Håkon Berger Steen

Topology optimised bolt placements

Using optimisation in combination with Abaqus to design and validate optimal bolt placements

Master's thesis in Engineering and ICT
Supervisor: Jan Torgersen
Co-supervisor: Håkon J.D Johnsen

June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Kunnskap for en bedre verden

Håkon Berger Steen

Topology optimised bolt placements

Using optimisation in combination with Abaqus to design and validate optimal bolt placements

Master's thesis in Engineering and ICT
Supervisor: Jan Torgersen
Co-supervisor: Håkon J.D Johnsen
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Norwegian University of
Science and Technology

Abstract

This thesis will focus on the use of optimization algorithms to suggest an optimal bolt placement with regards to deflection. The work is inspired by and supposed to be an addition to the design process that Thomas Røkke and Henrik Hoen Hersleth have been automating. The input and output of this program are based on this design process, and these parameters will be defined and discussed later.

The program is supposed to suggest an optimal bolt pattern for subsea manifold applications where the goal is to reduce the separation in and around hydraulic tubes. The solution to this problem is not necessarily a symmetric bolt pattern and this makes it challenging to establish a simple model to calculate the different deflections. The main focus of this thesis is to establish and design a model that can be used to suggest a good bolt pattern. The model will be used to design an objective function that tries to estimate the deformation in each tube, and this objective function can be used in optimization algorithms to find an optimal solution. The results of the different models will be verified using Finite Element Method (FEM) in the ABAQUS CAE tool.

Sammendrag

Arbeidet som er gjort i denne masteren sikter seg inn på lage et program som kan foreslå en optimal boltplassering. I Henrik Hoen Herslet sin masteroppgave har bolteplassering blitt merket som en iterativ og tidkrevende del av designprosessen til subsea manifolder. Programmet er designet rundt resultatene fra Thomas Røkke og Henrik Hoen Hersleth sine programmer, denne dataen er svært lik dagens tegninger så det vil være fullt mulig å bruke dette alene også.

Programmet skal anbefale en optimal bolteplassering for subsea manifolder der målet er å redusere deformasjonen rundt hydrauliske rør. Tidligere har det blitt brukt symmetriske boltmønstre, men med alle fremskrittene som er gjort innenfor produksjonsteknikker blir mer kompliserte design utforsket. Siden det ikke er mulig å bruke en numerisk løser (som Abaqus) direkte i en optimeringsalgoritme, iallefall uten at det går veldig tregt, må det etableres en forenklet modell som kan estimere separasjonen rundt rørene. Denne modellen vil deretter bli brukt til å designe en objektfunksjon som kan brukes i optimeringen.

Table of Contents

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Background 1

1.2 Problem description 2

1.3 Thesis objectives 3

1.4 Report structure 3

2 Requirements and theory **5**

2.1 Domain, objective function and constraints 5

2.2 Calculating the force and stress in a bolt 7

2.3 Minimize - SLSQP 10

2.4 Minimize - trust-constr 10

2.5 Differential evolution 11

2.6 Design of Experiment 11

2.7 Algorithm selection 13

2.8 Objective functions 14

3 Python implementation **16**

3.1 Support functions 16

3.2 Implementation of constraints 16

3.3 Minimization setup and call 17

3.4 Automating Abaqus simulations 18

3.5 Design of Experiment 19

4 Results and Discussion **22**

4.1 Testing objective functions 22
4.2 Design of experiment results 23
4.3 Local optimisation results 24
4.4 Global optimisation results 25

5 Discussion 27
5.1 Evaluating objective functions **f2** and **f3** 27
5.2 Design of Experiment 28
5.3 Algorithm selection 30
5.4 Evaluating **f4** 32

6 Conclusion and further work 34
6.1 Conclusion 34
6.2 Future work 35

Bibliography 37

Appendix 38
A boltLocation.py 38
B supportFunctions.py 46
C abaqus_functions.py 49
D run.py 61
E regression.py 64

List of Figures

1 A sketch of a possible problem with the hydraulic tube locations and radiuses. 6
2 Forces and moment on the centroid. 8
3 LMLHL run 21

4 HHHHH run 21

5 Deformation of case with lowest distance to bolt. . . . 22

6 Deformation of case with highest distance to bolt. . . . 22

7 Estimated stiffness vs numerical stiffness. 22

8 Header of regression summary 24

9 The parameters with p-values less than 0.05 24

10 A plot showing the bolt placements of Run 5 using the trust-constr-method with **f4**. 24

11 A plot showing the bolt placements of Run 8 using the SLSQP-method with **f4**. 24

12 A plot showing the bolt placements of Run 4 using differential evolution with **f5**. 26

13 A plot showing the bolt placements of Run 5 using differential evolution with **f5**. 26

List of Tables

1 Factors to check significance of. 19

2 Factorial design 19

3 The result of the 10 runs for both methods with objective function **f4**. The runs marked with * are runs that exceeded the iteration limit. 25

4 The result of 10 runs with differential evolution and objective function **f4** 26

1 Introduction

This section will cover the background and previous work done for the project. Further, the problem and boundary conditions will be presented and lastly the structure of the project report will be described.

1.1 Background

This project is a continuation of the work done by Røkke [2019], Hersleth [2020] and Steen [2020] [1], in cooperation with Aker Solutions. With the advances in manufacturing technologies they wanted to look at topology optimisation of their subsea manifolds. These are currently created by drilling holes into a rectangular box and this way creating the paths within the manifold. Design of a traditional manifold is a relatively simple process, but now Aker is interested in automating the design process for the next generation of manifolds.

Røkke was the first student to work on this project and he used the A-star algorithm to design the paths within a manifold. He was able to produce a topology optimised manifold that greatly reduced the volume of the material. The reader can read more about how this is done in Røkke [2019].

Hersleth was the preceding student and he started rewriting Røkke's program to Python, and continued making a GUI for the code. The next step in the design process was to decide the location of bolts. Hersleth ended his work with a proof of concept for designing the bolt pattern and optimising the bracket structure. Hersleth proposes a program that is trying to reduce the total arm between all bolts and tubes, but this code has problems with achieving low runtime when the precision is sufficiently high.

Steen was the third student to look at this design process, and continued the work on a bolt placement-script in his specialization project. The main goal of this specialization project was to look at improving the runtime and evaluating the objective function. During the project two problems that needed to be addressed were found; the solution was dependent on the initial guess, and the objective function did not

[1] This article is an unpublished report of the author's specialization project. Some of the theory and implementation details will be covered twice.

seem to be a good representation of the separation.

1.2 Problem description

The problem in this thesis is to decide the placement of bolts such that a flange can be fastened to the manifold surface. According to Hersleth this is one of the most time consuming step in the design of manifolds and today this is done manually. In his thesis he recommended that this process should be automated and continues with pseudo code that attempts to automate this. This pseudo code was a brute force program, and it had problems with scalability and precision. One example is that it used around 20 minutes to find an optimal bolt placement in relatively small problem[2]. He uses the tube placements that are generated by Røkke's program as input and tries to place bolts in an optimal way. The results of Røkke's program are also very similar to the sketches created today, so this program is also designed to be used alone. Figure 1 shows an example of a simple sketch with the most important variables. This is the placement and size of the tubes, and size of the flange. In the thesis more variables and inputs are used, but they are usually the same across problems. Examples on such variables may be the washer radius of bolts, the thickness of flange or thickness of the tube wall.

The constraints and objective function are designed around the requirements for subsea manifolds, which are a combination of industry standards and tacit knowledge. The goal is to place bolts in such a way that the flange and manifold are bolted together such that no leaks occur. The separation needs to be less than 26 μm to make sure there are no leaks. This requirement is interpreted as the goal and used as optimisation goal. Due to the nature of leaking, the separation is assumed to be measured around the tube edges and not on the flange edges. Other requirements that may not be covered as industry standards are where the bolts can be placed. Two bolts cannot be placed on top of each other, a bolt cannot be placed on top of a tube, all bolts must hold and the bolts need to be placed inside of the flange.

[2]More details about this can be found in Steen's specialization project.

1.3 Thesis objectives

The main focus for this thesis is to establish a simplified model that is able to describe the separation in the tubes, which can be translated into an objective function used in the optimisation. A simplified model is needed because it is not possible to combine the optimisation algorithms with a numerical solver. If all the steps in the optimisation algorithm were to be tested in Abaqus, the script would take too much time and have problems with illegal solutions. This problem is solved by establishing a simplified model designed to work in optimisation problems and then have this model optimised towards one or more criteria.

In Steen's specialization project (Steen [2020]) an attempt was made to establish the objective function and constraints. The objective function that was proposed showed problematic properties and it was recommended to improve the objective function. A new objective function was proposed, but not implemented and tested. The first idea for a model is based on beam theory and the assumption that most of the separation appears as a result of the moment from bolts to the tubes. This hypothesis is later disproved and an experiment is conducted to investigate the relevant parameters using design of experiments.

Another change from the specialization project is to investigate the effect of optimisation methods. With new objective functions the results seemed to be more dependent on the initial guess. The two main ways that was discussed to counteract this dependency were to add more constraints or to find a way to make good initial guesses. Adding more constraints does not seem like a good solution just to reduce dependency on initial guess, and may even make the program worse because it is not able to find new designs.

1.4 Report structure

The introduction has outlined the previous work in this project with Aker. In the following sections the theory and implementation of a bolt placing algorithm will be provided. Section 2 starts with defining the domain, objective function and constraints for a minimization problem. It is then followed by theory on how to calculate the forces in

bolts from external pressures. Three different optimisation methods are presented - SLSQP, trust-constr and differential evolution. Section 3 covers the implementation of the algorithm, which is split into three parts. First all the support functions are explained. These functions are not directly used by SLSQP, but are required to find results used in the optimisation algorithm. Then Section 3.2 is dedicated to the implementation of the constraints, where some changes are done to obtain continuous properties to satisfy the requirements of SLSQP. This is followed by Section 3.3 which gives an explanation on how to the minimization calls for the three optimisers. The automating of Abaqus modelling is covered in Section 3.4. In Section 4 the results are presented and then later discussed in Section 5. The different objective functions are discussed and compared, and the optimisation methods are compared with each other.

2 Requirements and theory

Notation

Subscript i, A_i, is used for the area of bolt i, and n is used to express the number of bolts. Accordingly subscript j, A_j, is used for tube j, and m is the total number of tubes. An example for the sum of the area of all bolts would be

$$A_{bolts} = \sum_{i=1}^{n} A_i$$

Further, d_{ij} is the distance between bolt i and tube j, and $\mathbf{x}$ is the vector with bolt placements, given by

$$\mathbf{x} = \begin{bmatrix} x_0 \\ y_0 \\ \vdots \\ x_n \\ y_n \end{bmatrix}$$

2.1 Domain, objective function and constraints

A number of simplifications are done to reduce the runtime and complexity of the code. The domain is designed to be regular with a homogeneous thickness h. It is up to the engineer to decide the size of the domain, but it should at least contain all the tubes with the possibility to place a bolt on the outside of each tube.

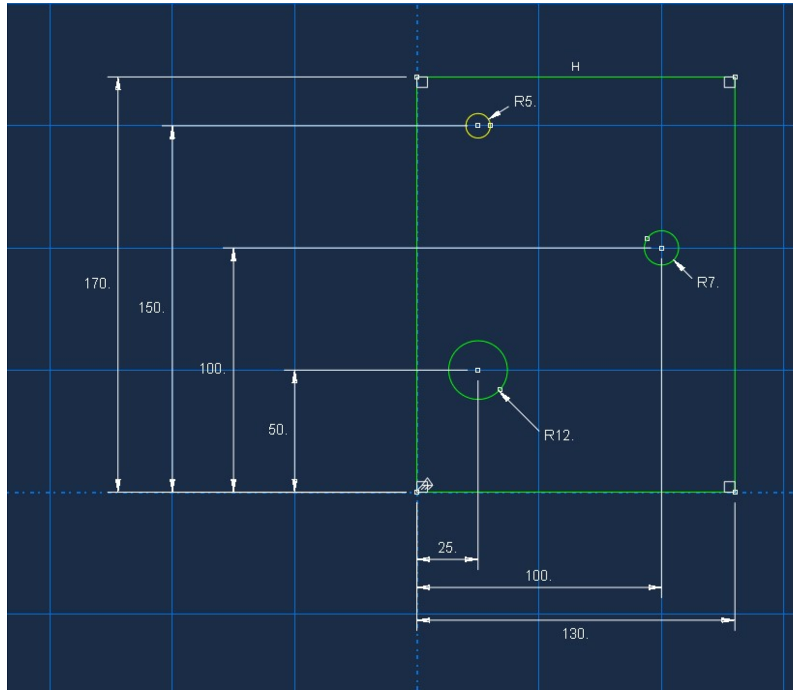


Figure 1: A sketch of a possible problem with the hydraulic tube locations and radiuses.

Figure 1 is an example of what a problem may look like. There are other parameters that need to be provided, but information about the hydraulic tubes is the main variable. The designer would also need to provide the number of bolts that needs to be placed.

Hersleth proposed to use an objective function that corresponds to finding the bolt pattern with the lowest total distance between all bolts and tubes, i.e. minimizing

$$\mathbf{f1}(\mathbf{x}) := \sum_{i}^{n} \sum_{j}^{m} d_{ij} \tag{1}$$

This function will be referred to as **f1**, as it is the first iteration of the objective function. Initially it can be hard to tell if this is a good or bad objective function. The advantages of Equation (1) are that it is a continuous function and it accounts for the main parameter that can be changed in this problem, position and distance. Benefits of continuous functions will be covered later.

Claiming that position, and indirectly distance, is the main parameter that can be changed, is supported by fundamental beam theory Bell [2015],

$$\delta = \frac{FL^3}{EI} \tag{2}$$

The forces and Young's modulus are already predetermined by the problem and the length is the parameter of highest order.

For this optimisation problem there are three constraints that need to be satisfied. No bolts should overlap each other, bolts should not overlap tubes and no bolts should break. These constraints can be expressed as

$$\begin{aligned} a(\mathbf{x}) &:= \forall_{k \neq l \in n} d_{k,l} \geq 2 \cdot r_i \\ b(\mathbf{x}) &:= \forall_{i,j} d_{i,j} \geq r_i + r_j + t_{wall} \\ c(\mathbf{x}) &:= \forall_i \sigma_i \geq \sigma_y \end{aligned} \tag{3}$$

where $a(\mathbf{x})$ must hold for all combinations of two bolts and t_{wall} is the thickness of the tube walls. Due to the continuous property of optimisation problems these constraints should not be implemented as straight boolean constraints, but rather as continuous functions. This rewriting will be covered in the section for Python implementation in Section 3.2.

