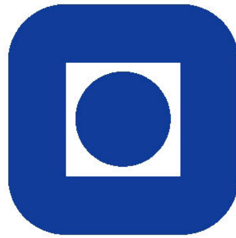
NTNU
Det skapende universitet

Evaluering av utmattingsberegninger for sveiser basert på DnV RP-C203

Remi Krister Sylvain Lanza

Produktutvikling og produksjon
Innlevert: juni 2015
Hovedveileder: Kjell Magne Mathisen, KT
Medveileder: Bjørn Haugen, IPM

Norges teknisk-naturvitenskapelige universitet
Institutt for konstruksjonsteknikk



NTNU

EVALUATION OF FATIGUE CALCULATIONS OF WELDS BASED ON DNV RP-C203

Remi Krister Sylvain Lanza

June 2015

MASTER'S THESIS

Faculty of Engineering Science and Technology
Department of Structural Engineerings

Supervisor : Assc. Professor Bjørn Haugen
Contact person : Tore Holmas (Aker Solutions)

THE NORWEGIAN UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF ENGINEERING DESIGN
AND MATERIALS

MASTER THESIS SPRING 2015
FOR
STUD.TECHN. REMI LANZA

EVALUATION OF FATIGUE CALCULATIONS OF WELDS BASED ON DNV RP-C203
Evaluering av utmattingsberegninger for sveiser basert på DnV RP-C203

One of the key aspects of mechanical design is to assess a structures ability to withstand repeated loading.

Offshore structures such as oil platforms and wind turbines with steel jacket structures are subject to dynamic loading which makes the welds prone to fatigue failure.
The fatigue life of offshore structures is commonly assessed through Finite Element analysis.

The design guide DnV RP-C203 describes several methods for fatigue analysis of welded tubular joints based on FEM results. Two of the methods are

1) Using beam elements and stress concentration factors (SCF) from Appendix B of RP-C203.
2) Using shell elements and computing stresses from extrapolation based on sections 4.2 of RP-C203.

This study will compare the method with respect to degree of approximation and relative safety factors. It is expected that method 1) give a more conservative estimate for fatigue damage than method 2). This is a reasonable assumption since method 1) represents a "coarse" model relative to 2).

The work can be started according to the following tentative tasks:

1. Study and document the recommended practice for fatigue evaluation using numerical methods from DNV RP C203 for tubular joints.
2. Become familiar with Usfos simulation package for dynamic analysis of offshore.
3. Create an application that converts a beam model of a joint to a shell element model of the same joint. This application reads and writes Usfos input files.
4. Based on a selection of typical joint geometries; compare the stresses from the shell and beam models.
5. Based on typical dynamic loading; compare the fatigue life based on the beam and shell models.

The scope of work for the thesis should be limited to two noded beam elements and 4 nodes shell elements with linear shape functions.

Formal requirements:

Three weeks after start of the thesis work, an A3 sheet illustrating the work is to be handed in. A template for this presentation is available on the IPM's web site under the menu "Masteroppgave" (http://www.ntnu.no/ipm/masteroppgave). This sheet should be updated one week before the master's thesis is submitted.

Risk assessment of experimental activities shall always be performed. Experimental work defined in the problem description shall be planed and risk assessed up-front and within 3 weeks after receiving the problem text. Any specific experimental activities which are not properly covered by the general risk assessment shall be particularly assessed before performing the experimental work. Risk assessments should be signed by the supervisor and copies shall be included in the appendix of the thesis.

The thesis should include the signed problem text, and be written as a research report with summary both in English and Norwegian, conclusion, literature references, table of contents, etc. During preparation of the text, the candidate should make efforts to create a well arranged and well written report. To ease the evaluation of the thesis, it is important to cross-reference text, tables and figures. For evaluation of the work a thorough discussion of results is appreciated.

The thesis shall be submitted electronically via DAIM, NTNU's system for Digital Archiving and Submission of Master's theses

Contact person: Tore Holmås, Aker Solutions.

Torgeir Welo
Head of Division

Bjørn Haugen
Professor/Supervisor

NTNU
Norges teknisk-
naturvitenskapelige universitet
Institutt for produktutvikling
og materialer

Acknowledgment

This thesis was written for a master's degree in Mechanical Engineering with specialization in Applied Mechanics.

I would like to thank my supervisor *Bjørn Haugen* for giving me the opportunity to chose this specific thesis. He gave me a subject which included several of the topics I am interested in, which lead me to enjoy the almost six months spent on programing, analysing and writing. He was also a huge help by discussing with me all the unclear issues encountered.
A special thanks is also owed to *Tore Holmås*, who has been part of the development of the USFOS software. He has been able to answer all my questions regarding technical problems.

The author of this work hereby declares that the work is made independently and in accordance to the rules set down by "Examination regulations" at the Norwegian University of Science and Technology (NTNU), Trondheim.

Sammendrag

I strukturnanalyser er det alltid flere metoder for å estimere strukturens indre spenninger. Noen er mindre nøyaktige, men og mindre komplekse, andre er mer presise men også mer tidskrevende.

I denne oppgave skal vi undersøke to forskjellige metoder på å analysere spenninger i rørknutepunkt. Disse metodene er foreslått av *DNV* i et dokument vi følger gjennom denne rapporten.

Den første metoden, som vi kaller *SCF-metoden*, er basert på spenninger beregnet fra en bjelkeelement-modell av strukturen. Den andre metoden er basert på skallelement-analyser av et enkelt rørknutepunkt. Vi kaller denne *skallelement-metoden*.

For å utføre en sammenligning av de to metodene, lager vi først et program som konverterer et knutepunkt fra en representasjon til en annen. Ved å gjøre dette kan vi lettere analysere flere tilfeller med mindre arbeidstid.

Vi kommer til å se gjennom rapporten hvordan spenningsforskjellene mellom de to metodene varierer sterkt på geometrien av knutepunktet og de påvirkende lastene. Disse spenningsvariasjonene er så *forstørret* når vi sammenligner utmattingsskaden forårsaket av bølgelaster. Vi ser deretter at skadene er meget sensitive på forandringer i spenninger.

Etter å ha sammenlignet spenninger og akkumulert skade, konkluderer rapporten med å beskrive årsakene til forskjellene og hvorfor de varierer. Fremtidig arbeid er deretter foreslått for å komme til en mer generell konklusjon.

Summary

In structural analysis there are always several methods for estimating the structures internal stresses. Some of which are less accurate, but also less complex, while others are precise but time consuming.

In this study we investigate two different methods of analysing stresses in tubular joints in jacket structures. These methods are proposed from *DNV* through a document which we will follow in this report.

The first method, which we will name the *SCF method*, is based on stresses calculated from a beam model of the structure. The second method is based on a shell element analysis of a singular tubular joint. We call this the *shell element method*.

To perform a comparison of the two methods, we first create a program which will convert a tubular joint from one representation to the other. By doing this we can more easily analyse multiple cases with less work.

We will see throughout the study how the stress difference between the two methods varies highly depending on geometrical parameters of the joints and loading conditions. These variation of stresses are then *magnified* when we compare the fatigue damage caused by wave loads on the structure. As we will see the damage caused is highly sensitive to changes in stress.

After having compared the stresses and the accumulated fatigue damage, the report concludes by describing some of the causes that make the differences vary. Future work based on the methods used in this document is then suggested to come to a more general conclusion.

Contents

Acknowledgment
Sammendrag
Summary
List of Figures
Abbreviations

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Structure of Thesis 3

2 Review of Literature **5**
2.1 DNV - Fatigue Design of Offshore Steel Structures 5
2.2 USFOS 5
2.3 Tubular joints 6
2.4 Stress Concentration Factors (SCF) 7
2.5 Force to Stress Matrix 8
2.5.1 Purpose 8
2.5.2 Method 8
2.6 Rainflow counting and Palmgen-Miners rule 10

3 Procedure and developed software **11**
3.1 Chapter introduction 11
3.1.1 Flow chart description 13
3.2 Software summary 14
3.3 beam2shell 15
3.3.1 Why create the program 15
3.3.2 Meshing 17
3.3.3 Main program 25
3.4 get_stress 31
3.5 read_stress 32
3.5.1 Purpose 32
3.5.2 Stress Transformation 32
3.5.3 Extrapolation 34
3.5.4 Membrane, Upper and Lower side stresses 34
3.5.5 Steps 35
3.6 shell_vs_beam 36

3.7 dmg_calc . . . 37
3.7.1 Procedure . . . 37
3.7.2 Rainflow counting . . . 38

4 Compare hotspot stresses **41**
4.1 Introduction . . . 41
4.1.1 Load and geometry . . . 42
4.1.2 SCF Method . . . 43
4.1.3 Shell element analysis . . . 43
4.2 Example and procedure . . . 44
4.2.1 Step 1 - Create beam model . . . 44
4.2.2 Step 2 - Define loads . . . 45
4.2.3 Step 3 - Extrapolation . . . 46
4.2.4 Step 4 - SCF method . . . 48
4.3 Results . . . 50
4.3.1 Presentation of Results . . . 50
4.3.2 Example of stress data output: . . . 52
4.3.3 Result discussion 1 . . . 53
4.3.4 Result discussion 2 . . . 63
4.3.5 Forces not considered by the SCF method . . . 63

5 Comparison of fatigue damage **65**
5.1 Structure . . . 65
5.2 Results . . . 67
5.3 Result discussion . . . 68

6 Conclusion and future work **73**

References . . . 75

Appendices **77**
Appendix A - Meshing module code . . . 77
Appendix B - Rainflow algorithm code . . . 85

List of Figures

2.1 Sub-joints . . . 6
2.2 Y-, K- and X-joint . . . 6
2.3 Forces defining K- or Y-connection . . . 7
2.4 Unit loads and degrees of freedom . . . 8
2.5 Hotspot elements . . . 9

3.1 Flow chart of procedure . . . 12
3.2 beam2shell flow chart . . . 16
3.3 Variables used for mesh creation . . . 17
3.4 Mesh: intersection layers (1) . . . 18
3.5 Mesh: intersection (2) . . . 19
3.6 Mesh: arcs . . . 19
3.7 Mesh: wrapping and unwrapping . . . 20
3.8 Mesh: bottom and extended sides . . . 21
3.9 Mesh: ellipses with deflection . . . 22
3.10 Mesh: branch . . . 22
3.11 Mesh: closing intersection . . . 23
3.12 Mesh: end beam elements . . . 23
3.13 Mesh: resulting mesh . . . 24
3.14 beam2shell: junction variables . . . 27
3.15 beam2shell: hotspots . . . 28
3.16 beam2shell: hotspot lengths . . . 28
3.17 beam2shell: beam unit vectors . . . 29
3.18 beam2shell: moment diagram . . . 30
3.19 element orientation . . . 32
3.20 element coordinates transformation . . . 33
3.21 extrapolation sketch . . . 34
3.22 upper-, membrane- and lower element stress . . . 34
3.23 FTS, F2S, STS illustration . . . 36
3.24 Rainflow: "peaks and valleys" . . . 38
3.25 Rainflow: cycle definition . . . 38
3.26 Rainflow: iterations . . . 39

4.1 Junction geometry definition . . . 42
4.2 Load cases for analysis . . . 42
4.3 Hotspots SCF method . . . 43
4.4 F2S Matrix . . . 47

4.5 Load case example 48
4.6 SCF's geometric parameters 48
4.7 Stress result variation example 51
4.8 Stress difference caused by extrapolation lengths changes 53
4.9 Stress variation changes caused by chord forces 54

5.1 Structure to be analyzed 65
5.2 First joint to be analyzed 66
5.3 Second joint to be analyzed 66
5.4 Damage results: amplitude SCF vs shell method 68
5.5 Damage results: Stress over time caused by different forces on junction 70
5.6 Damage results: Stress over time caused by different forces on junction (2) 71
5.7 Damage results: Stress over time caused by different forces on junction (3) 71

Abbreviations

d.o.f.	Degree of Freedom
DNV	Det Norske Veritas
F2S	Force too Stress matrix
FEA	Finite Element Analysis
FEM	Finite Element Method
FTS	Force Time Series matrix
GUI	Graphical User Interface
ID	Identity
SCF	Stress Concentration Factor
STS	Stress Time Series matrix

Chapter 1

Introduction

1.1 Motivation

In many structures the most important phase in the design process is to ensure their ability to withstand the potential forces acting on them. This is usually done by performing Finite Element Analysis (FEA), which discretize the structure in a number of so called elements representing the full structure and thus converting it to a solvable problem. A common decision structural engineers must make, is the balance of the FEM models' resolution (number of elements) and the cost of the analysis. The higher the resolution the more accurate is the answer, but at the expense of computational time and complexity.

This has led to multiple methods of improving the analyses' methods for specific structural cases, in the sense of using engineering experience together with FEM to reduce the computational complexity. The goal is always to have the most precise answer at the minimum cost. The specific case in this study is the fatigue analysis of offshore jacket structures. They are composed of multiple pipes welded together, and have to withstand the load of a platform or wind turbine together with the dynamic loading of wind and waves. The structural analyses of these structures are the basis for deciding the estimated lifetime of its usage and depending on the method used, it could represent a difference of several years in lifetime.

In our jackets the weakest point are the welds connecting the pipes, and what would break them is the fatigue damage caused by varying forces. This damage is based on the resulting stresses over time acting on the welds. Thus, the jackets' estimated lifetime will depend on how we calculate these stresses.

The design guide *DNV RP-C203* describes several methods for estimating the stress in welded tubular joints, and how to perform the fatigue analysis. The two methods relevant to this study are using beam elements with stress concentration factors (SCF) and using shell elements. The first mentioned method is based on a "coarser" model, or one could say a model with less resolution. The second is based on a high resolution model. In the nature of FEM analysis, and performing the analyses correctly, a coarser model tends to give a less stiff solution and is prone to give higher stresses than expected in reality. The second method would give us a more precise solution, but at the expense of computation and modeling time and complexity.

1.2 Problem Statement

Using DNV's guidelines [7] we will compare the two methods:

1) Using beam elements and stress concentration factors (SCF) from *Appendix B* of *RP-C203*.
2) Using shell elements and computing stresses from extrapolation based on sections 4.2 of *RP-C203*.

The comparison will be based on the resulting stresses acting on the welds, and how it affects the fatigue damage.

The first step would be to create a software with the following main capabilities:

- Read an USFOS beam model input file and create a shell model input file of a chosen tubular junction.
- Possibility of reading the forces acting on the junction over time.
- Automatically run several USFOS analysis.

Thereafter, by creating the required applications, compare the stresses resulting from the two different methods on a set of typical junctions.

On a typical jacket structure the fatigue damage difference will be analyzed for a few junctions. The expected result of the study is that method 1) will give higher stresses and fatigue damage than method 2). This is due to the model in method 1) being coarser than in method 2).

1.3 Structure of Thesis

Chapter 2 Review of Literature
A brief review of some of the concepts used in this study.

Chapter 3 Procedure and developed Software
In this chapter the programs developed for completing the study are discussed together with the methods behind the calculations.

Chapter 4 Comparison of Stresses
Using some of the software discussed in the previous chapter we do an in depth comparison of the two methods for calculating weld stresses. The effect of geometrical change in the pipe junction on the results are analysed. We use a set of typical geometries with different unit loads.
The chapter is composed of an explanation followed by an example before the results are presented and discussed.

Chapter 5 Comparison of Fatigue Damage
A second study is done to see how the results found in chapter 4 will affect the fatigue damage. Here we look at a realistic structure and load history.

Chapter 6 Conclusion and Future Work
A conclusion of the complete report and future work on the topic is suggested.

Chapter 2

Review of Literature

2.1 DNV - Fatigue Design of Offshore Steel Structures

"*Fatigue Design of Offshore Steel Structures*" is the title of the document "*DNV RP-C203*". As DNV works with risk management, they give free access to some of their practices so engineers can be prepared for what it requires to be licensed by DNV.

This document contains a huge amount of practices for very specific cases, which all involve fatigue and fracture assessment. For this report we are interested in what involves weld stresses in tubular joints. We will use the section "*3.3 Tubular joints and members*" in "*Chapter 3 - Stress Concentration Factors*" together with "*Appendix B - SCF's for tubular joints*" for using the SCF method. Section "*4.2 Tubular joints*" in "*Chapter 4 - Calculation of hotspot stress by finite element analysis*" will be used for analysing hotspot stresses using shell elements.

2.2 USFOS

USFOS is a structural analysis software widely used to perform non-linear static and dynamic analysis for offshore frame structures. It is used by oil companies and consultants , *for integrity assessment, collapse analyses and accidental load analyses of offshore jacket structures, topsides, jack-ups and other frame structures, intact or damaged* [2].

In this report we will use USFOS for all our simulations. Most input and ouput file read and written by USFOS can be edited by text editors. Throughout the study we will base ourself on USFOS' user's manual [6] to create, edit and process input and result files.

2.3 Tubular joints

The *DNV RP-C203* document gives some definitions about the tubular joint classifications. Below is a short summary of what is important for this report:

Tubular joints in our case are defined by one or more tubular beams (*braces*) welded to one larger tubular beam (*chord*) to form a joint.

A joint will always be composed of one *chord* and one or more *braces*. The joints are then categorized in *sub-joints*. We define each sub-joints by braces, including the chord, in the same plane (there is a tolerance of $\pm$ 15° to be considered the same plane). These sub-joints can then be defined as Y-,K- or X-connections or combinations of these.

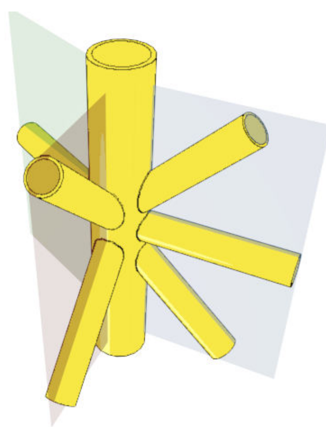


Figure 2.1: One joint with three sub-joints in different planes.

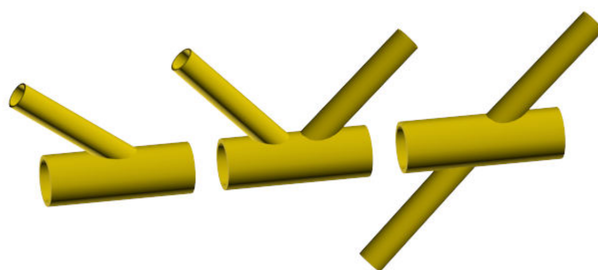


Figure 2.2: From left to right; Y-, K- and X-joint.

The *YKX* connections depend not only on the sub-joints geometry, but also on the axial loads the braces are carrying.

The cases we are interested in are sub-joints consisting of two braces on the same side of the chord. In other words, Y- and K-connections. If the axial load is balanced within *10%* on both braces, it is considered K. Any other load conditions mean the two braces are considered two Y-connections.

We will later in the report see that we are using two different sets of SCF values. One set is for loads on one brace, while the other set is for the case of balanced loads.

If: Fx1 = (1$\pm$0.1)Fx2, we consider the braces K, otherwise Y.

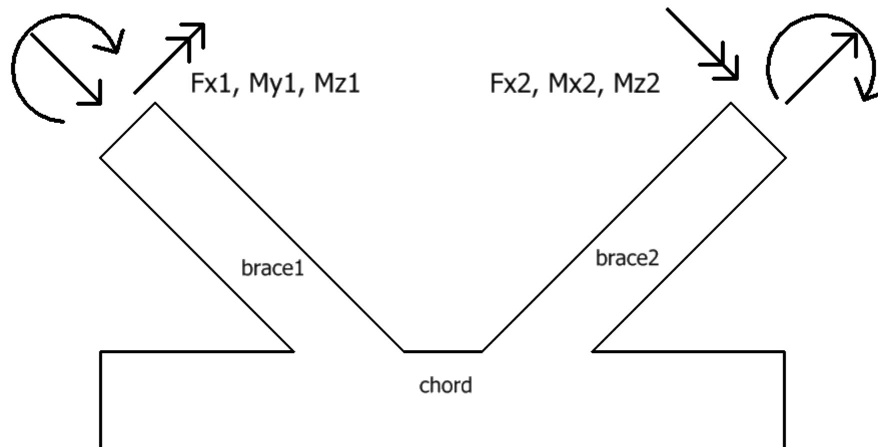


Figure 2.3: Forces on a K-geometry defines if it's a K- or Y-connection.

2.4 Stress Concentration Factors (SCF)

Stress concentration factors are used to simplify the process of calculating the stress in complex locations. The SCF is multiplied by a stress value (hence *factor*) often known as the nominal stress, calculated by a simplification of the problem or taken from an area near the location of interest. It is usually defined as the ratio between the real stress and the nominal stress. The SCF values are derived either through analytical mechanics or experiments. They are widely used for cracks and sharp corners, or any area where the shape of the model takes a sudden transition. The stress resulting from using the SCF can represent either the real stress in the same direction of the nominal stress, or a stress component in a different direction.

For our study we will use SCF factors formulated by the document *DNV RP-CP203* to estimate the stress in the area of transition between the brace and chord (the weld). The nominal stresses in this case are the axial stresses due to axial forces and moments in the brace. For different cases of geometry and load conditions 8 SCF's are given from a set of formulae. When using these factors we get an estimation of the stresses at 8 different points (hotspots) around the weld. How the SCF formulae are derived is not specified, neither is which directions these resulting stresses are representing. What is known is that the stresses are used for fatigue analysis on the weld. Considering that, we can assume their directions are normal to the weld, as that would be the proper stress components for its purpose.

2.5 Force to Stress Matrix

2.5.1 Purpose

Stress Time Series (STS) [3] is a method with the main purpose of reducing computation time. It will be used when comparing results from beam analysis and shell element analysis. As discussed earlier a pipe junction taken from a beam model will be modeled with shell elements. The beam model represents a larger structure with multiple dynamical load cases. The computation time of the analysis is relatively low due to the simplicity of the model. The shell model of the junction, which has a fine mesh, will consist of more degrees of freedom. Instead of using the imported forces over time from the outer ends of the junction and perform a long analysis, we will use the STS method.

2.5.2 Method

We start by executing a unit load analysis on each degree of freedom of the ends of the shell model. For this to be possible we need to restrict displacement in each degree of freedom once, either on one end point or dispersed on multiple end points. In our case we restrict all displacement at the end point to the left of the chord pipe, which leaves us with 18 degrees of freedom on a junction with two braces as seen in figure 2.4.

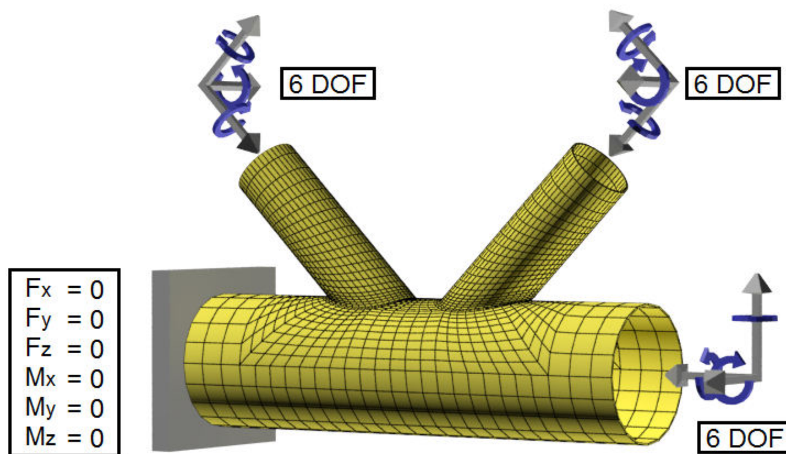


Figure 2.4: The arrows shows the degrees of freedom. The left chord end is fixed.

The stresses of interest are located at so called hotspots (see figure 2.5). Our hotspot stresses are the stresses normal to the weld at four spots on the brace, and four spots on the chord. For each unit load analysis we collect the values of these stresses and create the Force-to-Stress (F2S) matrix with dimensions *number of hotspots* $\times$ *degrees of freedom*.

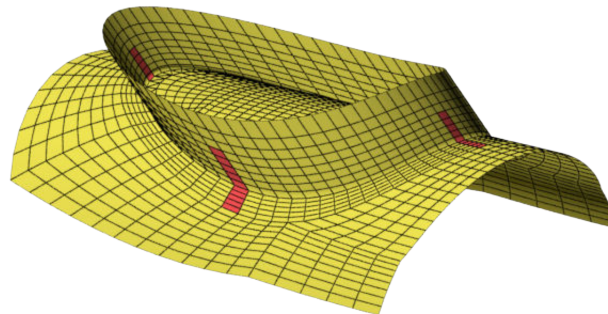


Figure 2.5: Highlighted elements are used for calculating hotspot stresses.

The Force Time Series matrix (FTS) includes the forces at each degree of freedom from the beams' end points for a number of time points, which are taken from the beam analysis result. The FTS matrix has dimensions *degrees of freedom* $\times$ *amount of time steps.*
Using superposition and assuming linear behaviour we obtain the Stress Time Series matrix (STS) by multiplying F2S by FTS. The STS matrix will then include all the stresses at each hotspot for each time step.

