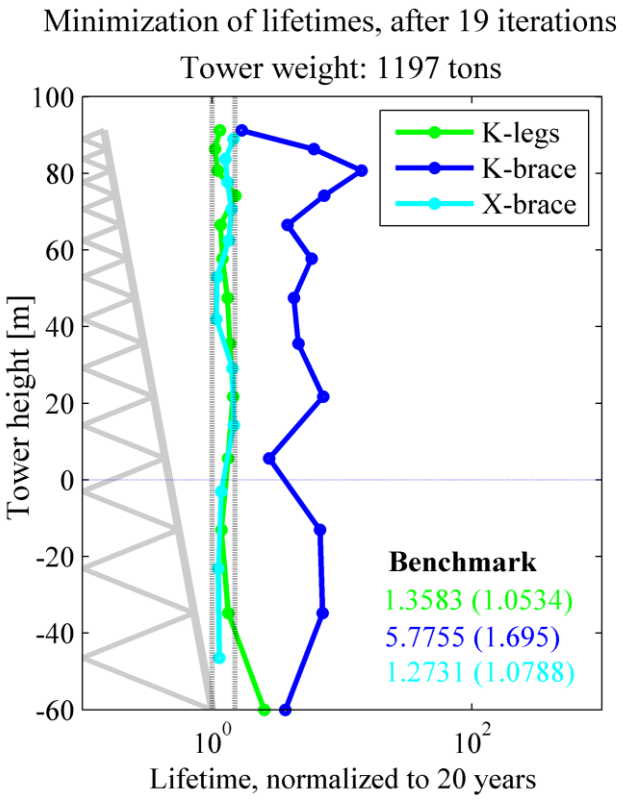

Figure 17: Minimum lifetime values for all sections, best result obtained with Minimization method

As previously mentioned, the SPSA method investigated in this study was implemented on top of an already existing program for optimization of lattice support structures [25]. That program is still a work in progress, but the current best results will be used here for comparison. The method used in the program is not based on any particular mathematical optimization technique, so we will just call it the "Minimization of Lifetimes" method. It is like the SPSA method based on an iterative approach. One time-domain simulation is performed per iteration. Based on these results the lifetime values for each node are evaluated. If a joint has an insufficient lifetime, its dimensions are increased a predefined amount, with a preference on increasing the thickness before the diameter. This is because studies show that an increase in thickness has a better ratio between increased lifetime and increased weight than that of increasing diameter. If a joint has an unnecessary high lifetime, its dimensions are reduced a similar predefined amount. This process is continued until all joints have a minimum lifetime within 1 to 1.5 times the design lifetime. Nowhere in this algorithm does the weight affect the result; it is just assumed that the weight will be at its lowest when all joints have a high utilization. As shown in Figure 17 this method produces a structure, with all joint lifetimes above the minimum, at 1197 tons. This is done in just 19 iterations, making it a very fast method. The method does not, however, search for other solutions in the vicinity that might utilize the material even better. The SPSA method is not capable of providing a solution in this few iterations, due to the random approach of the SPSA. Many of the changes of the variables in SPSA will actually make the structure worse than it was, but given enough iterations, these will be canceled out and you are left with a good design. Figure 18 show the optimization progress for the Minimization of Lifetimes method, compared to that of the SPSA method. Clearly, the convergence rate is not even comparable; the Minimization method reaches its solution in only 19 iterations, while the SPSA method uses 10 times as many, in addition to using twice as many function evaluations per iteration. But as we can see, the SPSA method is able to find solutions with significantly lower weight. Both methods clearly have their strengths and weaknesses, and to have a repertoire of different methods, with different characteristics, can be very useful to the designer. If he wants a quick decent solution, the Minimization method is a good choice, while he would need to use the SPSA method if he

wants to find the best possible solution. In the next section we will take a look at how these two methods can be combined for an efficient and accurate optimization.

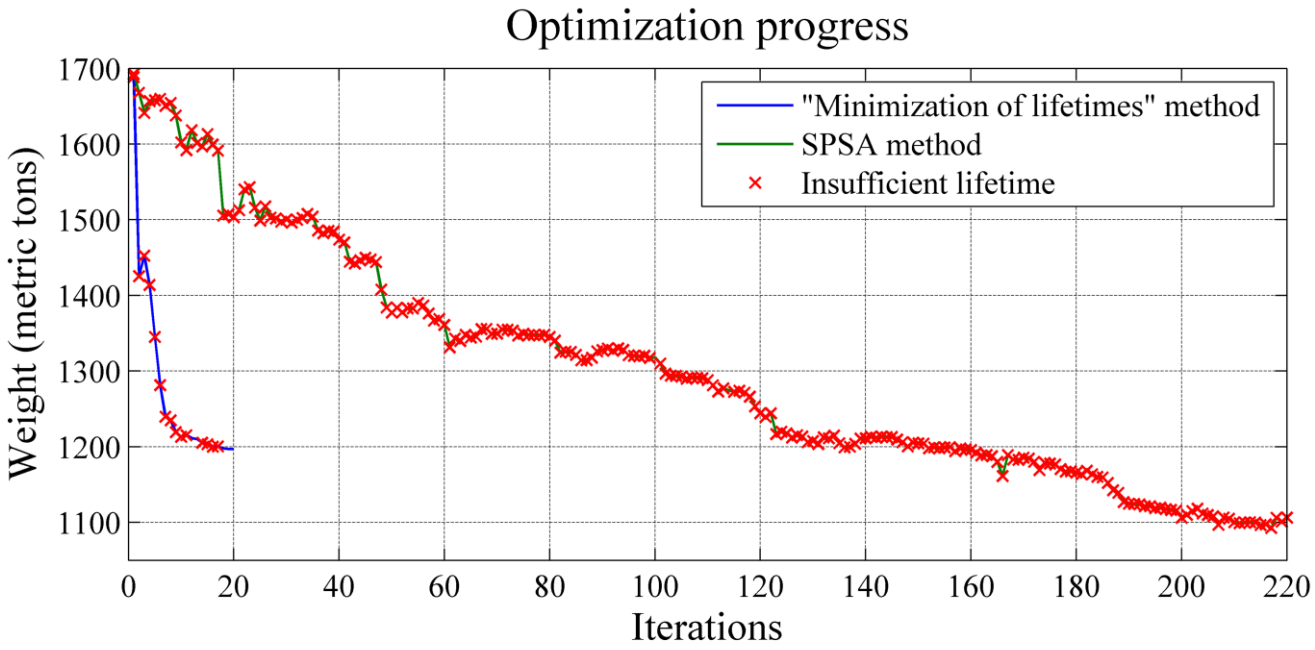

Figure 18: Optimization progress for the Minimization of Lifetime's method compared to that of the SPSA method

Starting from a «good» configuration

To investigate whether the SPSA can further improve on the solution from the Minimization of Lifetimes method, an optimization run was performed from the best solution obtained by that method. Because it was believed that the starting point was relatively near the optimal solution, both step size and perturbation width were reduced compared to a "normal" optimization. This is because in a "normal" optimization run, the algorithm would have taken several tens of iteration to reach a solution comparatively good to this starting point, thus giving the algorithm time to reduce a_k and c_k such that step size and perturbation width would be similarly small.

The result of this investigation was a structure that was more than 5% lighter than what was previously achieved within 40 iterations, and a full 8% lighter after 110 iterations. Figure 19 shows how the weight and the objective function decreased as the number of iterations increased. As we can see from the graphs, there is no sign of convergence as none of the graphs show signs of leveling out at the end. This means that if the optimization were allowed to run longer, it would probably improve the result even further. (Practical reasons limited the amount of time available for simulation.)

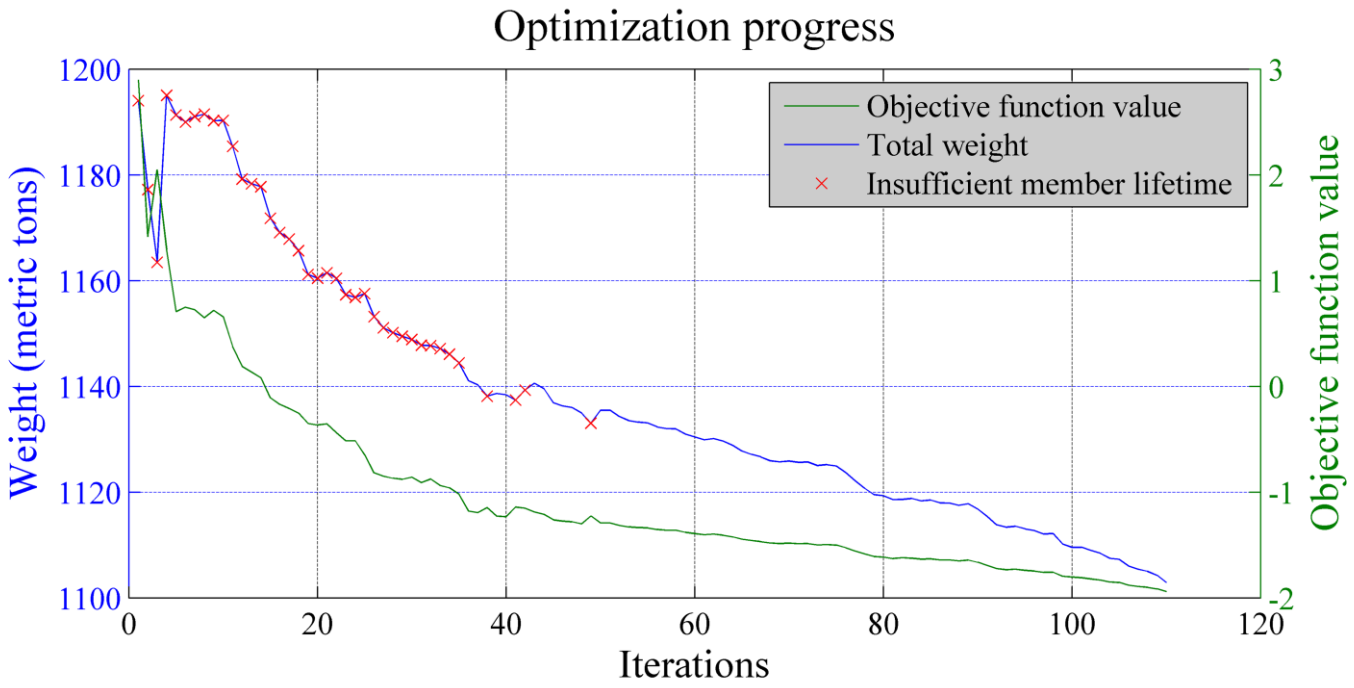

Figure 19: Optimization progress for weight on the left and objective function on the right, staring from the best result obtained by the Minimization of Lifetime's method