2.2 Calculating the force and stress in a bolt

This section shows how the stress in each bolt is calculated. The stress is required in order to make sure that no bolt is overloaded and the stress exceeds the yield strength. The pretension in bolts is assumed to be 75% of the yield strength, but this is a parameter that it would be easy to change or make problem dependent. Another assumption in this section is that the length of the threaded area is long enough to where the stresses in the threads can be disregarded. If any of these assumptions are wrong, it is possible to change or include them later.

By assuming that calculating the stress in bolts is a linear problem, the axial force in the bolts can be calculated from a bolt pattern and a set of normal forces. In a symmetric problem, where the bolt pattern and the forces are symmetric, the forces are evenly distributed among the bolts. However, this is not always the case for this code. Here the bolt pattern is changing with every step of the optimizer and the force distribution is required in every step.

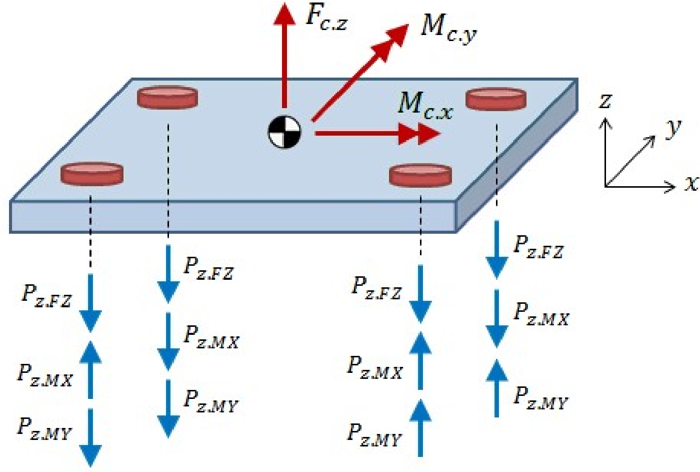


Figure 2: Forces and moment on the centroid.

Source: https://mechanicalc.com/reference/bolt-pattern-force-distribution

Figure 2 shows how the moment and normal forces on a centroid will contribute to the force in each bolt, which can be calculated by

$$F_i = P_{i,FZ} + P_{i,MX} + P_{i,MY} \tag{4}$$

$P_{i,FZ}$ is the easiest value to calculate, since the normal F_z is evenly distributed as a pressure over all bolts. By combining the formula for axial stress,

$$\sigma = \frac{N}{A}, \tag{5}$$

and the assumption that all bolts have the same area A_i, the formula is

$$P_{i,FZ} = \frac{F_{c,z}}{n} \tag{6}$$

$P_{i,MX}$ and $P_{i,MY}$ are harder to calculate, mostly because they are dependent on the bolt pattern[3]. The first step to finding these values is to calculate the centroid of the bolt pattern,

$$\begin{aligned} \bar{x} &= \frac{\sum_n x_i}{n} \\ \bar{y} &= \frac{\sum_n y_i}{n} \end{aligned} \tag{7}$$

For simplicity, denote

$$\Delta x_j = \bar{x} - x_j \quad and \quad \Delta y_j = \bar{y} - y_j$$

[3]This needs to be re-calculated in every step. In the implementation section it is shown how this can be used.

With the centroid calculated, the contribution to moment on the centroid can be calculated by

$$M_{c,x} = \sum_{j=1}^{m} F_j \cdot \Delta x_j$$
$$M_{c,y} = \sum_{j=1}^{m} F_j \cdot \Delta y_j \tag{8}$$

Using the formula for bending stress,

$$\sigma = \frac{M}{I_y} \cdot y, \tag{9}$$

it is possible to calculate the distribution of stress from $M_{c,x}$ and $M_{c,y}$ to each bolt. In order to use this formula the area moment of inertia, about x and y, for the pattern needs to be found. To do this the parallel axis theorem can be used which yields

$$I_x = \sum_{i}^{n} A_i \cdot \Delta x^2$$
$$I_y = \sum_{i}^{n} A_i \cdot \Delta y^2 \tag{10}$$

With the moment and area moment of inertia calculated, Equations (5) and (9) can be combined to calculate $P_{i,MX}$ and $P_{i,MY}$.

$$P_{i,MX} = \frac{M_{c,x}}{I_x} \cdot \Delta x \cdot A_i$$
$$P_{i,MY} = \frac{M_{c,y}}{I_y} \cdot \Delta y \cdot A_i \tag{11}$$

To check if a bolt is yielding, the stress in a bolt is required. This is calculated in accordance to Collins et al. [2010][p. 498] where the force in the bolt is

$$F_{bi} = \left(\frac{k_b}{k_b + k_m}\right) F_i + F_p, \tag{12}$$

where F_i are the forces from hydraulic tubes and F_p is the pretension in the bolt. Combining Equations (5) and (12) it is possible to find the stress in a bolt,

$$\sigma_i = \frac{F_{bi}}{A_{unthreaded}} \tag{13}$$

In this calculation the unthreaded area of a bolt is used, which will result in higher stresses and can be considered a safe value.

2.3 Minimize - SLSQP

SciPy is a module for Python which contains pre-written code to aid in scientific calculations. For this problem it is mainly the *optimize* file and the optimisation algorithms that are of interest; the documentation can be found in (The SciPy community). The *minimize()*-function has multiple required and conditional parameters. The main ones are the function to optimize, an initial guess, bounds, constraints, and the method to solve the minimization problem. The method that is being used in this code is the *SLSQP*-method, which is a quasi-Newton method using BFGS described by Kraft [1988]. *SLSQP* uses a Lagrange function consisting of the optimisation function, equality- and inequality constraints,

$$\mathcal{L}(x, \lambda, \sigma) = f(x) - \lambda b(x) - \sigma c(x) \tag{14}$$

In every step k, the method will calculate a new direction to look for a smaller value. The search direction is found by solving the quadratic subproblem Nocedal and Wright [2006], given by

$$\begin{aligned} \min_{d} \quad & f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T d + \frac{1}{2} d^T \nabla^2_{xx} \mathcal{L}(\mathbf{x}_k, \lambda_k, \sigma_k) d \\ \text{s.t.} \quad & b(\mathbf{x}_k) + \nabla b(\mathbf{x}_k)^T d \geq 0 \\ & c(\mathbf{x}_k) + \nabla c(\mathbf{x}_k)^T d = 0 \end{aligned} \tag{15}$$

The *SLSQP*-method should be considered a greedy algorithm, as it is looking for a direction to go next based on the steepest decline. This induces that the code may find local minimums, which tells that the user should always validate the results before continuing.

2.4 Minimize - trust-constr

Trust region methods are a different type of local optimisation algorithms compared to SLSQP. Trust region methods uses a model function to expand or contract the trust/search region, and the model function is often a quadratic function. This is done by calculating the ratio between the actual and modeled differentiations given some $\Delta\mathbf{x}$.

$$\rho = \frac{f(x) - f(x + \Delta\mathbf{x})}{m(x) - m(x + \Delta\mathbf{x})} \tag{16}$$

If ρ is sufficiently small the model is a bad representation and the region should be contracted, and if the ratio sufficiently large the region should be expanded.

For this thesis the regular implementation of SciPy's "trust-constr" will be used, as this method is the only trust region method in SciPy that supports constraints. The implementation that is used for inequality constraints is based on Byrd et al. [1999].

2.5 Differential evolution

Differential evolution is a global optimisation algorithm that is stochastic of nature. The method will find trial candidates by mutating other candidate solutions. In Storn and Price [1995] two strategies to find the candidate solutions are proposed, and scheme DE1 is referred to as *best1bin*.

best1bin creates a potential candidate, v,

$$v = \mathbf{x} + F \cdot (\mathbf{x}_1 - \mathbf{x}_2) \tag{17}$$

using the best candidate this far combined with 2 randomly selected vectors. Here $\mathbf{x}$ is the best candidate so far, $\mathbf{x}_1$ and $\mathbf{x}_2$ are two randomly selected candidates, and the generic mutation factor $F > 0$. If the potential candidate gives a better function value than the best candidate, the best candidate is updated. This is one iteration in differential evolution which is repeated.

In SciPy's implementation of this algorithm there is an option to polish the solution. If the solution is polished, the best candidate, after differential evolution, is used as the initial guess for the "trust-constr"-method.

2.6 Design of Experiment

The design of experiments is a method used to describe and explain the variation of information given conditions that are hypothesized to have a significant impact. According to Montgomery [2013] an experiment is defined as a "test or series of runs in which purposeful changes are made to the input variables of a process or system so that we may observe and identify the reasons for changes that may

be observed in the output response". This process involves two main steps - planning and conducting the experiment, and analyzing the resulting data.

In 1935 R. A. Fisher published a book about design of experiments and the book introduced multiple concepts. One of these concepts is the factorial design. In his book it is claimed that many scientists look at one factor at a time. In the cases they are not looking at only one factor it is not because of an ideal scientific procedure, but because it would be expensive and time consuming to do single variable tests. Fisher proceeds to claim that this is exaggerated, and it is possible to acquire knowledge from the results more efficiently(Fisher [1935][p. 97-99].)

A special case of factorial designs are 2-and 3-factorial designs. These are designs where each factor only has two or three levels respectively, with both having low/high level and the 3-factorial design also having a medium level. The benefit of these designs is that they allow for the fewest runs for each factor, which is really beneficial for the first experiments where there are multiple factors. This can then be used to filter out significant factors before a new experiment is conducted. The 2-factorial design is best suited where a linear response is expected from the factor, and the 3-factorial design is required if the response is not expected to be linear(Montgomery [2013]).

After the tests are ran, the next step is to analyse the results. This is done with a regression analysis of a model, where the model is a polynomial containing the parameters. For a model with two factors, x_1 and x_2, and their interactions, the model is

$$u = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \beta_{12} \cdot x_1 x_2, \tag{18}$$

where β_i is the coefficient for parameter x_i. The number of terms in the model increases with more base factors and factor interactions.

With a fitted model the next step is to evaluate the fit using the results. The main result values that will be considered in this thesis are the regular and adjusted R-squared values, and the p-value of each parameter. In regression analysis R-squared is a statistical measure of how good the model predicts the real values, and adjusted R-squared is a measure that in addition takes the number of observations and number of describing variables into account. R-squared can be calcu-

lated with

$$\begin{aligned}
\bar{y} &= \frac{1}{n}\sum_i y_i \\
SS_{tot} &= \sum_i (y_i - \bar{y})^2 \\
SS_{res} &= \sum_i (y_i - f_i)^2 \\
R^2 &= 1 - \frac{SS_{res}}{SS_{tot}}
\end{aligned} \tag{19}$$

where SS_{tot} is the total sum of squares and SS_{res} is the residual sum of squares. The residual sum of squares is based on the deviations between the predicted value, y_i, and the actual value, f_i.

The other results of interest used in this thesis are the p-values, which can be used to tell the significance level of a parameter. A p-value is used in null hypothesis testing, given a null hypothesis H_0. In regression analysis the null hypothesis is

$$H_0 : \beta_i = 0, \tag{20}$$

for each of the β_i coefficients in the model. The p-value for each parameter tests the null hypothesis that the coefficient is equal to zero (no effect), and this test is done by comparing the full model with the reduced model. A low p-value indicates that the null hypothesis should be rejected, because the reduced model has problems explaining every result. Thus it is a test of significance. The cutoff value for p-values can be found using a p-value table or by using the widely used cutoff at 5%. In a p-value table, the cutoff value is found as a function of the significance level, number of observations and number of terms in model.

2.7 Algorithm selection

The evaluation of the results will be focused around the selection of algorithm and the design of objective function. In Entner et al. [2019] a systematic framework is presented to find a suitable algorithm for a box-type crane. The goal is to select the best optimisation to help improve the solutions and a decision matrix is used to evaluate the functions. This matrix used to the suitable algorithm for the box-type crane is designed around multiple criteria.

In this thesis only some of the criteria will be used to evaluate the results, and the main criteria for this thesis are:

- **Convergence to optimum value:** The capability to always converge towards near-optimal solutions.
- **Generation of new design alternatives:** The capability to generate distinct design alternatives other than those obtained by small variations of the initial product configuration.
- **Computational resource need:** The computational resources, e.g., number of function evaluations or run times, required to obtain a high-quality solution. One aims at minimizing the need for computational resources.
- **Reproducibility:** The capability of an algorithm to provide the same, or close to, solution by rerunning the algorithm with the same setup. Reproducibility is to be maximized.

Criteria that are not used in this analysis are disregarded because they are irrelevant or equal for the different algorithms presented in this thesis. An example of such a criterion is "how hard it is to integrate in operating systems (CAD, CAE)", as this integration is the same for all objective functions and algorithms.

2.8 Objective functions

In this section an attempt is made to define the objective functions and also give some information about their background. Through the work several iterations have been tried and tested. In the specialization project leading up to this thesis (Steen [2020]), **f1** did not prove to be a good model for this problem, and it was suggested to work on a better objective function. The problem with **f1** was that it centered the bolts if there was free space in the middle, and the change suggested in the specialization was to reduce the distance to the closest bolt for each tube hole.

This leads to **f2**,

$$\mathbf{f2}(\mathbf{x}) := \sum_j min(d_{ij}), \tag{21}$$

where the idea is to reduce the distance to the closest bolt from each tube. This is done to counteract the centering effect in **f1**.

The third objective function is also designed around Equation (2) with the assumption that the moment of inertia is a function of the flange thickness and the bolt head radius. This extra function was an idea that came at the time where the Abaqus simulations were automated and it was thought of as an improvement to **f2**. It is given by

$$\mathbf{f3}(\mathbf{x}) := \sum_j (F_j \cdot min(d_{ij})^3) \tag{22}$$

After the Abaqus simulations were automated, it was clear that the functions based on beam theory were not sufficient. This discussion is covered in Section 5.1.