2.6 Rainflow counting and Palmgen-Miners rule

Rainflow counting is an algorithm much used in fatigue analysis. It reduces a data set of varying stresses to only the data needed for analyzing the amplitudes. It then allows the *Palmgen-Miner* rule to be implemented to calculate fatigue damage.

The Palmgen-Miner rule (equation 2.2) can, according to *DNV RP-C203*, be expressed as equation 2.1 under the assumption of linear cumulative damage, and is derived from equation 2.2 and 2.3. It dictates that failure is achieved when D reaches the value *1* (that value actually varies, but 1 is generally accepted as a good estimate).

$$D = \frac{1}{\bar{a}} \sum_{i=1}^{k} n_i (\Delta\sigma_i)^m \tag{2.1}$$

$$D = \sum_{i=1}^{k} \frac{n_i}{N_i} \tag{2.2}$$

$$N = \frac{\bar{a}}{\Delta\sigma^m} \tag{2.3}$$

n_i is the number cycles with amplitude $\Delta\sigma_i$ in the total number of cycles k. N_i is the number of cycles the given material with properties m and $\bar{a}$ can sustain with the amplitude $\Delta\sigma_i$ before failure. In our case of using the rainflow algorithm we calculate the damage obtained after each cycles as they are found, and n_i will always be 1.

Equation 2.3 is based on the SN representation of cycles to failure of a given material.

Chapter 3

Procedure and developed software

3.1 Chapter introduction

In this chapter we will go through the procedures followed to arrive at the final comparison results at the end of the report. A few programs and scripts have been created to reach the final goal and some of these will be described in details.
The comparison involves multiple steps of processing information and the use of different software. This makes it complicated to create one application that performs all steps. As we will see in this chapter the procedure uses several programs and scripts created by the author. At the end of the report we will discuss possible changes of how the process could be done to make it more automatic.

To first get a better understanding of every step, a flow chart on the next page shows how each program, script, and file are connected. A short description of the charts items is also included. Throughout the reading of this report, this flowchart can be used to see where in the process things are happening.

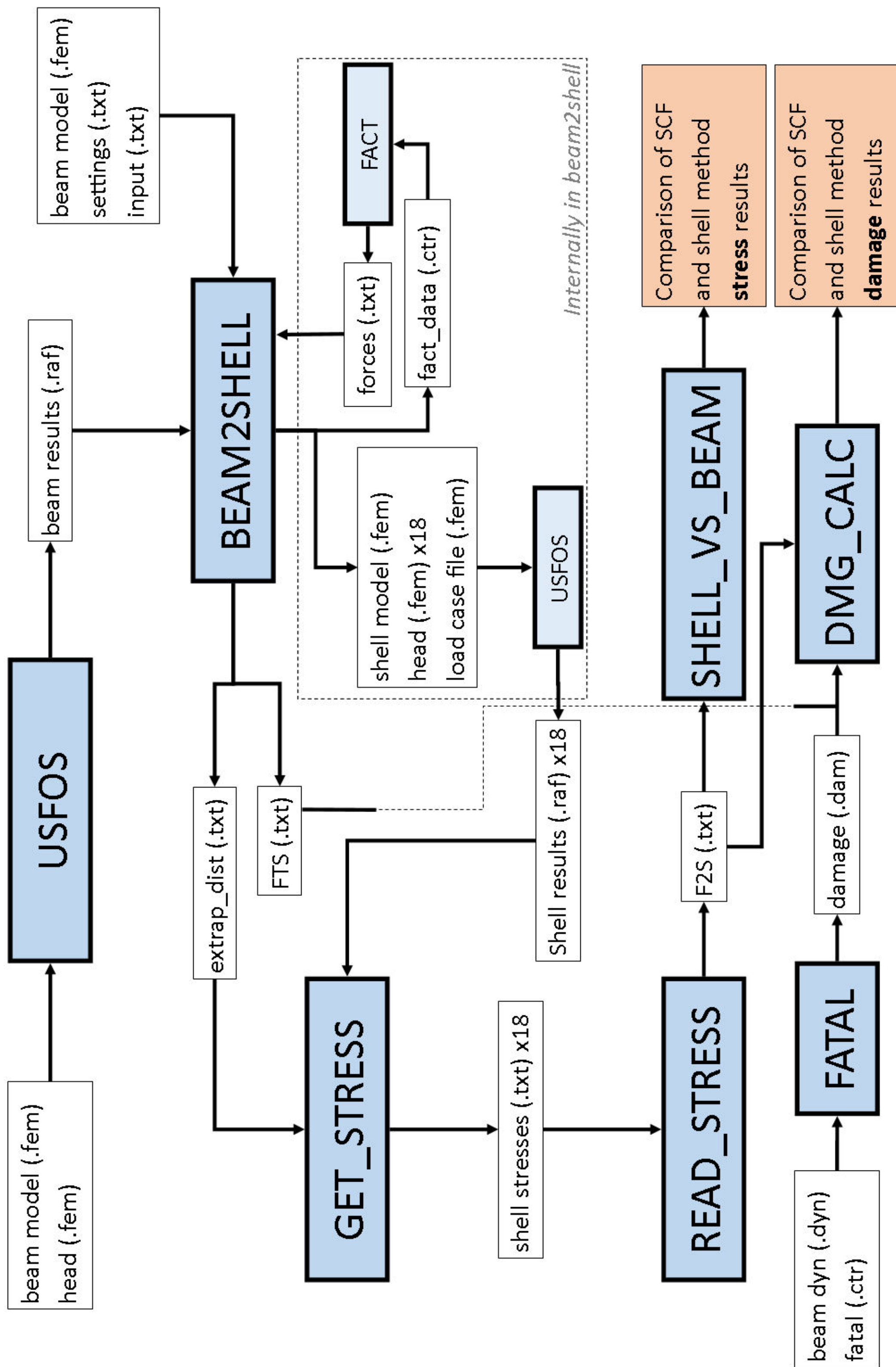


Figure 3.1: Flow chart of the procedure

3.1.1 Flow chart description

Files	**Description**
beam model	USFOS model file of beam element structure
head	USFOS file describing the analysis type
beam results	USFOS result file of the beam element analysis
settings	contains parameters for the meshing module of *beam2shell*
input	contains input information for *beam2shell*
fact_data	input file to run FACT, contains beam IDs
forces	output file of FACT, contains all forces for each time steps of the joints beams
FTS	FTS matrix created from *forces*
shell model	USFOS model file of shell element model
load case	USFOS file containing the 18 load cases for unit loads
extrap_dist	contains element lengths of the elements near hotspots
shell_results	USFOS result files for each unit loads
shell_stresses	contains all the relevant stresses of elements near hotspots
F2S	F2S matrix after processing *shell_stresses*
beam_dyn	USFOS output file containing forces per time step, required for FATAL to run
fatal	FATAL input file contaning SCF values for damage calculation
damage	FATAL output file containing total fatigue damage
Programs/Scripts	
USFOS	FEM software used for all FEM analysis in this report
BEAM2SHELL	Program written mainly for creating a tubular joint in shell elements
FACT	From the USFOS package, used for extracting beam forces to files
GET_STRESS	Script written for extracting hotspot stresses from USFOS result files
READ_STRESS	Program written for processing the hotspot stresses and creating the F2S matrix
SHELL_VS_BEAM	Script written for comparing stresses from the shell element and the SCF method
FATAL	From the USFOS package, used for calculating accumulated fatigue damage in a joint
DMG_CALC	Script created for comparing accumulated damage from the shell element and the SCF method

3.2 Software summary

beam2shell.exe

This is the main program used in the thesis. It reads an USFOS beam model file, and outputs a shell model file of a selected junction. It also reads (through *FACT*) the forces acting on the junction to create the FTS matrix to be used for calculating the STS matrix for fatigue analysis.

get_stress.au

A script that reads the relevant stress results from a shell analysis. Used in both Chapters 4 and 5 for creating the F2S matrix.

read_stress.exe

As *get_stress.au* has to be run for every unit load result files, *read_stress.exe* reads all the files created by *get_stress.au* to form the final F2S matrix. Used in both Chapters 4 and 5 for creating the F2S matrix.

shell_vs_beam.m

Reads the F2S matrices for several geometries to calculate the STS matrices, then compare the results with the SCF method and prints out readable result tables. Used in Chapter 4.

dmg_calc.m

Reads a F2S and a FTS matrix, then using rainflow counting it calculates the accumulated fatigue damage on one brace. These results are then compared with the results from FATAL. Used in Chapter 5.

3.3 beam2shell

3.3.1 Why create the program

There are two main reason for writing a program specifically for creating a pipe intersection mesh; firstly the shape's complexity and secondly the need of specific positions and orientation of certain elements. The intersection of two pipes or cylinders is a relatively complex function. We also need the elements to be well oriented to this function's path in order to be able to perform the calculations needed.

Most FEA software have their own meshing modules. We can categorize elements, both in 2D and 3D in quadrilateral and triangular elements. The average user of a FEA software would use the meshing module provided in the software to automatically create a mesh of a model. These algorithms work very well for triangular elements, which can easily form any model. But very often quadrilateral elements are preferred, because they would reduce the amount of elements required and thus decrease the analysis' calculation time, and also tend to give more accurate results. On the other hand, the algorithms for automatically mesh with quad elements are less powerful. They cannot mesh any arbitrary shape desired, and usually require human assistance to split and define the object in simpler geometrical parts (circles, spheres, squares, cubes, arcs, etc. . .).

In the case of pipe intersections, it would be possible for a person to use a normal FEA software to mesh the part although it is very time consuming to split it up in *quadifiable* parts. There would also be the need of repeating this operation if some geometrical variables are changed, for example the pipes radius or angle. A significant amount of working hours can be saved in cases like this, by creating a software able of handling it automatically.

The meshing module in this program is its main function, but it has two other functions as well. We will later look at hotspot stresses and fatigue damage accumulation. To acquire correct stresses we will use extrapolation and *beam2shell* will collect the needed element lengths for this purpose. For the fatigue damage we also need the forces over time acting on the junction, and these are found by implementing $FACT$ in the program.

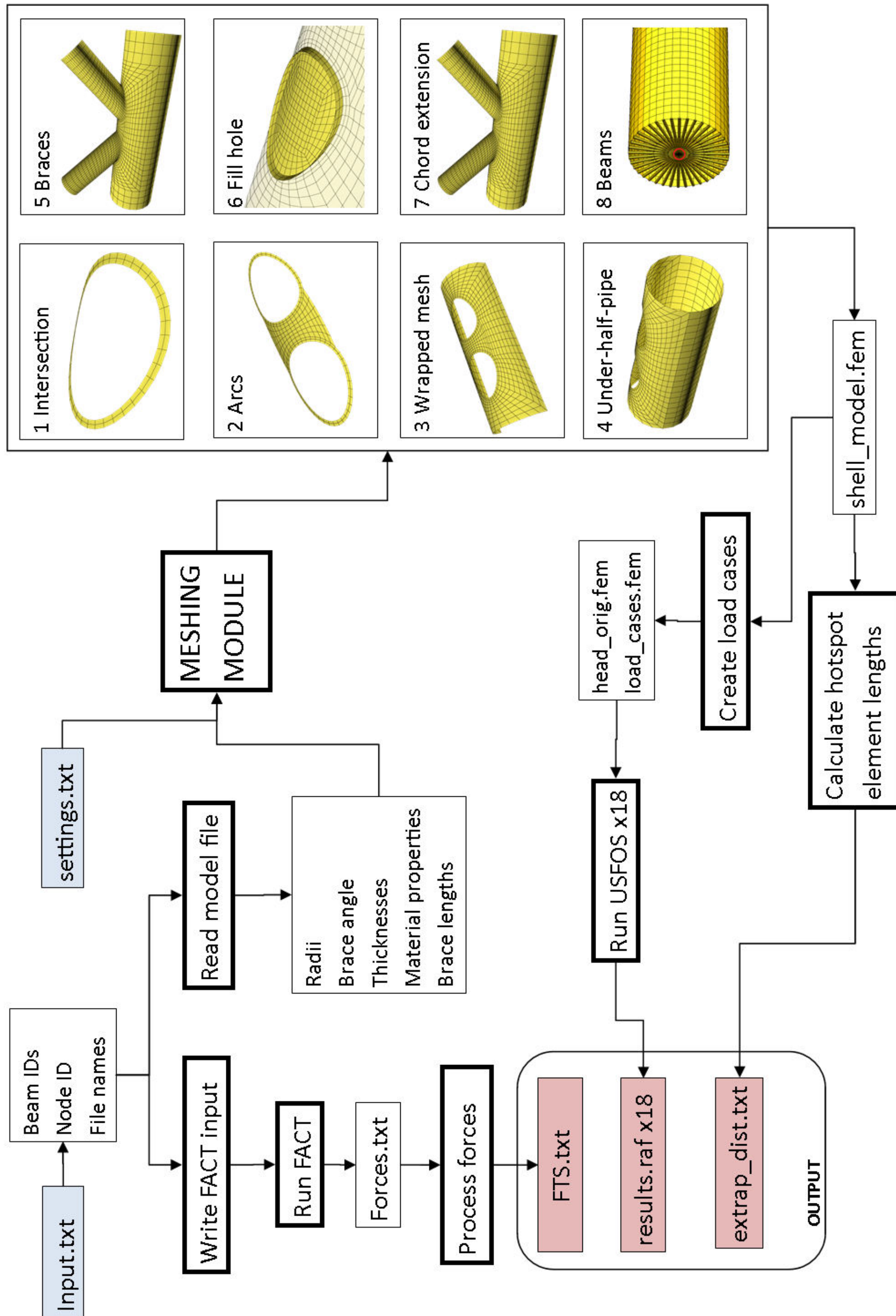


Figure 3.2: Flow chart showing the main components of what is done in *beam2shell*.

3.3.2 Meshing

About meshing algorithms

Most quad mesh algorithms work in an iterative way by defining certain boundaries and then starting to build elements along those. It then builds elements attached to the previous ones, and fill a whole surface or volume. At last it iteratively refines and fix distorted elements following certain algorithms. The procedure used in this case is a lot more mathematical, in the sense that the whole model is described in functions depending on the geometrical variables of the pipe. As there are no iterations, the mesh is created very quickly, but at the cost of being limited to specific tubular junctions.

The code behind the meshing module can be found in *Appendix A*.

Variables

We start by defining a few variables:

R_m	Main pipe radius (chord radius)
R_b	Branch pipe radius (brace radius)
ϕ	Angle between branch and main pipe
x_1	x-coordinate of the outer left part of main pipe
x_3	x-coordinate of the outer right part of main pipe
l_b	length of branch pipes
n	number of elements around intersection
t	Angle vector $[0,2\pi]$
x,y,z	Global coordinates with the origin in the junctions center
g	Gap between the braces

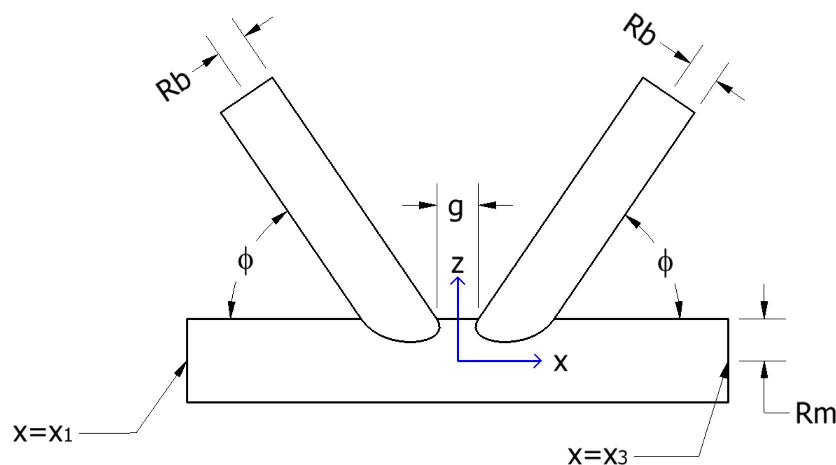


Figure 3.3: Sketch of junction

Intersection

The two intersections are described by parametric functions which can be shown to be:

$$\begin{cases} y = R_b \sin(t) \\ z = \sqrt{|R_m^2 - y^2|} \\ x = \frac{R_b}{\cos(\phi)} \cos(t) + z \tan(\phi) \end{cases} \tag{3.1}$$

The gap between can be calculated from the geometry of the beam model, and can be shown to be:

$$\frac{g}{2} = 2R_m \tan(\phi) - \frac{R_b}{\cos(\phi)} \tag{3.2}$$

This value is used if no other values are specified in the settings file.

Intersection layers

Both intersections are then surrounded by a number of layers defined by the same function, but with different parameters to increase their perimeters gradually. We can see the intersection functions as a distorted ellipse and introduce Rx and Ry radii and the virtual angle α, which are functions of the layer number. These values will be modified for each layers. The radii will increase, and the virtual angle α will be decreased and replace ϕ. By doing this, the layers will increase in size and morphing its shape from a distorted ellipse towards an ellipse (and eventually a circle). We get:

$$\begin{cases} y = R_y \sin(t) \\ z = \sqrt{|R_x^2 - y^2|} \\ x = R_x \cos(t) + z \tan(\alpha) \end{cases} \tag{3.3}$$

At this stage we have well defined the important elements around the intersection.

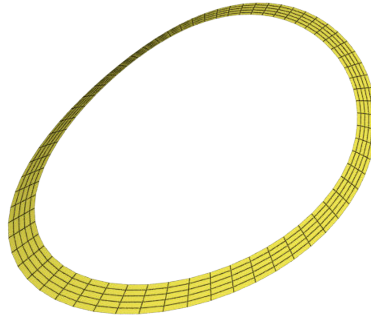
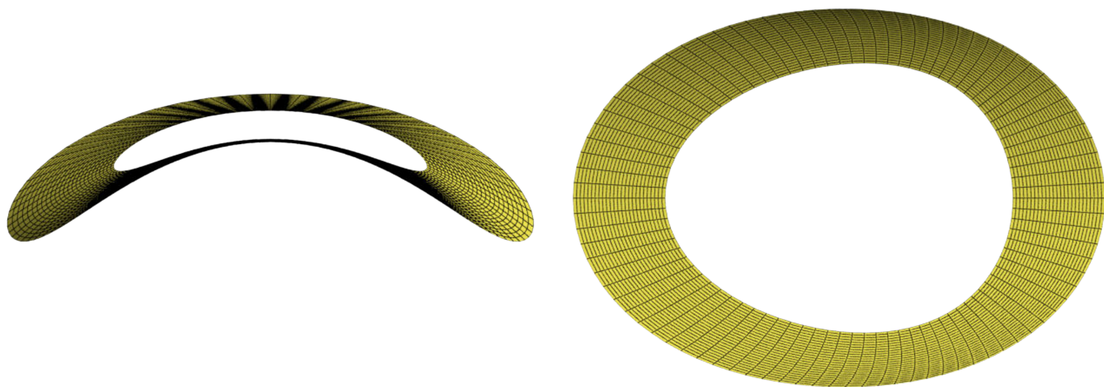


Figure 3.4: Five intersection layers. Note that with more layers, the morphing to an ellipse would be clearer.



(a) 25 layers seen from front view (b) 25 layers seen from top view

Figure 3.5: The layers morphs gradually to an ellipse.

The use of many layers is very useful to obtain a clean mesh when using only one brace. When two braces are used, multiple layers create complications for the mesh connection between the braces (the gap). In the report we will only look at junctions with two braces with 5 layers on each.

Arcs

The parts of the intersection's outer layers facing each other are connected by arcs with a node density which corresponds more or less to the density around the intersections. The arcs are defined by circles intersecting the corresponding nodes on the outer intersection layers, and an increasing radius. The user may change parameters adjusting the radius of the inner and outer arcs, the node density per arc and the amount of arcs.

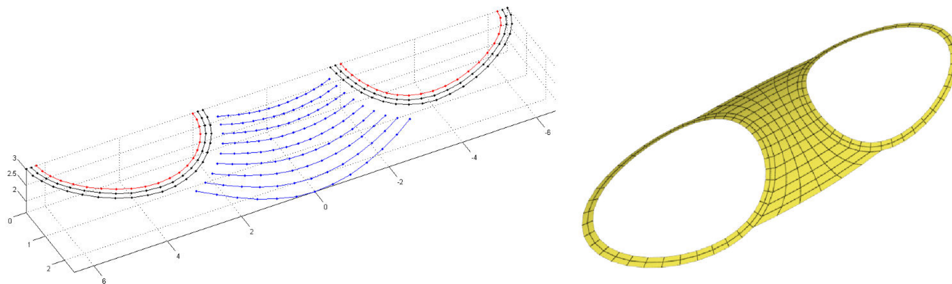


Figure 3.6: Left: arcs in blue. Right: Resulting mesh.

Perimeter and border

Now we can define two vectors. *Perimeter*, which includes all the nodes around the outer intersection layers and the outer arcs. And *Border* which includes nodes on a rectangle projected on the main pipe. All the coordinates from these vectors' nodes are then unwrapped from the pipes surface. For each of the nodes on the perimeter we define a line connecting it to the corresponding node on the border. Each line contains an amount of nodes. The dispersion of theses nodes are decided through a function *nonlinspace* increasing the distance between the nodes nearer to the outer border. Once all the coordinates are defined in the unwrapped space, they are wrapped back on the pipes surface. The reason for doing this wrapping and unwrapping, is that it is easier to define the node dispersion in a "flat" space. The x-coordinates stay the same, and the y-coordinates can be found with equation (3.4) and (3.5) [1]:

$$y_{wrapped} = R_m \sin(\frac{y_{unwrapped}}{R_m}) \tag{3.4}$$

$$y_{unwrapped} = R_m \arcsin(\frac{y_{wrapped}}{R_m}) \tag{3.5}$$

The z-coordinate is then found with equation (3.6):

$$z_{wrapped} = \sqrt{Rm^2 - y_{wrapped}^2} \tag{3.6}$$

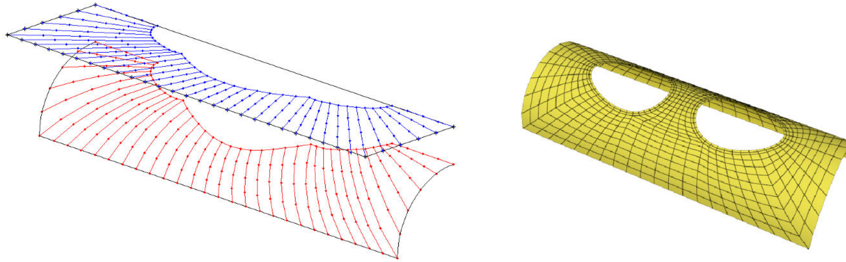


Figure 3.7: Left: Unwrapped nodes in blue and wrapped nodes in red. Right: Resulting mesh.

Completing the main pipe

The lower part of the main pipe, can easily be defined by half circles. The node dispersion of these are such that the elements in this section have similar sizes as the elements near the border. The pipe is also extended on both sides to reach the desired size.

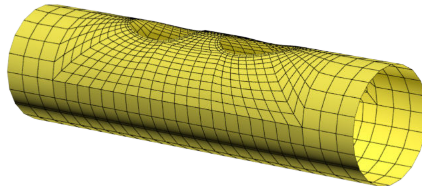


Figure 3.8: Resulting mesh after adding bottom and extended sides.