This is indeed a very interesting result. First, it shows that SPSA's ability to find good structural designs is unquestionable. But as we will see next, these results surpass what was achieved even after three times as many iterations, when the starting point was not equally good. It is therefore clear that SPSA is not a method that can be used without care. To go from an unfavorable starting point to a good final design requires an unserviceable amount of iterations; a good starting point is therefore critical if efficiency is important to the designer. A bad starting point does not only represent a challenge because of large time consumption and tie up of computational power; it is also more challenging to find appropriate parameters for such long runs. Parameters α and γ control how fast perturbation width and step size decline as a function of the number of iterations, and more effort would need to be used to calibrate these parameters if long runs (100+ iterations) are expected.

It is also interesting to see that the lightest design currently identified is not the design where all member lifetimes are shifted as close to the design lifetime as possible. (Compare Figure 17, which is the result of the Minimization of Lifetimes method, to Figure 20, which is the result of the SPSA method.) However, this does not necessarily mean that lightest possible design does not have most of the member lifetimes equal to the design lifetime. The only thing we can conclusively say is between the two best results currently obtained, the one with the lowest lifetime values is not the lightest. This can have several reasons. The lightest design might have more favorable ratios between thickness and diameter, resulting in lower stress concentration factors. It might also have a more favorable distribution of the load between legs and bracings, something that is not easy to account for in the more "manual" Minimization of Lifetimes algorithm. There might also be a more favorable ratio of the dimensions between adjacent sections. A major strength of the SPSA method is that it doesn't

need to consider factors like this. It simply tries two different designs, and moves towards the lightest. It is, however, reason to believe additional weight savings are possible if the utilization of this improved design can be improved. In other words, by shifting all joint lifetimes closer to the design lifetime, while still preserving the favorable ratios identified by the SPSA method. Figure 21 illustrates how the dimension has been changed by the SPSA algorithm.

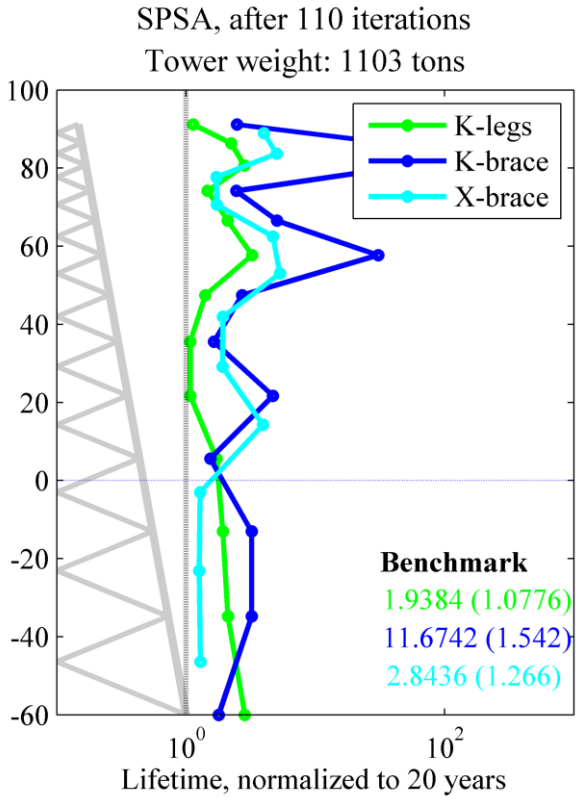

Figure 20: Minimum lifetime values for all sections, best results obtained with the SPSA method

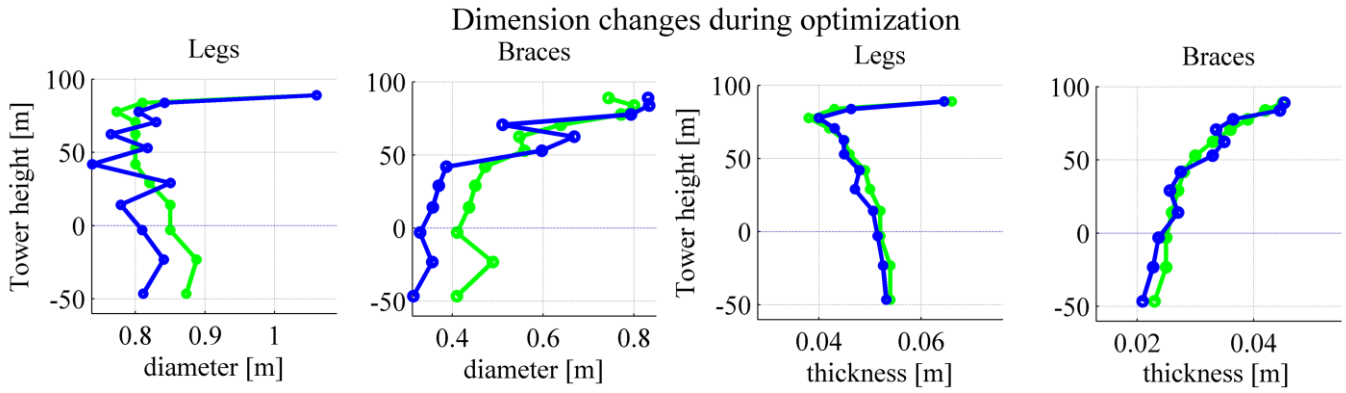

Figure 21: Dimensions before (green) and after (blue) optimization

A full-length optimization

During most of this study, optimization runs were not allowed to run for more than between 100 and 150 iterations. To get an impression of the behavior in the later part of the optimization process, one optimization run was allowed to run for more than 300 iterations. Based on the accumulated experience from the other simulations, the following parameters were chosen:

c = 0.0005 a = 0.000025 $\gamma = 0.1667$ $\alpha = 0.602$ A = 15

These parameters were not chosen to be exceptionally fast in the beginning, but more to emphasize consistent performance throughout. As we can see in Figure 22, the weight decreases steadily for almost 200 iterations. The objective function decreases a bit more rapidly in the beginning, when the worst lifetimes are filtered out, and then flattens out gradually as the number of iterations increases.

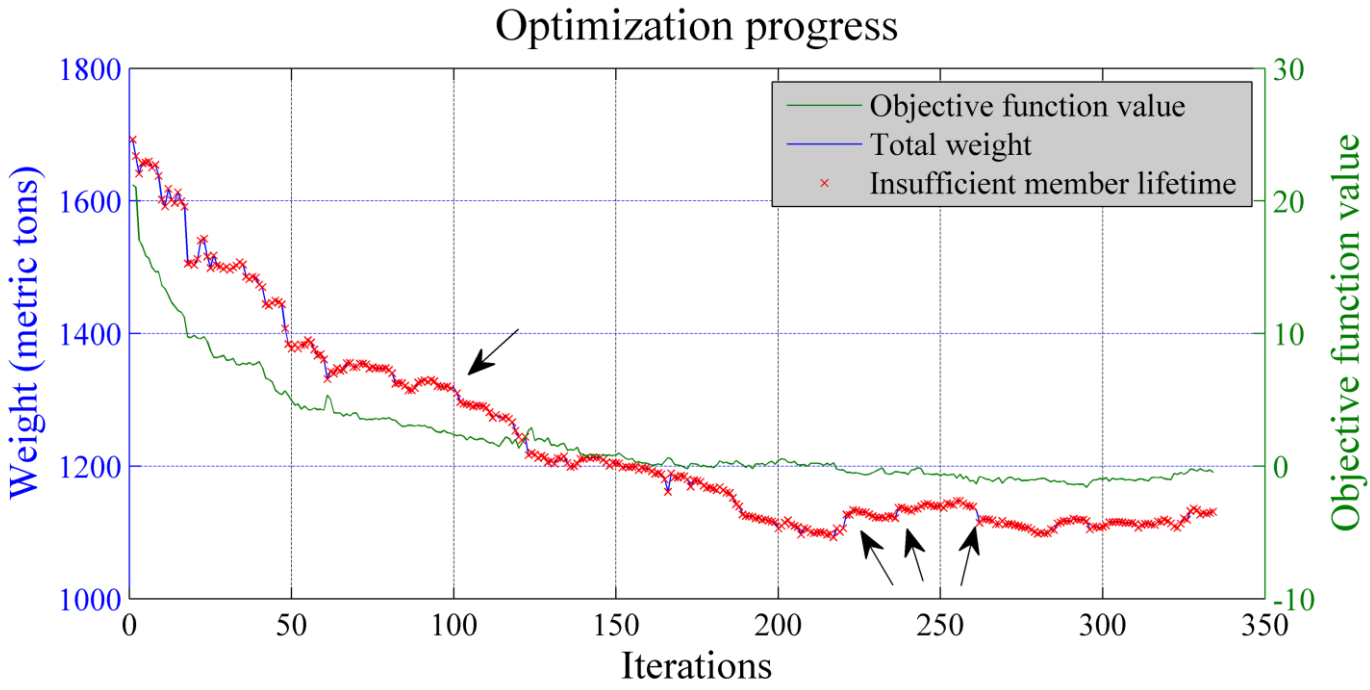

Figure 22: Optimization progress for full-length optimization. Arrows point to approved designs