The fourth objective function is based on the results of the experiment and after evaluating the earlier iterations of the objective functions. The selection of factors and their exponents will be discussed in Section 5.2. **f4** is given by

$$\mathbf{f4}(\mathbf{x}) := max \left\{ F_j \cdot \mathrm{f1}(\mathbf{x})^2 \cdot min\left(d_{ij}\right)^2 | j \right\} \tag{23}$$

Another change in **f4** is that it is not a sum for each tube, because this could lead into problems where the result could be optimized into having "all" the deformation around one tube. To prevent this from happening, the function is changed to only look at the worst tube. The function still needs to have differentiable properties, because the *differential evolution* algorithm in SciPy polishes the solution with trust-constr. The function is still continuous and remains its differentiable properties, but it is possible that two similar values will be based on two different tubes.

3 Python implementation

3.1 Support functions

All of the code covered in this section can be found in Appendix B.The functions in this file are not directly used in the optimisation, but they contain functions that are indirectly used for the optimisation. They are used to calculate force distribution and getting the bolt- and material specific values. In line 7-32 the function chooseBolt() is used to get the bolt-specific variables, and chooseBaseMaterial() does the exact same for the material. Another function that is used in the initialization is calculateForceOnCentroid() (lines 59-63). As the input is given as tube center, radius and pressure, this must be calculated as one force on the centroid. By summing the resultant force of each tube,

$$F_{c,z} = \sum_{j=0}^{m} P_j \cdot A_j, \tag{24}$$

the total normal force on the centroid is obtained. The following functions explained will be called in every step, because they are dependent on $\mathbf{x}$. For every step the force from hydraulic tubes must be calculated, using Equation (4). There are three contributing factors to this distribution - the total normal force on the centroid ($P_{i,FZ}$), the force from moment on centroid about x ($P_{i,MX}$) and about y ($P_{i,MX}$). By looking at Equation (6) it is clear that this is not dependant on bolt position, it is calculated once and then added in every step according to Equation (4). The function patternCentroid() (lines 65-72) finds the centroid of $\mathbf{x}_k$ using Equation (7), and patternInertia() (lines 74-80) finds the second moment of area using Equation (10). centroidMoment() (lines 82-88) calculates the moment on centroid about x and y using Equation (8). Finally the distribution of forces can be calculated using boltForces() (lines 47-56), where the calls to find the relevant variables are done, and the force in every bolt is found.

3.2 Implementation of constraints

As the local minimization methods are dependant on the constraint functions being continuous and differentiable, a correct implementation should return a value reflecting how wrong it is. Constraints in

SciPy.optimize are defined such that invalid solutions should return a negative value, and valid solutions should return a value greater than or equal to 0. With this in mind it is possible to re-write the constraints given in Equation (3). For the two first constraints, which are no overlapping between bolts or tubes, this can be implemented by using a punishment variable. The punishment variable starts at 0, and if there is overlapping, it subtracts a value that is increasing with lower distance. One way to obtain this can be

$$punishment- = \frac{1}{d_{i,j}} \tag{25}$$

The check for overlapping between two bolts is implemented as boltCrash() (lines 85-101) in Appendix A. The division by zero is covered by a try/catch block. To avoid lots of warnings being printed and a cluttered terminal window, 1nm is added to the distance. This value is added because subtracting could lead to subtracting negative values, which is an even worse result because that would be considered a valid solution by the program. tubeCrash() (lines 103-119) is an implementation of the constraint between bolts and tubes. This function returns a punishment based on the overlap and also uses Equation (25). The last constraint that is implemented is the test for bolt breaking, which is also an inequality constraint returning an increasing negative value if the stresses in the bolts are too high. boltBreak() (lines 121-140) is the function where this constraint is implemented, and in line 128 the forces in every bolt is calculated using boltForces(). The program then loops over every force and calculates the stress in a bolt according to Equation (13). forceBolt is equal to the force in the bolt from the operating load and pretensionOvershoot is F_p. With the total force in the bolt calculated, the stress in a bolt is calculated. The test to see if the boltStress is greater than ps, the bolt yield, then follows. If boltCheck is greater than zero, the excess stress is subtracted from the punishment.

3.3 Minimization setup and call

All the functions and variables used in the minimize functions are collected in *boltLocation.py* found in Appendix A. The constraints are defined, but the minimization call also needs the objective function and bounds in order to have all the required information. The bounds

are defined in line 32, as a list with the lower and upper values for each bolt that is multiplied by the number of bolts. The objective functions are implemented in lines 43-81. These functions are defined with the same notation as in Section 2.8, with small changes in **f3** and **f4**. The returned values are shortened by a factor of 1e-6 and 1e-12, so that the objective functions and constraints are of the same order. This change should have no effect on the ordering of different bolt placements, but it makes it easier to set the tolerances of optimisation methods.

Before the minimization calls, some extra options are defined. In line 152 and 152 these options are defined for the SLSQP-method and trust-constr-method. The options regulate the termination tolerances, the option to display more information about the optimisation and the maximum number of iterations.

The optimisation call is done on lines 179-181 with one line for each of the methods. In line 179 the differential evolution method is used - the input here is the objective function **f4**, the bounds and the constraints. This method does not require any initial guess - the length of **x** is found from the bounds and the function finds a good initial guess before the solution is polished with trust-constr. Another difference in this call is that the constraints need to be defined different, and they are defined on lines 148-150. The two local optimisation calls are done equally and these calls are shown in lines 180 and 181. The objective function **f4** is passed first and as these methods require an initial guess this is passed. All of the three methods return a result-object where values can be extracted in the same way. The lines from 182 and out are used to extract and save the results. These results are plotted and saved as a picture and they are saved as a JSON-file that is ready for the Abaqus simulation.

3.4 Automating Abaqus simulations

Abaqus CAE is the program that will be used for all the numerical simulations. Abaqus is selected because it is a stable, state of the art numerical solver, which is important to verify that the newly designed script is working as expected. The modelling process in Abaqus is very repetitive and time consuming if it was to be done manually. Fortunately Abaqus allows for multiple different automation methods to

help with this process - it is possible to make the *input*-files manually and it is possible to automate every action in Abaqus using Python. For this thesis the second option is used, and all the manifold-flange simulations are ran using a Python 2.7 script.

In Appendix C *abaqus_functions.py* can be found which contain the specific Abaqus functions, and *run.py* in Appendix D which tackles the translation from JSON-files[4] into Python-variables and function calls. The run file is altered for different cases and tests ran, where the file with Abaqus functions is the same. The Abaqus functions file is built up by first creating one simulation for this kind of problems. After the first simulation is ran, it is possible to get a **.rpy**-file that essentially is a Python replay file of all the commands that have been used in this simulation. The replay file paired with the Abaqus Scripting Reference Guide is used to build *abaqus_functions.py*.

3.5 Design of Experiment

After realising that previous iterations of objective functions proved to be bad heuristic functions for this problem, an experiment is conducted to investigate the relevant parameters using design of experiments. This conclusion is discussed more in Section 5.1. The experiment in this thesis is designed to create a model that is able to describe the separation in tubes. After the experiment is ran, these results are used in a regression analysis to see what parameters are important. Important parameters can then be used to design a more relevant simple objective function, and this new function will be completely heuristic and may not have any physical interpretation.

Factor	Degree
Total (Total distance - **f1**)	{1,2,3}
Min (Distance to closest bolt)	{1,2,3}
Max (Distance to furthest bolt)	{1,2,3}
Force (Radius of tube)	{1}
Tube_d (Distance to closest tube)	{1}

Table 1: Factors to check significance of.

Factor	Degree
Total	3-factorial
Min	3-factorial
Max	3-factorial
Force	2-factorial
Tube_d	2-factorial

Table 2: Factorial design

All the factors are calculated from a tube, i.e. the force in tube j, maximum distance from tube j to any bolt. The exception is the total-factor, which is calculated as **f1**. In order to create a good polynomial

[4] All the code for Abaqus scripts need to be in Python 2.7, but the other code in the thesis is written in Python 3.7. Because of this, the information is saved in JSON-files to transfer data between the two versions.

fit with higher degrees included, some of the factors requires a 3-factorial design. The factorial level of each factor is shown in Table 2.

An issue with this design is that there are high correlations between the factors, so the experiment is designed around this. The experiment is designed with 3 bolts and 2 tubes. In each run one value will be changed, and as there is no randomness the experiment is only done once. A quadratic flange is used where the size in x and y direction is 100mm, the thickness of flange is 20mm and the wall thickness of the tubes are 2mm. The values for the bolts and tubes are defined as Python dictionaries for each bolt. The number of tubes is decided to be two because the program aims to solve problems with multiple tubes.

```
bolt1 = {
    'L' : [10, 26],
    'M' : [10, 50],
    'H' : [10, 85]
}
bolt2 = {
    'L' : [25, 25],
    'M' : [50, 50],
    'H' : [75, 75]
}
bolt3 = {
    'L' : [26, 10],
    'M' : [50, 10],
    'H' : [85, 10]
}
tubelocation = { #mm [x,y]
    'L' : [[40,40],[60,60]],
    'H' : [[10,10],[90,90]]
}
tuberadius = { #mm
    'L' : [2, 2],
    'H' : [5, 5]
}
```

A run is created for every combination of these five variables. The simulation is ran in Abaqus, and the resulting separation is saved for both of the tubes. This result is later used in the regression analysis.

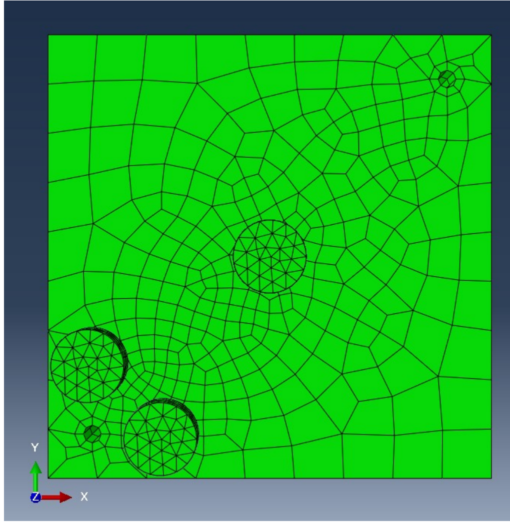


Figure 3: LMLHL run

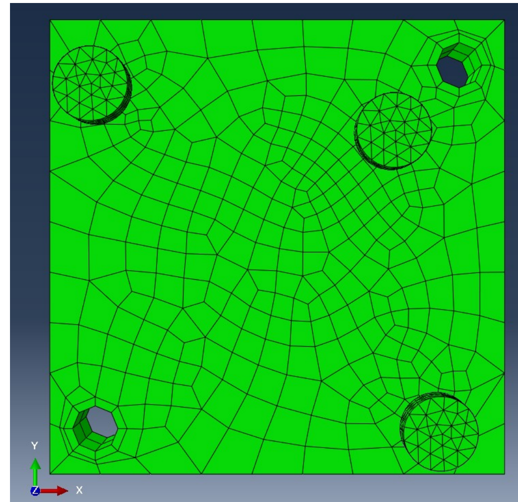


Figure 4: HHHHH run

Figure 3 shows the LMLHL run, where the three first letters indicate the level of bolt1-3, the fourth letter is the distance between tubes and the last letter is the radius of each tube.

The regression is done using the input and results from the Abaqus simulations. The output is used as is, but the input needs to be processed before it can be used in the regression analysis. The values of factors in Table 1 are calculated and used as input in statsmodels library[5] for Python. The most important input for this function is the formula, Equation (18), and a Pandas dataframe. The regression is done in Appendix E. The formula in line 96-126 is the full formula for every factor and degree, but this is not the same formula used in the results. This decision is explained in Section 4.2.

[5]This library is a lite version of R for Python.

4 Results and Discussion

4.1 Testing objective functions

A test was conducted to see if the beam theory inspired objective functions gave a good representation of the separation. This was done through a setup with 2 tubes and 3 bolts, where the distance to one bolt varied and the two other bolts where stationary. In Figure 5 and 6 the two end points of the test is shown.

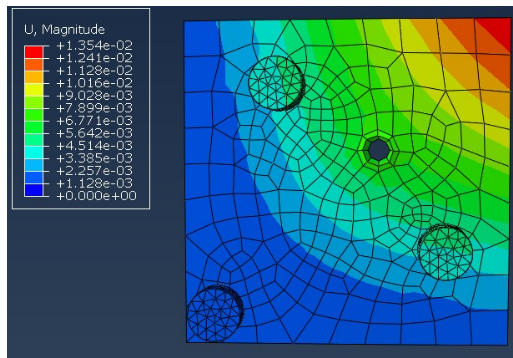


Figure 5: Deformation of case with lowest distance to bolt.

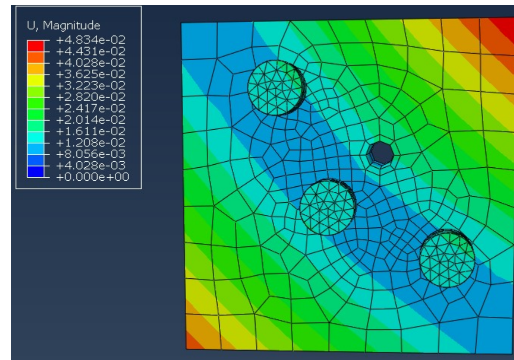


Figure 6: Deformation of case with highest distance to bolt.

Figure 7 shows a plot with the estimated stiffness and the numerically calculated stiffness. The estimated stiffness is found by estimating the deflection with **f3** and dividing by the force. The same procedure is followed for the numerical stiffness, where force is divided by the separation found in Abaqus.

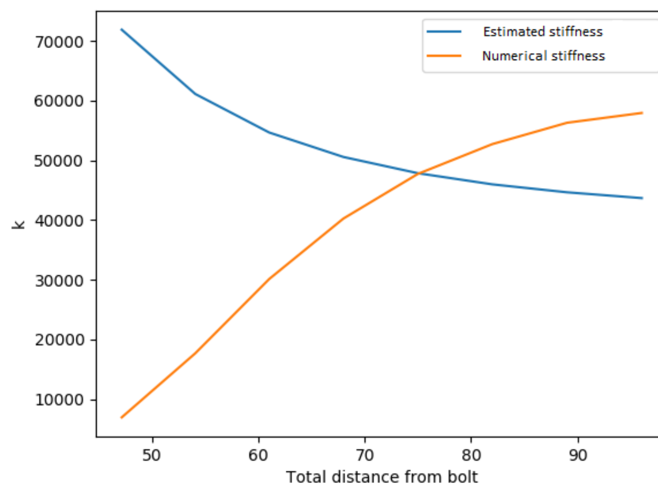


Figure 7: Estimated stiffness vs numerical stiffness.