Branches

The mathematical modeling of the branches mesh is slightly more complicated. First we define a function *rot_transl_def_ellipse*. The function will create an ellipse with radii matching an ellipse created by a cylinder cut, rotate the ellipse, translate it and give it a deflection (see figure 3.9). The rotation is done by using *Rodrigues' rotation formula* [4].

$$\mathbf{X}_r = \mathbf{X}\cos(\alpha) + \mathbf{k} \times \mathbf{X}\sin(\alpha) + \mathbf{k}(\mathbf{k} \cdot \mathbf{X})(1 - \cos(\alpha)) \tag{3.7}$$

A function, *deflection_func* is created to make a third order polynomial that will have zero value and first derivative value at L1 and L2, and -1 value and zero first derivative value at L1+p(L2-L1) where p∈[0,1]. This function is applied on both sides of the ellipse using symmetry. Starting from the bottom a certain amount of the deformed ellipses are placed over each other. The first ellipse will have a deflection matching the intersections z-deflection. The location and value of the maximum deflection depends on the branches angle and radius. For each precedent ellipse the deflection value is decreased. For the mesh's quality sake near the intersection, the first ellipses are parallel to the intersection. The next ellipses are rotated gradually such that they will end parallel to the circular top end of the branch. Applying Rodrigues' equation in our case gives us the position of the ellipses:

$$\mathbf{X}_r = \begin{bmatrix} R_x\cos(t) \\ R_y\cos(t) \\ f_{defl}(t) \end{bmatrix} \cos(\alpha) + \begin{bmatrix} f_{defl}(t) \\ 0 \\ -R_x\cos(t) \end{bmatrix} \sin(\alpha) + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} R_2\sin(t)(1-cos(\alpha)) + \begin{bmatrix} \Delta x + f_{defl}(t)\frac{\sin(\theta)}{\sin(\phi)} \\ 0 \\ \Delta z \end{bmatrix} \tag{3.8}$$

R_x,R_y	Ellipses radii depending on the branches angle and radius
$f_{defl}(t)$	z-deflection
θ	Angle between ellipses normal and branches angle
Δ x,Δ y, Δ z	Space translation
$f_{defl}(t)\frac{\sin(\theta)}{\sin(\phi)}$	Correction in x-direction after z-deflection
α	Orientation of ellipse

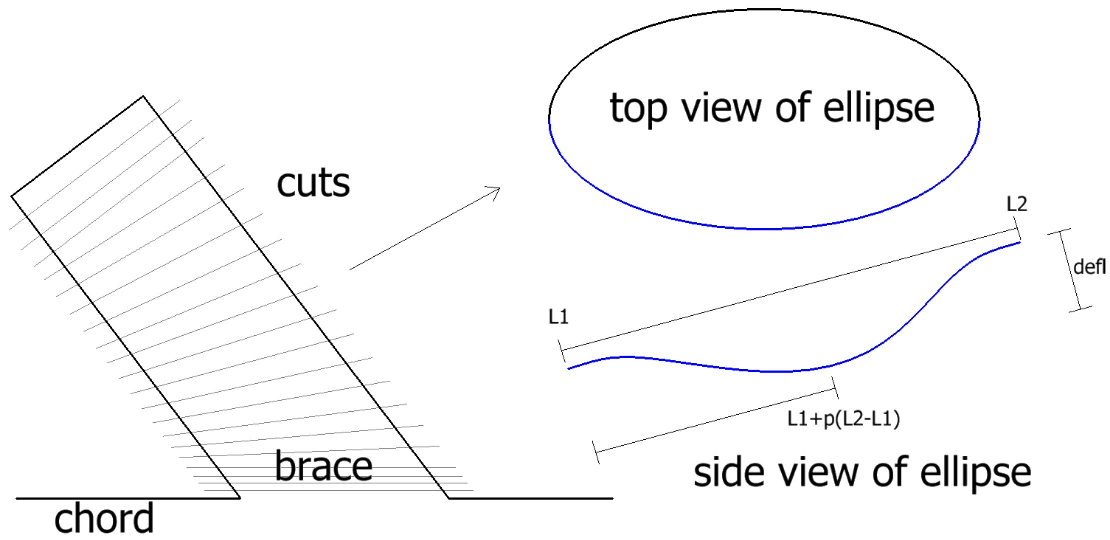


Figure 3.9: Ellipses with dimensions from a skewed cylinder cut gets a deflection added.

Figure 3.10: Left: first layer of elements, made of deformed ellipses. Right: Start of branch mesh.

Closing the intersection hole

A similar method to the one adopted between the perimeter and projected border in the previous section is used here. Each node inside the intersection is connected to a node on a square inside the intersection. This way the intersection shape is morphing to a square, and the hole can easily be closed off.

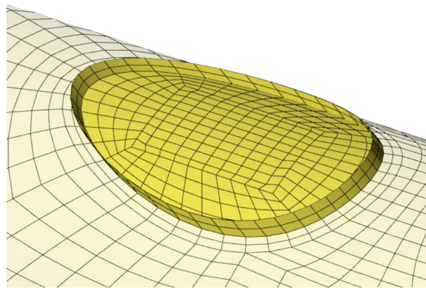


Figure 3.11: Resulting mesh after closing intersection.

Connection beams

To be able to add unit loads to the joints' ends, all the end nodes are connected to a common node. For the two chord and two brace ends, we add a beam to every node. All beams have their second end connected to a node in the center (figure 3.12) . By then adding high stiffness properties to the beam, the center node will act as a reference point for the loadings.

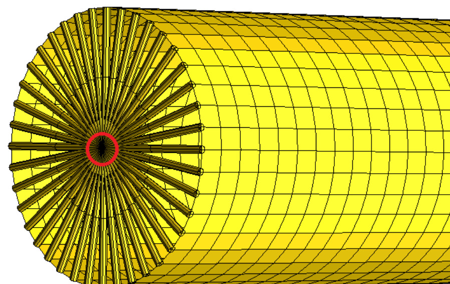


Figure 3.12: Beam elements connecting end nodes to one point.

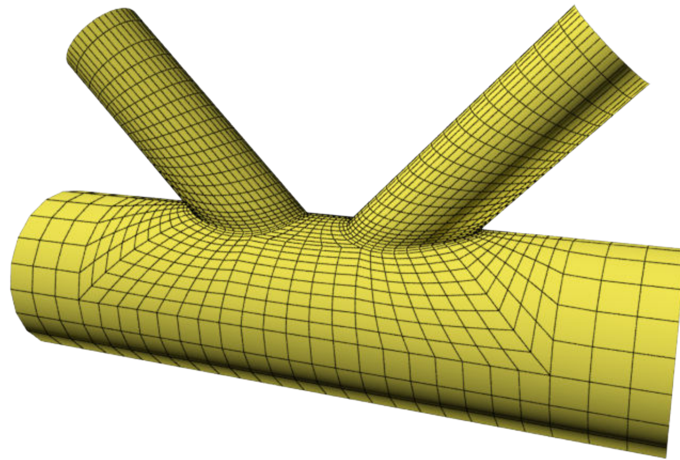


Figure 3.13: Resulting mesh

3.3.3 Main program

Input

The program has four methods of taking in the input:

load Main method, it reads an input.txt file created by the user.
manual The same input that would be in the input file is typed in the console.
create Geometric parameters are typed in the console to *only* create a mesh.
test A sample mesh is created.

Sample of the input file:

type:	K	K for two braces, Y for one brace
fem-file:	beam_model.fem	Name of the beam input file
raf-file:	results.raf	Name of the USFOS result file
nodenumber:	2	Node ID of junctions center
chord_id1:	1	Beam ID of junctions left chord part
chord_id2:	2	Beam ID of junctions right chord part
beam_id1:	3	Beam ID of junctions left brace
beam_id2:	4	Beam ID of junctions right brace
step:	0	Last step for extracting forces (0 for all steps)
branch:	0	Override beam length of braces (0 for no change)
x1:	0	Override beam length of left chord (0 for no change)
x3:	0	Override beam length of right chord (0 for no change)
usfosrun:	1	1 for running unit load analyses, 0 for mesh creation only
factrun:	0	1 for running FACT and extracting junction forces

Defining geometry and material parameters

We define a set of matrices:

node_id_vec	All node ID's connected to the junction
beam_id_vec	All beam ID's connected to the junction
geo_id_vec	All geometry ID's of the beams
mat_id_vec	All material ID's of the beams
end_np_vec	Defines which end of the beam is not at the junction's center
node_coords	Coordinates of nodes in *node_id_vec*
mat_prop	Material properties of materials in *mat_id_vec*
rad_vec	Radii of chord and braces.
thick_vec	Pipe thicknesses of chord and braces.

The beam ID's are given in the input file, and put in the *beam_id_vec* in the following order: left chord beam, right chord beam, left brace beam, right brace beam.

An extract of the beam model file can be seen below:

	Elem ID	np1	np2	material	geom	lcoor
BEAM	319	313	315	11	58	47
BEAM	320	315	316	11	58	53
BEAM	321	307	315	11	58	49
BEAM	322	303	315	11	58	55
BEAM	323	302	316	2	40	23
BEAM	324	303	316	2	40	38
BEAM	325	304	308	1	52	21

The beam input file is then being searched to locate all the relevant node ID's. Every beam ID is being checked if it is one of the ID's in *beam_id_vec*. When a correct beam ID is found, a check is done to see which node is not the junctions center, and is saved in *node_id_vec*. We then specify the value in *end_np_vec* to be the end point that is not the junction's center (1 or 2). The material and geometry ID's are saved in their corresponding vectors. For all vectors the items are arranged in the same order as described for *beam_id_vec*.
Thereafter we need to find the position of each node ID's. They are described as global x-,y-, z-coordinates in the same file as seen below:

	Node ID	X	Y	Z	Boundary code
NODE	51	−27.000	−13.500	25.000	
NODE	52	27.000	−13.500	25.000	
NODE	53	27.000	13.500	25.000	
NODE	54	−27.000	13.500	25.000	
NODE	57	−10.000	−13.500	25.000	
NODE	58	10.000	−13.500	25.000	

As they are found, the coordinates are placed row-wise in *node_coords* in the same order of the previous vectors of the corresponding beams.
The geometry and material descriptions need to be found. The file is searched again and the properties are placed in *mat_prop*, *rad_vec* and *thick_vec*. Below is an example of how the materials and geometries are defined:

	Geom ID	Do	Thick	Shear_y	Shear_z
PIPE	7	1.800	.075		
PIPE	8	1.800	.070		
PIPE	9	1.800	.067		
PIPE	10	1.800	.062		
PIPE	12	1.650	.060		

	MatID	E–mod	Poiss	Yield	Density	Thermal
MISOIEP	1	2.000E+11	3.000E−01	3.550E+08	7.850E+03	.000E+00
MISOIEP	2	1.800E+11	3.000E−01	3.400E+08	7.850E+03	.000E+00
MISOIEP	3	2.100E+11	3.000E−01	3.200E+08	7.850E+03	.000E+00
MISOIEP	4	2.100E+11	3.000E−01	3.100E+08	7.850E+03	.000E+00

As the chord is supposed to be one pipe, and not composed of two parts like it is represented in the beam model, a property check is performed. If the beams do not have the same diameter, thickness or material properties, a warning is given to the user, and the values are taken from the left chord. Because of the limitations of the meshing procedure, both braces should also have the same diameter.

We now want to find:

directions	Matrix containing the direction vectors of each beam
lengths	Vector containing the length of all beams
x1	Absolute value of local x-coordinate of left chord end point and length of left chord part.
x3	Local x-coordinate of right chord end and length of right chord part
lb	Length of braces
phi	Brace angles

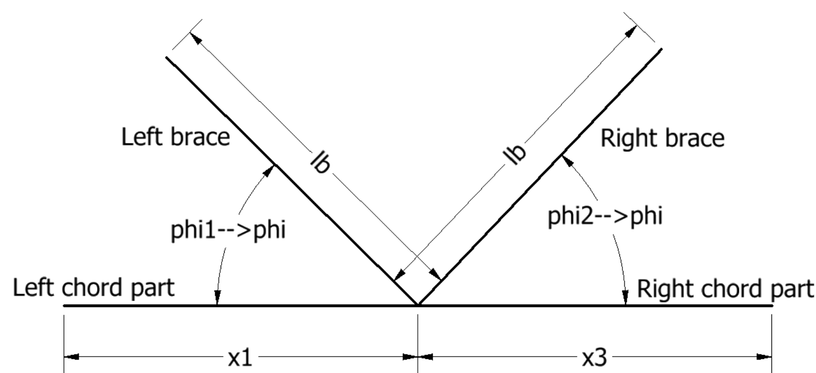


Figure 3.14: Pipe junction

We know the position of all the outer nodes of the junction, and as the *node_coords* matrix has these ordered like we defined earlier, we know which positions belong to which chord and brace part. By subtracting their position vectors by the position vector of the center node, we get each beam's direction vectors. The beam lengths are then found by calculating the direction vectors' lengths. From these values, we get *x1* and *x3*, the chord parts' lengths, and *lb* the braces' length. We want the braces to have the same lengths, and if they are different the values are averaged. In our local coordinate system where we create our shell model, the origin is located in the junction's center. Thus, *x1* and *x3* are also the x-coordinates (absolute value) of the chord's end points.
If values for *x1*, *x3* and *lb* are defined in the input file, the calculated values are overwritten.

To find the brace angles we use the dot product:

$$\mathbb{A} \cdot \mathbb{B} = \|A\|\|B\| \cos(\theta) \tag{3.9}$$

This is done on the left brace and left chord part, and right brace and right chord part. The two angles are defined as *phi1* and *phi2*. Again, if they are different, we average them to *phi* because of the limitations of the meshing procedure.

The geometric values found and calculated in this section are then sent to the meshing module described earlier and the mesh file is created. The user has the option to check the mesh, and make changes to the settings file and rerun the process until the mesh is satisfying.

Extrapolation distances

After the mesh file is completed, *beam2shell* will find the lengths of the elements around the hotspots. There are totally 8 hotspots, four on the brace, and four on the chord. The stresses are taken from the elements, but the extrapolation lengths are taken from the corresponding nodes.

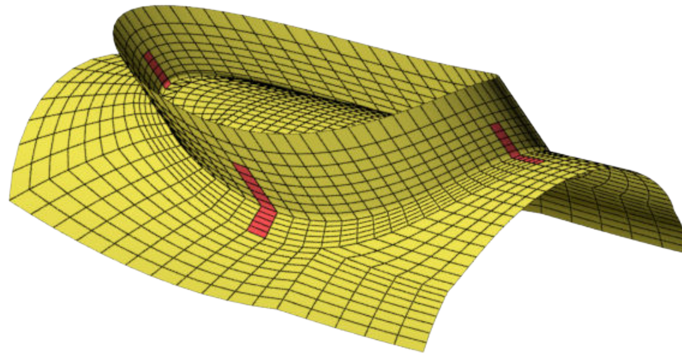


Figure 3.15: Hotspot elements are shown in red.

Because we know how many elements and nodes are around the intersection, and the method that is used to define the ID's in the meshing module, we can easily find the hotspots' element and node ID's.

The function used for this is separate from the meshing module, and will read the final mesh file after it has been created. The nodes' coordinates are defined similarly as we saw earlier in the beam model file. After finding the coordinates of each node, lengths are calculated by subtracting the coordinate of the neighbour node.

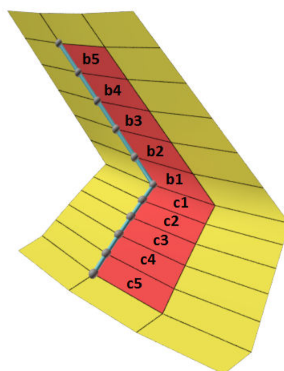


Figure 3.16: We find the lengths between the nodes on the elements b_i and c_i

Creating FTS

FACT, a utility software to USFOS is used to extract the forces going into the junction over time from the USFOS result file. FACT requires an input text file to run and *beam2shell* writes this file and runs FACT. The output file is a text file which shows the forces for the six degrees of freedom of both end points of the relevant beams, for every calculated time steps. This file is then read and processed to create the FTS matrix.

A beam's local coordinate system is defined by its x-axis going from end point 1 to end point 2. The other axes are defined from the x-axis and a unit vector $\vec{u}$ described in the USFOS model file for each beam. The y- and z-axes are then:

$$\vec{y} = \vec{u} \times \vec{x} \tag{3.10}$$

$$\vec{z} = \vec{x} \times \vec{y} \tag{3.11}$$

If there is no defined unit vector, USFOS defines one through an algorithm. In our case, to simplify the programing, we manually add unit vectors to the right chord and the two braces. By defining them as seen on the figure below (all vectors are in the plane of the junction), we force the z-axis to have the same direction as the unit loads in the shell analysis (see figure 2.4).

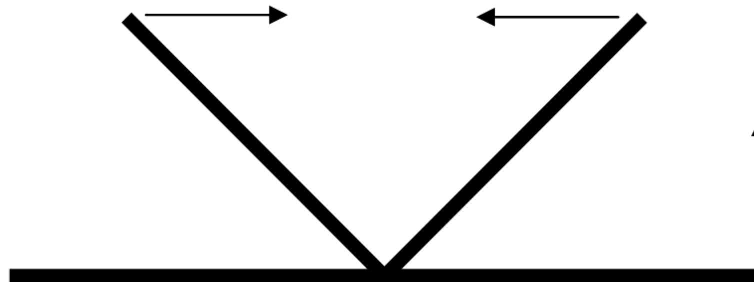


Figure 3.17: Direction of unit vectors.

In the unit loads the positive x-direction is inwards towards the junctions center. If end point 1 is then at the junction's center, we switch the direction of the forces in x and y direction to match the convention (in other words, rotating 180° around the z-axis). If end point 1 is at the opposite end, we leave the forces as they are.

The SCF method is based on the beam forces at the end point located at the junctions center. It will only use stresses originated from axial force and moment in and out of plane. In the shell analysis we place all our loads at the outer end of the braces and chord. What happens then, is that we get a moment from both the moment force and the shear force. This will drastically increase the resulting moment in the junction's center. To address this problem and keep the shear effect from the shear force, we subtract the moment caused by the shear force from the moment force.

This will result in that the forces corresponding to Fy and Fz in the FTS matrix will represent both moment and shear, while My and Mz will be the

remaining moment force. As Fx is axial force, and Mx is torsion, there is no need for any corrections.

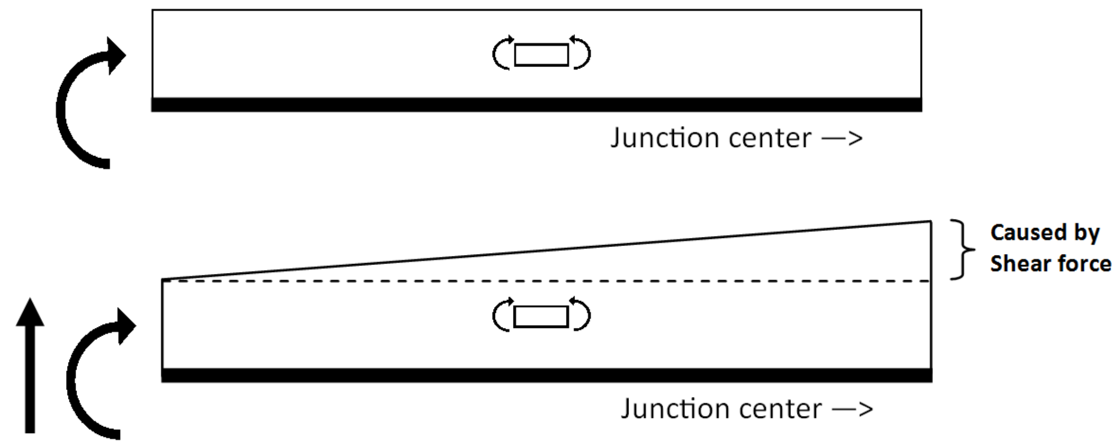


Figure 3.18: Top: resulting moment at the junction from the SCF method. Bottom: resulting moment if no correction is done in the shell method.

Analysis

A load case file *load_cases.fem* is written by the program. According to the brace angles and the center beam nodes at the right chord end and both braces ends, 18 unit loads are defined in each degree of freedom. This can be seen in Chapter 2 figure 2.4. *beam2shell* will then run USFOS for each load cases, and for that it creates a new *head.fem* file every run, defining the analysis and the corresponding load case.

3.4 get_stress

After we have performed 18 unit load cases, there is an important amount of information to extract. There are in all 8 hotspots we want to check, for each of those we want the stresses at 5 different distances from the weld, and for some of those we want the stress component in 3 directions. In addition we will extract all the relevant stresses from both the middle and upper side of the elements (lower side stress can be calculated from middle and upper). This makes a total of 140 stresses per load case, giving a total of 2720. The reason for extracting all this information, and its significance will be discussed later in the report.
To extract it all, 3 possible methods were considered:

Manually: Manually enter the USFOS GUI and collect each values.

Unix shell script: Create a script using Unix shell, which should make it possible to print all the needed values.

AutoIT script: Create a script in the AutoIT language, which will in some sense do the *manually* method automatically with a macro, and pasting each values in a text file.

While the second method would be the most effective, we have encountered difficulties making it work properly due to lack of information on the scripting method. The method used in the thesis was using AutoIT and is shortly described below:

- The scripts need 2 input values, n, the amount of nodes around the brace intersection, and *B_ID* the last element number of the analyzed brace. Since the method of creating the mesh is done in a very systematic way, we can easily calculate the relevant element ID's from only these two numbers.
- Using a loop going through the list of elements, AutoIT performs the steps required to extract the stresses from the USFOS GUI and paste them in a text file.
- This script is run for every load case, and we are left with 18 files that need to be processed to create the F2S matrix.

3.5 read_stress

3.5.1 Purpose

read_stress.exe is written with C++ and reads the 18 result files created by *get_stress*, and *extrap_dist.txt* created by the meshing module, which includes the respective lengths of the relevant elements. We also need to input the brace angle, the element layer numbers to extrapolate from and the type of stress considered (membrane or upper side).

To better understand what happens in *read_stress.exe* three concepts will be briefly explained:

- Shell element stress transformation
- Stress extrapolation
- Shell stress types

3.5.2 Stress Transformation

As mentioned we want to access the stresses in the direction normal to the weld. In USFOS we can extract the S_{xx}, S_{yy} and S_{xy} stress of an element, which are, respectively stress component in x-axis, y-axis and shear. If the element is a perfect rectangle, the element's local coordinate system has direction from node 1 to node 2 as x-axis and 1 to 4 as y-axis.

The elements around the intersection on the chord are part of the *intersection layers* mentioned earlier. These elements are rectangular with their x-axis perpendicular to the intersection. Thus we only need the S_{xx} stress. On the brace, the mesh geometry is different, and the two non-horizontal lines are angled with the brace's angle. To find the component parallel to the intersection we need to use S_{xx}, S_{yy} and S_{xy} for each element and do a transformation.

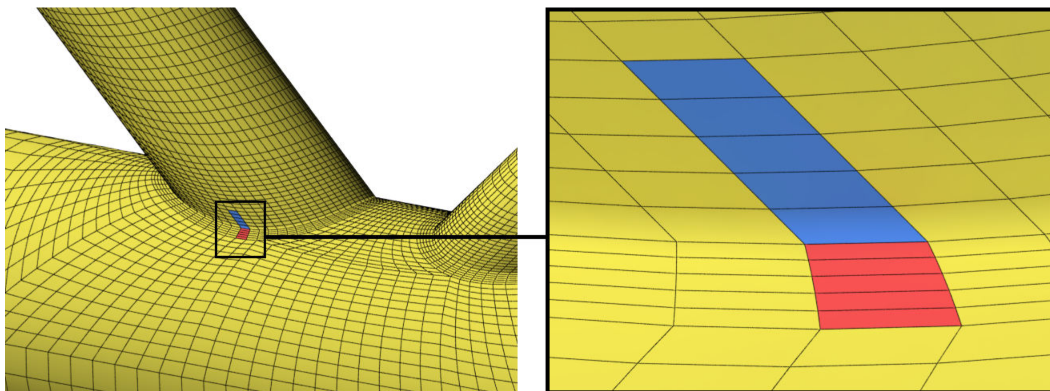


Figure 3.19: Elements on the brace are angled. Elements on the chord are rectangular.

For non-rectangular quad elements, the local coordinate system is decided through these steps:

1. Define directions node 1 to node 2 and node 1 to node 4 as temporary x- and y axes.

2. Calculate the angle between the axes, θ

3. Calculate $\alpha = (\pi/4 - \theta)/2$

4. Add α to both temporary axes making them perpendicular, and define them as new local axes.

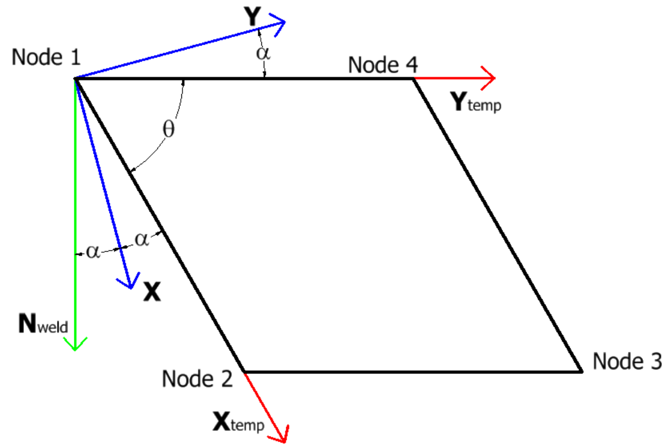


Figure 3.20: Element coordinate transformation

We can then find the component in N_{weld} direction as seen above, by doing a clockwise rotation of α degrees. The component parallel to the brace has the same direction as X_{temp}, and require a counter-clockwise rotation of α degrees. (Note that this rotation is for one side of the brace, on the other side, the elements requires an opposite rotation.)