Unfortunately, almost none of the iterations provide acceptable design in terms of lifetimes. One design around hundred iterations, and a couple at the end, are all that meet the necessary lifetime requirements (indicated by arrows in Figure 22). A probable solution, as already mentioned, that would help on this situation would be to set the minimum of the objective function a bit into the approved lifetime domain. For even comparison to the other plots, this was not done here. For the majority of the iterations, however, the members breaching the constraint were few and had only barely too low lifetimes, so they could easily be adjusted manually if that was necessary. On the positive side, we can see that if the optimization is allowed to run long enough, the solution is actually getting very close to what we achieved when we started at a much better starting point. One of the approved designs towards the end

had a weight of 1132 tons, only 28 tons behind what was achieved when starting from a 1200 ton starting point. Some of the other iterations were even lighter, but then with some of the members slightly under the design lifetime. Figure 23 shows a curve fitted with a second order polynomial to the weight measurements. It demonstrates the tendency of the weight and looks to be converging towards just over 1100 tons. This gives an indication of what it takes for the method to converge, but the results shown here, with little change after 200 iterations, do not translate directly to other cases. A different starting point might require more or less iterations to converge, but it shows that convergence is possible, and can be expected within a three-digit amount of iterations. Even though this proves that given enough time, the SPSA can converge towards a good solution, it is still reason to question whether any designer would have the time to wait for so many time-consuming iterations to finish before he gets his results.

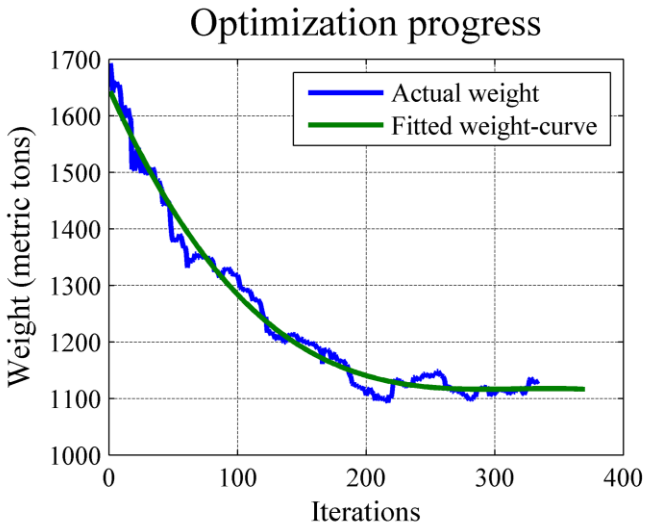

Figure 23: Green curve showing the tendency of the weight (blue)

5.4 Accuracy

In Figure 24 the member dimensions for the 1132 ton design found in the full-length optimization is shown. With the exception of brace thickness, there are very few similarities with the dimensions shown in Figure 21, which are the final dimensions after optimizing from a good starting point. There is 28 tons separating them, and neither of the optimizations was fully converged, but the difference is so significant that it seems unlikely that they are converging towards the same solutions. These are interesting results that might indicate that there are several local minima's in the design space. A more thorough investigation would be necessary to determine whether the method will converge towards the same solution, independent of the starting position and parameters chosen (for the same objective function). This has unfortunately not been possible to carry out, as it would require optimization runs to run until they converge, and during the study it became clear that that would require more iterations and more computational time than what was possible within the assigned timeframe.

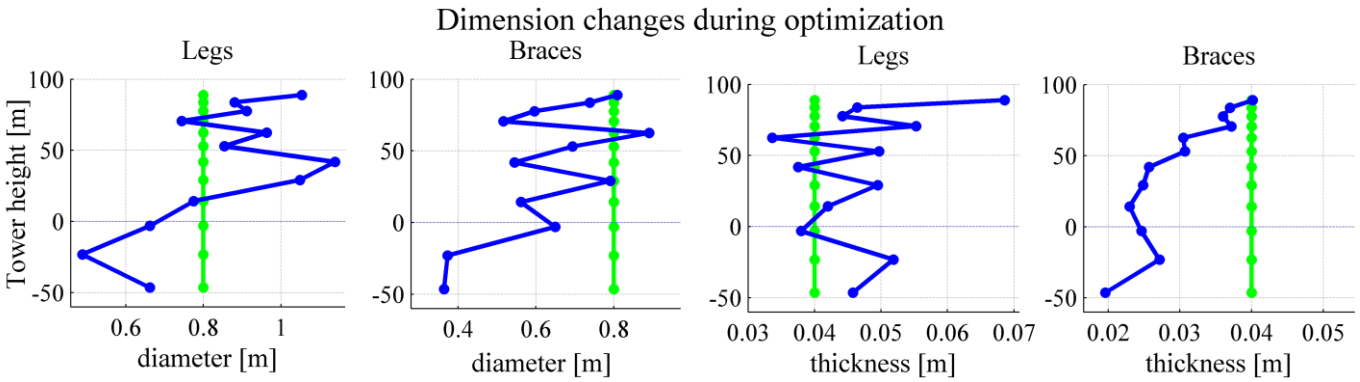

Figure 24: Dimensions before (green) and after (blue) optimization

6. Discussion and conclusion

6.1 Discussion

The SPSA method has some appreciated qualities, and has shown promising results in optimizing structural design of offshore lattice towers. It is, however, not given in which stage of the design process it has the greatest potential. To rely solely on SPSA to find a good design from a very rough first guess can prove inordinately expensive in terms of computational effort and time consumption. Yet to find a decent first guess does not necessarily require very much work, and can be time well spent. Despite good final designs with the SPSA method, in terms of computational time it seems difficult to compete with the Minimization of Lifetimes algorithm. That algorithm does take some shortcuts, which results in slightly lesser optimal results, but it is superior in terms of computational time. Not only does it produce good results after few iterations, it also makes do with only one function evaluation per iteration. A very promising procedure will be to first use the Minimization algorithm to find a "refined guess" from which an SPSA optimization can be performed. This will make the work easy for the designer, as he can make a first starting point without giving much thought to its position, since the minimization algorithm can take almost any input and bring it to a reasonable good design within a few iterations. Then the final optimization is left to the SPSA method, who initially rapidly improves the design, but can also pretty much keep improving as long as it is allowed to run, albeit at a lower rate. An experienced designer should also, by using a couple of manual trial simulations, be able to produce a decent first guess. Then this can be improved further directly with the SPSA method, allowing for a simpler code.

A concern with doing optimization on a detail level this high is the increased complexity in production. Unless adjustments are done after the optimization, there will be a large number of members where almost all have different dimensions. This will complicate the logistics related to production. Some economy of scale is of course possible for large wind farms, given that many of the turbines are optimized for the same site conditions. But with more efficient optimization techniques it is possible to optimize each individual structure, at least if there are differences in water depth, etc. This could again result in every single structure being unique, with its own dimensions on every member. Taking the optimization to the extreme is therefore not necessarily a smart move, and a thorough economical evaluation is necessary. This is regardless outside the scope of this thesis, where total weight of the structure is used consistently as the indicator of cost.

All simulations in this report have been on just one load case. How to optimize a structure that satisfies all the hundreds of different load cases suggested by IEC and other design standards is a major challenge. Given the time consumption spent to run one SPSA optimization, it is clearly not practically possible to do this for all load cases. Some load cases are naturally more demanding on the structure than others. An idea could be to identify some of the more challenging load cases, and run the optimization using SPSA on them. This will make a good foundation for other techniques to adjust the design if it proves insufficient under other loads. The idea would be to use SPSA (perhaps in a combination with the Minimization of Lifetimes method) to come up with a good foundation for further development, not to use it as the concluding stage of the development. More thorough analysis of this problem is suggested as future work.

This report has focused exclusively on fatigue limit state (FLS) lifetime. This was done because the load case that was used was a load case that typically leads to a design driven by fatigue. However, this is not the case for all load cases, and for those situations there is no problem using the same method to optimize with respect to ultimate limit state (ULS). The code written in this study does even extract ULS-values, and can easily be reconfigured to use those for optimization instead of the FLS values. It would also, with some modifications, be possible to use both the FLS and the ULS values simultaneously as input to the optimization.

Results shown in this report is based on a limited number of simulations. To accurately study the effect of the various parameters, a systematical approach with many more simulations would be required. It has therefore not been possible to give conclusive recommendations on values for the parameter. Optimal parameters might not even be possible to obtain, as both the perturbations and the performance will change from time to time, and the best parameters one can find is those who perform best in average. However, the result does very clearly show the potential of the method and could therefore be seen as a "proof of concept". Based on the many simulations that have been performed, a good understanding of the workings of the method has been achieved, and when the underlying data has been insufficient for conclusive results, further investigation has been suggested.

6.2 Conclusion

This paper has shown that simulation- based optimization using Spall's SPSA method is indeed a viable technique for optimization of lattice support structures for offshore wind energy. The method has been implemented and tested on a full height reference tower, and it has, through several examples, been shown that the method is able to find solutions that are on par with or superior to those of algorithms currently in use. The greatest concern with the method is the number of iterations it requires to reach a solution. The per-iteration performance is much lower than alternative methods, e.g. the Minimization of Lifetimes method developed by D. Zwick. However, if the initial starting point is well thought out, significant savings, in terms of iterations used, can be achieved. It is therefore suggested to use other methods (e.g. the Minimization of Lifetimes method) to come up with a refined starting point, and then use the SPSA method for further optimization.