4.2 Design of experiment results

The experiment is conducted to try and find a connection between measurable parameters and the resulting separation, in order to establish an objective function for the optimisation problem. The regression has been ran multiple times with different parameters. The results have varied from R-squared values as low as 0.465[6] and up to 0.983. A common thing for the results with R-squared values higher than 0.9 is that complex interactions are the most significant parameters, but the level of significance is varying between the regressions. One of the regression results is presented in Figure 8 and 9. This result is chosen because it has only 10 significant parameters and the fit is still good. The best regression results are achieved with 85 significant parameters. In Section 5.2 it will be discussed more in detail why these regression results are not used.

The regression summary includes more information about every parameter combination than Figure 9 shows, but this information is not used. For this model that would be 45 different parameters with coefficients, p-values and t-statistics. To prevent having too many results to look at, the parameters are filtered on the p-values and only parameters with low p-values are shown. The coefficients are β_i-values in Equation (18) and I(Total ** 2):I(Min ** 1):Force:Tube_d is the parameter where these factors are multiplied together. A positive coefficient indicates that the separation, u, along the tube edges increases with increasing parameter value, and negative coefficients indicates a reduction in separation.

[6] These values where found early with a low amount of factors and interactions

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      u   R-squared:                       0.931
Model:                            OLS   Adj. R-squared:                  0.908
Method:                 Least Squares   F-statistic:                     41.18
Date:                Thu, 20 May 2021   Prob (F-statistic):           1.50e-59
Time:                        12:35:27   Log-Likelihood:                 108.25
No. Observations:                 180   AIC:                            -126.5
Df Residuals:                     135   BIC:                             17.18
Df Model:                          44
Covariance Type:            nonrobust
```

Figure 8: Header of regression summary

```
Parameter                                              coeff       p-value
---------------------------------------------------  ------------  -----------
I(Total ** 2):I(Min ** 1):Force:Tube_d                3.08796e-10  7.59145e-08
I(Total ** 2):Force:Tube_d                           -3.13783e-09  0.000641836
I(Total ** 2):Force                                  -2.70422e-11  0.00154126
I(Total ** 2):I(Min ** 2):I(Max ** 2):Force:Tube_d    8.86735e-15  0.00368518
I(Min ** 2):I(Max ** 3):Force:Tube_d                 -1.61323e-13  0.00420156
I(Total ** 2):I(Min ** 2):I(Max ** 3):Force:Tube_d   -1.35532e-16  0.00980017
I(Total ** 2):I(Min ** 2):I(Max ** 3):Force           1.38356e-14  0.0194537
I(Total ** 2):I(Min ** 2):I(Max ** 2):Force          -7.4136e-13   0.0298754
I(Min ** 2):I(Max ** 2):Force:Tube_d                  7.88388e-12  0.0349007
I(Total ** 2):I(Min ** 3):I(Max ** 2):Force:Tube_d   -9.3456e-17   0.0487721
10
R-squared: 0.931         Adj. R-squared: 0.908
```

Figure 9: The parameters with p-values less than 0.05

4.3 Local optimisation results

In this section the results of running the two local optimisation methods 10 times are shown, and Figure 10 and 11 shows the worst results for the two optimisation methods.

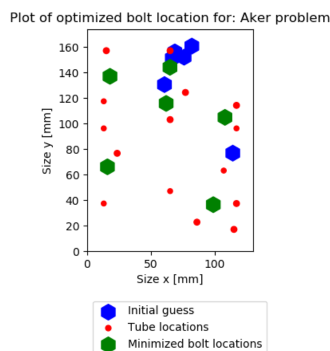


Figure 10: A plot showing the bolt placements of Run 5 using the trust-constr-method with **f4**.

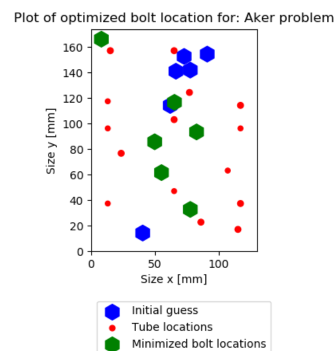


Figure 11: A plot showing the bolt placements of Run 8 using the SLSQP-method with **f4**.

Table 3 shows the results of all the 10 runs, with the function values and the numerical values. The function values are all calculated using *abaqus_functions* with random bolt placements as input.

Run	Trust Function Value	Trust Numerical Value	SLSQP Function Value	SLSQP Numerical Value
Run 1	31.2	15.3 μm	65.7*	24.9 μm
Run 2	33.5	17.6 μm	32.8	17.1 μm
Run 3	87.9	33.3 μm	41.7	18.4 μm
Run 4	34.2	16.6 μm	72.6	34.2 μm
Run 5	80.5	67.5 μm	34.6	19.1 μm
Run 6	25.2	16.6 μm	40.6	20.4 μm
Run 7	28.6	22.0 μm	33.6	19.4 μm
Run 8	45.8	30.2 μm	43.7	17.2 μm
Run 9	32.4	17.4 μm	64.2*	41.2 μm
Run 10	28.7	23.3 μm	40.9	25.8 μm

Table 3: The result of the 10 runs for both methods with objective function **f4**. The runs marked with * are runs that exceeded the iteration limit.

The highest deformation is around the same tube for all the runs - the lower right tube always has the highest deformation on the lower right side. In Run 1 and Run 9, with SLSQP solver, the optimisation terminated at 1500 iterations due to reaching the iteration limit. For Run 1 that resulted in an illegal state with two bolts overlapping, this result will not be used in any explanation. An important note about these results is that the function values are not meant to be equal to the numerical value, and the objective function does not have a dimension. This should be considered as a factor or value that tries to estimate order of the separation for each tube.

4.4 Global optimisation results

The same test is carried out for the differential evolution algorithm, with 10 runs to see how the results, function and numerical values, vary.

Run	Function Value	Numerical Value
Run 1	23.8	18.7 μm
Run 2	33.8	19.5 μm
Run 3	28.5	16.4 μm
Run 4	23.7	19.1 μm
Run 5	24.4	19.1 μm
Run 6	24.1	18.3 μm
Run 7	23.5	18.1 μm
Run 8	24.1	18.7 μm
Run 9	30.3	19.4 μm
Run 10	24.1	19.0 μm

Table 4: The result of 10 runs with differential evolution and objective function **f4**

The highest deformation is around the same tube for all the runs, with the lower right tube having the highest separation. In Figure 12 and 13 the bolt locations are shown for the two runs with the same numerical value.

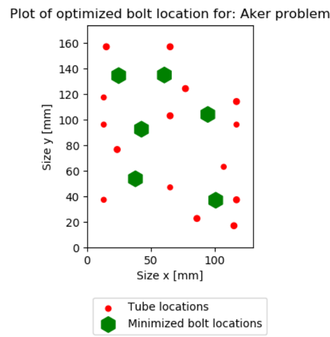


Figure 12: A plot showing the bolt placements of Run 4 using differential evolution with **f5**.

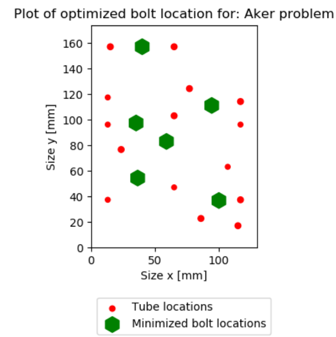


Figure 13: A plot showing the bolt placements of Run 5 using differential evolution with **f5**.

5 Discussion

5.1 Evaluating objective functions f2 and f3

*The main focus of this section is to discuss the results of the two objective functions **f2** and **f3**. These results were obtained before the experiment and it is the main reason to why the experiment was done.*

The first results from testing **f2** and **f3** were promising, as they reduced the maximum separation around the tubes with a good initial guess. This indicates that the functions with SLSQP can be used to polish an existing solution, as it can estimate the separation from small placement changes. In Figure 7 it becomes clear the assumption that the deflection is closely tied to beam theory is wrong. The numerical results indicate that the stiffness and deflection are related to more than just the force and total length, and it also indicates that the total length can have an inverse reaction on the separation. Even though the stiffness is only calculated with **f3**, this plot disregards the use of both **f2** and **f3**. This is done because the only varying parameter in these simulations is the placement of the one bolt, thus the distance is the only measured difference and neither of the functions would be able to describe this effect.

These variations can be explained by looking at Figure 6. In this simulation it is possible to see the "hinge"-line that arises and that the entire flange will rotate around this line. This indicates that the function may be closer tied to the second moment of area and that spreading the bolts out can reduce the hinge-line and create a constrained area. Figure 5 shows how the spread out bolts contribute with more stiffness because no hinge-line is present.

Another problem with **f2** and **f3** is that they may "forget" about bolts, which can happen if one or more bolts are not the closest to any tube. If that happens it will not be part of the objective function in any way, and if the placement of that bolt is changed it will not affect the objective function, but it will have an impact on the simulation. The extreme case of this issue is if one bolt is the closest to all the tubes. In this case the objective function will think that this is the only bolt that contributes to the stiffness.

5.2 Design of Experiment

The main results of the design of experiment are shown in Figure 8 and 9. Figure 8 shows the header that is automatically created by statsmodels and it gives information about the fit of the regression. Figure 9 shows the significant parameters sorted on the p-value. The selected analysis is chosen because it is the most efficient analysis. In this context efficient means that it is able to describe the separation with a simpler model, the R^2 is good with 45 terms and only 10 significant parameters. It is important to remember that this is a purely heuristic model and that these values and parameters do not need to correlate to any theory, only that these parameters can describe the separation.

A recurring result from all the regression analyses is that the more complex interactions always are the most significant parameters, and that Force is part of the interaction. The exact combination that is the most important changes between each analysis. In an answer on JMP Blog, Phil Kay states that bad correlation between factors does not imply that it is impossible to make good predictions, but that it can be hard to understand the effect of the factors. This is very much what is seen for the different models, the significance of a parameter can very high in one model and for the next model it is further down the list.

The biggest problem with using the most significant parameter from Figure 9 as an objective function is the effect of increasing distance between tubes, Tube_d. This is probably caused by a flaw in the experiment where there is no real case where tubes are close together and far away from the center of the bolts. Regardless of this, the parameter is a good representation of the most significant parameter, it is combined of Total**2, Min**1, Force and Tube_d. In some of the models Max is also part of the most significant parameter, but because of the correlation between factors it is hard to get a good understanding of these factors.

The results of the experiment supports the claim that **f2** and **f3** should be discarded as objective functions, and that a more complex function should be implemented. **f4** is the objective function that is made as a result. The new function is based on the results of the experiment and

the most significant parameter, I(Total ** 2):I(Min ** 1):Force:Tube_d.

- **I(Total**2)**: This is the total distance between each bolt and each tube squared, and it is calculated using $\mathbf{f1}^2$. Although this factor alone made the solution worse, it also helps negate the hinge problem. Because the force in each tube is about the same, this will help center the solution in the middle of all forces which helps reduce the moment in any direction and thus reduce the hinge effect. The effect of this can be seen in Figure 12 where the bolts are placed along the diagonal from top-left to bottom-right. This is different from the initial guess where the bolts are symmetrically placed along the x- and y-axis.
- **I(Min**1)**: This is the minimum distance from each tube to any bolt. This factor has also proved to be a bad estimator alone, but in combination it is present in most parameters that are significant. In the good solutions, this factor seems to have a spreading effect - it counteracts the centering effect in the way that it pulls bolts towards the tubes.
- **Force**: This factor is part of all the significant parameters which is expected. The force is what generates the separation and its size is therefor of importance.
- **Tube_d**: This is a measure of the distance to the closest tube. Although this factor is part of the significant parameters, it is a hard factor to include in any objective function because this value decreases when the separation increases. In order to use this in a objective function a new term would have to be added and this could lead to negative estimations, which is something that cannot happen in the simulations.

The other significant parameters in Figure 9 indicate that higher degrees of Min and Max also are needed to make a good model. This leads to the other change in **f4**, the degree of min is increased to cubed from linear. The goal of this change is to reduce the centering effect and indirectly force the algorithm to maximise the second moment of area. With Tube_d removed and the degree of Min increased, **f4** is obtained and defined as Equation (23).

5.3 Algorithm selection

The three optimisation methods that have been used will be compared using the criterions from Section 2.7. An important difference between trust-constr and SLSQP compared to differential evolution is that the first two are local optimisation algorithms, while differential evolution aims to be a global optimisation algorithm. When it comes to runtime and computational power required "There ain't no such thing as a free lunch", so it is expected that differential evolution performs worse on this point. The question is how the extra runtime compares to the extra manual work required by the local optimizers.

Trust-constr

- Convergence to optimum value: As seen in Figure 3, Run 3 and Run 5 trust-constr will not always converge to the optimum value.
- Generation of new design alternatives: As this is a local optimisation algorithm and these algorithms are principally dependent on the initial guess, new designs are not something that can be expected. In some cases it may be able to find new design alternatives, but this may be just as much a result of the objective function and not the optimisation method.
- Computational resource need: In terms of memory usage this method is not very demanding. The runtime of this optimisation usually lies between 1-5 seconds and in some cases it can run up to 20 seconds. It is probably possible to improve this with better tolerances and maximum iterations for when to terminate the optimisation.
- Reproducibility: With the same initial guess this method will get the same results, but with different random guesses it does not give the same results.

SLSQP

- Convergence to optimum value: As seen in Figure 3, this method is not able to converge to the optimum value given a random guess. For two of the runs it was not even able to find a valid solution within 1500 iterations.

- Generation of new design alternatives: As this is a local optimisation algorithm and these algorithms are principally dependent on the initial guess, new designs are not something that can be expected. In some cases it may be able to find new design alternatives, but this may be just as much a result of the objective function and not the optimisation method.
- Computational resource need: The SLSQP-method is the best method with computational resources in mind - it performs the least amount of function evaluations and it is the fastest. This function could probably be cut off at 100 iterations because the runs were under 100 or above 1500 runs.
- Reproducibility: This method will yield the same results with the same inputs, but it is dependent on the initial guess.

Differential evolution

- Convergence to optimum value: As seen in Figure 4, this method yields low values for the objective function. What is more interesting is the stability in numerical value.
- Generation of new design alternatives: The random factor in differential evolution combined with the parameters indicates that it may be able to find unexpected design that gives low function values.
- Computational resource need: This algorithm is notably slower than the two local optimisation algorithms. The differential algorithm can be split into two steps - first it is trying to make a good initial guess as stated by Equation (17), and the second step is a trust-constr optimisation used to polish the solution. The second step is done to make sure that it finds the local minimum in what is expected to be the global minimum area. Both of these steps can take up to 20 seconds to do, making the total time as high as 40 seconds. Due to the way it is implemented in SciPy it is not possible to change the options of the trust-constr optimisation, but this can probably be improved so the second step becomes faster.
- Reproducibility: The first step of differential evolution is a way to make random educated guesses, so the initial guess sent to trust-constr will vary. It is therefore expected that the solutions will not

be the same, but they are similar based on the bolt placements and the objective functions. This specific solution may be affected by the quasi continuous objective function, and that the function value can be the same with different bolt placements.

From the results of Section 4.3 and 4.4, differential evolution performs better and more stable compared to the local optimisation algorithms, while trust-constr seems to be the better local optimisation algorithm. This can be a result of differential evolution being an extension to trust-constr - it first makes an educated guess for a good initial guess, before it polishes the solution with trust-constr.

Even though differential evolution is the slower of the three algorithms, it is still the preferred algorithm. This is because it does not require any initial guess and it seems to generate good initial guesses on its own. Without the requirement of an initial guess, the program could also be shortened by a few line. This change also helps in making the program dependent on the choice of bolt type and number of bolts.

5.4 Evaluating f4

By closer examination of Table 3 and 4 it becomes clear that **f4** is not able to describe all the separation, or stiffness, in the solutions. The differences between the objective function and the numerical separation are bigger using the local optimisers compared to differential evolution. If the approval of objective functions were done solely on local optimisation algorithms **f4** should be disapproved, but with differential evolution it is another story. When differential evolution is used, this objective function seems to be even more stable based on the numerical value compared the function value. The most obvious reason for this partial correlation is that only one term of the experimental model is used, where the best model is built with 256 terms.

A concern with designing the objective function on the experiment was to see how it performed on more complex problems. The idea behind design of experiments is to perform a series of small runs to identify how different factors affect the result, and the experiment conducted is very different from the "Aker problem". Only 2 tubes and 3 bolts are used in the experiment, while the "Aker problem" consists of 15 tubes and 6 bolts. The two main differences between these problems is the

difference in scale and the different relations between bolts and tubes. Even though the objective function shows bad correlation between estimated separation and numerical separation, it is a good sign with regards to robustness and scalability. The definitions of robustness and scalability are taken from Entner et al. [2019]. Robustness is a measure of how good the optimiser handles small changes in the problem, and scalability is a measure of how good dimensional changes are handled.

6 Conclusion and further work

6.1 Conclusion

This thesis described a method to automate the placement of bolts in subsea manifold applications. The method presented aims to propose a good bolt placement that is able to join a flange and manifold together with very small acceptable separation. With the progress that is made within 3D-manufacturing, these new methods to propose and test designs are required. Subsea manifolds is a field where the new manufacturing technologies can be used to save resources and money, but the design is not yet automated.

The 'bolt placement' application utilises existing numerical solvers in combination with a new program in order to recommend and evaluate a solution. Optimisation is used to propose these new solutions and the thesis focuses on the development and verification of this optimisation problem. For this specific optimisation the design of objective function and the selection of solver are the most important issues. As the goal is to automate with little to none manual work, a combination that gives a good global candidate is required.

The combination of differential evolution and **f4** seems to be a good optimiser for this type of problems. In the real life problem the program is able to deliver consistent solutions which are significantly better than the manually made solution. The bolt placement that is used as the initial guess in the "Aker problem" is the humanly made solution, and even though this solution passed simulation, the solution has been problematic. Abaqus reported a separation of 25.6 μm, which is very close the requirement of 26 μm. The results shown in Table 3 and 4 indicate that the objective function is not able to describe the separation, but these results alone are not enough to disapprove of this solution. Without more realistic problems it is hard to identify if the partial correlation is a result of the reduced model, the difference in the problem, or if it is a combination of both factors. When differential evolution is used, all of the suggested bolt placements meets the requirements. This makes it hard to identify if there is something that needs to be worked on or if the program is ready to be used. Nevertheless, more testing should be done on real life problems to see how the program performs.

6.2 Future work

Different tests and adaptations of the objective function have been left for the future due to time and other factors. The natural steps to test program:

1. It could be interesting to see how the optimiser performed in other real life examples where human developed solution is used as a measure. This could also be used to test and compare other objective functions.
2. The Abaqus modelling should be reviewed by a field expert or someone with more experience. This should be done to assure that the method used to verify solutions is correct and used in the industry. If the simulations are done, it is possible that the experiment and analysis needs to be reconducted. This is because the optimisation is specially designed for this modelling. With the changes in the modelling, this may change what parameters are important to model the separation.
3. If the proposed solutions are accepted by an industry expert, then a physical test should be conducted.
4. More details probably needs to be added to the program. Some of the tubes may have valves or other elements tied to the hole which may take up space, this would need to implemented in the constraint that checks for crash between bolts and tubes. The bolts may also have washers that will increase the space they occupy. These details could be added as optional parameters with default values.

Using the code

The code included in the appendix is not all of the code used throughout this thesis, it is just the most important files and functions. In Appendix A *boltLocation.py* is found where the optimisation is done, Appendix B contains the support functions that is used by *boltLocation.py* to calculate distribution of force in bolts. *run.py* found in Appendix D is the Python file ran in Abaqus which reads data from a JSON-file and initiates the Abaqus functions. All the Abaqus functions are defined in *abaqus_functions.py* which is found in Appendix C.

Bibliography

T. Røkke. Exploring the design freedom in additive manufacturing for manifold blocks with minimum cost algorithms. 2019.

H. H. Hersleth. Designing next generation hydraulic manifolds with minimal material usage and competitive lead times. 2020.

H. B. Steen. Designing manifolds using optimization. 2020.

K. Bell. *Konstruksjonsmekanikk: Del II - fasthetslære*. Fagbokforlaget, 2015.

J. A. Collins, H. Busby, and G. Staab. *Mechanical Design of Machine Elements and Machines 2nd ed.* Wiley, 2010. ISBN 978-0-470-41303-6.

The SciPy community. Scipy documentation. URL https://docs.scipy.org/doc/scipy/reference/optimize.html.

D. Kraft. A software package for sequential quadratic programming. 1988.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag Berlin Heidelberg, 2006. ISBN 978-3-540-35445-1.

R. Byrd, M. E. Hribar, and J. Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM J. Optim.*, 9:877–900, 1999.

R Storn and KV Price. Differential evolution–a simple and efficient adaptive scheme for global optimization over continuous spaces. institute of company secretaries of india, chennai. *Technical Report TR-95-012*, 1995.

D. Montgomery. *Design and analysis of experiments (8th ed.)*. John Wiley & Sons, INC, 2013. ISBN 978-1118-14692-7.

R. A. Fisher. *The design of Experiments*. Oliver and Boyd, London: 33 Paternoster Row, E.C., 1935.

Doris Entner, Philipp Fleck, Thomas Vosgien, Clemens Münzer, Steffen Finck, Thorsten Prante, and Martin Schwarz. A systematic approach for the selection of optimization algorithms including end-user requirements applied to box-type boom crane design. *Applied System Innovation*, 2:20, 07 2019. doi: 10.3390/asi2030020.

Appendix

A boltLocation.py