An arbitrary counter-clockwise rotation of β is done by using the transformation matrix and applying it to our stress tensor matrix:

$$T = \begin{bmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{bmatrix} \tag{3.12}$$

$$\begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix} = T \begin{bmatrix} S_{xxtemp} & S_{xytemp} \\ S_{xytemp} & S_{yytemp} \end{bmatrix} T^T \tag{3.13}$$

The new S_{xx} can then be shown to be:

$$S_{xx} = S_{xxtemp} \cos(\beta)^2 + S_{xytemp} \cos(\beta) \sin(\beta) + S_{yytemp} \sin(\beta)^2 \tag{3.14}$$

3.5.3 Extrapolation

DNV's procedure has defined the values for extrapolation lengths, which are dependent on the thicknesses and radii of the model. In our case, as the program *beam2shell* does not let us create elements at the exact positions desired, we will not achieve the exact extrapolation lengths. Anyhow, the lengths are calculated and can be accessed when the mesh is created and refinements can be done to achieve values near the proposed ones. We can also access more than two different extrapolation points, and will in some of the analyses compare the different extrapolations lengths to see how it affects the result.

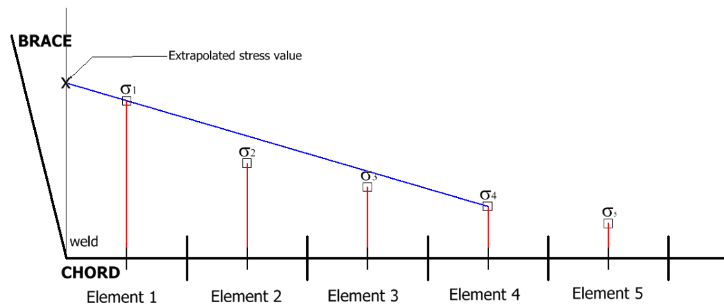


Figure 3.21: Extrapolation with element 1 and 4

3.5.4 Membrane, Upper and Lower side stresses

Shell elements are used for simplifying the simulation of thin objects by being designed for taking the models full thickness in one element. As the stresses will to a certain degree vary over the thickness, it can be useful to know the stresses at different element heights. In USFOS we can chose to extract membrane, lower side, upper side and bending stress. Membrane stress can be explained in two different ways. Either as the average stress over the thickness, or the elements stress without considering the bending effect. The membrane stress is the stress at the center, so it will obviously not receive any effect from element bending.

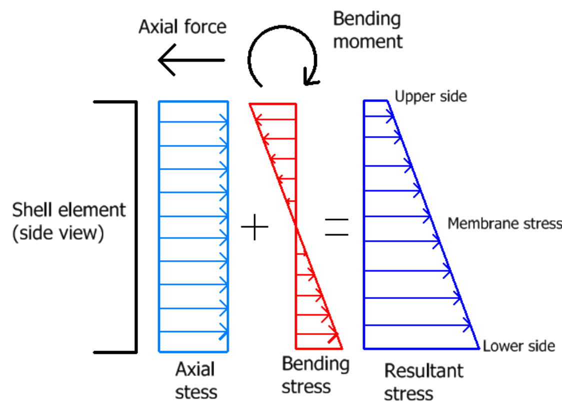


Figure 3.22: Membrane, upper and lower stress due to axial force and moment.

3.5.5 Steps

- Convert the element lengths from *extrap_dist.txt* to distances from the brace intersection to the extrapolation points. The element stresses represent the stress value at the elements center and this needs to be accounted for when calculating the distances.
- For each result file corresponding to each load case, the stresses (membrane or upper side) are read and saved in matrices $S_{xxbrace}$, $S_{xxchord}$, $S_{yychord}$ and $S_{xychord}$. The hotspots are then arranged row-wise, and the element layers column-wise.
- Having all the values in matrices, we can easily perform extrapolations with two of the columns to get the predicted stress at the weld.
- A transformation is done with the extrapolated values of S_{xx}, S_{yy} and S_{xy} on the brace to get the component S_{norm}, normal to the weld, and S_{brace} parallel with the brace.
- We end up with 4 stresses on the chord, 4 on the brace normal to the weld, and 4 parallel with the brace. These are printed column-wise for each load case and represents the F2S matrix

3.6 shell_vs_beam

shell_vs_beam.m is written in MATLAB and will read the F2S matrices for a number of geometries, and using a given load case from the user, compare the results with the SCF method. The steps performed are described below:

- The load case chosen is represented in a column vector with the 18 degrees of freedom of the junction, which is the equivalent of the FTS matrix (or *vector* in this case, as we only have one time step).
- The SCF method considers only 6 degrees of freedom, axial force, moment in plane and out of plane on both braces. The defined positive direction of the forces in the SCF method and shell element method are different. Considering this, a *LOAD* vector is created from the *FTS* vector, consisting of the 6 loads for the SCF method.
- From the *LOAD* vector and the geometry of the junction the axial stresses from each load are calculated. This information can then be used in the SCF formulae given in *DNV-RP C203 Appendix B*. A check is done on whether or not the loads are balanced in the braces, which will change the SCF formulae used.
- The SCF formulae together with the calculated stresses and the equation 3.3.1 in *DNV RP-C203 Chapter 3* give us the predicted weld stresses by the *SCF method*.
- By multiplying the F2S matrix with the FTS vector we get the weld stresses predicted by the *shell element method*.
- Both predicted stresses are then printed out, and are the values used in chapter 4 to discuss the differences between the two methods.

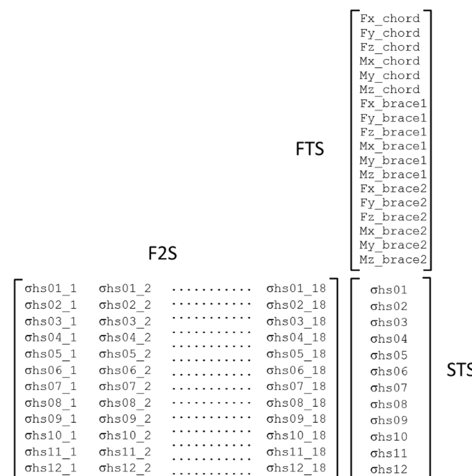


Figure 3.23: Illustration of connection between F2S, FTS and STS

3.7 dmg_calc

3.7.1 Procedure

dmg_calc is written in MATLAB, and is used to calculate the accumulated fatigue damage on a brace. It then compares it with the damage calculated with FATAL which is based on stresses from the SCF method.
The steps taken in this script:

- The FTS matrix which was created from *beam2shell*, and the F2S matrix from *read_stress* are read.
- The STS matrix is calculated similarly as in *beam_vs_shell*, but note that the FTS and STS matrix will now have one column for each time steps in the analysis. In the fatigue damage calculation we also look away from the four hotspot stresses which are parallel with the brace, leaving us with only 8 hotspot stresses normal to the weld.
- A rainflow counting algorithm is then used on the time history of each hotspots, in other words, on each row of the STS matrix. (This is described in details in the next section).
- The rainflow algorithm delivers the accumulated damage for each hotspot.
- The accumulated damage for each hotspot as well as the sum of the damage on brace and chord are compared with the results from FATAL.

3.7.2 Rainflow counting

As mentioned in Chapter 2, Rainflow counting is a method often used for estimating the fatigue damage from a stress history. A rainflow algorithm was written and implemented in *dmg_calc* and will briefly be explained. The code used for the rainflow script can be found in *Appendix B*.

- For a stress history of a hotspot, which consists of one data point for each time step, we check every point for being a turning point. All points which are not, are removed.

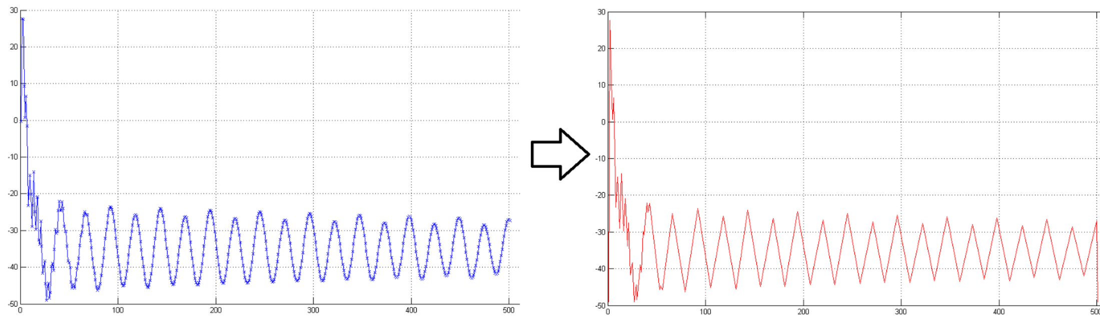


Figure 3.24: The left graph shows a stress history. On the right, only the turning points are left (*"Peaks and Valleys"*)

- As we use *Palmgren-Miners Rule* we need to find every full cycle, its amplitude and then its damage. A full cycle is defined by the condition $\delta 1 > \delta 2$ and $\delta 3 > \delta 2$ on 4 consecutive points. The cycle's damage is then:

$$\delta_{damage} = \frac{1}{\bar{a}} \sigma_{amp}^{m}; \tag{3.15}$$

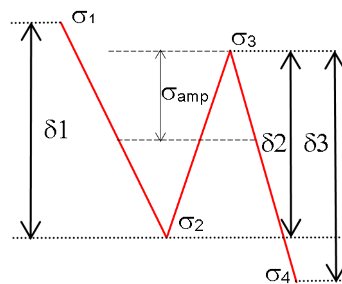


Figure 3.25: Definition of a cycle.

- After all the cycles are found we remove σ_2 and σ_3 for every cycle. As seen in figure 3.26 we are then left with new cycles. This process is repeated until there are no cycles left. (A special case is used when we are left with only three points, but this will not be discussed.)

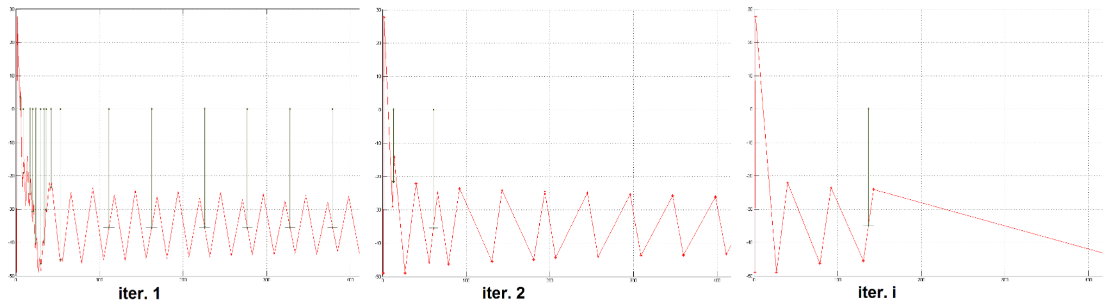
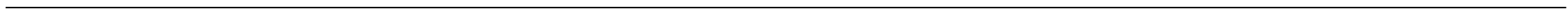


Figure 3.26: The vertical lines shows the locations of each cycles found. These are removed in the next iteration. The iterations are continued until there are no more cycles.

- The damage for each cycle found are summed and give us the total damage of one hotspot. The whole process is repeated for 8 hotspots.



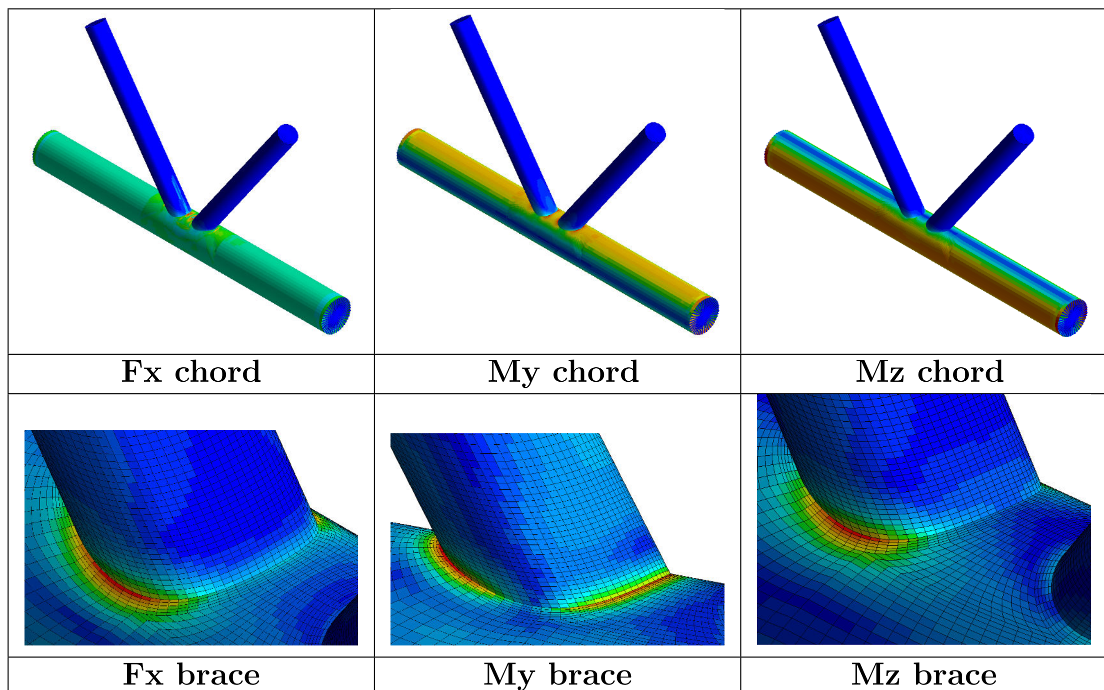
Chapter 4

Compare hotspot stresses

4.1 Introduction

In this chapter we will investigate the stresses arising from the shell element and the SCF method. We will look at how they compare to each other under the same geometry and load conditions, and how the differences change with different conditions.

On the figures below we see the Von Mises stress distribution of some of the unit loads analysis used for creating the F2S matrix.



4.1.1 Load and geometry

To start with, we define 9 geometries with two-brace junctions. We limit ourselves to equal angle and radius on both braces. All geometries will have a unit length on both chord and brace beams, equal pipe thickness and material on the whole model and equal gap between the braces. The variables which will change for each geometry will be the radius ratio between the brace and chord pipes, and the brace angles. To prevent free translations in any direction we fix all degrees of freedom at the left end of the chord.

For each of the brace angles; 35°, 55° and 75°, we will do the analysis with brace to chord radius ratio; 0.35, 0.55, 0.75

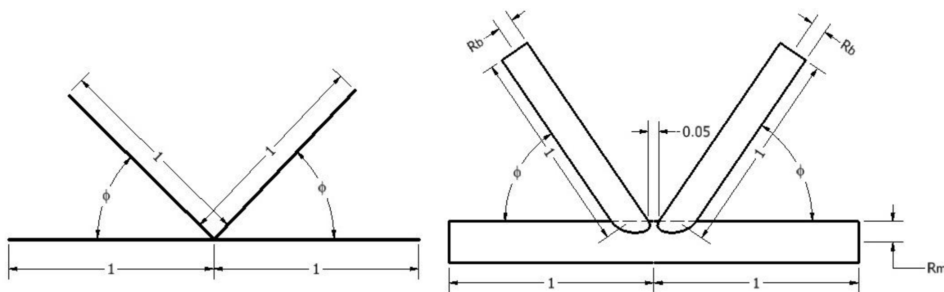


Figure 4.1: Definition of geometry. Left: Beam model. Right: Shell model

In addition to different geometries, we will for each of them look at different load cases. An important fact to note is that the SCF values calculated from *DNV RP-C203* do not take all forces into the junction in account when they are calculated. How these non-accounted for forces affect the shell element analysis will be investigated further.

Below we can see the load cases we will analyze. Each one of them have independent SCF formulae for the hotspots. In addition we will have a case where we combine all loads.

Figure 4.2: Six load cases

4.1.2 SCF Method

The approach DNV's method calculates hotspot stresses is by using the axial stress from axial loading and the axial stresses from in plane and out of plane bending of the brace. These are then scaled with SCF values, which are dependent on the junction's geometry parameters and load conditions. The hotspot stresses can be calculated for 8 points around the brace, for both the chord and brace. As mentioned in Chapter 2, we assume the values calculated with this method represent the stress components normal to the weld.

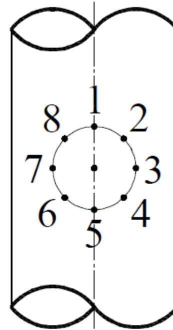


Figure 4.3: 8 hotspots around the brace

4.1.3 Shell element analysis

In the shell model analysis we will limit ourselves to 4 hotspots on the chord and brace. The stresses we are interested in are the stress components normal to the weld. For each analysis we will extract stresses from the brace and chord directed to the weld, as well as parallel to the brace on the brace.

A well-known fact about FEM analysis is that stresses near corners tend to reach unrealistic values and are very dependent on the mesh density. We will work around that problem by using extrapolation of stresses. This is done by looking at the stress values at two points away from the corner, and then do a linear extrapolation. By extracting stresses for 5 elements per hotspots we can try different extrapolation distances to observe the effect.

4.2 Example and procedure

Before proceeding to the results, an example will be shown to better understand the method. For each step, the technique for doing the complete analysis is explained.

4.2.1 Step 1 - Create beam model

Example:

Create an USFOS input file *beam_model.fem* defining the geometry of a pipe junction with brace angle 55° and radius ratio 0.55:

```
                MODEL FILE, phi = 0.959931, radius ratio = 0.55        HEADER
                                2                      2

         Node ID            X       Y       Z     Boundary code        NODE COORDINATES
NODE        1              -1       0       0     1 1 1 1 1 1
NODE        2               0       0       0
NODE        3               1       0       0
NODE        4       -0.573576       0   0.819152
NODE        5        0.573576       0   0.819152

         ELEMID      np1       np2       mat      geo                  BEAM DEFINITIONS
BEAM        1         1         2         1        1
BEAM        2         2         3         1        1
BEAM        3         2         4         1        2
BEAM        4         2         5         1        2

         Geom ID      D          Thick                                 BEAM GEOMETRY DEFINTIONS
PIPE        1        0.2         0.005
PIPE        2        0.11        0.005

         matno    E-mod     poiss    yield    density  therm exp       MATERIAL DEFINITIONS
MISOIEP  1        1e+014    0.3      1e+008   1000     1e-5
```

It should be noted that it is not strictly necessary to create an USFOS input file of the geometry, as for the purpose of this chapter we only look at unit loads. The beam model will not be run in USFOS, the nominal stresses will be calculated by classic mechanics. However, we create the input files to show how *beam2shell* can read them and create the shell model form there.

Full analysis:

9 USFOS files are created with *create_beam_model.exe* written with C++ which let's the user choose an angle, a radius, an angle increment and a radius increment. Giving us 9 models with combinations of angles=[35°, 55°, 75°] and radius ratio =[0.35, 0.55, 0.75].

4.2.2 Step 2 - Define loads

Example:

Using *beam2shell.exe* we create an USFOS shell model input file. In *beam2shell*'s input file, we need to specify the beam model file we just created, and turn off *run_usfos*. In the settings file we specify the gap between the braces to be 0.05 and the number of intersection layers to be 5. The program will create a model file *shell_model.fem*, this file can be opened in USFOS to check the mesh. If the mesh is not satisfactory we can change other parameters in the settings file and create a new model file.
extrapol_dist.txt will also be created and includes the lengths and ID's of the potential elements that can be used for extrapolation, together with the proposed lengths from DNV. When the mesh is acceptable we proceed to manually write a *load_case.fem* and a *header.fem* file which defines a static analysis of one load. We include only an axial load in the left brace. The load is defined in a global coordinate system and a transformation is needed.

```
             ID   NodeID      fx      fy      fz      Mx  My   Mz       USFOS LOAD CASE FILE
NODELOAD      1   22781    0.5736     0   -0.8192     0    0    0
```

Full analysis:

For each of the 9 beam model files, we run *beam2shell.exe*, but this time we turn on *usfos_run*. This way the program will automatically create a load case file including 18 unit loads (one for each d.o.f.). USFOS will be run 18 times, and for each time the header file is updated to specify which load case to be performed. For each of the 9 models we are left with 18 USFOS result files and the extrapolation data.

4.2.3 Step 3 - Extrapolation

Example:

In the result file from USFOS we can manually go in and check the relevant element stresses. We want to check the stresses for hotspot 4 on the brace, parallel to its direction. First we open *extrapolation_data.txt* and get the data we need.

```
# Chord element/node ids:
 1  20  37  53
145  164  181  197
289  308  325  341
433  452  469  485
577  596  613  629
721  740  757  773
# Brace element ids:
9821   9840   9857   9873
9893   9912   9929   9945
9965   9984   10001  10017
10037  10056  10073  10089
10109  10128  10145  10161
# Element widths;
0.0017578     0.00172524    0.001758     0.00169967
0.0017578     0.00174526    0.001758     0.0017203
0.0017578     0.00176688    0.001758     0.00174201
0.0017577     0.00178988    0.001757     0.00176546
0.0017578     0.00181469    0.001758     0.00179056
0.00259174    0.00280621    0.00259139   0.00280058
0.00256099    0.0027752     0.00256116   0.00276968
0.00253082    0.0027454     0.00253094   0.00273994
0.00250257    0.00271705    0.00250234   0.00271153
0.0024756     0.00270146    0.00247572   0.00270992

DNV proposed extrapolation lengths:
Brace              : a = 0.00331662     b = 0.010779
Chord (crown)      : a = 0.00331662  b = 0.00770257
Chord (saddle)     : a = 0.00331662  b = 0.00872665

brace_el_id        : 10054
brace_node_id      : 10180
n                  : 72
```

We use element ID 10089 and 9945, which are the second and fourth nearest elements to the weld line. The lengths given are defined as the length between node 1 and node 2 in each element. And we calculate:

Distance a from center of element ID 10089 to the weld line = 0.0027099 + 0.0027115/2

Distance b from center of element ID 9945 to the weld line = 0.0027099 + 0.0027115 + 0.0027399 + 0.0027697/2

a = **0.004066** and b = **0.009546**

We go in USFOS and collect S_{xx} and S_{yy} from the two elements:

$$
\begin{array}{llll}
S_{xx10089} & = \text{-}4325.17 & S_{xx9945} & = \text{-}3486.97 \\
S_{yy10089} & = \text{-}2935.24 & S_{yy9945} & = \text{-}2395.65 \\
S_{xy0089} & = \ 1422.54 & S_{xy9945} & = \ 859.79
\end{array}
$$

As mentioned earlier we need to do a transformation to find the stresses in the correct directions. Using equation 3.14 we get:

$$S_{weld10089} = \mathbf{-1720.2}$$
$$S_{weld9945} = \mathbf{-1531.4}$$

The extrapolation is performed:

$$S_{weld} = \frac{b}{b-a}(S_{weld10089} - S_{weld9945}) + S_{weld9945} = \mathbf{-5856.6}$$

Full Analysis:

For each of the 9 geometries we have 18 load cases with each 8 hotspots and 5 corresponding element layers to extract both membrane and upper side stresses from. To extract all these stresses we use the program *get_stresses.au*. The program has to be run for each result file while the *results.raf* file is open in USFOS. It will collect all the relevant stresses and print them out to a text file.

The program *read_stresses.exe* can be run for each geometry and will read the 18 text files created by *get_stresses.au* together with the geometries corresponding *extrapolation_dist.txt* file. The program asks for extrapolation layers and stress type, and then performs the necessary calculations to create the F2S matrix.

We are left with a F2S matrix for each geometry, where we can read every hotspot stress to each load case. The F2S matrix of the model considered in the example is shown below. Notice that the 8th hotspot stress (row 8) in load case 7 (column 7) is the result from our example.