6.3 Future work

Although this study has illustrated and proved some important aspects of use of the SPSA method in structural design of offshore wind turbines, there are still unanswered questions waiting for additional research. Here are some suggestions for future work:

- Include more variables. Number of legs, number of sections, variable section height, bottom leg distance and top leg distance are all variables with major impact on structural performance. It would be interesting to see if these variables can be included in the same optimization process, and how that affects the performance of the optimization.
- Account for natural frequency. The current algorithm does not account for the natural frequency of the structure and can therefore produce a final solution that is unsuitable for actual implementation. Further research could investigate whether it is possible to include frequency as a factor in the optimization process, helping it avoid critical frequencies like 1p and 3p.
- During this study, all optimization has been done based on one single load case. Real world development does, however, require testing with hundreds of load cases, if design standard recommendations are to be followed. Further work can therefore investigate how the optimization procedure can coexist and interact with this large number of load cases, resulting in a design that is in guaranteed to comply with all requirements.

References

1. Det Norske, v., *Fatigue design of offshore steel structures: April 2008*2008, Høvik: Det norske veritas. 130 s.
2. Czyzewski, A. *Wind energy gets serial*. 2012 30.04 [cited 2012 07.06]; Available from: http://www.theengineer.co.uk/in-depth/the-big-story/wind-energy-gets-serial/1012449.article.
3. Jamieson, P., *Innovation in wind turbine design*. 1st ed2011, Hoboken, N.J.: Wiley.
4. Gosavi, A., *Simulation-based optimization : parametric optimization techniques and reinforcement learning*. Operations research/computer science interfaces series2003, Boston: Kluwer Academic Publishers. xxvii, 554 p.
5. Kiefer, J. and J. Wolfowitz, *Stochastic Estimation of the Maximum of a Regression Function.* Annals of Mathematical Statistics, 1952. **23**(3): p. 462-466.
6. Blum, J.R., *Multidimensional Stochastic Approximation Methods.* Annals of Mathematical Statistics, 1954. **25**(4): p. 737-744.
7. Snyman, J.A., *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*2005, Boston, MA: Springer Science+Business Media, Inc.
8. Spall, J.C., *Introduction to stochastic search and optimization : estimation, simulation, and control*. Wiley-Interscience series in discrete mathematics and optimization2003, Hoboken, N.J.: Wiley-Interscience. xx, 595 p.
9. Sawaragi, Y., H. Nakayama, and T. Tanino, *Theory of multiobjective optimization*. Mathematics in science and engineering1985, Orlando: Academic Press. xiii, 296 p.
10. Spall, J.C., *A Stochastic Approximation Technique for Generating Maximum Likelihood Parameter Estimates.* American Control Conference, 1987 , vol., no., pp.1161-1167, 10-12 June 1987, 1987.
11. Spall, J.C., *Multivariate stochastic approximation using a simultaneous perturbation gradient approximation.* Automatic Control, IEEE Transactions on, 1992. **37**(3): p. 332-341.
12. Spall, J.C. *SPSA.* [cited 2012 07.06]; Available from: http://www.jhuapl.edu/spsa/index.html.
13. Spall, J.C., *An overview of the simultaneous perturbation method for efficient optimization.* Johns Hopkins Apl Technical Digest, 1998. **19**(4): p. 482-492.
14. Twidell, J. and G. Gaudiosi, *Offshore wind power*2009, Brentwood: Multi-Science Pub. Co. xi, 357 p.
15. Lynn, P.A., *Onshore and offshore wind energy : an introduction*2012, Chichester, West Sussex ; Hoboken, NJ: Wiley.
16. Hau, E., *Wind turbines : fundamentals, technologies, application, economics*. 2nd English ed2006, Berlin ; New York: Springer. xviii, 783 p.
17. Vries, W.d., *Support Structure Concepts for Deep Water Sites, UpWind final report WP 4.2.* 2011.
18. Schijve, J., *Fatigue of structures and materials*2008, New York: Springer.
19. Nowitech. *Nowitech*. 2012 [cited 2012 07.06]; Available from: http://www.sintef.no/Projectweb/Nowitech/.

20. Muskulus, M., *The full-height lattice tower concept.* Energy Procedia (in press), 2012.

21. Law, A.M. and M.G. McComas, *Simulation-based optimization.* Proceedings of the 2000 Winter Simulation Conference, Vols 1 and 2, 2000: p. 46-49.

22. Negm, H.M. and K.Y. Maalawi, *Structural design optimization of wind turbine towers.* Computers & Structures, 2000. **74**(6): p. 649-666.

23. Long, H.Y. and G. Moe, *Preliminary Design of Bottom-Fixed Lattice Offshore Wind Turbine Towers in the Fatigue Limit State by the Frequency Domain Method.* Journal of Offshore Mechanics and Arctic Engineering-Transactions of the Asme, 2012. **134**(3).

24. Long, H., G. Moe, and T. Fischer, *Lattice Towers for Bottom-Fixed Offshore Wind Turbines in the Ultimate Limit State: Variation of Some Geometric Parameters.* Journal of Offshore Mechanics and Arctic Engineering-Transactions of the Asme, 2012. **134**.

25. Zwick, D., M. Muskulus, and G. Moe, *Iterative optimization approach for the design of full-height lattice towers for offshore wind turbines.* Energy Procedia (in press), 2012.

26. Sadegh, P. and J.C. Spall, *Optimal random perturbations for stochastic approximation using a simultaneous perturbation gradient approximation.* Proceedings of the 1997 American Control Conference, Vols 1-6, 1997: p. 3582-3586.

27. Spall, J.C., *Implementation of the simultaneous perturbation algorithm for stochastic optimization.* Aerospace and Electronic Systems, IEEE Transactions on, 1998. **34**(3): p. 817-823.

28. Maryak, J.L. and D.C. Chin, *Global random optimization by simultaneous perturbation stochastic approximation.* Ieee Transactions on Automatic Control, 2008. **53**(3): p. 780-783.

29. Fu, M.C. and S.D. Hill, *Optimization of discrete event systems via simultaneous perturbation stochastic approximation.* Iie Transactions, 1997. **29**(3): p. 233-243.

30. Spall, J.C., *Accelerated second-order stochastic optimization using only function measurements.* Proceedings of the 36th Ieee Conference on Decision and Control, Vols 1-5, 1997: p. 1417-1424.

31. Spall, J.C., *A one-measurement form of simultaneous perturbation stochastic approximation.* Automatica, 1997. **33**(1): p. 109-112.

32. Chin, D.C., *A More Efficient Global Optimization Algorithm-Based on Styblinski and Tang.* Neural Networks, 1994. **7**(3): p. 573-574.

33. Spall, J.C. and J.A. Cristion, *Nonlinear Adaptive-Control Using Neural Networks - Estimation with a Smoothed Form of Simultaneous Perturbation Gradient Approximation.* Statistica Sinica, 1994. **4**(1): p. 1-27.

34. Amzallag, C., et al., *Standardization of the Rainflow Counting Method for Fatigue Analysis.* International Journal of Fatigue, 1994. **16**(4): p. 287-293.

35. Gere, J.M. and B.J. Goodno, *Mechanics of materials*. Brief ed2011, Stamford, Conn.: Cengage Learning. xix, 618 p.

36. IEC, *Wind Turbines - Part 3: Design requirements for offshore wind trubines*, 2009.

37. Sadegh, P., *Constrained optimization via stochastic approximation with a simultaneous perturbation gradient approximation.* Automatica, 1997. **33**(5): p. 889-892.

Appendix

Appendix A - Objective functions A-1
Appendix B - Code B-1
Main optimization code: B-1
Objective function code: B-13

Appendix A - Objective functions

Selected tested objective functions with accompanying comments. Complete (and more accurate) list can be found as code in appendix B.

Objective function	Comments
All function does only consider the lowest lifetime value registered per joint. Some only consider the lowest value in the whole member.	
$\left(\sqrt{\frac{\sum_{i=1}^{n} NLT_i^2}{n}} - 1\right)^{-1} + \left(\frac{weight}{1000}\right)^2$	Does not work. An optimal root mean squared value can be achieved even though several lifetimes are insufficient. Might work if summing over all NLT < 1.
$(\min(NLT) - 1)^{-1} + \frac{Weight}{1500}$	One lifetime was not enough to decide which direction is the best. And there is also a discontinuity at lifetime = 1.
$\left(\frac{\sum_{NLT_i<1} NLT_i}{\sum_{NLT_i<1} 1}\right)^{-1} + \frac{weight}{1500}$	The extreme values close to zero lifetime creates instability.
$\left(-log\left(\left(\frac{\sqrt{\frac{\sum_{i=1}^{n} NLT_i^2}{n}}}{1.2}\right)^2\right)\right)^2 + \frac{weight}{1500}$	Does not work. The target value within the root can be achieved even though several lifetimes are insufficient
$\sqrt{\frac{\sum_{NLT_i<1} (NLT_i - 1.25)^2}{\sum_{NLT_i<1} 1}} + \left(\frac{weight - 1200}{1000}\right)^2$	Too dominated by the weight term
$a1 * \exp\left(-\left(\frac{(avg(NLT) - b1)}{c1}\right)^2\right) + a2 * \exp\left(-\left(\frac{(avg(NLT) - b2)}{c2}\right)^2\right) + \left(\frac{weight}{1500}\right);$	Can work, but not very intuitive.. *curve-fitted function
$\sum_{NLT_i<1} (NLT_i - 1.25)^2 + \left(\frac{weight - 1200}{50}\right)$	Takes too long to get rid of the lowest lifetime values
$\sum_{NLT_i<1} ((NLT_i - 1.25)^2 + (NLT_i - 1.1)^{20}) + \left(\frac{weight - 1200}{50}\right)$	Working very well! Recommended!

Appendix B - Code

The most relevant code is given here. The “Main optimization code” contains the actual optimization algorithm, while the “Objective function code” contains the code for the various objective functions that has been investigated. For the program to run, several other files are necessary.