```
1 from scipy.sparse.construct import random
2 from supportFunctions import *
3
4 #Pip-modules
5 from scipy.optimize import minimize,
  ↪ differential_evolution, NonlinearConstraint
6 import numpy as np
7 import matplotlib.pyplot as plt
8 from itertools import combinations
9 import json
10 import random
11
12 from Examples.exampleProblems import problems, problem
13
14 #Problem specific variables
15 tubeLocation =
  ↪ [[10,10],[90,30],[45,30],[25,70],[90,75],[30,100],[85,100]]
16 tubePressures = [34.5, 34.5, 34.5, 34.5, 34.5, 34.5,
  ↪ 34.5]
17 tubeRadius = [5., 6., 6., 7., 4., 3., 2.]
18 sizeX, sizeY = 150,110 #Size of domain [mm]
19 initial_guess = [65.2,33.8,51.8,50.8,76.4,59.4,52.8,7.76]
  ↪ #Initial guess x0, y0, ... xn, yn
20
21
22 h = 20 #Thickness
23 boltType = "M8"
24 materialChoice = "AISI316L"
25
26
27 numberOfBolts = int(len(initial_guess)/2)
28 boltSize, boltRadius, s, d, emodbolt, boltArea, ps =
  ↪ chooseBolt(boltType, numberOfBolts) #Getting bolt
  ↪ specific values
29 xl,yl = [0+boltRadius]*2 #Calculating the lower bounds
```

```
30 xu,yu = sizeX-boltRadius,sizeY-boltRadius #Calculating
   ↪ the upper bounds
31 tubeforces, forceOnCentroid =
   ↪ calculateForceOnCentroid(tubePressures, tubeRadius)
   ↪ #Calculating the total force on the centroid
32 bounds = [(xl,xu), (yl,yu)]*numberOfBolts #Creating the
   ↪ full bounds matrix that is needed for the
   ↪ minimize-function
33 sumBoltSize = numberOfBolts*boltSize #Total bolt area
34 pretensionOvershoot = ps*boltSize*0.75 #Pretension in
   ↪ bolts, [N]
35
36
37
38 #Bolt break static variables
39 emodbase, yieldStress =
   ↪ chooseBaseMaterial(materialChoice) #Getting material
   ↪ specific values
40 km = emodbase*np.pi*(s**2 - d**2)/4*h #Calculating
   ↪ stiffness for material
41 kb = emodbolt*boltArea/h #Calculating stiffness for the
   ↪ bolt
42
43 def f1(boltLocation):
44     sum = 0
45     for i in range(0,len(boltLocation),2):
46         for tube in tubeLocation:
47             sum+=np.sqrt((boltLocation[i]-tube[0])**2 +
               ↪ (boltLocation[i+1]-tube[1])**2)
48     return sum
49
50 def f2(boltLocation):
51     sum = 0
52     for tube in tubeLocation:
53         mini = np.inf
54         for i in range(0,len(boltLocation),2):
55             distance =
               ↪ np.sqrt((boltLocation[i]-tube[0])**2 +
               ↪ (boltLocation[i+1]-tube[1])**2)
```

```
56              mini = min(mini,distance)
57          sum+= mini
58      return sum
59
60  def f3(boltLocation):
61      sum = 0
62      for j, tube in enumerate(tubeLocation):
63          mini = np.inf
64          for i in range(0,len(boltLocation),2):
65              distance =
                 ↪  np.sqrt((boltLocation[i]-tube[0])**2 +
                 ↪  (boltLocation[i+1]-tube[1])**2)
66              mini = min(mini,distance)
67          sum+= tubeforces[j]*(mini**3)/1e6
68      return sum
69
70  def f4(bolt_location):
71      max_result = -np.inf
72      total = f1(bolt_location)
73      for j, tube in enumerate(tubeLocation):
74          mini = np.inf
75          maxi = -np.inf
76          for i in range(0,len(bolt_location),2):
77              distance =
                 ↪  np.sqrt((bolt_location[i]-tube[0])**2 +
                 ↪  (bolt_location[i+1]-tube[1])**2)
78              mini = min(mini,distance)
79              maxi = max(maxi,distance)
80          max_result = max(max_result, tubeforces[j] *
             ↪  (total**2) * (mini**2) * 1e-12)
81      return max_result
82
83
84
85  def boltCrash(boltLocation, boltR = boltRadius,
     ↪  washerDist = 0):
        '''
86
87      In-equality constraint
88
```

```
89      :param: boltLocation is a 1-D list with
    ↪   x0,y0,x1,y1,...
90      :returns: A punishment value based on how much
    ↪   overlapping there is between bolts
91      '''
92      punishment = 0
93      minDist = (washerDist+boltR)*3
94      for combo in combinations(np.reshape(boltLocation,
          ↪ (numberOfBolts,2)),2):
95          avstand = distance(combo[0],combo[1])
96          if avstand < minDist:
97              try:
98                  punishment -= 1/(avstand+1e-6)
99              except ZeroDivisionError:
100                 punishment -= 1e10
101     return punishment
102
103 def tubeCrash(boltLocation, boltR = boltRadius,
    ↪   washerDist = 1, tube_width = 2):
104     '''
105     In-equality constraint
106     :param: boltLocation is a 1-D list with
    ↪   x0,y0,x1,y1,...
107     :returns: A punishment based on how much overlapping
    ↪   there is between bolts and tubes
108     '''
109     punishment = 0
110     minDist = boltR + washerDist + tube_width
111     for bolt in np.reshape(boltLocation,
          ↪ (numberOfBolts,2)):
112         for i, tube in enumerate(tubeLocation):
113             avstand = distance(bolt, tube)
114             if avstand < (minDist+tubeRadius[i]):
115                 try:
116                     punishment -= 1/ (avstand+1e-6)
117                 except ZeroDivisionError:
118                     punishment -= 1e10
119     return punishment
120
```