-185.187	268.454	-7892.43	58.5021	-7767.51	-253.303	-3093.04	-6457.07	-109166	4311.64	-108227	6516	924.003	-7851.67	134.32	6066.62	1872.99	8068.98
-6.27793	-259.453	726.217	-1007.38	704.998	288.207	-10173.8	-285294	31992.5	200288	23371.2	290414	-4604.35	92037.8	-22542.4	-57004.5	-24942.4	-95142.3
-249.875	-151.795	-9456.2	-173.088	-8277.7	182.218	-187.197	3174.94	109036	-4303.66	106366	-3540.71	2329.4	-665.276	-5302.09	269.712	-2423.66	726.011
10.4939	119.552	778.809	1755.31	682.146	-252.666	-10232.4	287097	30410.7	-203595	21652.9	-292127	-4696.09	-93348.7	-23027.1	57739.5	-25479.6	96295.4
-10.9806	84.4227	-1181.38	-45.4249	-1269.34	-62.989	-2285.91	-5281.84	-70639.8	4300.91	-69248	5256.3	111.697	-4434.83	3990.13	3245.4	4098.83	4569.19
-176.519	1331.92	-3360.55	-333.072	-3017.06	-1249	-5866.68	-183483	32930.2	127068	27529.5	186311	-2554.32	58444.7	-20550.8	-33217.1	-21132.5	-60243.8
-92.6906	-20.7304	-3310.52	184.421	-2945.64	-2.54237	-184.666	-22.1423	55793.1	3446.05	54465.9	411.547	816.731	-2.47674	-2714.66	142.996	-1721.64	-49.2577
-163.293	-1328.78	-3352.1	337.646	-3009.23	1255.53	**-5856.65**	183591	32955.8	-127145	27550.3	-186410	-2542.88	-58470.7	-20552	33235.6	-21134.5	60278.9
-10.9806	84.4227	-1181.38	-45.4249	-1269.34	-62.989	-2285.91	-5281.84	-70639.8	4300.91	-69248	5256.3	111.697	-4434.83	3990.13	3245.4	4098.83	4569.19
-241.14	2249.54	-4562.72	890.601	-4024.01	-2243.67	-3753.44	-125442	27492.7	81164.7	24238.4	127622	-1405.48	37786.4	-16120.4	-18697.3	-15908.1	-38980.3
-92.6906	-20.7304	-3310.52	184.421	-2945.64	-2.54237	-184.666	-22.1423	55793.1	3446.05	54465.9	411.547	816.731	-2.47674	-2714.66	142.996	-1721.64	-49.2577
-232.464	-2248.41	-4557.61	-888.045	-4019.27	2248.29	-3745.88	125498	27508.3	-81207.6	24251	-127672	-1397.17	-37799.4	-16120.1	18705.7	-15908.2	38998.4

Figure 4.4: F2S Matrix

4.2.4 Step 4 - SCF method

Example:

We want to compare the result with *DNV*'s guidelines. In the *RP-C203* document we go to *Appendix B – SCF's for tubular joints* and find the case of axial load on one brace only:

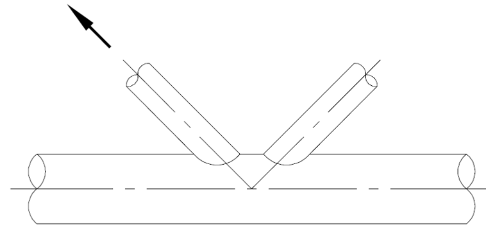


Figure 4.5: Axial load on one brace from *RP-C203* corresponds to the negative of load case 7 used in the F2S matrix.

The SCF for a *saddle* point (hotspots 2 and 4) on the brace is referred to equation (3) in the DNV document:

$$SCF = 1.3 + \gamma\tau^{0.52}\alpha^{0.1}(0.187 - 1.25\beta^{1.1}(\beta - 0.96))\sin(\theta)^{2.7-0.01\alpha} \tag{4.1}$$

There are a few parameters we need to calculate first which are dependent on the models geometry:

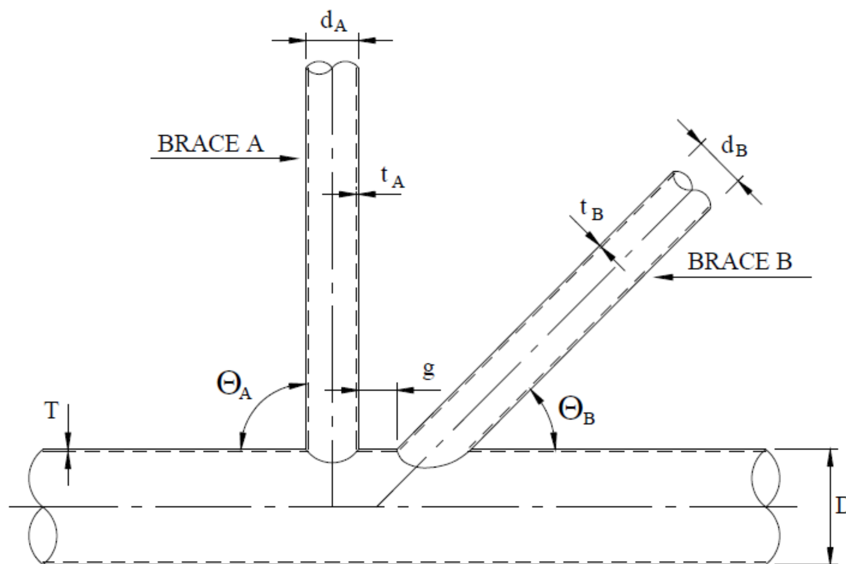


Figure 4.6: SCF's geometric parameters

$$
\begin{aligned}
\theta & & & & &= 55\\
\gamma &= \frac{D}{2T} &=& \frac{0.2}{2\cdot 0.005} &&= 20\\
\tau &= \frac{t}{T} &=& \frac{0.005}{0.005} &&= 1\\
\alpha &= \frac{2L}{D} &=& \frac{2\cdot 1}{0.2} &&= 10\\
\beta &= \frac{d}{D} &=& \frac{0.11}{0.2} &&= 0.55
\end{aligned}
$$

From equation 4.4 we get: SCF = **8.0829**

In the same document in *3.3 Tubular joints and members* we find the equation for hotspot 7, which will correspond to our hotspot 4:

$$S_{hs4} = SCF_{as}\sigma_x + SCF_{mop}\sigma_{mx} \tag{4.2}$$

σ_{mx} is zero as we don't have any moments. σ_x is the stress due to the axial loading and can be calculated by:

$$\sigma_x = \frac{F}{Area} = \frac{-1}{\pi dt} = \frac{1}{\pi 0.11 \cdot 0.005} = -578.75 \tag{4.3}$$

From equation 4.2 we get: S_{hs4} = **-4678**

Full analysis:

The MATLAB script *shell_vs_beam.m* will read the F2S matrix of all geometries and apply the FTS matrix defined by the loading conditions in the script. STS is then calculated for each geometry. It also calculates the corresponding stresses from the DNV's guidelines and prints out the results. In the next section we will look at those results.

4.3 Results

4.3.1 Presentation of Results

The goal of this chapter is to see how hotspot stresses calculated from *DNV*'s SCF formulae differ from stresses acquired from extrapolated shell element analysis results. We will see how these differences vary with geometry and loading and determine which factors in the SCF formulae cause the variation.

The script *beam_vs_shell.m* was used and calculates the ratios between the two methods for each hotspot for each geometric condition. The results are plotted to get a better overview of the variations and eventually find some relations to the geometric factors. This is done independently for every load case.

An example of all the data acquired from comparing the two methods is presented in section 4.3.2. Three stress ratios are shown, where the hotspot stress from shell element analysis is divided by the corresponding stress from the SCF method.

sh. chord / b. chord	: hotspot stress ratio normal to weld on chord
sh. brace / b. brace	: hotspot stress ratio normal to weld on brace
sh. brace (par) / b. brace	: hotspot stress ratio parallel to brace on brace

The hotspot stress ratios are plotted against each other for either constant angle and varying radius ratio, or the opposite. This will give a better overview of how the two variables are affecting the results.
To analyze all this data, we will also calculate a few values that will measure the behaviour of the stress ratios' dependency of the angle and radius ratios.

We define these terms, all regarding one specific hotspot:

Stress range	:	The smallest and largest stress ratio.
Max. var. ang.	:	Largest stress ratio variation caused by change in angle *(largest e_i value)*
Avg. var. ang.	:	The average of all e_i values.
Max. var. rad.r.	:	Largest stress ratio variation caused by change in radius ratio. *(largest d_i value)*
Avg. var. rad.r.	:	The average of all d_i values.

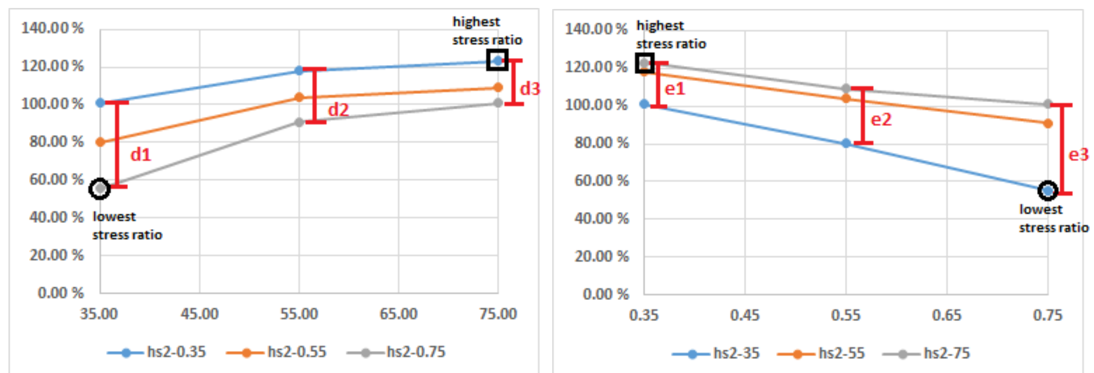


Figure 4.7: Stress ratios vs Angle and Radius ratio

Multiple plots like the two seen above will be used on the next pages. On both graphs, the vertical axis shows the shell method to SCF method stress ratio. The horizontal axis on the left and right figure are angle and radius ratio respectively. While for each line the angle or radius ratio are kept constant.

4.3.2 Example of stress data output:

Axial loading on one brace, Upper stress, extrap. points 2 and 4

Geom.	Angle	Radius ratio	Hot-spot	Beam stress		Shell stress			sh. chord / b. chord	sh. brace / b. brace	sh. Brace (par) / b. beam
				chord	brace	chord (weld)	brace (weld)	brace (par)			
1	35	0.35	1	-5.22E+03	-2.33E+03	-3.95E+03	-3.77E+03	-3.77E+03	**75.60 %**	162.00 %	162.00 %
			2	-7.72E+03	-3.49E+03	-8.80E+03	-5.09E+03	-2.11E+03	114.00 %	146.00 %	60.50 %
			3	-5.22E+03	-2.33E+03	-5.12E+02	-3.94E+02	-3.94E+02	9.81 %	16.90 %	16.90 %
			4	-7.72E+03	-3.49E+03	-8.65E+03	-5.08E+03	-2.11E+03	112.00 %	146.00 %	60.50 %
2	35	0.55	1	-2.94E+03	-1.20E+03	-2.28E+03	-2.14E+03	-2.14E+03	**77.50 %**	178.00 %	178.00 %
			2	-5.37E+03	-2.31E+03	-5.29E+03	-2.69E+03	-9.77E+02	98.40 %	117.00 %	42.40 %
			3	-2.94E+03	-1.20E+03	6.68E+02	1.74E+02	1.74E+02	-22.70 %	-14.50 %	-14.50 %
			4	-5.37E+03	-2.31E+03	-5.10E+03	-2.66E+03	-9.74E+02	95.00 %	116.00 %	42.20 %
3	35	0.75	1	-2.18E+03	-8.29E+02	-1.49E+03	-1.37E+03	-1.37E+03	**68.50 %**	165.00 %	165.00 %
			2	-3.43E+03	-1.50E+03	-3.17E+03	-1.57E+03	-4.92E+02	92.50 %	104.00 %	32.70 %
			3	-2.18E+03	-8.29E+02	1.06E+03	3.83E+02	3.83E+02	-48.90 %	-46.20 %	-46.20 %
			4	-3.43E+03	-1.50E+03	-3.12E+03	-1.55E+03	-4.86E+02	91.10 %	103.00 %	32.30 %
4	55	0.35	1	-5.26E+03	-2.33E+03	-4.95E+03	-4.60E+03	-4.60E+03	**94.00 %**	198.00 %	198.00 %
			2	-1.36E+04	-7.01E+03	-1.64E+04	-1.00E+04	-6.71E+03	120.00 %	143.00 %	95.80 %
			3	-5.26E+03	-2.33E+03	-1.75E+03	-1.19E+03	-1.19E+03	33.20 %	51.00 %	51.00 %
			4	-1.36E+04	-7.01E+03	-1.64E+04	-1.00E+04	-6.71E+03	121.00 %	143.00 %	95.80 %
5	55	0.55	1	-2.98E+03	-1.20E+03	-3.09E+03	-2.29E+03	-2.29E+03	**104.00 %**	190.00 %	190.00 %
			2	-9.42E+03	-4.68E+03	-1.02E+04	-5.87E+03	-3.75E+03	108.00 %	125.00 %	80.20 %
			3	-2.98E+03	-1.20E+03	-1.87E+02	-1.85E+02	-1.85E+02	6.29 %	15.40 %	15.40 %
			4	-9.42E+03	-4.68E+03	-1.02E+04	-5.86E+03	-3.75E+03	109.00 %	125.00 %	80.10 %
6	55	0.75	1	-2.22E+03	-8.29E+02	-2.09E+03	-1.33E+03	-1.33E+03	**94.30 %**	160.00 %	160.00 %
			2	-5.98E+03	-2.96E+03	-5.93E+03	-3.63E+03	-2.20E+03	99.10 %	123.00 %	74.40 %
			3	-2.22E+03	-8.29E+02	2.32E+02	1.04E+01	1.04E+01	-10.50 %	-1.26 %	-1.26 %
			4	-5.98E+03	-2.96E+03	-6.04E+03	-3.61E+03	-2.19E+03	101.00 %	122.00 %	74.00 %
7	75	0.35	1	-5.29E+03	-2.33E+03	-5.08E+03	-4.29E+03	-4.29E+03	**96.10 %**	184.00 %	184.00 %
			2	-1.76E+04	-1.01E+04	-2.18E+04	-1.41E+04	-1.30E+04	124.00 %	139.00 %	129.00 %
			3	-5.29E+03	-2.33E+03	-2.64E+03	-2.13E+03	-2.13E+03	49.90 %	91.40 %	91.40 %
			4	-1.76E+04	-1.01E+04	-2.19E+04	-1.41E+04	-1.30E+04	124.00 %	139.00 %	129.00 %
8	75	0.55	1	-3.00E+03	-1.20E+03	-3.30E+03	-1.93E+03	-1.93E+03	**110.00 %**	160.00 %	160.00 %
			2	-1.22E+04	-6.78E+03	-1.34E+04	-8.37E+03	-7.66E+03	110.00 %	123.00 %	113.00 %
			3	-3.00E+03	-1.20E+03	-9.05E+02	-5.15E+02	-5.15E+02	30.20 %	42.90 %	42.90 %
			4	-1.22E+04	-6.78E+03	-1.35E+04	-8.37E+03	-7.65E+03	111.00 %	123.00 %	113.00 %
9	75	0.75	1	-2.24E+03	-8.29E+02	-2.34E+03	-1.23E+03	-1.23E+03	**104.00 %**	148.00 %	148.00 %
			2	-7.67E+03	-4.25E+03	-8.06E+03	-5.55E+03	-4.98E+03	105.00 %	131.00 %	117.00 %
			3	-2.24E+03	-8.29E+02	-3.89E+02	-2.66E+02	-2.66E+02	17.40 %	32.10 %	32.10 %
			4	-7.67E+03	-4.25E+03	-8.15E+03	-5.52E+03	-4.96E+03	106.00 %	130.00 %	117.00 %

4.3.3 Result discussion 1

Membrane vs Upper Side Stress

We choose to concentrate on the upper stress, as we can suppose it is on the surface of the weld there could be cracks. (This is argued for and performed in an older study [5]). The upper stress was checked vs the membrane stress for a few cases, and as expected they are much higher. As an example hotspot 2 stresses under axial loading was 80-120% of the SCF method, while membrane stress was as low as 5-30%. Similar results were found for different hotspots and load cases.

Extrapolation points

Results have been checked for two sets of interpolation points for both type of stresses. The first set is using the second and fourth layer of elements around the intersections. This corresponds to extrapolation points that are off with around up to 20% from DNV's proposed values. The second set is using the element layer 1 and 2. These distances are around 25% of the proposed values. Interestingly, results from both sets give very similar values.

	Geom.	Angle	Radius ratio	Hot-spot	Shell stress			sh. chord / b. chord	sh. brace / b. brace	sh. Brace (par) / b. beam
					chord (weld)	brace (weld)	brace (par)			
extrap. points 2 + 4	1	35	0.35	1	-3.95E+03	-3.77E+03	-3.77E+03	**75.60 %**	162.00 %	162.00 %
				2	-8.80E+03	-2.11E+03	-5.09E+03	114.00 %	60.50 %	146.00 %
				3	-5.12E+02	-3.94E+02	-3.94E+02	9.81 %	16.90 %	16.90 %
				4	-8.65E+03	-2.11E+03	-5.08E+03	112.00 %	60.50 %	146.00 %
extrap. points 1 + 2	1	35	0.35	1	-4.07E+03	-3.79E+03	-3.79E+03	**77.80 %**	163.00 %	163.00 %
				2	-8.89E+03	-2.10E+03	-5.12E+03	115.00 %	60.30 %	147.00 %
				3	-5.41E+02	-2.54E+02	-2.54E+02	10.30 %	10.90 %	10.90 %
				4	-8.75E+03	-2.10E+03	-5.12E+03	113.00 %	60.30 %	147.00 %
extrap. points 2 + 4	8	75	0.55	1	-3.30E+03	-1.93E+03	-1.93E+03	**110.00 %**	160.00 %	160.00 %
				2	-1.34E+04	-7.66E+03	-8.37E+03	110.00 %	113.00 %	123.00 %
				3	-9.05E+02	-5.15E+02	-5.15E+02	30.20 %	42.90 %	42.90 %
				4	-1.35E+04	-7.65E+03	-8.37E+03	111.00 %	113.00 %	123.00 %
extrap. points 1 + 2	8	75	0.55	1	-3.44E+03	-2.22E+03	-2.22E+03	**115.00 %**	184.00 %	184.00 %
				2	-1.39E+04	-8.00E+03	-8.82E+03	115.00 %	118.00 %	130.00 %
				3	-8.83E+02	-6.18E+02	-6.18E+02	29.40 %	51.40 %	51.40 %
				4	-1.41E+04	-8.00E+03	-8.81E+03	116.00 %	118.00 %	130.00 %

Stress difference* geom. 1		
chord (weld)	brace (weld)	brace (par)
-3 %	-1 %	-1 %
-1 %	0 %	-1 %
-6 %	36 %	36 %
-1 %	0 %	-1 %

Stress difference* geom. 8		
chord (weld)	brace (weld)	brace (par)
-4 %	-13 %	-13 %
-4 %	-4 %	-5 %
2 %	-17 %	-17 %
-4 %	-4 %	-5 %

Figure 4.8: *Stress differences are defined by (stress24-stress12)/stress12.

Reaction forces

The pipe joints presented in the example are only affected by the axial force in the left brace and the boundary conditions in the left end of the chord. This is one of the analysis used to calculate the F2S matrix. These load conditions would not be very realistic as the right part of the chord is "loose". In reality this end would be affected by certain forces.

The SCF formulae are intended to be used for joints in real structures subjected to realistic load conditions. To take this into account we will compare some of the results above with results arising from the same conditions but with in addition a half unit load in x- and z-direction at the chords right end. This would simulate some reaction forces at the end.

The SCF formulae for axial loading on one brace has two alternatives for the *crown* (hotspots 1 and 3) stresses on chord and brace. The first does not take into account chord stresses (eq. 4.4), while the second varies with the ratio of chord bending stress to brace axial stress (eq. 4.5).

$$\gamma^{0.2}\tau(2.65 + 5(\beta - 0.65)^2) + \tau\beta(C_2\alpha - 3)\sin(\theta) \tag{4.4}$$

$$\gamma^{0.2}\tau(2.65 + 5(\beta - 0.65)^2) - 3\tau\beta\sin(\theta) + \frac{\sigma_{BendingChord}}{\sigma_{AxialBrace}}SCF_{att} \tag{4.5}$$

The results presented earlier use the first alternative. When adding the forces on the chord we also switch to the second alternative. One of the effects caused by that is presented below:

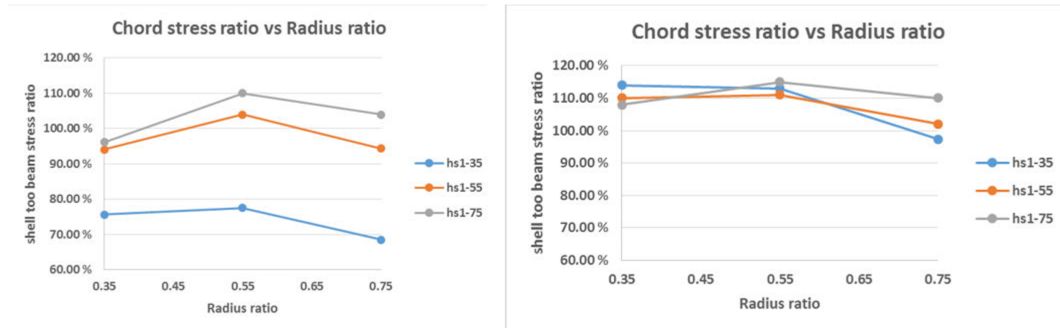


Figure 4.9: Left: Without chord forces. Right: With chord forces.

We see the variations in stress ratio due to the brace angle is considerably smaller when we include the chord forces. The same effect can also be shown for changes in brace radius.

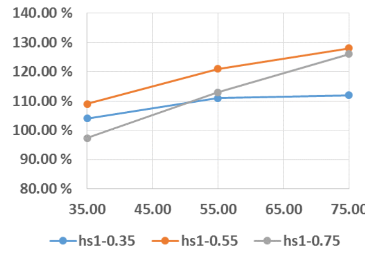
On the next pages stress ratio variations are analyzed. (When referred to *, the SCF method gives zero stress value.)

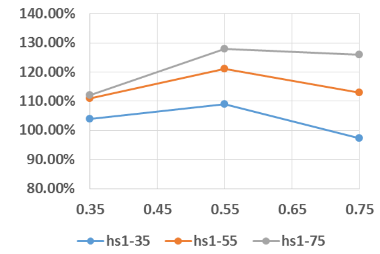
Stress ratio vs. angle | **Stress ratio vs. radius ratio**

HOTSPOT 1 CHORD

Stress range	: 97%-128%
Max/avg. var. ang.	: 29%/18%
Max/avg. var. rad.r.	: 16%/12%

Relatively good correlation
Increasing with higher angle

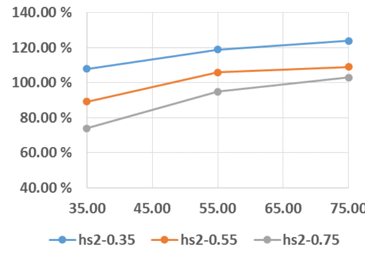


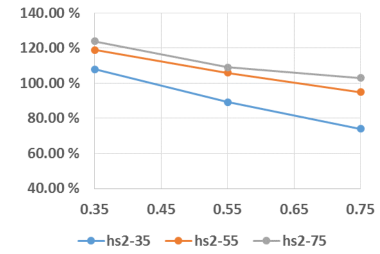


HOTSPOT 2 CHORD

Stress range	: 74%-124%
Max/avg. var. ang.	: 29%/22%
Max/avg. var. rad.r.	: 34%/26%

Relatively good correlation
Decreasing with higher brace radius
Increasing with higher angle

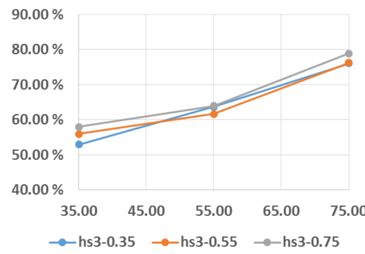


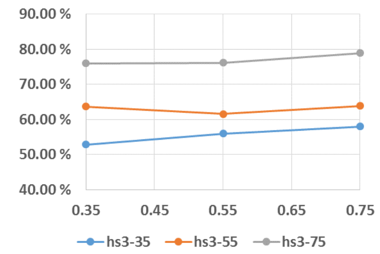


HOTSPOT 3 CHORD

Stress range	: 53%-79%
Max/avg. var. ang.	: 23%/21%
Max/avg. var. rad.r.	: 5%/3%

Slightly low stresses
Increasing with higher angle

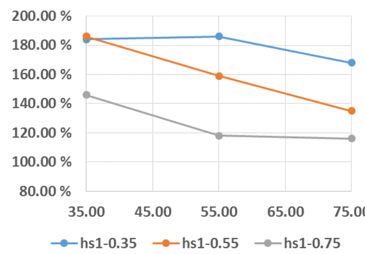


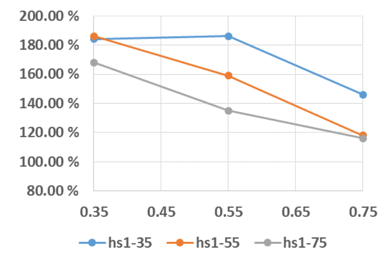


HOTSPOT 1 BRACE

Stress range	: 116%-186%
Max/avg. var. ang.	: 51%/33%
Max/avg. var. rad.r.	: 68%/53%

High stresses
Decreasing with higher angle
Decreasing with higher brace radius

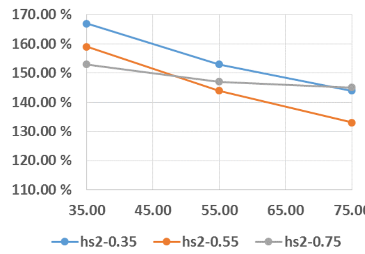


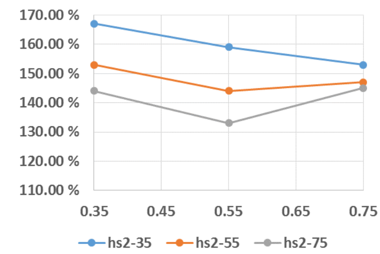


HOTSPOT 2 BRACE

Stress range	: 133%-167%
Max/avg. var. ang.	: 26%/19%
Max/avg. var. rad.r.	: 14%/11%