Main optimization code:

Input:

```
project: The name of the project
leg_num: number of legs
sec_num: number of sections
members: member dimensions
tow_hei: height of the tower
top_dis: distance between legs at top
bot_dis: distance between legs at bottom
NodeK: K-joint locations
NodeX: X-joint locations
wat-dep: water depth
ang_opt: Constant (1) or varying(2) brace angle
locations: folder directories
```

```
function SPSA_fullpara_optimization(project, leg_num, sec_num, members, tow_hei,
top_dis, bot_dis, NodeK, NodeX, ang_bra_ho, wat_dep, ang_opt, locations)

% ======================================================================
% FUNCTION SPSA_optimization
% ======================================================================
% written by Håvard Molde, 03/2012
% based on "optimization" written by Daniel Zwick, 10/2011


% ----------------------------------------------------------------------
% 0) predefinitions
% ----------------------------------------------------------------------

    % Open matlabpool for increased computational speed (parallization)
        distcomp.feature( 'LocalUseMpiexec', false );
        matlabpool open 2;

    % including soil modelling in FEDEM: (1)-true / (0)-false
        soil=1;

    % counter for number of FEDEM runs
        runs=0;
        runs_err=0;

    % variable for record of dimension changes
        dimch=cell(sec_num);

    % split initial project name
        pro1=project(1:8);
        pro2=str2double(project(10:12));

    % create model batch, data and figures directory
        dir_nam{1}=[locations{1,2} '\FEDEM\Analysis\FEDEM_Models\' pro1];
        dir_nam{2}=[locations{1,2} '\FEDEM\PostProcessing\data\' pro1];
```

```
        dir_nam{3}=[locations{1,2} '\FEDEM\PostProcessing\figures\' pro1];
        for i=1:length(dir_nam)
            if (exist(dir_nam{i},'dir') ~= 2)
                cmd_dir{i}=sprintf('mkdir %s', dir_nam{i});
                system(cmd_dir{i});
            end
        end



    % create validity matrix for member dimensions, limited by SCF parameters
    fil_nam=['scf_validity/' num2str(sec_num) 'sec_935_' num2str(bot_dis) 'm.mat'];
    if (exist(fil_nam,'file') == 2)
        load(fil_nam)
    else

        % diameter range
            D=0.001:0.001:2.0;    % NB! maximum diameter size 2,0m

        % thickness range
            T=0.001:0.001:0.1;    % NB! maximum thickness size 100mm

        % reserve member matrix for all possible and allowed values
            mem_matrix=cell(sec_num);
            mem_valid=zeros(length(D),length(T));

        % calculation for one brace-to-leg rate: 0.50
            for i=1:length(T)
                for j=1:length(D)
                    thi=T(i);
                    dia=D(j);
                    for k=1:sec_num
                        mem_matrix{k}=[dia 0.5*dia thi 0.5*thi];
                    end
                    % stress concentration factor
                        [scf val]=scf_calculation(sec_num, mem_matrix, tow_hei,…
                        top_dis, bot_dis, ang_bra_ho);
                    if (mean(mean(val(:,:))) == 1)    % valid SCF
                        mem_valid(int16(1000*dia),int16(1000*thi))=1;
                    end
                end
            end
            save(fil_nam, 'mem_valid')
    end


% ------------------------------------------------------------------
% 1) START OPTIMIZATION LOOP
% ------------------------------------------------------------------

    % Initiation and coefficient selection
        alpha = 0.602;
        gamma = 0.101;
        A = 15;
        a = 0.000005;
        c = 0.0002;

    % Choose objective-function
        obj_func = 24;

    % number of members changed for initial design
        nmc=0;
```

```
% tower weight for initial design [kg]
    M(1)=weight_calculation(leg_num, sec_num, members, NodeK)/1000;

% write to development-file. Can be removed
    fidx=fopen([locations{2,1} '\Matlab\lists\' project(1:8)
        '_development.txt'],'at');
    fprintf(fidx, 'Constants:\n');
    fprintf(fidx, 'a = %d, c = %d, A = %d, alpha = %d, gamma = %d\n', a, c, A,…
         alpha, gamma);
    fprintf(fidx, '\n');
    fprintf(fidx, 'Initial members dimentions: \n');
    for i = sec_num:-1:1
        fprintf(fidx, '% 1.4f % 1.4f % 1.4f % 1.4f\n', members{i}(1,1),…
                      members{i}(1,2), members{i}(1,3), members{i}(1,4));
    end
    fprintf(fidx, '\n');
    fclose(fidx);


iter = 0;
opt_run=0;
while (opt_run == 0)
    iter = iter+1;

    ak=a/(iter+1+A)^alpha;
    ck=c/(iter+1)^gamma;

% leveling of dimensions between diameter and thickness
    lev = [20 20 1 1 1 1 1 1];

% Generate a SP-vector
    delta = zeros(sec_num,8);
    for i = 1:sec_num
        delta(i,1:4) = 2*round(rand(1,4))-1;
        members_plus{i} = members{i}+ck*delta(i,:).*lev;
        members_minus{i} = members{i}-ck*delta(i,:).*lev;

        % Checking that leg dimentions are larger than brace dimentions
        % Picking randomly the largest or smallest value
        for j = [1 3]
            if members_plus{i}(j+1) > members_plus{i}(j)
                dir = 2*round(rand(1,1))-1;
                members_plus{i}(j+1/2*(1+dir)) = members_plus{i}(j+1/2*(1-dir));
            end
            if members_minus{i}(j+1) > members_minus{i}(j)
                dir = 2*round(rand(1,1))-1;
                members_minus{i}(j+1/2*(1+dir)) = members_minus{i}(j+1/2*(1-dir));
            end
        end

        % Checking that every dimentions are above minimum:
            members_minus{i}(members_minus{i}(1:2) < 0.1) = 0.1;
            members_minus{i}(members_minus{i}(1:4) < 0.005) = 0.005;
            members_plus{i}(members_plus{i}(1:2) < 0.1) = 0.1;
            members_plus{i}(members_plus{i}(1:4) < 0.005) = 0.005;


        % checking that the diameter is larger then twise the tickness:
        for j = [1 2]
            if 2*members_plus{i}(j+2) > members_plus{i}(j)
                members_plus{i}(j) = 2*members_plus{i}(j+2);
            elseif 2*members_minus{i}(j+2) > members_minus{i}(j)
```

```
                    members_minus{i}(j) = 2*members_minus{i}(j+2);
                end
            end
        end

        % Check that top leg diameter is larger than minimum (larger
        % requirements then the rest of the tower)
            members_minus{sec_num}(members_minus{sec_num}(1) < 0.5) = 0.5;
            members_plus{sec_num}(members_plus{sec_num}(1) < 0.5) = 0.5;
            if members_minus{sec_num}(3) < 0.06
                members_minus{sec_num}(3) = 0.06;
            end
            if members_plus{sec_num}(3) < 0.06
                members_plus{sec_num}(3) = 0.06;
            end

        % write to development-file. Can be removed
            fidx=fopen([locations{2,1} '\Matlab\lists\' project(1:8)
                        '_development.txt'],'at');
            fprintf(fidx, 'Members_plus dimentions: \n');
            for i = sec_num:-1:1
                fprintf(fidx, '% 1.4f % 1.4f % 1.4f % 1.4f\n', members_plus{i}(1,1),…
                        members_plus{i}(1,2), members_plus{i}(1,3),
members_plus{i}(1,4));
            end
            fprintf(fidx, '\n');
            fprintf(fidx, 'Members_minus dimentions: \n');
            for i = sec_num:-1:1
                fprintf(fidx, '% 1.4f % 1.4f % 1.4f % 1.4f\n', members_minus{i}(1,1),…
                        members_minus{i}(1,2), members_minus{i}(1,3),
members_minus{i}(1,4));
            end
            fprintf(fidx, '\n');
            fclose(fidx);



    % update cross sectional area and moment of inertia for new design
        for i=1:sec_num
            for k=1:2
                members_plus{i}(4+k) = pi*((members_plus{i}(k))^2-(members_plus{i}(k)-…
                        2*members_plus{i}(k+2))^2)/4;
                members_plus{i}(6+k) = pi*((members_plus{i}(k))^4-(members_plus{i}(k)-…
                        2*members_plus{i}(k+2))^4)/64;

                members_minus{i}(4+k) = pi*((members_minus{i}(k))^2-…
                        (members_minus{i}(k)-2*members_minus{i}(k+2))^2)/4;
                members_minus{i}(6+k) = pi*((members_minus{i}(k))^4-…
                        (members_minus{i}(k)-2*members_minus{i}(k+2))^4)/64;
            end
        end


    % header
        % actual time
            clk=datestr(clock);
        % tower weight saved in last iteration
            M(2)=weight_calculation(leg_num, sec_num, members, NodeK)/1000;
            tws=M(1)-M(2);
            M(1)=M(2);
        % time since start
            % a) seconds
                tss_s=toc;
```

```
            % b) minutes and seconds
                tss_m=fix(tss_s/60);
                tss_s=rem(tss_s,60);
            % c) hours, minutes and seconds
                tss_h=fix(tss_m/60);
                tss_m=rem(tss_m,60);

clc
fprintf('+-------------------------------------------------------------+\n')
fprintf('| Lattice Tower Optimization, %s          |\n',clk)