```
121 def boltBreak(boltLocation):
122     '''
123     In-equality constraint
124     :param: boltLocation is a 1-D list with
     ↪  x0,y0,x1,y1,...
125     :returns: A punishment based on the excess stress on
     ↪  the bolts
126     '''
127     punishment = 0
128     forces = boltForces(boltLocation, numberOfBolts,
         ↪  boltSize, boltRadius, tubeLocation,
         ↪  tubePressures, tubeRadius, forceOnCentroid)
129     for force in forces:
130         #Calculate force in material
131         forceBolt = (kb/(km+kb)) * force
132         forceMaterial = (km/(km+kb)) * force
133         #Calculate the needed pretenstion and the total
             ↪  forces and stresses
134         pretension = forceMaterial + pretensionOvershoot
135         totalForceBolt = forceBolt + pretension
136         boltStress = totalForceBolt / boltArea
137         boltCheck = boltStress - ps
138         if boltCheck > 0:
139             punishment -= boltCheck
140     return punishment
141 
142 con1 = {'type': 'ineq', 'fun': boltCrash} #Adding a
     ↪  connection between function and the type of
     ↪  constraint
143 con2 = {'type': 'ineq', 'fun': tubeCrash} #Adding a
     ↪  connection between function and the type of
     ↪  constraint
144 con3 = {'type': 'ineq', 'fun': boltBreak} #Adding a
     ↪  connection between function and the type of
     ↪  constraint
145 cons = ([con1, con2, con3])
146 
147 
148 constraints = (NonlinearConstraint(boltCrash,0, np.inf),
```

```
149     NonlinearConstraint(tubeCrash,0, np.inf),
150     NonlinearConstraint(boltBreak,0, np.inf))
151
152 valg_slsqp = {'ftol': 1e-2, 'disp': True, 'maxiter':1500}
    ↪  #ftol:1e-2 means tolerance in optimization function
153
154 valg_trust = {'gtol': 1e-1, 'xtol': 1e-1, 'disp': True,
    ↪  'maxiter':1500}
155
156 optimized_results = []
157
158
159 for j in range(10):
160     init_problem = problems[0]
161     sizeX, sizeY, tubeLocation, tubePressures,
        ↪  tubeRadius, initial_guess, name, run =
        ↪  init_problem.getValues()
162     name+=str(j)
163     numberOfBolts = int(len(initial_guess)/2)
164     boltSize, boltRadius, s, d, emodbolt, boltArea, ps =
        ↪  chooseBolt(boltType, numberOfBolts) #Getting bolt
        ↪  specific values
165     xl,yl = [0+boltRadius]*2 #Calculating the lower
        ↪  bounds
166     xu,yu = sizeX-boltRadius,sizeY-boltRadius
        ↪  #Calculating the upper bounds
167     tubeforces, forceOnCentroid =
        ↪  calculateForceOnCentroid(tubePressures,
        ↪  tubeRadius) #Calculating the total force on the
        ↪  centroid
168     bounds = [(xl,xu), (yl,yu)]*numberOfBolts #Creating
        ↪  the full bounds matrix that is needed for the
        ↪  minimize-function
169     sumBoltSize = numberOfBolts*boltSize #Total bolt
        ↪  area
170
171     for i in range(numberOfBolts):
172         initial_guess[i*2] = random.uniform(xl, xu)
173         initial_guess[i*2+1] = random.uniform(yl, yu)
```

```
174
175
176     print([round(x, 3) for x in tubeforces],
         ↪  len(tubeforces))
177
178     try:
179         result = differential_evolution(f4,
             ↪  bounds=bounds, constraints=constraints)
180         #result = minimize(f4, initial_guess, method =
             ↪  'slsqp', bounds=bounds, constraints=cons,
             ↪  options=valg_slsqp)
181         #result = minimize(f4, initial_guess, method =
             ↪  'trust-constr', bounds=bounds,
             ↪  constraints=cons, options=valg_trust)
182         fitted_params = result.x
183         optimized_result = problem(sizeX, sizeY,
             ↪  tubeLocation, tubePressures,
184             tubeRadius, initial_guess,
                 ↪  optimized_location=fitted_params.round(2).tolist(),
185             name=name)
186         optimized_results.append(
             ↪  json.dumps(optimized_result.__dict__))
187         for i in range(numberOfBolts):
188             print('Bolt-{0}: x={1:.2f}, y={2:.2f}
                 ↪  '.format(i, fitted_params[i*2],
                 ↪  fitted_params[i*2 +1]))
189         plt.figure()
190         plt.ion()
191         plt.title('Plot of optimized bolt location for:
             ↪  {0}'.format(init_problem.name))
192         plt.xlim(0,sizeX)
193         plt.ylim(0,sizeY)
194         plt.xlabel("Size x [mm]")
195         plt.ylabel("Size y [mm]")
196         plt.subplots_adjust(bottom=0.30)
197         plt.gca().set_aspect('equal', adjustable='box')
198         boltPlotSize =
             ↪  [np.pi*boltRadius**2]*numberOfBolts
199         tubePlotSize = [np.pi*x**2 for x in tubeRadius]
```

```
200         plt.scatter(initial_guess[0::2],
             ↪ initial_guess[1::2], c='b', marker= 'h',
             ↪ s=boltPlotSize, label='Initial guess')
             ↪ #Initial guess plotted in blue hex
201         xval = [x[0] for x in tubeLocation] #Temp
             ↪ variable to get the plot nice
202         yval = [x[1] for x in tubeLocation] #Temp
             ↪ variable to get the plot nice
203         plt.scatter(xval, yval, c='r', marker='o',
             ↪ s=tubePlotSize, label='Tube locations')
204         plt.scatter(fitted_params[0::2],
             ↪ fitted_params[1::2], c='g', marker='h',
             ↪ s=boltPlotSize, label='Minimized bolt
             ↪ locations') #Tubes plotted in green circles
205         plt.legend(bbox_to_anchor=(0,-0.2), loc="upper
             ↪ left")
206         plt.show()
207         file_name =
             ↪ 'Results/Slsqp/sqlpqs{0}.png'.format(j)
208         print(file_name)
209         print(f4(fitted_params))
210         plt.savefig(file_name)
211         #input("Press any key to remove the plot")
212         plt.close()
213     except ValueError:
214         print('Value Error in {0}'.format(
             ↪ init_problem.name))
215         raise ValueError
216 
217 print_bool = True
218 if print_bool:
219     with open('optimized_results_de_10.json', 'w') as
         ↪ write_file:
220         json.dump(optimized_results, write_file,
             ↪ indent=4)
```

B supportFunctions.py

```
from scipy.optimize import minimize
import numpy as np
import matplotlib.pyplot as plt
from itertools import combinations


def chooseBolt(boltType, NumberofBolts):
    # -- This function initializes relevant bolt
     ↪ parameters, depending on the bolt type you
     ↪ desire --
    boltSize = 0
    boltRadius = 0
    s = 0
    d = 0
    ps = 0
    emodbolt = 210
    boltarea = 0
    if boltType == "M8":
        boltSize = 36.6 # Area of bolt without threads
        boltRadius = 8. # Max cap radius for M8 bolt
        s = 13.27
        d = 8
        emodbolt = 210000 #[N/mm2]
        boltarea = (4**2)*np.pi
        ps = 641.9 #[N/mm2]
    elif boltType == "M10":
        boltSize = 58 # Area of bolt without threads
        boltRadius = 18.48 # Max cap radius for M8 bolt
        s = 16.27 #[Nut size]
        d = 10 #[Diameter]
        emodbolt = 210000 #[N/mm2]
        boltarea = (5**2)*np.pi
        ps = 641.9 #[N/mm2]
    return boltSize, boltRadius, s, d, emodbolt,
     ↪ boltarea, ps


```

```
35 def chooseBaseMaterial(materialType):
36     # Return parameters depending on materialtype
37     emod = 0
38     yieldstress = 0
39     if materialType == "AISI316L":
40         emod = 210000
41         yieldstress = 250000
42     if materialType == "AISI1010":
43         emod = 205000 #[MPa]
44         yieldstress = 305000 #[MPa]
45     return emod, yieldstress
46
47 def
   ↪   boltForces(boltLocations,numberOfBolts,boltSize,boltRadius,tubeLo
   ↪   tubePressures, tubeRadius, forceOnCentroid):
48     boltForces = np.zeros(numberOfBolts)
49     centroid =
       ↪   patternCentroid(boltLocations,numberOfBolts)
50     patternInertiaX, patternInertiaY =
       ↪   patternInertia(boltLocations,boltSize,boltRadius,centroid,num
51     centroidMomentX, centroidMomentY =
       ↪   centroidMoment(tubeLocation,tubePressures,tubeRadius,centroid
52     for i in range(numberOfBolts):
53         boltForces[i]= forceOnCentroid/numberOfBolts + \
54
                                     ↪   ((centroidMomentX*(boltLocations[i*2+1]-c
                                     ↪   + \
55
                                     ↪   ((centroidMomentY*(boltLocations[i*2]-cen
56     return boltForces
57
58
59 def calculateForceOnCentroid(tubePressures, tubeRadius):
60     tubeforces = []
61     for i in range(len(tubePressures)):
62
           ↪   tubeforces.append(tubePressures[i]*(np.pi*tubeRadius[i]**
63     return tubeforces, sum(tubeforces)
64
```

```
65 def patternCentroid(boltLocations,numberOfBolts):
66     xCentroid = 0
67     yCentroid = 0
68     for i in range(0,len(boltLocations),2):
69         xCentroid += boltLocations[i]
70         yCentroid += boltLocations[i+1]
71     centroid =
        ↪ np.array([xCentroid/numberOfBolts,yCentroid/numberOfBolts])
72     return centroid
73 
74 def
    ↪ patternInertia(boltLocations,boltSize,boltRadius,centroid,numberOf
75     patternInertiaX = 0
76     patternInertiaY = 0
77     for i in range(numberOfBolts):
78         patternInertiaX += (((centroid[1])**2  -
            ↪ boltLocations[i*2+1]) **2) * boltSize
79         patternInertiaY += (((centroid[0])**2  -
            ↪ boltLocations[i*2]) **2) * boltSize
80     return patternInertiaX,patternInertiaY
81 
82 def centroidMoment(tubeLoc, tubePressures, tubeRadius,
    ↪ centroid):
83     centroidMomentX = 0
84     centroidMomentY = 0
85     for i in range(len(tubeLoc)):
86         centroidMomentX += (tubePressures[i]*np.pi *
            ↪ tubeRadius[i]**2) * (tubeLoc[i][1] -
            ↪ centroid[1])
87         centroidMomentY += (tubePressures[i]*np.pi *
            ↪ tubeRadius[i]**2) * (tubeLoc[i][0] -
            ↪ centroid[0])
88     return centroidMomentX,centroidMomentY
89 
90 def distance(center1, center2):
91     return np.sqrt((center1[0]-center2[0])**2 +
        ↪ (center1[1]-center2[1])**2)
```

C abaqus_functions.py

```
1  # -*- coding: mbcs -*-
2  from part import *
3  from material import *
4  from section import *
5  from assembly import *
6  from step import *
7  from interaction import *
8  from load import *
9  from mesh import *
10 from optimization import *
11 from job import *
12 from sketch import *
13 from visualization import *
14 from connectorBehavior import *
15 import numpy as np
16
17
18 def create_flange(model, sizeX, sizeY,height,
     ↪  boltlocation, tubeLocation, tubeRadius,
     ↪  wall_thickness):
19     sizeX2 = sizeX/2
20     sizeY2 = sizeY/2
21     number_of_bolts = len(boltlocation)/2
22     number_of_tubes = len(tubeLocation)
23     #Create part Flange
24     model.ConstrainedSketch(name='__flangeSketch__',
        ↪  sheetSize=200.0)
25
        ↪  model.sketches['__flangeSketch__'].rectangle(point1=(0.0,
        ↪  0.0),
26         point2=(sizeX, sizeY))
27     model.Part(dimensionality=THREE_D, name='Flange',
        ↪  type=
28         DEFORMABLE_BODY)
29     model.parts['Flange'].BaseSolidExtrude(depth=height,
        ↪  sketch=
30         model.sketches['__flangeSketch__'])
```

```
31      del model.sketches['__flangeSketch__']
32
33
34      model.ConstrainedSketch(gridSpacing=7.14,
         ↪ name='__flangeSketch__',
35          sheetSize=285.65, transform=
36          model.parts['Flange'].MakeSketchTransform(
37          sketchPlane=model.parts['Flange'].faces[4],
38          sketchPlaneSide=SIDE1,
39          sketchUpEdge=model.parts['Flange'].edges[7],
40          sketchOrientation=RIGHT, origin=(sizeX2, sizeY2,
             ↪ height)))
41
         ↪ model.parts['Flange'].projectReferencesOntoSketch(filter=
42          COPLANAR_EDGES,
             ↪ sketch=model.sketches['__flangeSketch__'])
43
         ↪ model.sketches['__flangeSketch__'].sketchOptions.setValues(
44          gridOrigin=(-sizeX2, -sizeY2))
45      del model.sketches['__flangeSketch__']
46
47      #
48      p = model.parts['Flange']
49      e = p.edges
50      f = p.faces
51      face = f.findAt(coordinates=(1.0,1.0,height))
52      edge = e.findAt(coordinates=(sizeX,sizeY - 1,0.0))
53
54
55      #Create a reference to the top surface for use later
56      face_ind=face.index
57      side1Faces=f[face_ind:face_ind+1]
58      model.parts['Flange'].Surface(side1Faces=side1Faces,
         ↪ name='top-surface')
59
60
61      #Cut out space for bolts and tubes
62      model.ConstrainedSketch(gridSpacing=7.14,
         ↪ name='__cutSketch__',
```

```
63          sheetSize=285.65, transform=
64          model.parts['Flange'].MakeSketchTransform(
65          sketchPlane=face,
66          sketchPlaneSide=SIDE1,
67          sketchUpEdge=edge,
68          sketchOrientation=RIGHT, origin=(sizeX2, sizeY2,
             ↪  height)))
69
        ↪   model.parts['Flange'].projectReferencesOntoSketch(filter=
70          COPLANAR_EDGES,
             ↪  sketch=model.sketches['__cutSketch__'])
71
72
73      #Sketch the holes for bolt
74      for i in range(number_of_bolts):    #
75          xi = i*2
76          yi = xi+1
77          xc = boltlocation[xi] - sizeX2
78          yc = boltlocation[yi] - sizeY2
79
             ↪  model.sketches['__cutSketch__'].CircleByCenterPerimeter(c
80              xc, yc), point1=(xc, yc+4.04))
81
82
83      #Sketch the holes for tubes
84      for i in range(number_of_tubes):
85          xi = tubeLocation[i][0]
86          yi = tubeLocation[i][1]
87          xc = xi - sizeX2
88          yc = yi - sizeY2
89          r = tubeRadius[i]
90
             ↪  model.sketches['__cutSketch__'].CircleByCenterPerimeter(c
91              xc, yc), point1=(xc, yc+r))
92
93
94      #Make the cut using the sketch made
95
        ↪   model.parts['Flange'].CutExtrude(flipExtrudeDirection=OFF,
```