High stresses
Decreasing with higher angle

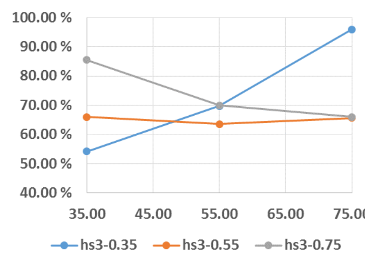


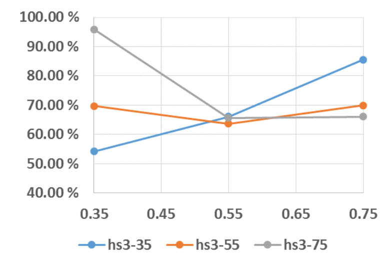


HOTSPOT 3 BRACE

Stress range	: 54%-96%
Max/avg. var. ang.	: 42%/21%
Max/avg. var. rad.r.	: 31%/23%

No consistent variation



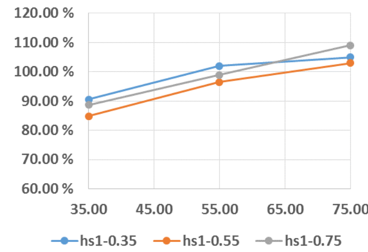


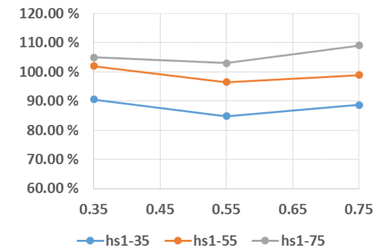
Stress ratio vs. angle | Stress ratio vs. radius ratio

HOTSPOT 1 CHORD

Stress range	: 85%-109%
Max/avg. var. ang.	: 20%/17%
Max/avg. var. rad.r.	: 6%/6%

Good correlation

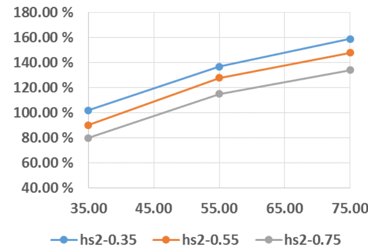


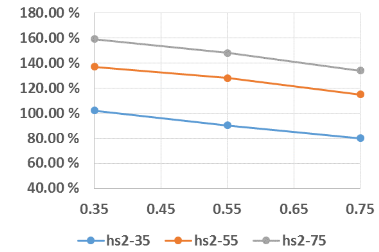


HOTSPOT 2 CHORD

Stress range	: 80%-159%
Max/avg. var. ang.	: 58%/56%
Max/avg. var. rad.r.	: 25%/23%

Increasing with higher angle

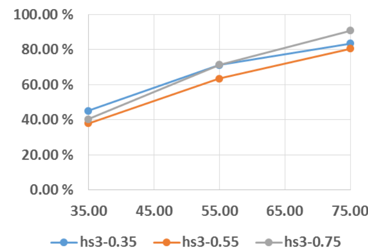


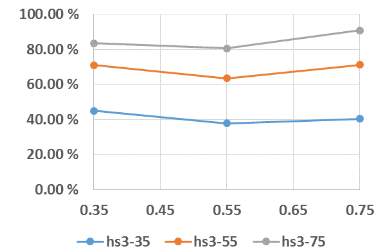


HOTSPOT 3 CHORD

Stress range	: 38%-91%
Max/avg. var. ang.	: 50%/44%
Max/avg. var. rad.r.	: 10%/8%

Low stress for small angle
Increasing with higher angle

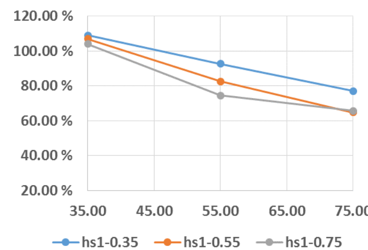


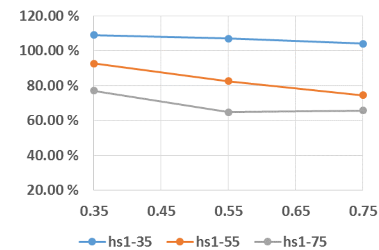


HOTSPOT 1 BRACE

Stress range	: 65%-109%
Max/avg. var. ang.	: 42%/38%
Max/avg. var. rad.r.	: 18%/12%

Relatively good correlation
Decreasing with higher angle

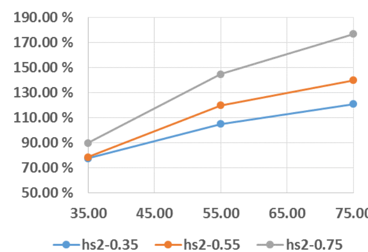


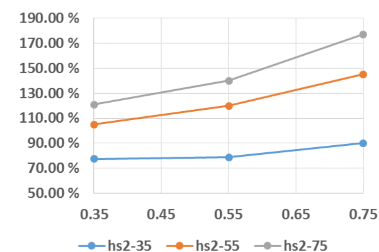


HOTSPOT 2 BRACE

Stress range	: 77%-177%
Max/avg. var. ang.	: 87%/64%
Max/avg. var. rad.r.	: 56%/36%

High variation
Increasing with higher brace radius
Increasing with higher angle

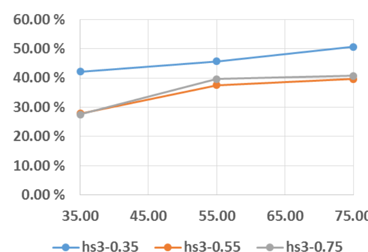


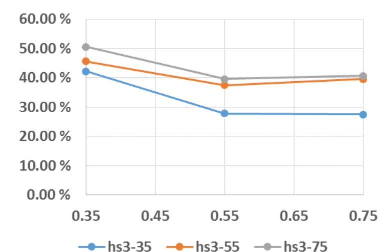


HOTSPOT 3 BRACE

Stress range	: 27%-51%
Max/avg. var. ang.	: 13%/11%
Max/avg. var. rad.r.	: 15%/11%

Low stresses



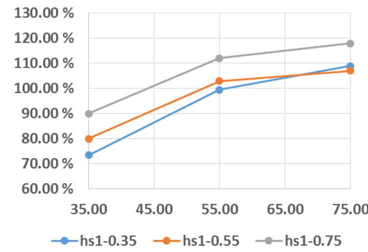


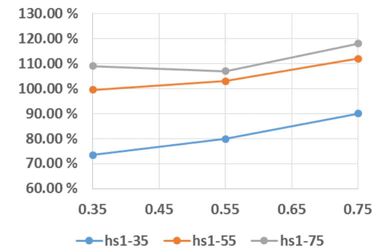
Stress ratio vs. angle | Stress ratio vs. radius ratio

HOTSPOT 1 CHORD

Stress range	: 73%-118%
Max/avg. var. ang.	: 36%/30%
Max/avg. var. rad.r.	: 17%/13%

Relatively good correlation
Increasing with higher angle

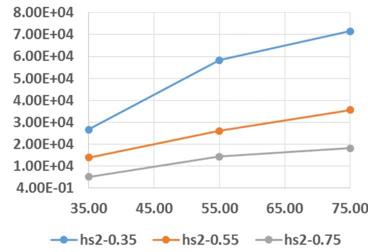


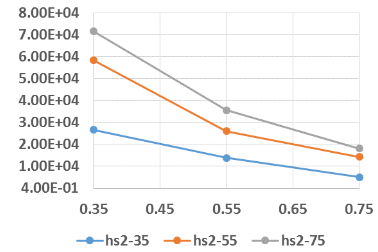


HOTSPOT 2 CHORD

Stress range	: N/A
Max/avg. var. ang.	: N/A
Max/avg. var. rad.r.	: N/A

*

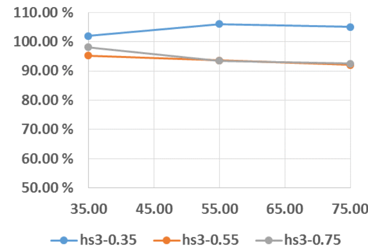


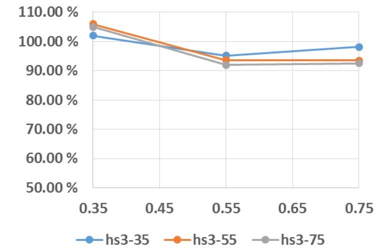


HOTSPOT 3 CHORD

Stress range	: 92%-106%
Max/avg. var. ang.	: 6%/4%
Max/avg. var. rad.r.	: 13%/11%

Good correlation

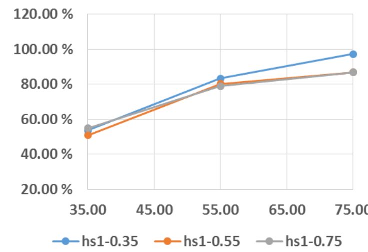


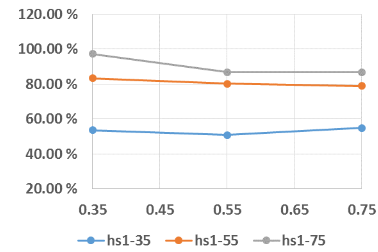


HOTSPOT 1 BRACE

Stress range	: 51%-97%
Max/avg. var. ang.	: 44%/37%
Max/avg. var. rad.r.	: 11%/6%

Low stress for small angle
Increasing with higher angle

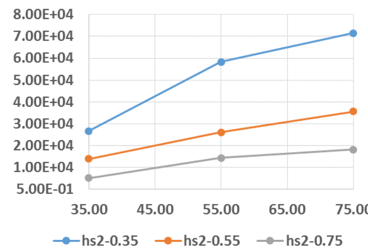


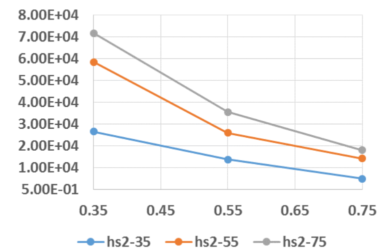


HOTSPOT 2 BRACE

Stress range	: N/A
Max/avg. var. ang.	: N/A
Max/avg. var. rad.r.	: N/A

*

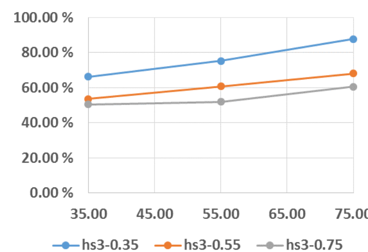


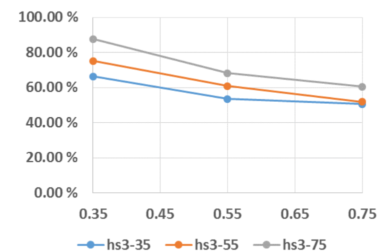


HOTSPOT 3 BRACE

Stress range	: 50%-88%
Max/avg. var. ang.	: 21%/15%
Max/avg. var. rad.r.	: 27%/22%

Slightly low stress



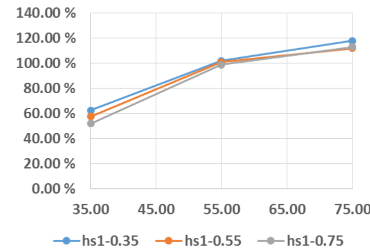


HOTSPOT 1 CHORD

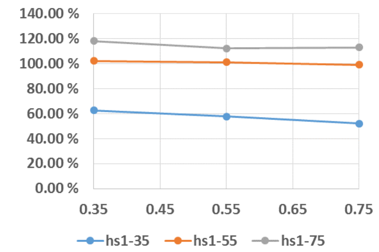
Stress range : 52%-118%
Max/avg. var. ang. : 61%/57%
Max/avg. var. rad.r. : 10%/6%

Low stress for small angle
Increasing with higher angle

Stress ratio vs. angle



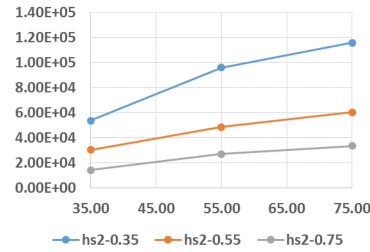
Stress ratio vs. radius ratio

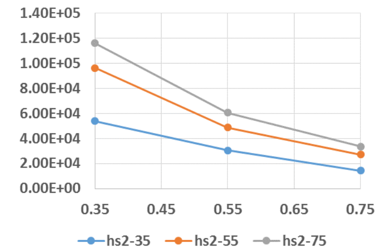


HOTSPOT 2 CHORD

Stress range : N/A
Max/avg. var. ang. : N/A
Max/avg. var. rad.r. : N/A

*

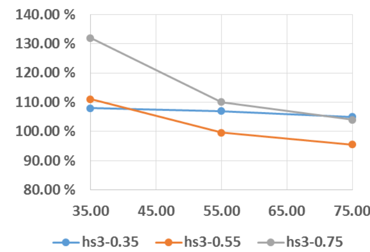


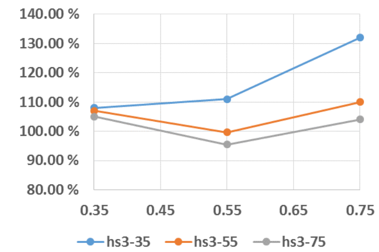


HOTSPOT 3 CHORD

Stress range : 95%-132%
Max/avg. var. ang. : 28%/16%
Max/avg. var. rad.r. : 24%/15%

Relatively good correlation

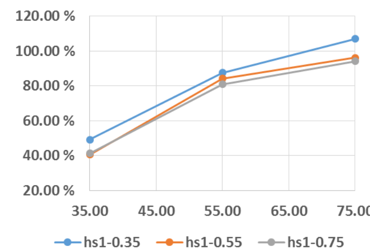


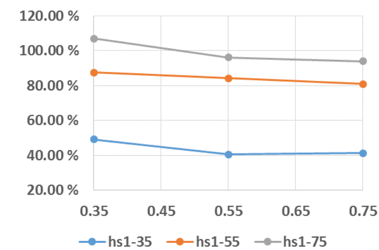


HOTSPOT 1 BRACE

Stress range : 41%-107%
Max/avg. var. ang. : 57%/55%
Max/avg. var. rad.r. : 13%/9%

Low stresses for small angle
Increasing with higher angle

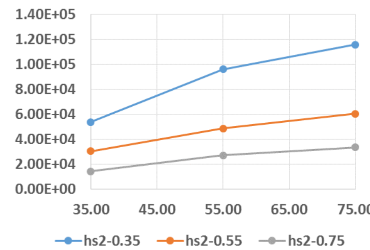


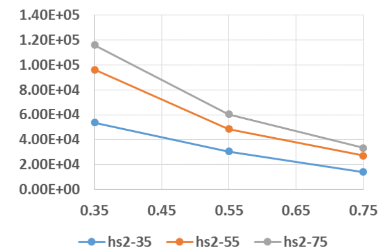


HOTSPOT 2 BRACE

Stress range : N/A
Max/avg. var. ang. : N/A
Max/avg. var. rad.r. : N/A

*

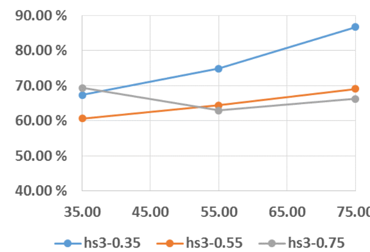


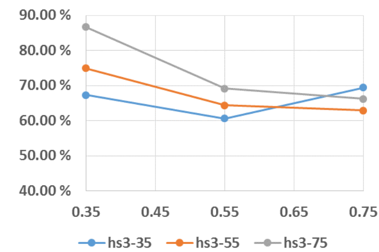


HOTSPOT 3 BRACE

Stress range : 61%-87%
Max/avg. var. ang. : 19%/11%
Max/avg. var. rad.r. : 21%/14%

Slightly low stresses





LOAD CASE 5/6 - MOMENT OUT OF PLANE ON ONE BRACE / TWO BRACES

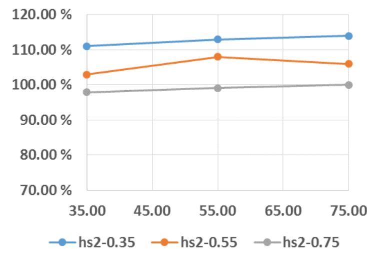
MOP ONE BRACE

HOTSPOT 2 CHORD

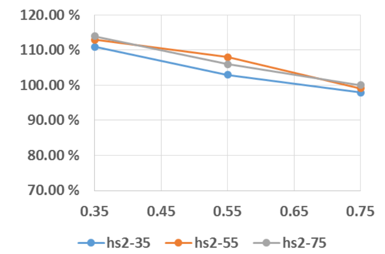
Stress range : 98%-114%
Max/avg. var. ang. : 5%/3%
Max/avg. var. rad.r. : 14%/13%

Good correlation
Increasing with higher angle

Stress ratio vs. angle



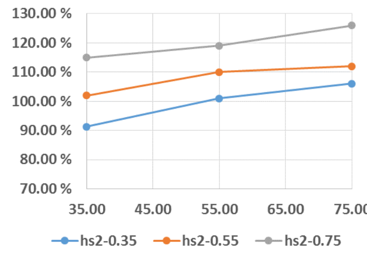
Stress ratio vs. radius ratio

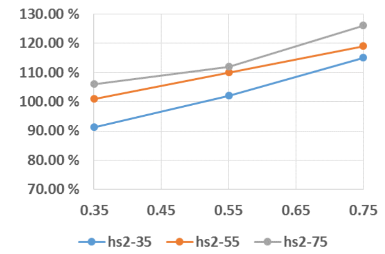


HOTSPOT 2 BRACE

Stress range : 91%-126%
Max/avg. var. ang. : 15%/12%
Max/avg. var. rad.r. : 24%/21

Good correlation
Increasing with higher brace radius



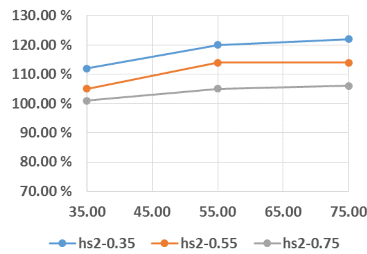


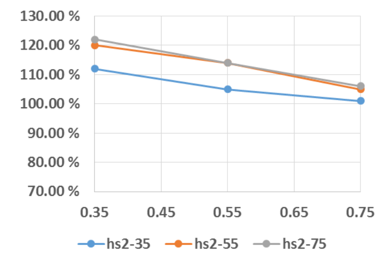
MOP TWO BRACES

HOTSPOT 2 CHORD

Stress range : 101%-122%
Max/avg. var. ang. : 11%/8%
Max/avg. var. rad.r. : 16%/14%

Good correlation

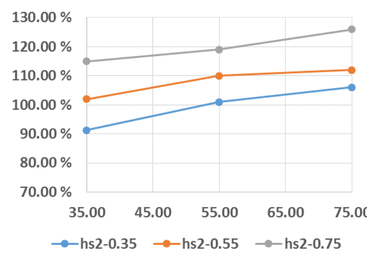


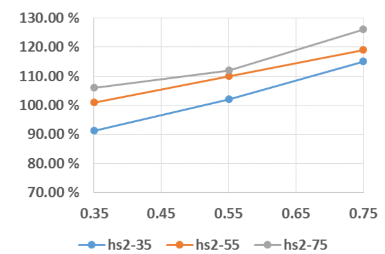


HOTSPOT 2 BRACE

Stress range : 92%-130%
Max/avg. var. ang. : 20%/16%
Max/avg. var. rad.r. : 25%/20%

Good correlation





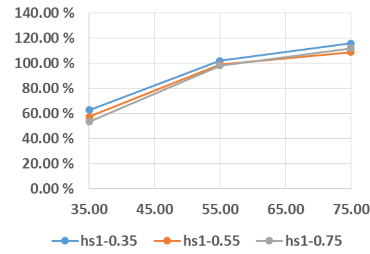
LOAD CASE 7 - MULTIPLE FORCES

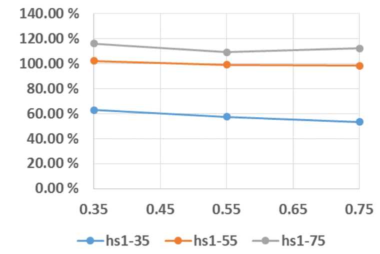
Stress ratio vs. angle | **Stress ratio vs. radius ratio**

HOTSPOT 1 CHORD

Stress range : 53%-116%
Max/avg. var. ang. : 59%/55%
Max/avg. var. rad.r. : 9%/7%

Relatively good correlation
Slightly low stresses for small angle

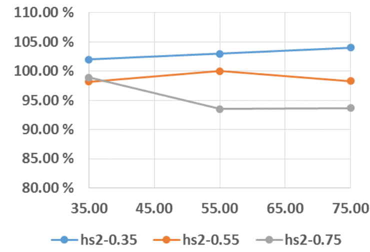


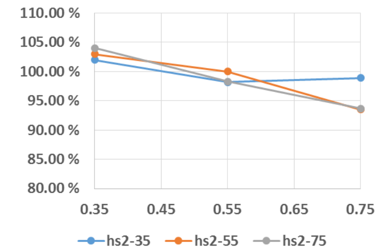


HOTSPOT 2 CHORD

Stress range : 94%-104%
Max/avg. var. ang. : 5%/3%
Max/avg. var. rad.r. : 10%/8%

Good correlation

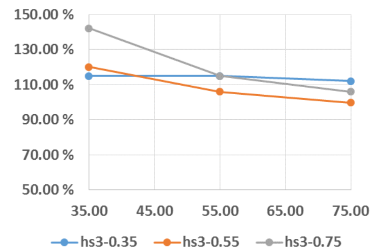


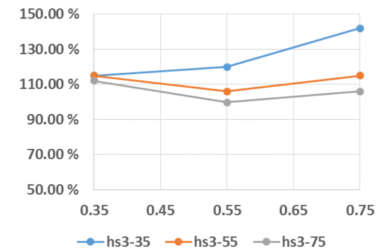


HOTSPOT 3 CHORD

Stress range : 100%-142%
Max/avg. var. ang. : 36%/20%
Max/avg. var. rad.r. : 27%/16%

Slightly high stresses

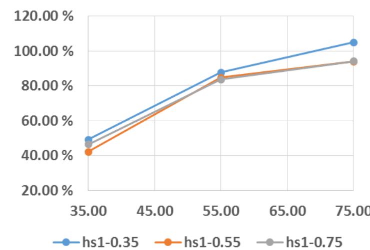


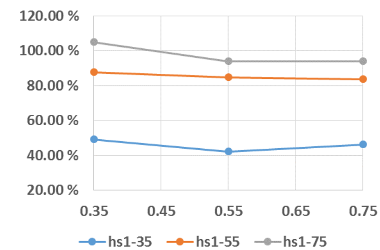


HOTSPOT 1 BRACE

Stress range : 42%-105%
Max/avg. var. ang. : 56%/52%
Max/avg. var. rad.r. : 11%/7%

Low stresses for small angle

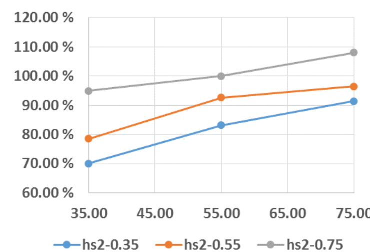


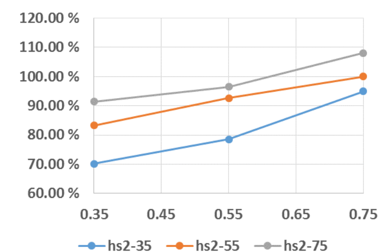


HOTSPOT 2 BRACE

Stress range : 70%-108%
Max/avg. var. ang. : 21%/18%
Max/avg. var. rad.r. : 25%/20%

Relatively good correlation
Increasing with higher brace radius
Increasing with higher angle

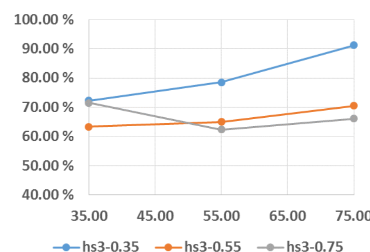


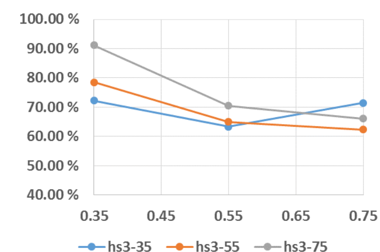


HOTSPOT 3 BRACE

Stress range : 62%-91%
Max/avg. var. ang. : 19%/12%
Max/avg. var. rad.r. : 25%/17%

Slightly low stresses





4.3.4 Result discussion 2

As seen on the previous pages, every hotspot has been analysed individually. An overall conclusion is not obvious. Some results are higher than the SCF method, and some are lower, but an overall average would show that the shell analysis gives slightly less stress.