fprintf('+-------------------------------------------------------------+\n')
fprintf('| Number of runs performed (error):               %3.0f (%3.0f) |\n',runs,…
      runs_err);
fprintf('| Number of iteration completed :                        %3.0f |\n',iter-1);
fprintf('| Number of members changed in last iteration:           %3.0f |\n',nmc);
fprintf('| Tower weight saved in last iteration [t]:              %3.0f |\n',tws);
fprintf('| Time since start:                              %3.0fh %2.0fm %2.0fs…
      |\n',tss_h,tss_m,tss_s);
fprintf('+-------------------------------------------------------------+\n')
fprintf('\n\n')


project_a = [project(1:12) 'a'];
project_b = [project(1:12) 'b'];

% write FEDEM input file using slightly increased dimensions
    fedem_input(project_a, sec_num, members_plus, soil, locations)
    fprintf('\n')
% write FEDEM input file using slightly decreased dimensions
    fedem_input(project_b, sec_num, members_minus, soil, locations)
    fprintf('\n')

    lock_limit = 0;

close all hidden

parfor eval = 1:2
    % 1 - evaluation using added delta-values
    % 2 - evaluation using subtracted delta-values


    % ---------------------------------------------------------------------
    % 2) build and run FEDEM model
    % ---------------------------------------------------------------------


     if eval == 1;

         % run FEDEM analysis
             fil_mod=[locations{1,2} '\FEDEM\Analysis\FEDEM_Models\' project_a(1:8)…
                  '\' project_a '.fmm'];
             cmd_run=sprintf('"C:\\Program Files (x86)\\Fedem Simulation Software R6.0-
…                 i6\\fedem.exe" -f %s -solve dynamics',fil_mod);
             fprintf([project_a ': Start FEDEM run ...\n'])
             system(cmd_run);
             fprintf([project_a ': FEDEM run finished\n'])

         % ULS/FLS analysis
             sim_fil=[locations{1,1} '/FEDEM/Analysis/FEDEM_Models/' project_a(1:8)…
                   '/' project_a '_RDB/response_0001/'];
```

```
            SPSA_HSS_analysis(project_a, leg_num, sec_num, members_plus, tow_hei,…
                        top_dis, bot_dis, NodeK, NodeX, ang_bra_ho, sim_fil,
lock_limit,…
                    locations)

     else
         pause(10)

         % run FEDEM analysis
             fil_mod=[locations{1,2} '\FEDEM\Analysis\FEDEM_Models\' project_b(1:8)…
                  '\' project_b '.fmm'];
             cmd_run=sprintf('"C:\\Program Files (x86)\\Fedem Simulation Software R6.0-
…                 i6\\fedem.exe" -f %s -solve dynamics',fil_mod);
             fprintf([project_b ': Start FEDEM run ...\n'])
             system(cmd_run);
             fprintf([project_b ': FEDEM run finished\n'])

         % ULS/FLS analysis
             sim_fil=[locations{1,1} '/FEDEM/Analysis/FEDEM_Models/' project_b(1:8)…
                  '/' project_b '_RDB/response_0001/'];
             SPSA_HSS_analysis(project_b, leg_num, sec_num, members_minus, tow_hei,…
                        top_dis, bot_dis, NodeK, NodeX, ang_bra_ho, sim_fil,
lock_limit,…                        locations)
     end

end         % of parallell loop

runs = runs + 2;

fprintf('\n\n')



    % ---------------------------------------------------------------------
    % 3) analyse results
    % ---------------------------------------------------------------------

    % header
        fprintf('| 2) Post Processing                                          |\n')
        fprintf('+-----------------------------------------------------------+\n')

for eval = 1:2

        if eval == 1;
            project = [project(1:12) 'a'];
        else
            project = [project(1:12) 'b'];
        end

        feedback=fedem_feedback(project, locations);
        if (feedback == 0)
            fprintf('\n');
            fprintf('=====>   WARNING - Simulation ')
            fprintf('%s',project)
            fprintf(' failed :-(   <=====\n')
            runs_err=runs_err+1;
        end

        % header
            fprintf('| 3) Benchmark results %s                       |\n', project)
            fprintf('+-----------------------------------------------------------+\n')

        % read benchmark results
```

```
            fil_nam=[locations{1,1} '/FEDEM/PostProcessing/data/' project(1:8) '/'…
                        project '_BMi.mat'];
            load(fil_nam)
            element={'Legs  ';'Braces'};
            warnings={'ULS-warnings:'; 'FLS-warnings:'};

        % check for ULS/FLS performance and mark planned improvements
            % variable mark(a,b)=c
                % a - sec_num
                % b - (1)-legs / (2)-braces
                % c - (+) decrease in thickness
                %     (0) no change
                %     (-) increase in thickness
                mark=zeros(sec_num,2);

             for w=2:2   % (1)-ULS, (2)-FLS
                fprintf('%s\n',warnings{w});
                for k=1:sec_num


**************************************************************************

                    % LEGS
                    if ((bmi{w}(1,k) < 1.0) || (bmi{w}(2,k+1) < 1.0))
                        fprintf('    %s in section %2.0f < 0.5 (%f)\n', element{1}, k,…
                                    min([bmi{w}(1,k) bmi{w}(2,k+1)]))
                        mark(k,1)= -1;
                    elseif ((bmi{w}(1,k) > lock_limit) && (bmi{w}(2,k+1) > lock_limit)…
                                    && (w == 2))   % decrease for FLS only
                        mark(k,1)= +1;
                    end

                    % BRACES
                    if ((min(bmi{w}([3 5],k)) < 1.0) || (min(bmi{w}([4 6],k+1)) < 1.0)…
                              || (min(bmi{w}(7:10,k)) < 1.0))
                        fprintf('    %s in section %2.0f < 0.5 (%f)\n', element{2}, k,…
                                    min([min(bmi{w}([3 5],k)) min(bmi{w}([4 6],k+1))…
                                          min(bmi{w}(7:10,k))]))
                        mark(k,2)= -1;
                    elseif ((min(bmi{w}([3 5],k)) > lock_limit) &&…
                         (min(bmi{w}([4 6],k+1)) > lock_limit) &&
(min(bmi{w}(7:10,k))...                          > lock_limit) && (w == 2))   % decrease
for FLS only
                        mark(k,2)= +1;
                    end

                    %
**************************************************************************

                end
                fprintf('\n');
            end
            fprintf('\n');

            if eval == 1
                mark_a = mark;
                bmi_a = bmi;
            else
                mark_b = mark;
                bmi_b = bmi;
            end
```

```
end         % of evaluation

        project = project(1:12);        % Remove a/b ending.

        M_a=weight_calculation(leg_num, sec_num, members_plus, NodeK)/1000;
        M_b=weight_calculation(leg_num, sec_num, members_minus, NodeK)/1000;

        % loss function evaluation
            yplus=SPSA_loss(M_a, bmi_a, obj_func);
            yminus=SPSA_loss(M_b, bmi_b, obj_func);




% ---------------------------------------------------------------------
% 4) change member dimensions
% ---------------------------------------------------------------------

    % header
        fprintf('| 4) Update topology                                          |\n')
        fprintf('+-----------------------------------------------------------+\n')

    % save member dimensions in record variable before changing
        for i=1:sec_num
            for j=1:4
                dimch{iter}(i,j)=members{i}(j);     % diameter,(1)legs and (2)braces
[m]
                                                    % thickness,(3)legs and (4)braces
[m]
            end
        end
        dimch{iter}(1,5)=-tws;                      % delta tower weight [t]
        dimch{iter}(1,6)=weight_calculation(leg_num, sec_num, members, NodeK)/1000;
                                                    % total tower weight [t]

    % save record variable to file
        fil_nam=[locations{1,1} '/FEDEM/PostProcessing/data/' project(1:8) '/'…
                  project '_dimch.mat'];
        save(fil_nam, 'dimch')

    % apply new project name for new member dimensions
        pro2=pro2+1;

    % create project name
        if (pro2 < 10)
            project=[pro1,'_00',int2str(pro2)];
        elseif ((pro2 >= 10) && (pro2 < 100))
            project=[pro1,'_0',int2str(pro2)];
        else
            project=[pro1,'_',int2str(pro2)];
        end

    % gradient approximation. Applying max step size
        g_temp = (yplus-yminus)./(2*ck);
        if g_temp <= 0
            g_temp=max([g_temp -0.005/ak]);
            ghat = g_temp./delta(:,1:4);
        elseif g_temp > 0
            g_temp = min([g_temp 0.005/ak]);
            ghat = g_temp./delta(:,1:4);
        end
```

```
% reset number of members changed
        nmc=0;

% update members estimate
    % Changing members diameter within: (0.05....2.0)m, and