```
96          sketch=model.sketches['__cutSketch__'],
              ↪ sketchOrientation=
97          RIGHT, sketchPlane=face,
98          sketchPlaneSide=SIDE1, sketchUpEdge=
99          edge)
100     del model.sketches['__cutSketch__']
101
102
103     #Sketch the holes for tubes and tube walls
104     p = model.parts['Flange']
105     e = p.edges
106     f = p.faces
107     face = f.findAt(coordinates=(1.0,1.0,height))
108     edge = e.findAt(coordinates=(sizeX,sizeY - 1,0.0))
109     model.ConstrainedSketch(gridSpacing=7.14,
          ↪ name='__PartitionSketch__',
110         sheetSize=285.65, transform=
111         p.MakeSketchTransform(
112         sketchPlane=face,
113         sketchPlaneSide=SIDE1,
114         sketchUpEdge=edge,
115         sketchOrientation=RIGHT, origin=(sizeX2, sizeY2,
              ↪ height)))
116     p.projectReferencesOntoSketch(filter=
117         COPLANAR_EDGES,
              ↪ sketch=model.sketches['__PartitionSketch__'])
118
119     for i in range(number_of_tubes):
120         xi = tubeLocation[i][0]
121         yi = tubeLocation[i][1]
122         xc = xi - sizeX2
123         yc = yi - sizeY2
124         r = tubeRadius[i]
125
              ↪ model.sketches['__PartitionSketch__'].CircleByCenterPerim
126             xc, yc), point1=(xc, yc+r))
127
              ↪ model.sketches['__PartitionSketch__'].CircleByCenterPerim
128             xc, yc), point1=(xc, yc+r+wall_thickness))
```

```
129
130
131     #Partition top-surface
132     model.parts['Flange'].PartitionFaceBySketch(
133         faces = face,
134         sketch = model.sketches['__PartitionSketch__']
135     )
136     del model.sketches['__PartitionSketch__']
137
138
139     #Create a surface for each tube wall
140     p = model.parts['Flange']
141     f = p.faces
142     for i in range(number_of_tubes):
143         r = tubeRadius[i]
144         xc = tubeLocation[i][0]
145         yc = tubeLocation[i][1] + r + wall_thickness/2
146         surface_name = 'TubeSurf-'+str(i+1)
147         face = f.findAt(coordinates=(xc,yc,height))
148         face_ind=face.index
149         side1Faces=f[face_ind:face_ind+1]
150         p.Surface(side1Faces=side1Faces,
            ↪   name=surface_name)
151
152     model.Material(name='AISI316L')
153     model.materials['AISI316L'].Density(table=((7.8e-06,
        ↪   ), ))
154     model.materials['AISI316L'].Elastic(table=((210000.0,
        ↪   0.3), ))
155     model.HomogeneousSolidSection(material='AISI316L',
        ↪   name=
156         'AISI316L', thickness=None)
157     model.parts['Flange'].SectionAssignment(offset=0.0,
158         offsetField='', offsetType=MIDDLE_SURFACE,
            ↪   region=Region(
159
            ↪   cells=model.parts['Flange'].cells.getSequenceFromMask(
160         mask=('[#1 ]', ), )), sectionName='AISI316L',
            ↪   thicknessAssignment=
```

```
161             FROM_SECTION)
162
163
164         # End of Flange part
165
166     def create_bolt(model):
167         #Create "bolt"-part  M8-bolt
168         #TODO Different bolt parameters
169         model.ConstrainedSketch(name='__boltSketch__',
            ↪   sheetSize=200.0)
170
            ↪   model.sketches['__boltSketch__'].ConstructionLine(point1=(0.0
171             -100.0), point2=(0.0, 100.0))
172
            ↪   model.sketches['__boltSketch__'].FixedConstraint(entity=
173             model.sketches['__boltSketch__'].geometry[2])
174
            ↪   model.sketches['__boltSketch__'].rectangle(point1=(0.0,
            ↪   0.0),
175             point2=(30.0, 4.0))
176         model.sketches['__boltSketch__'].undo()
177
            ↪   model.sketches['__boltSketch__'].rectangle(point1=(0.0,
            ↪   0.0),
178             point2=(4.0, 30.0))
179
            ↪   model.sketches['__boltSketch__'].rectangle(point1=(0.0,
            ↪   30.0),
180             point2=(8.0, 38.0))
181
            ↪   model.sketches['__boltSketch__'].autoTrimCurve(curve1=
182             model.sketches['__boltSketch__'].geometry[4],
                ↪   point1=(
183             1.78766822814941, 29.8395004272461))
184
            ↪   model.sketches['__boltSketch__'].autoTrimCurve(curve1=
185             model.sketches['__boltSketch__'].geometry[10],
                ↪   point1=(
186             2.45284080505371, 30.0059661865234))
```

```
187    model.Part(dimensionality=THREE_D, name='Bolt', type=
188        DEFORMABLE_BODY)
189    model.parts['Bolt'].BaseSolidRevolve(angle=360.0,
190        flipRevolveDirection=OFF, sketch=
191        model.sketches['__boltSketch__'])
192    del model.sketches['__boltSketch__']
193
194
195    #Create surface on part level for contact with
       ↪  flange
196    model.parts['Bolt'].Surface(name='ContactSurf',
       ↪  side1Faces=
197
           ↪  model.parts['Bolt'].faces.getSequenceFromMask(('[#4
           ↪  ]', ),
198        ))
199
200
201    #Create edge for the boundary conditions on bolt
202    model.parts['Bolt'].Set(edges=
203
           ↪  model.parts['Bolt'].edges.getSequenceFromMask(('[#40
           ↪  ]', ),
204        ), name='BCEdge')
205
206
207    #Assign material properties
208    #TODO Material as parameter
209    model.Material(name='Bolt')
210    model.materials['Bolt'].Density(table=((7.8e-06, ),
       ↪  ))
211    model.materials['Bolt'].Elastic(table=((210000.0,
       ↪  0.3), ))
212    model.HomogeneousSolidSection(material='Bolt',
       ↪  name='Bolt',
213        thickness=None)
214    model.parts['Bolt'].SectionAssignment(offset=0.0,
       ↪  offsetField=
215        '', offsetType=MIDDLE_SURFACE, region=Region(
```

```
216
        ↪ cells=model.parts['Bolt'].cells.getSequenceFromMask(mask=
217      '[#1 ]', ), )), sectionName='Bolt',
        ↪ thicknessAssignment=FROM_SECTION)
218
219 def create_instance_flange(model):
220     #Create instance Flange
221     model.rootAssembly.Instance(dependent=OFF,
        ↪ name='Flange-1',
222         part=model.parts['Flange'])
223     #Seed part instance Flange-1
224
        ↪ model.rootAssembly.seedPartInstance(deviationFactor=0.1,
225         minSizeFactor=0.1, regions=(
226         model.rootAssembly.instances['Flange-1'], ),
            ↪ size=11.0)
227     #Generate mesh Flange-1
228     model.rootAssembly.generateMesh(regions=(
229             model.rootAssembly.instances['Flange-1'], ))
230
231 def create_instance_bolts(model, boltlocation):
232     number_of_bolts = len(boltlocation)/2
233     #Create instance bolts, rotate and translate to
        ↪ correct position
234     for i in range(number_of_bolts):
235         xi = i*2
236         yi = xi+1
237         boltname = 'Bolt-'+str(i+1)
238         model.rootAssembly.Instance(dependent=OFF,
            ↪ name=boltname, part=
239             model.parts['Bolt'])
240         model.rootAssembly.rotate(angle=90.0,
            ↪ axisDirection=(10.0, 0.0,
241             0.0), axisPoint=(0.0, 0.0, 0.0),
                ↪ instanceList=(boltname, ))
242
            ↪ model.rootAssembly.translate(instanceList=(boltname,
            ↪ ), vector=
243             (boltlocation[xi], boltlocation[yi], -10))
```

```
244
245     #Loop to add stuff to bolts
246     for i in range(number_of_bolts):
247         boltname = 'Bolt-'+str(i+1)
248         bcname = 'BC-'+str(i+1)
249         tiename = 'Constraint'+str(i+1)
250         #Boundary condition bolts
251         model.EncastreBC(createStepName='Initial',
             ↪  localCsys=None,
252             name=bcname, region=
253
                 ↪  model.rootAssembly.instances[boltname].sets['BCEdge']
254         #Constraint between bolts and flange
255         model.Tie(adjust=ON, master=
256
                 ↪  model.rootAssembly.instances['Flange-1'].surfaces['to
257             , name=tiename,
                 ↪  positionToleranceMethod=COMPUTED, slave=
258
                 ↪  model.rootAssembly.instances[boltname].surfaces['Cont
259             , thickness=ON, tieRotations=ON)
260         #Set mesh controls Bolts
261         model.rootAssembly.setMeshControls(elemShape=TET,
             ↪  regions=
262
                 ↪  model.rootAssembly.instances[boltname].cells.getSeque
263             mask=('[#1 ]', ), ), technique=FREE)
264         #Set element type Bolts
265
             ↪  model.rootAssembly.setElementType(elemTypes=(ElemType(
266             elemCode=C3D20R, elemLibrary=STANDARD),
                 ↪  ElemType(elemCode=C3D15,
267             elemLibrary=STANDARD),
                 ↪  ElemType(elemCode=C3D10,
                 ↪  elemLibrary=STANDARD)),
268             regions=(
269
                 ↪  model.rootAssembly.instances[boltname].cells.getSeque
270             mask=('[#1 ]', ), ), ))
```

```
271         #Seed part instance Bolts
272
             ↪  model.rootAssembly.seedPartInstance(deviationFactor=0.1,
273             minSizeFactor=0.1, regions=(
274             model.rootAssembly.instances[boltname], ),
                 ↪  size=3.0)
275         #Generate mesh Bolts
276         model.rootAssembly.generateMesh(regions=(
277             model.rootAssembly.instances[boltname], ))
278
279 def create_step(model, step_name, previous_step_name =
     ↪  'Initial'):
280     #Create step
281     model.StaticStep(name=step_name, nlgeom=OFF,
         ↪  previous=previous_step_name)
282
283 def create_connection_rp(model, bolt_location,
     ↪  tube_location, tube_radius, tube_pressures, height,
     ↪  step_name):
284     number_of_bolts = len(bolt_location)/2
285     number_of_tubes = len(tube_location)
286     #Create reference point for each tube and connecting
         ↪  it with the surface
287     assembly = model.rootAssembly
288     location = ()
289     for i in range(number_of_tubes):
290
291         r = tube_radius[i]
292         xc = tube_location[i][0]
293         yc = tube_location[i][1]
294         rp_name = 'RP-'+str(i+1)
295         coupling_name = 'Coupling-'+str(i+1)
296         surface_name = 'TubeSurf-'+str(i+1)
297         instance_flange = assembly.instances['Flange-1']
298         rp_set_name = 'RP-Set-'+str(i+1)
299         load_size = tube_pressures[i] * np.pi *
             ↪  (tube_radius[i]**2)     #[N]
300         load_name = 'Load-'+str(i+1)
301         #Log the reference point(rp) after creation
```

```
302         rp = assembly.ReferencePoint(
303             point=(xc,yc, height),
304             instanceName = rp_name
305         )
306         #Create a set with the rp
307         rp_set = assembly.Set(
308             name = rp_set_name,
309             referencePoints =
              ↪  (assembly.referencePoints[rp.id], )
310         )
311         #Add a coupling between set of rp and surface
312         model.Coupling(
313             name = coupling_name,
314             surface =
              ↪  instance_flange.surfaces[surface_name],
315             controlPoint = rp_set,
316             influenceRadius = WHOLE_SURFACE,
317             couplingType = DISTRIBUTING
318         )
319         #Add tube load[i] on rp_set[i]
320         model.ConcentratedForce(
321             name = load_name,
322             createStepName = step_name,
323             region = rp_set,
324             cf3 = load_size
325         )
326 
327 def create_job(model, job_name, start_job=False,
  ↪  job_parallell = False):
328     mdb.Job(atTime=None, contactPrint=OFF,
      ↪  description='', echoPrint=OFF,
329         explicitPrecision=SINGLE,
          ↪  getMemoryFromAnalysis=True, historyPrint=OFF,
330         memory=90, memoryUnits=PERCENTAGE, model=model,
          ↪  modelPrint=OFF,
331         multiprocessingMode=DEFAULT, name=job_name,
          ↪  nodalOutputPrecision=SINGLE,
332         numCpus=1, numGPUs=0, queue=None,
          ↪  resultsFormat=ODB, scratch='', type=
```

```
333         ANALYSIS, userSubroutine='', waitHours=0,
              ↪  waitMinutes=0)
334     if start_job:
335
              ↪  mdb.jobs[job_name].submit(consistencyChecking=OFF)
336         if not job_parallell:
337             mdb.jobs[job_name].waitForCompletion()
338
339 def create_output_set(model ,set_name, tube_radius,
      ↪  tube_location):
340     number_of_tubes = len(tube_location)
341     edgeList = []
342     assembly = model.rootAssembly
343     instance_flange = assembly.instances['Flange-1']
344     for i in range(number_of_tubes):
345         r = tube_radius[i]
346         xc = tube_location[i][0]
347         yc = tube_location[i][1]
348         edgeList.append(instance_flange.edges.findAt((xc,
              ↪  yc+r, 0), ))
349
350     set = assembly.Set(name = set_name,edges =
          ↪  part.EdgeArray(edgeList))
351     return set
352
353 def print_highest_disp(odb, set_name, step_name,
      ↪  job_name):
354     center = odb.rootAssembly.nodeSets[set_name]
355     last_frame = odb.steps[step_name].frames[-1]
356     displacement = last_frame.fieldOutputs['U']
357     centerDisplacement =
          ↪  displacement.getSubset(region=center)
358
          ↪  print('################################################')
359     print(job_name)
360     node = centerDisplacement.values[0]
361     for v in centerDisplacement.values:
362         if v.magnitude > node.magnitude:
363             node = v
```

```
364
365     print('Position = ', node.position,'Type =
         ↪  ',node.type)
366     print('Node label = ', node.nodeLabel)
367     print('Displacement magnitude =', node.magnitude)
368
369     return [job_name, node.nodeLabel, node.magnitude]
```

D run.py