What is observed the most often is that the shell analysis usually gives higher stresses with higher angles than the SCF method predicts. This could hint to that the way the brace angle affects the SCF formulae could be revised. The angle affects most formulas with the factor $\sin(\theta)^n$ where n is varying for each formula. Though, before considering any revision, a much broader analysis should be done. In our case we have limited us to only two geometrical changes; the brace to chord radius ratio and the brace angle. Other changes could be the brace to chord thickness ratio and the gap distance between the braces. More variations of brace radius and brace angle should also be applied and a more extensive analysis of the effect of extrapolation distances would also be required.

4.3.5 Forces not considered by the SCF method

When using the SCF method the six force and moment components in the chord are not considered. One exception, as we saw in section 4.3.3, is for the crown hotspots' SCF under axial load on one brace, where the SCF accounts for the bending stress in the chord.

Without going to deeply in this matter, we will look at how chord forces affect the hotspot stresses in a shell element analysis. We already have all the F2S matrices for the different geometries, and from these we can see how much unit loads in the chord affect the different hotspots.

To compare the values we take the stress value under axial loading on one brace for hotspot 1. This value varies between *-3.6e3* and *-7.27e3*. For the sake of simplicity we average of these values to get *C=-5e3*.

Results in % for geometry 5 (angle=55, radius ratio=0.55)

	Unit load in:					
	Fx	Fy	Fz	Mx	My	Mz
Chord hotspot	$hotspotstress/C$					
1	3.70	-5.37	157.85	-1.17	155.35	5.07
2	0.13	5.19	-14.52	20.15	-14.10	-5.76
3	5.00	3.04	189.12	3.46	165.55	-3.64
4	-0.21	-2.39	-15.58	-35.11	-13.64	5.05
Brace hotspot	$hotspotstress/C$					
1	0.22	-1.69	23.63	0.91	25.39	1.26
2	3.53	-26.64	67.21	6.66	60.34	24.98
3	1.85	0.41	66.21	-3.69	58.91	0.05
4	3.27	26.58	67.04	-6.75	60.18	-25.11

The results were checked for all the 9 geometries, and agree very well with the results above. The first thing to note is the stress caused by Fz and My. We see that hotspot 1 and 3 on the chord gets highly affected. This agrees very well with having a corrected SCF formula for axial loading on one brace.

In section 3.3.3, regarding the calculation of the FTS matrix, we mentioned a subtraction of the moment caused by shear force. For the unit loads in this chapter we neglect this contribution as we do not use shear force and its corresponding moment force together. This is why on the previous page the results from Fy and Mz, and Fz and My are very similar.

Other values are much smaller, but not necessarily neglectable. There are two other factors not yet considered. Firstly, we compared the stresses with a stress arising from axial loading, while a unit moment in the brace can give stresses 10-50 times higher. This reduces highly the effect chord forces have on the hotspot stresses. Secondly, and most importantly, we have only looked at unit loads in every case. The pipe junctions are in reality part of bigger structures subjected to wave loads. The loads acting on the brace may all have different magnitude depending on where the junction is located in the structure. In the next chapter we will look more into how different the forces are from each other in a real scenario.

Chapter 5

Comparison of fatigue damage

5.1 Structure

The structure chosen for this analysis is the jacket from an offshore windmill. It is attached in the seabed and subjected to waves.

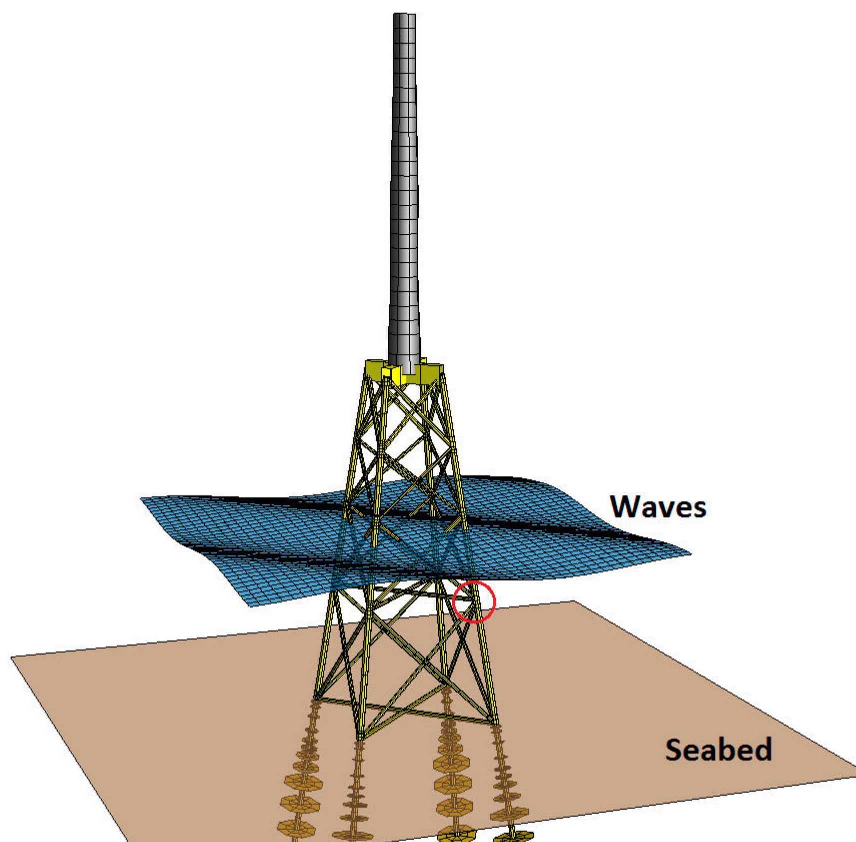


Figure 5.1: The joint we analyze is circled in red.

Taking a closer look at the first joint we are analyzing, we see there are 5 braces. As mentioned earlier we look at braces in the same plane. Our "subjoint" is a K-joint like seen in figure 5.2.

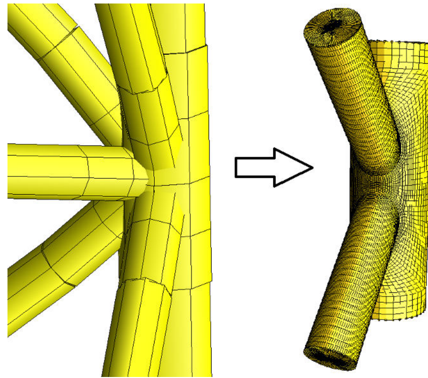
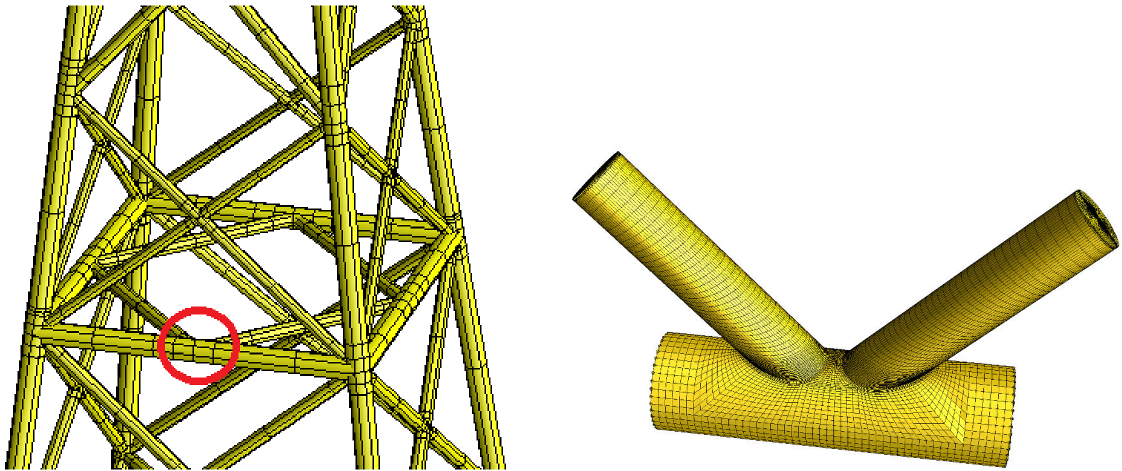


Figure 5.2: Beam model of full joint to the left. Shell model of "subjoint" to the right.

For a second analysis, a modification of the structure was done by adding a few horizontal joints as seen in figure 5.3a.



(a) Beams were added to create new joints. (b) Mesh of new joint.

Figure 5.3: Joint in the horizontal plane.

5.2 Results

With the FTS matrix from *beam2shell* and the F2S matrix from *read_stress*, we use *dmg_calc* presented in section 3.7 to calculate the damage from the shell method.

With the *results.dyn* file from the beam structure USFOS analysis and *FATAL* we get the damage calculated from the SCF method. The SCF values which can vary depending on load on one brace only or balanced load on both, are not varying through the analysis. The values need to be specified as constant throughout the analysis. As seen in the results, the comparison is done for both sets of SCF values. Where SCF1 and SCF2 corresponds to the SCF values from loading on one brace and two braces respectively.

Joint analysis 1:

Hotspot	**Damage [·1e8]**			**Ratio**	
chord	**Shell**	**SCF1**	**SCF2**	**Shell/SCF1**	**Shell/SCF2**
hs1:	0.4477	0.7009	0	63.88%	N/A
hs2:	0.6393	10.0340	23.8350	6.37%	2.68%
hs3:	2.2060	3.0851	1.6912	71.50%	130.44%
hs4:	0.6295	2.3451	6.0373	26.84%	10.43%
Total:	3.9225	16.1651	31.5635	**24.27%**	**12.43%**
brace					
hs1:	0.0758	0.3083	0	24.59%	N/A
hs2:	2.7593	11.3150	21.9980	24.39%	12.54%
hs3:	1.2145	4.7268	2.8794	25.70%	42.18%
hs4:	2.1639	1.9785	30.7231	109.37%	37.02%
Total:	6.2135	18.3286	30.7231	**33.90%**	**20.22%**

Joint analysis 2:

Hotspot	**Damage [·1e8]**			**Ratio**	
chord	**Shell**	**SCF1**	**SCF2**	**Shell/SCF1**	**Shell/SCF2**
hs1:	69.13	78.41	10.85	88.17%	637.11%
hs2:	832.99	351.57	882.08	236.93%	94.44%
hs3:	75.19	142.84	30.48	52.64%	246.66%
hs4:	876.58	639.27	1649.00	137.12%	53.16%
Total:	1853.89	1212.09	2572.41	**152.95%**	**72.07%**
brace					
hs1:	14.42	0	0	N/A	N/A
hs2:	185.98	87.67	181.37	212.14%	102.54%
hs3:	7.91	17.63	0.27	44.88%	2964.50%
hs4:	222.25	112.96	344.62	196.75%	64.491%
Total:	430.56	218.25	526.26	**197.27%**	**81.81%**

FATAL disregard damages smaller than a certain value, which is why we see some of the values as zero in the tables above. The waves used in the analysis of the second joint were much higher and frequent than in the first. This is why we see a big difference in damage.

5.3 Result discussion

What is explained in this section is not meant as a method for understanding the exact damage differences, but rather understand from *where* the differences of the two methods lies, and how they *could* be quantified. (All the numbers in this section are taken from the first analysis.)

As we have seen in the description of the Rainflow algorithm, the fatigue damage is based on the stress amplitude, and is independent of the mean stress. We also know that the two methods are proportional to the force applied. For example, consider axial force on one brace, the resulting stress on a hotspot is:

$$\sigma_{shell,i} = F2S(i)F = aF \tag{5.1}$$

$$\sigma_{SCF,i} = \sigma_x SCF_i = \frac{SCF_i}{A}F = bF \tag{5.2}$$

Where a and b are constant for the analysis. We then define:

$$p = \frac{\sigma_{shell,i}}{\sigma_{SCF,i}} = constant \tag{5.3}$$

The same can be shown for forces in other directions.
If we then have one stress cycle between σ_{SCF1} and σ_{SCF2} from the SCF method, we can define the respective stresses from the shell element method as $p\sigma_{SCF1}$ and $p\sigma_{SCF2}$, where p is the ratio between both stresses.

The amplitudes are then (figure 5.4):

$$\sigma_{ampSCF} = \frac{\sigma_{SCF1} - \sigma_{SCF2}}{2} \tag{5.4}$$

$$\sigma_{ampshell} = p\frac{\sigma_{SCF1} - \sigma_{SCF2}}{2} \tag{5.5}$$

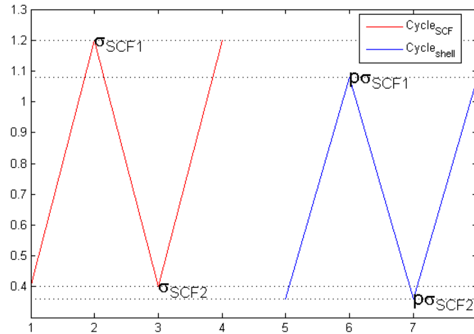


Figure 5.4: Visualization of amplitude difference.

We understand that the amplitudes from the two methods are related with the same proportions as the stresses.

The rainflow algorithm calculates the damage from the STS matrix, where the stresses from different forces are added together. So an actual amplitude will look like:

$$\sigma_{ampshell} = \frac{1}{2}(p_1 \Delta\sigma_{1SCF} + p_2 \Delta\sigma_{2SCF} + p3 \Delta\sigma_{3SCF}) + \delta\sigma_{ampshell} \tag{5.6}$$

$$\delta\sigma_{ampshell} = \frac{1}{2}(\Delta\sigma_4 + \ldots \Delta\sigma_i \ldots + \Delta\sigma_{18}) \tag{5.7}$$

Where $i \in [1, 2, 3]$ correspond to Fx, My and Mz on the brace, accounted for in the SCF method, and $i \in [4, \ldots, 18]$ to the other forces accounted for only in the shell element method. For now we neglect $\delta\sigma$.

In the definition of cycle damage in equation 3.13 we see the amplitude is raised to the m'th. m will generally be in the range 3-5, which means that a small difference in stress will lead to large change in damage. With the junctions F2S matrix we can check how much these differences are.

		shell / SCF (p)	
		chord	**brace**
Axial	1	111.00%	151.00%
	2	68.50%	51.00%
	3	11.70%	41.40%
	4	69.20%	51.00%
Mip	1	76.90%	76.90%
	3	110.00%	83.00%
Mop	2	86.50%	90.00%
	4	86.70%	90.00%

Averaging all the stress ratios we get approximately 75%. With a m value of 3 this means a reduction of damage of almost 60%. This is a very rough estimate. Depending on the loading case, the amplitudes of axial force might be much higher than the momentum amplitudes, or vise-versa. Averaging the p values will therefore not be the best estimate, but it will give an idea of the magnitude.

We can define $\Delta\sigma_{SCF}$ to be a constant stress amplitude for a hotspot from a force F. We then define the other forces to give a stress amplitude with ratios q_i to $\Delta\sigma_{SCF}$. The damage can then be expressed as:

$$D_{shell} = \frac{1}{2^m \bar{a}}(q_1 p_1 + q_2 p_2 + q_3 p_3)^m \Delta\sigma_{SCF}^m + \delta D \tag{5.8}$$

$$D_{SCF} = \frac{1}{2^m \bar{a}}(q_1 + q_2 + q_3)^m \Delta\sigma_{SCF}^m \tag{5.9}$$

The idea is just to show that a stress ratio (p_i) deviating from *1* affects the total damage more if it corresponds to a force generating high stress amplitudes (high q_i)

As an example, the graphs below show the stress arising from Fx, Fz and My on the joint. We must remember (from 3.3.3 on the FTS calculation) that Fz give shear and momentum forces. (Fy and My are left out as they give much smaller amplitudes.)

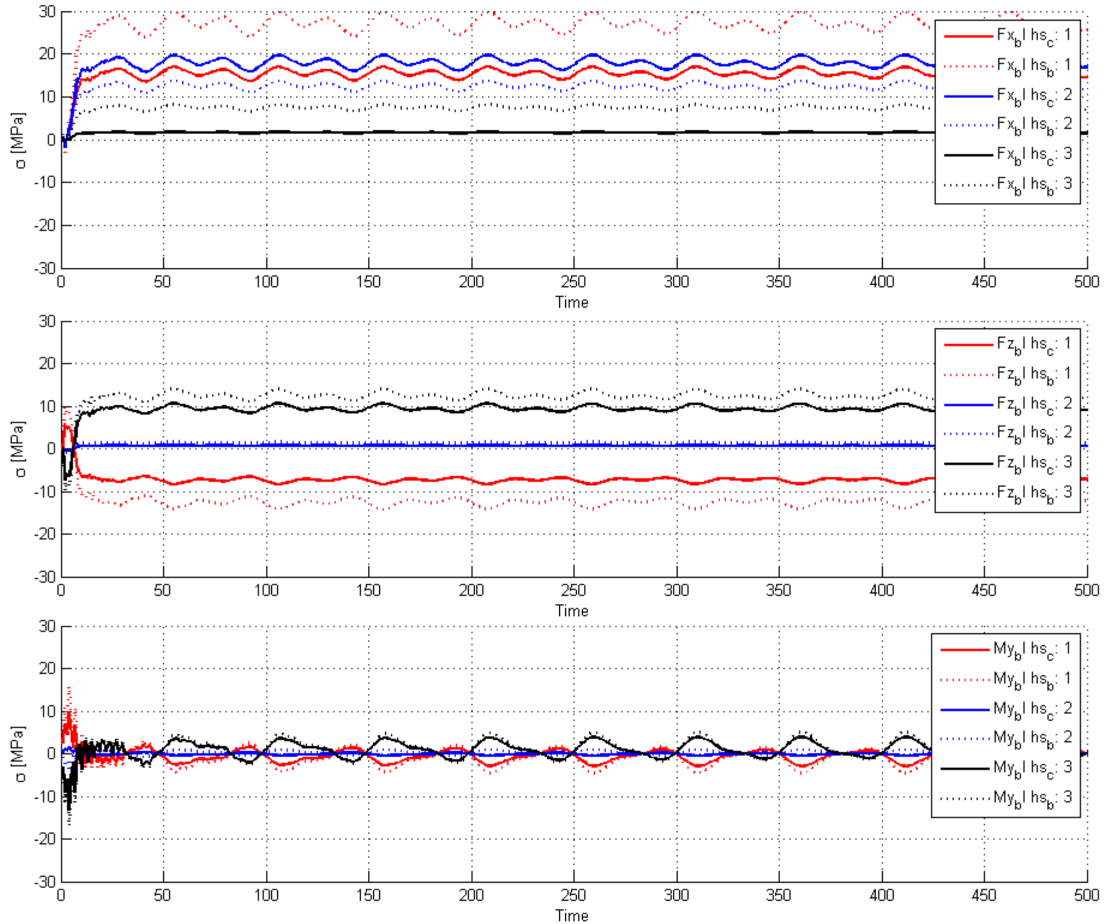


Figure 5.5: Stress on hotspots from specific forces (1)

From this data a few points can be concluded:

- *Fx* gives relatively large amplitudes on brace hotspot 1, thus large q for a high p
- *Fx* gives small amplitudes on hotspot 3, thus low q for a low p.
- Fz and My results in stress amplitudes on hotspot 1 which seems to be near opposite of amplitudes from axial force, thus an opposite sign on q.

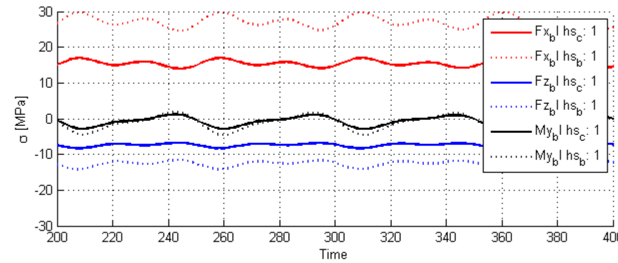


Figure 5.6: Hotspot 1 stress amplitudes seem to *cancel* each other.

To find out what lies in $\delta\sigma_{amplitude}$ and δD we need to look at the forces not accounted for by the SCF method. Checking the effect of these forces on the different hotspots, we can see that it is only the axial force on the chord that gives a non-neglectable amplitude. (This can be understood by the fact that the chord is a vertical beam supporting the structure).

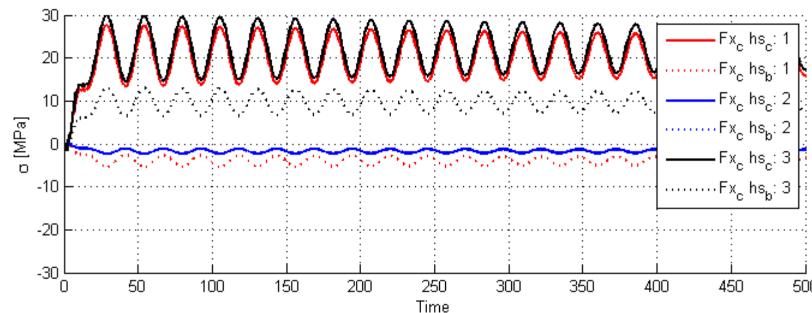
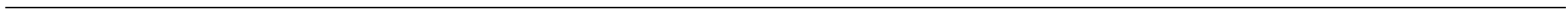


Figure 5.7: Stress amplitude caused by axial force on chord.

Considering this, we should suspect the large amplitudes on the chords hotspots 1 and 3 to give a bigger damage than the SCF method would predict. But as seen in the results this is not the case. The shell method still gives only around 60-70% of the SCF method damage. This could for example be due to other forces causing amplitudes canceling some of the amplitudes we have looked at. But the same hotspot numbers on the brace have considerably lower damage ratio (25%) and much lower amplitudes caused by axial force on the chord.

As for the second joint, we can see some very high ratios, up to 637% for hotspot 1 on the chord. Although they are high, the actual damage value is low compared to the other hotspots (one order of magnitude) so it doesn't affect the total damage ratio of the joint.



Chapter 6

Conclusion and future work

As we have seen in Chapter 4, there are many factors affecting how the shell element method compares to the SCF method. From our results the hotspot stresses are on *average* very near the SCF method stresses. Still, we see some large variations for specific hotspots.

We tried to see how the increase in brace angle and brace radius affected the difference between the two methods. It seemed that in general, an increase in angle leads to a larger difference between the methods while the change in radius had in most cases a smaller effect.

We have also seen how the chord forces not accounted for in the SCF method, affect the results. The in-plane moment can give large contribution to certain hotspots, while the other forces had less effect.

The extrapolation distance didn't highly change the stress results, but small changes in stress could mean large changes in damage. A more extensive research on this is recommended.

As for the accumulated damage in Chapter 5, we only tested two different joints, but observed in general a smaller total damage with the shell element method. What is important to note here, is that the damage is highly sensitive to the stresses, which again are depending on the type of loads acting. The forces acting on the joint from the structure can vary importantly depending on the loading scenario and structure. The point being that it is difficult to pin down a conclusion for *how much* difference there is between the two methods. There are too many variables in place, and a more in depth analysis should be performed to come to a more general conclusion.

Throughout this report, the methods used could be applied to create a more automatic process. This way more variables could be tested more efficiently and we might find a more satisfying conclusion.

Suggested future work

- The meshing module used here creates the mesh automatically, but still needs the user to intervene to perform final refinements through the *settings file*. Creating a more efficient meshing program would be complicated, but would enable the possibility to analyze a larger amount of different geometries without refining every mesh.
- The meshing module should also be able to create elements at specific extrapolation distances around the brace. We could then follow *DNV*'s requirements more correctly.
- The method that was used to extract the shell element stresses from USFOS was quite tedious. It should be possible to create a script in *Unix shell* that could perform this task much faster.
- Multiple scripts were used here, and the process was applied with many steps. We might want to combine all these elements to one program that does everything from reading an USFOS model and creating a mesh in order to analyse and compare the stress and damage differences.
- Having done these changes, we could have two different modes on the program. *One* performing multiple analyses on a set of predefined geometries on a larger scale than the one we did in Chapter 4, and adding the variables tube thickness and brace gap. Thereafter automatically compare the results with the SCF method. A *second* mode to read a set of joints from an USFOS structure model, mesh and analyse them and automatically go through the steps required to compare the damage.

References

[1] Tom M. Apostol and Mamikon A. Mnatsakanian. “Unwrapping Curves from Cylinders and Cones”. In: *The Mathematical Association of America* 114 (2007), pp. 388–392.

[2] USFOS Reality Engineering. 2015. URL: www.usfos.no.

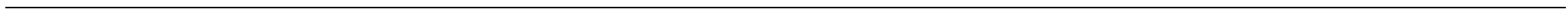
[3] Tore Holmas. *FAT31 Theory*. 2009.