    % thickness within (0.002...0.1)m, while makeing sure diameter >
    % diameter > 2*thickness
    for i = 1:sec_num
        for j = 1:2
                temp = members{i}([j j+2]) - ak*ghat(i,[j j+2]).*lev([j j+2]);
                members{i}(j+2) = min([temp(2) 0.1]);
                members{i}(j+2) = max([members{i}(j+2) 0.005]);
                members{i}(j) = min([temp(1) 2.0]);
                members{i}(j) = max([members{i}(j) 0.1 2*members{i}(j+2)]);
                nmc = nmc +2;
        end
        % If brace dimentions are larger than leg dimentions: use the
        % average on both.
        for k = [1 3]
            if members{i}(k) < members{i}(k+1)
                members{i}(k) = 0.5*(members{i}(k+1)+members{i}(k));
                members{i}(k+1) = members{i}(k);
                if k == 1
                    members{i}(k+1) = max([members{i}(k+1) 2*members{i}(k+3)]);
                    members{i}(k) = max([members{i}(k+1) 2*members{i}(k+2)]);
                end
            end
        end
    end
    % Check that top leg diameter is larger than minimum (larger
    % requirements then the rest of the tower)
    members{sec_num}(members{sec_num}(1) < 0.5) = 0.5;
    if members{sec_num}(3) < 0.06
        members{sec_num}(3) = 0.06;
    end

    % write to development-file.
        fidx=fopen([locations{2,1} '\Matlab\lists\' project(1:8)…
                    '_development.txt'],'at');
        fprintf(fidx, 'yplus      yminus      ak          ck…
                     ghat      step \n');
        fprintf(fidx, '% 9.4f % 10.4f % 13.8f % 12.8f % 11.2f % 8.4f\n', yplus,…
                     yminus, ak, ck, ghat(1,1), ghat(1,1)*ak);
        fprintf(fidx, '\n');
        fprintf(fidx, '-------------------------------------------------------…
                     ------------ \n');
        fprintf(fidx, 'Iteration number: %d\n', iter+1);
        fprintf(fidx, 'Member dimentions: \n');

        for i = sec_num:-1:1
            fprintf(fidx, '% 1.4f % 1.4f % 1.4f % 1.4f\n', members{i}(1,1), …
                     members{i}(1,2), members{i}(1,3), members{i}(1,4));
        end

        fprintf(fidx, '\n');
        fprintf(fidx, 'nmc: %d\n', nmc);
        fprintf(fidx, '\n');
        fclose(fidx);


    % update cross sectional area and moment of inertia for new design
    for i=1:sec_num
        for k=1:2
```

```
                members{i}(4+k) = pi*((members{i}(k))^2-(members{i}(k)-…
                                2*members{i}(k+2))^2)/4;
                members{i}(6+k) = pi*((members{i}(k))^4-(members{i}(k)-…
                                2*members{i}(k+2))^4)/64;
            end
        end

        if all(mark == 0) % terminate if converged
            % terminate while loop
                fprintf('Terminate while loop\n');
            % save command window output to log-file
                diary([locations{1,1} '/FEDEM/PostProcessing/logfiles/' pro1 '.txt'])
            break
        end
        fprintf('\n')

         if iter > 100  % terminate if reached maximum number of itertions
             % terminate while loop
                fprintf('Terminate while loop\n');
             % save command window output to log-file
                 diary([locations{1,1} '/FEDEM/PostProcessing/logfiles/' pro1 '.txt'])
             break
         end
        fprintf('\n\n')


    % save new project parameters to file (only when improvements are done)
        if (opt_run == 0)
            fil_nam=['parameters/',project,'.mat'];
            save(fil_nam, 'tow_hei', 'top_dis', 'bot_dis', 'wat_dep', 'leg_num', …
                   'sec_num', 'ang_opt', 'members')
            write_project_list(project);
        end

    % save command window output to log-file
        diary([locations{1,1} '/FEDEM/PostProcessing/logfiles/' pro1 '.txt'])

% ------------------------------------------------------------------------
% 5) plot changes in member dimensions
% ------------------------------------------------------------------------

    % convert tower weight record from cell to vector
        for m=1:iter
            MT(m)=dimch{m}(1,6);
        end

    % plot member dimension record
        figure
            subplot(2,2,1)
                hold on
                axis_min=min(dimch{1}(:,1))-0.010;
                axis_max=max(dimch{1}(:,1))+0.010;
                for k=1:iter
                    col_tag=1-(0.1+k/iter)/1.5;
                    plot(dimch{k}(:,1), NodeX(:,3), '-mo', 'color', [col_tag col_tag…
                         col_tag], 'LineWidth', 1, 'MarkerSize', 3)
                    if (min(dimch{k}(:,1)) < axis_min)
                        axis_min=min(dimch{k}(:,1))-0.002;
                    end
                    if (max(dimch{k}(:,1)) > axis_max)
                        axis_max=max(dimch{k}(:,1))+0.002;
                    end
                end
```

```
    axis([axis_min axis_max NodeK(1,3) 10*ceil(NodeK(sec_num+1,3)/10)])
    line([axis_min axis_max],[0 0],'Color','b','LineStyle',':')
    xlabel('diameter [mm]')
    ylabel('Tower height [m]')
    title('Legs')
    hold off
subplot(2,2,2)
    hold on
    axis_min=min(dimch{1}(:,2))-0.010;
    axis_max=max(dimch{1}(:,2))+0.010;
    for k=1:iter
        col_tag=1-(0.1+k/iter)/1.5;
        plot(dimch{k}(:,2), NodeX(:,3), '-mo', 'color', [col_tag col_tag…
                  col_tag], 'LineWidth', 1, 'MarkerSize', 4)
        if (min(dimch{k}(:,2)) < axis_min)
            axis_min=min(dimch{k}(:,2))-0.002;
        end
        if (max(dimch{k}(:,2)) > axis_max)
            axis_max=max(dimch{k}(:,2))+0.002;
        end
    end
    axis([axis_min axis_max NodeK(1,3) 10*ceil(NodeK(sec_num+1,3)/10)])
    line([axis_min axis_max],[0 0],'Color','b','LineStyle',':')
    xlabel('diameter [mm]')
    title('Braces')
    hold off
subplot(2,2,3)
    hold on
    axis_min=min(dimch{1}(:,3))-0.010;
    axis_max=max(dimch{1}(:,3))+0.010;
    for k=1:iter
        col_tag=1-(0.1+k/iter)/1.5;
        plot(dimch{k}(:,3), NodeX(:,3), '-mo', 'color', [col_tag col_tag …
                  col_tag], 'LineWidth', 1, 'MarkerSize', 3)
        if (min(dimch{k}(:,3)) < axis_min)
            axis_min=min(dimch{k}(:,3))-0.002;
        end
        if (max(dimch{k}(:,3)) > axis_max)
            axis_max=max(dimch{k}(:,3))+0.002;
        end
    end
    axis([axis_min axis_max NodeK(1,3) 10*ceil(NodeK(sec_num+1,3)/10)])
    line([axis_min axis_max],[0 0],'Color','b','LineStyle',':')
    xlabel('thickness [mm]')
    ylabel('Tower height [m]')
    title('Legs')
    hold off
subplot(2,2,4)
    hold on
    axis_min=min(dimch{1}(:,4))-0.010;
    axis_max=max(dimch{1}(:,4))+0.010;
    for k=1:iter
        col_tag=1-(0.1+k/iter)/1.5;
        plot(dimch{k}(:,4), NodeX(:,3), '-mo', 'color', [col_tag col_tag…
             col_tag], 'LineWidth', 1, 'MarkerSize', 4)
        if (min(dimch{k}(:,4)) < axis_min)
            axis_min=min(dimch{k}(:,4))-0.002;
        end
        if (max(dimch{k}(:,4)) > axis_max)
            axis_max=max(dimch{k}(:,4))+0.002;
        end
    end
    axis([axis_min axis_max NodeK(1,3) 10*ceil(NodeK(sec_num+1,3)/10)])
    line([axis_min axis_max],[0 0],'Color','b','LineStyle',':')
    xlabel('thickness [mm]')
```

```
                title('Braces')
                suptitle(['Dimension changes during optimization, run ' project(1:8)])
                hold off
        fig_name=[locations{1,1} '/FEDEM/PostProcessing/figures/' project(1:8) '/'
pro1…                    '_optimization_dimensions.fig'];
        hgsave(fig_name)

        figure
            hold on
            for k=1:iter
                bar(k,dimch{k}(1,5))
            end
            box off
            xlabel('Iteration steps')
            ylabel('\Delta tower weight [t]')
            h1=gca;
            h2=axes('Position',get(h1,'Position'));
            plot(MT,'-mo', 'color', 'green', 'LineWidth', 2, 'MarkerSize', 4);
            box off
            ylabel('Total tower weight [t]')
            set(h2,'YAxisLocation','right','Color','none','XTickLabel',[])
            set(h2,'XLim',get(h1,'XLim'),'Layer','top')
            title('Tower weight')
            suptitle(['Tower weight changes during optimization, run ' project(1:8)])
            hold off
        fig_name=[locations{1,1} '/FEDEM/PostProcessing/figures/' project(1:8) '/'
pro1…                    '_optimization_weight.fig'];
        hgsave(fig_name)


    % benchmark plots
        if (iter > 1)
            SPSA_BM_plots(project, locations)
%             fprintf('Ferdig med SPSA_BM_plots \n')
            SPSA_scf_member_plot(project, tow_hei, sec_num, top_dis, bot_dis,…
                        ang_bra_ho, locations)
%             fprintf('Ferdig med SPSA_scf_member_plots \n')
        end

close all hidden

% ---------------------------------------------------------------------
% 6) END OPTIMIZATION LOOP
% ---------------------------------------------------------------------

    end
end   % of function
```