```
1  # -*- coding: mbcs -*-
2  from part import *
3  from material import *
4  from section import *
5  from assembly import *
6  from step import *
7  from interaction import *
8  from load import *
9  from mesh import *
10 from optimization import *
11 from job import *
12 from sketch import *
13 from visualization import *
14 from connectorBehavior import *
15
16 import numpy as np
17 import json
18
19 print('###############################')
20 ######################################################################
21 # Reload the example problems in case anything is
     ↪  changed
22 # Changes are not tracked unless Abaqus is restarted
     ↪  (Cached files)
23 try:
24     reload(Examples.exampleProblems)
25     reload(abaqus_functions)
26 except NameError:
```

```
27      print(NameError)
28      import Examples.exampleProblems
29      import abaqus_functions
30  # Problem specific variables
31
32  problems = Examples.exampleProblems.problems
33
34  name_extenstions = ['slsqp','trust', 'de'] #TODO . set
    ↪  the first to 'slsqp'
35  name_extenstion = name_extenstions[2]
36
37  file = 'optimized_results_'+ name_extenstion +'_10.json'
38  full_json = json.load(open(file))
39
40  class read_data(object):
41      def __init__(self, jdata):
42          self.__dict__ = json.loads(jdata)
43
44      def get_values(self):
45          return self.__dict__.values()
46
47
48  results = []
49
50
51  for v in full_json:
52      data = read_data(v)
53      print(data.get_values())
54      sizeX = data.x
55      sizeY = data.y
56      tube_location = data.tubeLocation
57      tube_pressures = data.tubePressures
58      tube_radius = data.tubeRadius
59      initial_guess = data.initial_guess
60      model_name  = str(data.name)
61      run_job = data.run
62      optimized_result = data.optimized_location
63      #Calculated values
64
```

```
65      wall_thickness = 2
66      extraBolt = 10
67      print('Deg',model_name)
68
69      step_name = 'External-Loading'
70      height = 20
71      problem_results = []
72      for i in range(2):
73          if i == 0:
74              continue
75              bolt_location = initial_guess
76              job_name = 'job-'+model_name.replace(' ','-')
77              m = mdb.Model(name = model_name)
78          else:
79              bolt_location = optimized_result
80              job_name = 'job-'+model_name.replace('
                 ↪  ','-')+'-' + name_extenstion + '-opt'
81              m = mdb.Model(name = model_name+' opt')
82          abaqus_functions.create_flange(m,sizeX=sizeX,
             ↪  sizeY=sizeY, height=height,
             ↪  boltlocation=bolt_location,
83              tubeLocation=tube_location,
                 ↪  tubeRadius=tube_radius,
                 ↪  wall_thickness=wall_thickness)
84          abaqus_functions.create_bolt(m)
85          abaqus_functions.create_instance_flange(m)
86          abaqus_functions.create_instance_bolts(m,
             ↪  boltlocation=bolt_location)
87
             ↪  abaqus_functions.create_step(m,step_name=step_name)
88          abaqus_functions.create_connection_rp(m,
             ↪  bolt_location=bolt_location,
             ↪  tube_location=tube_location,
89
                 ↪  tube_radius=tube_radius,tube_pressures=tube_pressures
                 ↪  height=height, step_name=step_name)
```

```
90          set = abaqus_functions.create_output_set(model=m,
             ↪  set_name='TUBE-EDGES',
             ↪  tube_radius=tube_radius,
             ↪  tube_location=tube_location)
91          print(job_name)
92          abaqus_functions.create_job(m,job_name=job_name,
             ↪  start_job=True)
93
94          odb = session.openOdb(job_name+'.odb')
95
             ↪  problem_results.append(abaqus_functions.print_highest_dis
             ↪  set_name='TUBE-EDGES', step_name=step_name,
             ↪  job_name=job_name))
96      results.append(problem_results)
97
98
99  print('Biggest displacement around bottom tube edges')
100 for problem in results:
101     print(problem[0])
102     #print(problem[1])
```

E regression.py

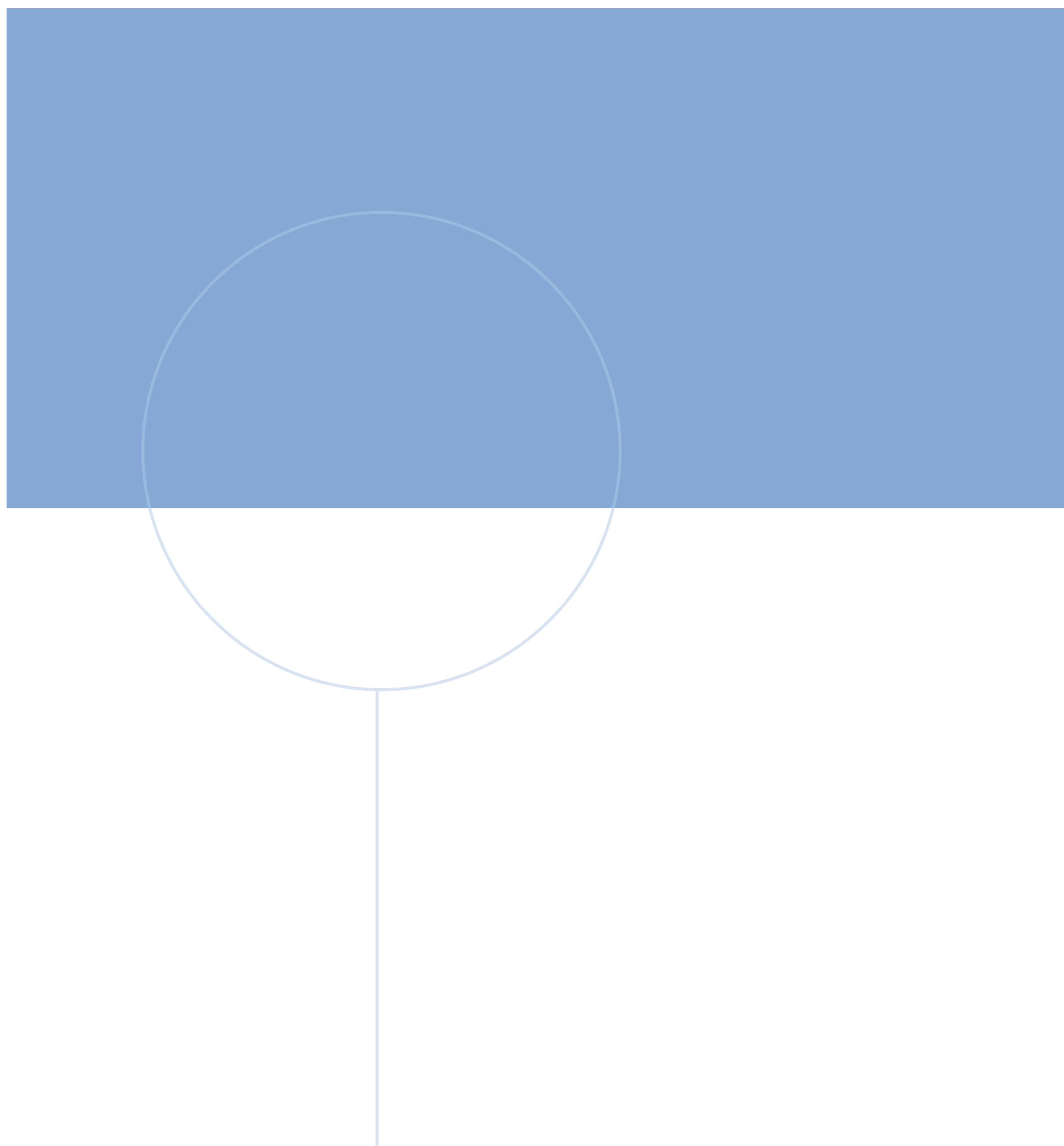
```
1   import pandas as pd
2   import numpy as np
3   import json
4   from tabulate import tabulate
5
6   from values import *
7   from statsmodels.formula.api import ols
8
9
10
11  class case():
12
13      def __init__(self,b1,b2,b3,t,r):
14          self.boltlocations =
             ↪  bolt1[b1]+bolt2[b2]+bolt3[b3]
```

```
15          self.tubelocations = np.array(tubelocation[t])
16          self.tuberadius = np.array(tuberadius[r])
17          self.tubeforces =
             ↪  np.pi*tubepressure*(self.tuberadius**2)
18          self.name = b1+b2+b3+t+r
19
20      def distance(self):
21          d = []
22          for tubex, tubey in self.tubelocations:
23              di =[]
24              for i in
                 ↪  range(int(len(self.boltlocations)/2)):
25                  boltx = self.boltlocations[i*2]
26                  bolty = self.boltlocations[i**2+1]
27                  dii = np.sqrt((boltx-tubex)**2 +
                     ↪  (bolty-tubey)**2)
28                  di.append(dii)
29              d.append(di)
30          return d
31
32      def tube_distance(self, t):
33          if t=='H':
34              return np.sqrt(80**2 + 80**2)
35          elif t=='L':
36              return np.sqrt(20**2 + 20**2)
37
38  filepath = 'Results/results.txt'
39
40  with open(filepath, 'r') as jsonfile:
41      data = json.load(jsonfile)
42
43  tube0 = data['Tube-0']
44  tube1 = data['Tube-1']
45
46  y0 = [y[1] for y in tube0]
47  y1 = [y[1] for y in tube1]
48
49  total_distance0 = []
50  total_distance1 = []
```

```
51 min_distance0 = []
52 min_distance1 = []
53 max_distance0 = []
54 max_distance1 = []
55 tube_force0 = []
56 tube_force1 = []
57 tube_distance0 = []
58
59
60 for value in tube0:
61     b1, b2, b3, t, r = value[0][-5:]
62     casei = case(b1,b2,b3,t,r)
63     distance0, distance1 = casei.distance()
64     #Tube-0
65     total_distance0.append(sum(distance0))
66     min_distance0.append(min(distance0))
67     max_distance0.append(max(distance0))
68     tube_force0.append(casei.tubeforces[0])
69     tube_distance0.append(casei.tube_distance(t))
70     #Tube-1
71     total_distance1.append(sum(distance1))
72     min_distance1.append(min(distance1))
73     max_distance1.append(max(distance1))
74     tube_force1.append(casei.tubeforces[1])
75
76 y = y0+y1
77 total_distance = total_distance0 + total_distance1
78 min_distance = min_distance0 + min_distance1
79 max_distance = max_distance0 + max_distance1
80 tube_force = tube_force0 + tube_force1
81 tube_distance=tube_distance0 + tube_distance0
82
83
84 df = pd.DataFrame(
85     {
86         "u": y,
87         "Total": total_distance,
88         "Min": min_distance,
89         "Max": max_distance,
```

```
90            "Force": tube_force,
91            "Tube_d": tube_distance
92        }
93 )
94
95
96 formel = "u ~  \
97            + I(Total**1)*I(Min**1)*I(Max**1)*Force*Tube_d \
98            + I(Total**1)*I(Min**1)*I(Max**2)*Force*Tube_d \
99            + I(Total**1)*I(Min**1)*I(Max**3)*Force*Tube_d \
100           + I(Total**1)*I(Min**2)*I(Max**1)*Force*Tube_d \
101           + I(Total**1)*I(Min**2)*I(Max**2)*Force*Tube_d \
102           + I(Total**1)*I(Min**2)*I(Max**3)*Force*Tube_d \
103           + I(Total**1)*I(Min**3)*I(Max**1)*Force*Tube_d \
104           + I(Total**1)*I(Min**3)*I(Max**2)*Force*Tube_d \
105           + I(Total**1)*I(Min**3)*I(Max**3)*Force*Tube_d \
106       \
107           + I(Total**2)*I(Min**1)*I(Max**1)*Force*Tube_d \
108           + I(Total**2)*I(Min**1)*I(Max**2)*Force*Tube_d \
109           + I(Total**2)*I(Min**1)*I(Max**3)*Force*Tube_d \
110           + I(Total**2)*I(Min**2)*I(Max**1)*Force*Tube_d \
111           + I(Total**2)*I(Min**2)*I(Max**2)*Force*Tube_d \
112           + I(Total**2)*I(Min**2)*I(Max**3)*Force*Tube_d \
113           + I(Total**2)*I(Min**3)*I(Max**1)*Force*Tube_d \
114           + I(Total**2)*I(Min**3)*I(Max**2)*Force*Tube_d \
115           + I(Total**2)*I(Min**3)*I(Max**3)*Force*Tube_d \
116       \
117           + I(Total**3)*I(Min**1)*I(Max**1)*Force*Tube_d \
118           + I(Total**3)*I(Min**1)*I(Max**2)*Force*Tube_d \
119           + I(Total**3)*I(Min**1)*I(Max**3)*Force*Tube_d \
120           + I(Total**3)*I(Min**2)*I(Max**1)*Force*Tube_d \
121           + I(Total**3)*I(Min**2)*I(Max**2)*Force*Tube_d \
122           + I(Total**3)*I(Min**2)*I(Max**3)*Force*Tube_d \
123           + I(Total**3)*I(Min**3)*I(Max**1)*Force*Tube_d \
124           + I(Total**3)*I(Min**3)*I(Max**2)*Force*Tube_d \
125           + I(Total**3)*I(Min**3)*I(Max**3)*Force*Tube_d \
126           "
127
128 Reg = ols(formula=formel, data=df)
```

```
129 Fit = Reg.fit()
130
131 names = Fit.model.exog_names
132
133 test_results = []
134 formel2 = 'u  ~'
135 for name, coeff, pvalue in zip(names, Fit.params,
    ↪  Fit.pvalues):
136     if pvalue < 5e-2:
137         test_results.append([name, coeff, pvalue])
138
139
140 test_results_sorted = sorted(test_results, key= lambda
    ↪  x:x[2], reverse=True)
141 #print(Fit.summary())
142
143 print(tabulate(test_results_sorted, headers=['Parameter',
    ↪  'coeff', 'p-value']))
144 print(len(test_results_sorted))
145 print('R-squared: {0:.3f}\t Adj. R-squared:
    ↪  {1:.3f}'.format(Fit.rsquared, Fit.rsquared_adj))
```



NTNU
Kunnskap for en bedre verden