Objective function code:

Input:

```
weight: total tower weight
bmi: lifetime values for all members
alt: function selector


function loss = SPSA_loss(weight, bmi, alt)

[r c] = size(bmi{2});
temp = 0;
teller = 0;

switch alt;

    % (1) -----------------------------------------------------
    case 1
    % using root mean square of  all liftimes, and weight term

    % calculate root mean square of the BMi-values for FLS
        for i = 1:r
            for j = 1:c
                temp = temp + (bmi{2}(i,j))^2;
            end
        end

        RMS_bmi  = sqrt(temp/(r*(c-1)));

    % defining loss-function
        loss = 1/(RMS_bmi - 1) + (weight/1000)^2;

    % (2) -----------------------------------------------------
    case 2
    % Not used

    % (3) -----------------------------------------------------
    case  3
    % use minimum lifetime-value, and weight term

        min_bmi = min(nonzeros(bmi{2}));

    % defining loss-function
        loss = 1/(min_bmi - 1) + (weight/1500);

    % (4) -----------------------------------------------------
    case  4
    % use minimum lifetime-value, no weight term

        min_bmi = min(nonzeros(bmi{2}));

    % defining loss-function
        loss = 1/(min_bmi - 1)^3;

    % (5) -----------------------------------------------------
    case  5
    % use average of all lifetimes < 1, and weight term

        for i = 1:r
            for j = 1:c
                if (bmi{2}(i,j) < 1) && (bmi{2}(i,j) > 0)
```

```
                temp = temp + bmi{2}(i,j);
                teller = teller + 1;
            end
        end
    end

    avg_bmi = temp/teller;
    f


% defining loss-function
    loss = 1/(avg_bmi); %+ (weight/1500);

% (6) ------------------------------------------------------
case  6
% calculate root mean square of the BMi-values for FLS
    for i = 1:r
        for j = 1:c
            temp = temp + (bmi{2}(i,j))^2;
        end
    end

    RMS_bmi  = sqrt(temp/(r*(c-1)));

% defining loss-function
    loss = (-log((RMS_bmi./1.2).^2)).^2 + (weight/1500);

% (7) ------------------------------------------------------
 case  7
% calculate root mean square of the minumum for X- and K- braces, and all leg BMi-
  values for FLS
    for i = 1:c
        temp = temp + (min([bmi{2}(3,i) bmi{2}(8,i)]))^2;
        temp = temp + (min([bmi{2}(4,i) bmi{2}(7,i)]))^2;
        temp = temp + (min([bmi{2}(5,i) bmi{2}(10,i)]))^2;
        temp = temp + (min([bmi{2}(6,i) bmi{2}(9,i)]))^2;
        temp = temp + bmi{2}(1,i)^2 + bmi{2}(2,i)^2;
    end

    RMS_bmi  = sqrt(temp/(6*(c-1)));

% defining loss-function
    loss = ((-log(RMS_bmi./1.25)).^2)./RMS_bmi.^0.22 + (weight/1500);

% (8) ------------------------------------------------------
 case  8
% calculate root mean square of the minumum for X- and K- braces, and all leg BMi-
  values for FLS
    for i = 1:c-1
        temp = temp + (min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)…
              bmi{2}(7,i) bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)])))^2;
        temp = temp + (min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)])))^2;
    end

    RMS_bmi  = sqrt(temp/(2*(c-1)));

% defining loss-function
    loss = ((-log(RMS_bmi./1.5)).^2)./RMS_bmi.^0.22 + (weight/1500);

% (9) ------------------------------------------------------
case  9
% use average of all lifetimes < 1, and weight term
```

```
    for i = 1:r
        for j = 1:c
            if (bmi{2}(i,j) < 1) && (bmi{2}(i,j) > 0)
                temp = temp + bmi{2}(i,j);
                teller = teller + 1;
            end
        end
    end

    avg_bmi = temp/teller;

    % defining loss-function
    loss = ((-log(avg_bmi./1.5)).^2)./avg_bmi.^0.22 + (weight/1500);

% (10) ------------------------------------------------------
case  10
% use minimum lifetime-value, and weight term

    min_bmi = min(nonzeros(bmi{2}));

    % defining loss-function
    loss = ((-log(min_bmi./1.5)).^2)./min_bmi.^0.22 + (weight/1500);

% (11) ------------------------------------------------------
case  11
% use minimum lifetime-value, without weight term

    min_bmi = min(nonzeros(bmi{2}));

    % defining loss-function
    loss = ((-log(min_bmi./1.5)).^2)./min_bmi.^0.22;

% (12) ------------------------------------------------------
case  12
% calculate average of the minumum for X- and K- braces and leg BMi-values for FLS
    for i = 1:c-1
        temp = temp + min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)…
               bmi{2}(7,i) bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
        temp = temp + min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
    end

    avg_bmi  = temp/(2*(c-1));

% defining loss-function
    loss = ((-log(avg_bmi./1.5)).^2)./avg_bmi.^0.22;% + (weight/1500);

% (13) ------------------------------------------------------
case  13
% calculate average of 1/minumum for X- and K- braces and leg BMi-values for FLS
    for i = 1:c-1
        temp = temp + 1/(min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)…
               bmi{2}(7,i) bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)])));
        temp = temp + 1/(min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)])));
    end


    avg_bmi  = (2*(c-1))/temp;


% defining loss-function
    loss = ((-log(avg_bmi./1.5)).^2)./avg_bmi.^0.22;% + (weight/1500);
```

```
    % (14) ------------------------------------------------------
    case  14
    % calculate average of 1/minumum for X- and K- braces and leg BMi-values less than
1     for FLS. Insert into curve-fitted function
        for i = 1:c-1
            prove = 1/(min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)
bmi{2}(7,i)...
                    bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)])));
            if prove > 1
                temp = temp + prove;
                teller = teller +1;
            end
            prove = 1/(min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)])));
            if prove > 1
                temp = temp + prove;
                teller = teller +1;
            end
        end

        avg_bmi  = teller/temp;

        % Loss function and parameters fund by Gaussian curve fitting with
        % two terms, and input values x=[0.01 1 1.5 10 50 150 200], y=[15 0.5 0 0 0 0
0]
        a1 =       -1.85;
        b1 =     0.02498;
        c1 =      0.1342;
        a2 =   8.502e+14;
        b2 =      -18.12;
        c2 =       3.228;

    % defining loss-function
        loss = a1*exp(-((avg_bmi-b1)./c1).^2) + a2*exp(-((avg_bmi-b2)./c2).^2) +
                  (weight/1500);

     % (15) ------------------------------------------------------
    case  15
    % calculate average of 1/minumum for X- and K- braces and leg BMi-values for FLS.
      Insert into curve-fitted function
        for i = 1:c-1
            temp = temp + 1/(min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)...
                        bmi{2}(7,i) bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1)
bmi{2}(9,i)])));
            temp = temp + 1/(min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)])));
        end

        avg_bmi  = (2*(c-1))/temp;

        % Loss function and parameters fund by Gaussian curve fitting with
        % two terms, and input values x = [0.01 1 1.5 10 50 150 200], y = [15 0.5 0 0 0
0 0]
        a1 =       -1.85;
        b1 =     0.02498;
        c1 =      0.1342;
        a2 =   8.502e+14;
        b2 =      -18.12;
        c2 =       3.228;

    % defining loss-function
        loss = a1*exp(-((avg_bmi-b1)./c1).^2) + a2*exp(-((avg_bmi-b2)./c2).^2) +…
                  (weight/1500);
```

```
    % (16) ------------------------------------------------------
    case  16
    % calculate sum of |1.5/(minumum for X- and K- braces and leg BMi-values) -1| for
FLS
        for i = 1:c-1
            temp = temp + abs(1.5/(min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)…
                        bmi{2}(7,i) bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1)
bmi{2}(9,i)]))))-1);
            temp = temp + abs(1.5/(min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]))))-1);
        end

%         loss = temp + (weight/1500)^4;          % (1)
%         loss = temp + ((weight-1300)/250)^4;    % (2)
%         loss = temp + ((weight-1200)/200)^2;    % (3)
        loss = temp + ((weight-1300)/25);    % (4)
%         loss = temp + ((weight-1200)/250)^3;    % (5)

    % (17) ------------------------------------------------------
    case  17
      % Find lowest brace lifetimes in each section, if less than 1, sum
      0.5*|1.5/(lifetime-1)|. Do the same for legs.

        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)
bmi{2}(7,i)…                                bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1)
bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (abs(1.5/bmi_value-1))/0.5;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (abs(1.5/bmi_value-1))/0.5;
            end
        end

        loss = temp + ((weight-1300)/10);

    % (18) ------------------------------------------------------
    case 18
      % Find lowest brace lifetimes in each section, if less than 1, include in RMSD
      Do the same for legs. (target RMSD-value: 1.25)

        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1)
bmi{2}(7,i)…                                bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1)
bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2;
                teller = teller +1;
            end
        end

        RMSD_bmi  = sqrt(temp/teller);

        loss = RMSD_bmi + ((weight-1200)/1000);

    % (19) ------------------------------------------------------
    case 19
```

```
      % Find lowest brace lifetimes in each section, if less than 1, include in MSD
      Do the same for legs. (target MSD-value: 1.75)

        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                       bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.75)^2;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.75)^2;
                teller = teller +1;
            end
        end

        MSD_bmi  = (temp/teller);

        loss = MSD_bmi + ((weight-1200)/1000);

    % (20) ------------------------------------------------------
    case 20
      % Find lowest brace lifetimes in each section, if less than 1, include in SSD
      Do the same for legs. (target SSD-value: 1.25)

        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                        bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2;
                teller = teller +1;
            end
        end

        SSD_bmi  = temp;

        loss = SSD_bmi + ((weight-1200)/500);

    % (21) ------------------------------------------------------
    case 21
      % Find lowest brace lifetimes in each section, if less than 1, include in SSD
      Do the same for legs. (Target SSD-value: 1.5)
        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                        bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.5)^2;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.5)^2;
                teller = teller +1;
            end
        end

        SSD_bmi  = temp;
```

```
        loss = SSD_bmi + ((weight-1200)/500);

    % (22) ------------------------------------------------------
    case 22
      % Find lowest brace lifetimes in each section, if less than 1, include in SSD
with  two terms. Do the same for legs. (Target SSD-value: 1.25 and 1.1)

        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                     bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^10;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^10;
                teller = teller +1;
            end
        end

        SSD_bmi  = temp;

        loss = SSD_bmi + ((weight-1200)/500);

    % (23) ------------------------------------------------------
    case 23
      % Find lowest brace lifetimes in each section, if less than 1, include in SSD
with  two terms. Do the same for legs. (Target SSD-value: 1.25 and 1.1)
        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                     bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^30;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^30;
                teller = teller +1;
            end
        end

        SSD_bmi  = temp;

        loss = SSD_bmi + ((weight-1200)/500);

    % (24) ------------------------------------------------------
    case 24
      % Find lowest brace lifetimes in each section, if less than 1, include in SSD
with                  two terms. Do the same for legs. (Target SSD-value: 1.25 and
1.1)
        for i = 1:c-1
            bmi_value = min(nonzeros([bmi{2}(3,i) bmi{2}(8,i) bmi{2}(4,i+1) bmi{2}(7,i)
…                     bmi{2}(5,i) bmi{2}(10,i) bmi{2}(6,i+1) bmi{2}(9,i)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^20;
                teller = teller +1;
            end
            bmi_value = min(nonzeros([bmi{2}(1,i) bmi{2}(2,i+1)]));
            if bmi_value < 1
                temp = temp + (bmi_value-1.25)^2 + (bmi_value-1.1).^20;
```

```
            teller = teller +1;
        end
    end

    SSD_bmi  = temp;

    loss = SSD_bmi + ((weight-1200)/50);
end
```