[4] Don Koks. *Explorations in Mathematical Physics*. 2006. Chap. 4, p. 147.

[5] M.R. Morgan and M.M.K. Lee. “Stress Concentration Factors in Tubular K-Joints under In Plane Moment Loading”. In: *Journal of Structural Engineering* (1998), pp. 382–390.

[6] USFOS. *USFOS User’s Manual*. 2014. Chap. 6.

[7] Det Norske Veritas. *DNV-RP-C203 - Fatigue Design of Offshore Structures*. 2012.



Appendix A - Meshing module code

Below is the code of the *beam2shell*'s meshing module written in the C++ language. Only the part which calculates the coordinates is included. The remaining code which defines each element by each node numbers is left out.

```
1
2   #include "stdafx.h"
3   #include <iostream>
4   #include <cmath>
5   #include <Eigen/Dense>
6   #include <fstream>
7   #include <string>
8   #include <iomanip>
9   #include "calc_functions.h"
10
11  using namespace std;
12  using namespace Eigen;
13
14
15  VectorXd non_linspace();                        //functions included
16  VectorXd center_spaced();
17  pair<VectorXd, VectorXd> deflection_func();
18  MatrixXd rot_transl_elps();
19  MatrixXd rot_transl_defl_elps();
20  MatrixXd rot_transl_defl_int();
21
22
23
24
25  VectorXd calc(double x1,double x3,double phi,double Rm, double Rb,double l,
26  double thick_m,double thick_bl, double thick_br, MatrixXd mat_prop)
27  {
28
29          //LOAD SETTING VALUES:::::::::::::::::::::
30          int layers, abn, el, nr, nl,n;
31          float drf, bbf, spcr_p, ref_b, rad1, rad2, gap, nb1_fac, nb2_fac, nb3_fac, dpos_inc, dpos_inc2, p;
32          string skip;
33
34          ifstream settings;
35          settings.open("settings.txt");                  //loading parameters from settings file
36          settings >> skip >> n >> skip >> layers >> skip >> drf >> skip >> bbf
37                  >> skip >> abn >> skip >> rad1 >> skip >> rad2 >> skip >> el
38                  >> skip >> spcr_p >> skip >> ref_b >> skip >> nr >> skip >> nl
39                  >> skip >> gap >> skip >> nb1_fac >> skip >> nb2_fac >> skip >> nb3_fac
40                  >> skip >> dpos_inc >> skip >> dpos_inc2 >> skip >> p;
41
42
43          settings.close();
44
45          //some of the code is prepared for enabling the possibility of having different
46          //angle and radius on the braces.
47
48          //INPUT
49          float x2, x4, x5, Rb1, Rb2, z4, z5;
50          float g0_1, g0_2, g1, g2, dg, phi1, phi2, es;
51
52          x2 = 0;                                         //x-coordinate of center node (junction center)
53          Rb1 = Rb;   Rb2 = Rb;
54          if (!n % 8 == 0){ n = n - n % 8; }
55
56          es      = 2 * M_PI*Rb1 / n;                     //approximate element size
57          phi1    = -phi; phi2 = phi;                     //left brace has "negative" angle
58
59          phi1    = -(M_PI / 2 - abs(phi1));              //angle is defined differently in intersection function
60          phi2    = (M_PI / 2 - abs(phi2));
61
62
63          z4 =  cos(abs(phi1))*(1 + Rm / cos(abs(phi1))); //coordinates of left brace end
64          x4 = -sin(abs(phi1))*(1 + Rm / cos(abs(phi1)));
65
66          z5 = cos(phi2)*(1 + Rm / cos(phi2));            //coordinates of right brace end
67          x5 = sin(phi2)*(1 + Rm / cos(phi2));
68
69          if (gap == 0) {                                 //calculating gap if no gap in setting file
70                  g1 = Rm*tan(abs(phi1)) - Rb1/cos(abs(phi1));
71                  g2 = Rm*tan(phi2) - Rb2/cos(phi2);
72          }
73          else {
74                  g1 = gap / 2;
75                  g2 = gap / 2;
76          }
77
78          //Printing out info to console
79          cout << "Original gap between intersections: " << g0_1 + g0_2 << endl;
80          cout << "New gap between intersections     : " << g1 + g2 << endl;
81          cout << "Branch angle                      : " << phi * 180 / M_PI << endl;
82          cout << "Braces lengths                    : " << l << endl;
83          cout << "Left chord length                 : " << -x1 << endl;
84          cout << "Right chord length                : " << x3 << endl;
```

```
85          cout << "Main pipe radius                   : " << Rm << endl;
86          cout << "Branch pipe radius                 : " << Rb << endl;
87          cout << "Approximate element size           : " << es << "\n" << endl;
88
89          if (g1 + g2 < 0){ cout << "Warning: Space between intersections is less than zero.\n" << endl; }
90
91
92
93          //SPACING CONSTANTS
94          int spc1 = 21;  int spc2 = 8; int spc3 = 5;                 //spacing for file print out
95
96          //CREATE FILE
97          ofstream myfile;
98          myfile.open("shell_model.fem");
99
100         //NODE TITLE                                                  //print out to .fem file
101         myfile << "'"   << setw(spc1) << "Node ID" << setw(spc1);
102         myfile << ""    << "X" << setw(spc1) << "Y" << setw(spc1) << "Z";
103         myfile << setw(spc1) << "Boundary code" << endl;
104
105         int NodeID = 0;
106
107         ///////////////////////
108         //INTERSECTIONS INNER//
109         ///////////////////////
110         VectorXd ti1, ti2, Xi1, Xi2, Yi1, Yi2, Zi1, Zi2;
111
112         ti1 = VectorXd::LinSpaced(n,0,2*M_PI*(1-(double)1/n));          //angle vector for intersection
113         ti2 = VectorXd::LinSpaced(n,0,2*M_PI*(1-(double)1/n));
114
115
116         //FIRST INTERSECTION
117         Yi1 = Rb1*ti1.array().sin();
118         Zi1 = (pow(Rm, 2) - Yi1.array().pow(2)).array().pow(0.5);
119         Xi1 = (Rb1 / cos(phi1))*ti1.array().cos() + tan(phi1)*Zi1.array();
120
121         Xi1 = Xi1.array() - Xi1(0);
122         Xi1 = Xi1.array() - g1;
123
124         for (int i = 0; i <= n-1; i++)          //print out to .fem file
125         {
126                 NodeID = NodeID + 1;
127                 myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
128                 myfile << Xi1(i) << setw(spc1) << Yi1(i) << setw(spc1) << Zi1(i) << endl;
129
130         }
131         int nid1 = NodeID;
132
133         //SECOND INTERSECTION
134         Yi2 = Rb2*ti2.array().sin();
135         Zi2 = (pow(Rm, 2) - Yi2.array().pow(2)).array().pow(0.5);
136         Xi2 = (Rb2 / cos(phi2))*ti2.array().cos() + tan(phi2)*Zi2.array();
137
138         Xi2 = Xi2.array() - Xi2(n / 2);
139         Xi2 = Xi2.array() + g2;
140
141         for (int i = 0; i <= n-1; i++)          //print out to .fem file
142         {
143                 NodeID = NodeID + 1;
144                 myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
145                 myfile << Xi2(i) << setw(spc1) << Yi2(i) << setw(spc1) << Zi2(i) << endl;
146
147         }
148         int nid2 = NodeID;
149         cout << "- Intersections complete." << endl;
150
151
152         VectorXd Xi1_temp = Xi1;                                      //saving node coordinates
153         VectorXd Xi2_temp = Xi2;
154         VectorXd Yi1_temp = Yi1;
155         VectorXd Yi2_temp = Yi2;
156         VectorXd Zi1_temp = Zi1;
157         VectorXd Zi2_temp = Zi2;
158
159         ///////////////////////
160         //INTERSECTION LAYERS//
161         ///////////////////////
162         float dR1, dR2, k1, k2, Ry1, Rx1, Ry2, Rx2, d_alpha1,d_alpha2,alpha1,alpha2;
163         float Xi1_start, Xi1_end, Xi2_start, Xi2_end;
164
165         Xi1_start = Xi1(0); Xi1_end = Xi1(n / 2);           //saving outer x-coordinates of intersection
166         Xi2_start = Xi2(0); Xi2_end = Xi2(n / 2);
167
168         dR1     = drf*layers*es;                            //Total change in intersection radius left brace
169         dR2     = drf*layers*es;                            //Total change in intersection radius right brace
170         k1      = 2 * Rb1 / (Xi1_start - Xi1_end);          //Ratio between X and Y radius of intersection
171         k2      = 2 * Rb2 / (Xi2_start - Xi2_end);
172
173         d_alpha1 = 0.15*phi1;       d_alpha2 = 0.15*phi2;   //angle change of "virtual" angle of new layers
174
175         for (int i = 1; i <= layers; i++)
176         {
177
178                 //first intersection (left brace)
179                 alpha1  = phi1 - d_alpha1*((double)i / layers);
```

```
180                 Ry1             = Rb1 + dR1*((double)i / layers);
181                 Rx1             = Ry1 / k1;
182
183                 Yi1 = Ry1*ti1.array().sin();
184                 Zi1 = (pow(Rm, 2) - Yi1.array().pow(2)).array().pow(0.5);
185                 Xi1 = Rx1*ti1.array().cos() + tan(alpha1)*Zi1.array();
186
187                 Xi1 = Xi1.array() - (Xi1(n / 2) + Xi1(0)) / 2;          //translation of layer to place it around
188                 Xi1 = Xi1.array() + (Xi1_start + Xi1_end) / 2;          //previous layer
189
190                 for (int i = 0; i <= n - 1; i++)
191                 {
192                         NodeID = NodeID + 1;
193                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
194                         myfile << Xi1(i) << setw(spc1) << Yi1(i) << setw(spc1) << Zi1(i) << endl;
195
196                 }
197
198
199                 //second intersection (right brace)
200                 alpha2 = phi2 - d_alpha2*((double)i / layers);
201                 Ry2 = Rb2 + dR2*((double)i / layers);
202                 Rx2 = Ry2 / k2;
203
204                 Yi2 = Ry2*ti2.array().sin();
205                 Zi2 = (pow(Rm, 2) - Yi2.array().pow(2)).array().pow(0.5);
206                 Xi2 = Rx2*ti2.array().cos() + tan(alpha2)*Zi2.array();
207
208                 Xi2 = Xi2.array() - (Xi2(n / 2) + Xi2(0)) / 2;
209                 Xi2 = Xi2.array() + (Xi2_start + Xi2_end) / 2;
210
211                 for (int i = 0; i <= n - 1; i++)
212                 {
213                         NodeID = NodeID + 1;
214                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
215                         myfile << Xi2(i) << setw(spc1) << Yi2(i) << setw(spc1) << Zi2(i) << endl;
216
217                 }
218
219         }
220         int nid3 = NodeID;
221         cout << "- Intersection layers complete." << endl;
222
223         //////////////////
224         //BETWEEN ARCS/////
225         //////////////////
226
227         int bp1, bp2, nid3sym;
228
229         bp1 = n/4 + abn;                                             //bp1 and bp2 are node numbers of intersection
230         bp2 = round(bbf*abs(Xi1_start - Xi2_end) / es);              //for first and last arc
231         VectorXd Xa, ta1, ta2, ta;
232         MatrixXd Ya, Za, XA(bp2,bp1), YA(bp2,bp1), ZA(bp2,bp1);
233         double r1, h;
234
235         //straight line (middle line between all arcs)
236         Xa = VectorXd::LinSpaced(bp2, Xi1(0), Xi2(n / 2));
237         Ya = MatrixXd::Zero(bp2,1);
238         Za = (MatrixXd::Ones(bp2,1))*Rm;
239
240         for (int j = 1; j <= bp2 - 2; j++)
241         {
242                 NodeID = NodeID + 1;
243                 myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
244                 myfile << Xa(j) << setw(spc1) << Ya(j) << setw(spc1) << Za(j) << endl;
245         }
246
247
248         //ARCS FIRST SIDE
249         double Ra1 = (12.5 + 12.5*rad1)*Rb, Ra2 = (15 + 10 * rad2)*Rb;          //Arcs radius (first and last)
250
251         for (int i = 1; i <=bp1; i++)
252         {
253
254                 r1      = Ra1+(Ra2-Ra1)*((double)i / bp1);                      //radius changing for each arcs
255                 alpha1  = M_PI / 2 + asin(abs(Xi1(i)) / r1);
256                 alpha2  = M_PI / 2 - asin(Xi2(n / 2 - i) / r1);                 //alpha1,2 and h are parameters for getting
257                 h       = cos(M_PI / 2 - alpha1)*r1;                            //correct arc node coordinates.
258
259                 ta = VectorXd::LinSpaced(bp2,alpha1,alpha2);                    //ta - angle vector for arcs
260
261                 Xa = r1*ta.array().cos();                                       //arc coordinates
262                 Ya = -(h - Yi1(n / 2 - i)) + r1*ta.array().sin();
263                 Za = (pow(Rm, 2) - Ya.array().pow(2)).array().pow(0.5);
264
265
266                 XA.col(i - 1) = Xa;                                             //Needed for creating arcs on opposite
267                 YA.col(i - 1) = Ya;                                             //side with symmetry
268                 ZA.col(i - 1) = Za;
269
270                 for (int j = 1; j <= bp2-2; j++)
271                 {
272                         NodeID = NodeID + 1;
273                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
274                         myfile << Xa(j) << setw(spc1) << Ya(j) << setw(spc1) << Za(j) << endl;
```

```
275
276                     }
277             }
278
279
280             nid3sym = NodeID;
281
282             //ARCS OTHER SIDE (using symetry)
283             for (int c = 0; c <= bp1-1; c++){
284             for (int j = 1; j <= bp2 - 2; j++)
285             {
286                     NodeID = NodeID + 1;
287                     myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
288                     myfile << XA(j,c) << setw(spc1) << -YA(j,c) << setw(spc1) << ZA(j,c) << endl;
289             }
290             }
291             int nid4 = NodeID;
292             cout << "- Arcs complete." << endl;
293
294
295             //////////////////////////////
296             //CONNECT SUROUNDING NODE/////
297             ////////////////////////////
298             int bp4, N;
299             N = 2 * (n - 2 * bp1 + 1) + 2 * (bp2 - 2);                  //N - amount of nodes around full perimeter
300             VectorXd X_per(N), Y_per(N);                                //(part of last brace layers, and outer arcs)
301             bp4 = n / 2 - bp1+1;
302
303             X_per << Xi2.head(bp4), (Xa.segment(1, bp2 - 2)).reverse(),     //X-per, Y-per, node coordinates of full perimeter
304                     Xi1.segment(bp1, n - (2 * (bp1 - 1) + 1)), Xa.segment(1, bp2 - 2),
305                     (Xi2.segment(1, n / 2 - bp1)).reverse();
306
307             Y_per << Yi2.head(bp4), ((Ya.col(0)).segment(1, bp2 - 2)).reverse(),
308                     Yi1.segment(bp1, n - (2 * (bp1 - 1) + 1)), -(Ya.col(0)).segment(1, bp2 - 2),
309                     -(Yi2.segment(1, n / 2 - bp1)).reverse();
310
311
312             //////////////////////////////
313             ///UNWRAP PERIMETER NODES///////
314             ////////////////////////////
315             VectorXd Y_per_u;
316             Y_per_u = Rm*((1 / Rm)*Y_per).array().asin();                //Unwrapping node coordinates to "flatspace"
317
318             //////////////////////////////
319             /////////BORDERS LIMITS///////
320             ////////////////////////////
321             double y_lim, x_lim_l, x_lim_r;                             //Defining limits to where to connect
322             double L1, L2,beta;                                         //perimeter nodes
323             int n_end, n_side;
324
325             y_lim   = 0.5*M_PI*Rm;                                      //limits
326             x_lim_l = X_per(N / 2) - p*Rm / 2;
327             x_lim_r = X_per(0) + p*Rm / 2;
328
329             L1 = x_lim_r - x_lim_l;                                     //side lengths
330             L2 = y_lim;
331
332             n_end   = round((double)N*L2 / (4 * L2 + 2 * L1));          //number of nodes at borders
333             n_side  = 0.5 * ((double)N - 4*(double)n_end+2);
334             beta    = (2*L1 + 4 * L2) / (double)N;                      //space between nodes
335
336             //////////////////////////////
337             /////////BORDERS NODES///////
338             ////////////////////////////
339             VectorXd y_end1, y_end2, y_end3, x_side1, x_side2, Xborder_all(N),Yborder_all(N);
340             MatrixXd x_end1, x_end2, x_end3, y_side1, y_side2;
341
342             y_end1  = VectorXd::LinSpaced(n_end,0,y_lim);               //calculating coordinates of nodes around border
343             x_end1  = x_lim_r*MatrixXd::Ones(n_end,1);
344             x_side1 = VectorXd::LinSpaced(n_side,x_lim_r-beta, x_lim_l+beta);
345             y_side1 = y_lim*MatrixXd::Ones(n_side,1);
346             y_end2  = VectorXd::LinSpaced(2*n_end-1, y_lim, -y_lim);
347             x_end2  = x_lim_l*MatrixXd::Ones(2*n_end-1,1);
348             x_side2 = x_side1.reverse();
349             y_side2 = -y_side1;
350             y_end3  = VectorXd::LinSpaced(n_end-1, -y_lim, 0-beta);
351             x_end3  = x_lim_r*MatrixXd::Ones(n_end-1,1);
352
353             Xborder_all << x_end1, x_side1, x_end2, x_side2, x_end3;
354             Yborder_all << y_end1, y_side1, y_end2, y_side2, y_end3;
355
356             ////////////////////////////////////////////////
357             /////////CREATE WRAPED AND UNWRAPED LINES//////
358             //////////////////////////////////////////////
359             VectorXd Xs_u, Ys_u, Ys_w, Zs_w;
360             int ns;
361             ns =n / 4 + el;
362             for (int i=0; i <= N-1; i++)
363             {
364                 //Unwrapped coordinates between perimeter and border
365                     Xs_u = non_linspace(X_per(i), Xborder_all(i),spcr_p,ns);
366                     Ys_u = non_linspace(Y_per_u(i), Yborder_all(i), spcr_p, ns);
367
368             //Wrapped coordinates between perimerter and border
369                     Ys_w = Rm*((1 / Rm)*Ys_u).array().sin();
```

```
370                 Zs_w = (pow(Rm, 2) - Ys_w.array().pow(2)).array().pow(0.5);
371
372
373
374                 for (int j = 1; j <= ns-1; j++)
375                 {
376                         NodeID = NodeID + 1;
377                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
378                         myfile << Xs_u(j) << setw(spc1) << Ys_w(j) << setw(spc1) << Zs_w(j) << endl;
379                 }
380
381         }
382
383         int nid5 = NodeID;
384         cout << "- Wraped mesh complete." << endl;
385
386
387
388         //::::::::::::::::::::::::::::::::::::::::::
389         //:::BRANCH PIPE MESH::::::::::::::::::::::
390         //::::::::::::::::::::::::::::::::::::::::::
391
392         double aphi1,aphi2,dLb, Lb, Lb3, Lb2, Lb1, xpos0_l,xpos0_r, d, dmax, dpos, dpos0, R1l,R1r, theta, spcr;
393         VectorXd angv_l,angv_r, Zdef;
394         Vector3d pos0_l, pos0_r, pos, move_pos_r, move_pos_l;
395         MatrixXd XYZ;
396         int Nb1, Nb2, Nb3;
397
398         aphi1 = M_PI / 2 - phi1;            //changing angle definition
399         aphi2 = M_PI / 2 - phi2;
400
401
402
403         Lb = pow(pow(x4 +Rm*tan(abs(phi1)), 2) + pow(z4 - Rm, 2), 0.5);//(this could possibly be changed to "lb" original branch length)
404         Lb2 = 0.80*Lb;                                              //Lbi, different length parts of branch pipe
405         Lb3 = ref_b*Lb;                                             //they have different mesh density
406         Lb1 = Lb - Lb2 - Lb3;
407
408         Nb1 = round(Lb1 / (nb1_fac * 3.00*es));                     //Nbi, number of element layers on each branch parts
409         Nb2 = round(Lb2 / (nb2_fac * 2.00*es));
410         Nb3 = round(Lb3 / (nb3_fac * 1.00*es));
411
412         angv_l = non_linspace(M_PI / 2, aphi1, 1.4, Nb2);           //Vectors consisting of angle to which the ellipses will rotate
413         angv_r = non_linspace(M_PI / 2, aphi2, 1.4, Nb2);
414         dmax = 1.0*abs(Zi1_temp(0) - Zi1_temp(n / 4));              //max deflection of first ellipse
415
416         //positions for translating ellipses
417         xpos0_l = (Xi1_start + Xi1_end) / 2;    xpos0_r = (Xi2_start + Xi2_end) / 2;
418         pos0_l << xpos0_l, 0, Rm;               pos0_r << xpos0_r, 0, Rm;
419         move_pos_l << cos(aphi1), 0, sin(aphi1);    move_pos_r << cos(aphi2), 0, sin(aphi2);
420
421         dpos0 = (Xi1_temp(n / 4) - Xi1_temp(0)) / (Xi1_temp(n / 2) - Xi1_temp(0));
422
423         spcr = 1;
424
425
426         ///BRANCH LEFT/////////////////////////////////
427         //CREATE BEGINING OF PIPE WITH PARALLEL CIRCLES
428         for (int i = 0; i <= Nb1 - 1; i = i + 1)
429         {
430                 dLb = (Lb - Lb1*(double)i / ((double)Nb1));
431                 pos = dLb*move_pos_l + pos0_l;
432                 MatrixXd XYZ = rot_transl_elps(Rb1, Rb1, (M_PI / 2 - aphi1), pos, n);
433
434                 //FOR EACH NODES, WRITE COORDINATES ON FILE:
435                 for (int j = 0; j <= (n - 1); j = j + 1){
436
437                         NodeID = NodeID + 1;
438                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
439                         myfile << XYZ(j, 0) << setw(spc1) << XYZ(j, 1) << setw(spc1) << XYZ(j, 2) << endl;
440                 }
441         }
442
443         //CREATE REST OF BRANCH PIPE, USING ROTATED AND DEFLECTED ELLIPSES
444         for (int i = 0; i <= (Nb2 + Nb3 - 1); i = i + 1)
445         {
446                 if (i >= Nb2)
447                 {
448                         R1l = Rb1 / sin(angv_l(Nb2 - 1));
449                         theta = angv_l(Nb2 - 1) - aphi1;
450                         dpos = 1-((0.5 - 0.45) / (M_PI / 2 - M_PI / 16))*(angv_l(Nb2 - 1) - M_PI / 16) - 0.45 + dpos_inc;
451                         dLb = Lb3*(1 - pow( ((double)i - (double)Nb2) / ((double)Nb3),0.65) );
452
453                 }
454                 else
455                 {
456                         R1l = Rb1 / sin(angv_l(i));
457                         theta = angv_l(i) - aphi1;
458                         dpos =1 - ((0.5 - 0.45) / (M_PI / 2 - M_PI / 16))*(angv_l(i) - M_PI / 16) - 0.45 + dpos_inc;
459                         dLb = Lb2*(1 - pow((double)i / ((double)Nb2),.8)) + Lb3;
460                 };
461
462
463                 d       = dmax*(double)i / ((double)Nb2 + (double)Nb3);
464
```

```
465                     pos = dLb*move_pos_l.array() + pos0_l.array();
466
467
468                             XYZ = rot_transl_defl_elps(R1l, Rb1, d, dpos, theta, n, pos, spcr, aphi1);
469
470
471                     //FOR EACH NODES, WRITE COORDINATES ON FILE:
472                     for (int j = 0; j <= (n - 1); j = j + 1){
473
474                             NodeID = NodeID + 1;
475                             myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
476                             myfile << XYZ(j, 0) << setw(spc1) << XYZ(j, 1) << setw(spc1) << XYZ(j, 2) << endl;
477
478                     }
479             }
480
481             int nid6 = NodeID;
482             cout << "- Left brace complete." << endl;
483
484             ///BRANCH RIGHT/////////////////////////////////
485             //CREATE BEGINING OF PIPE WITH PARALLEL CIRCLES
486             for (int i = 0; i <= Nb1 - 1; i = i + 1)
487             {
488                     dLb = (Lb - Lb1*(double)i / (double)Nb1);
489                     pos = dLb*move_pos_r + pos0_r;
490                     MatrixXd XYZ = rot_transl_elps(Rb2, Rb2, (M_PI / 2 - aphi2), pos, n);
491
492                     //FOR EACH NODES, WRITE COORDINATES ON FILE:
493                     for (int j = 0; j <= (n - 1); j = j + 1){
494
495                             NodeID = NodeID + 1;
496                             myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
497                             myfile << XYZ(j, 0) << setw(spc1) << XYZ(j, 1) << setw(spc1) << XYZ(j, 2) << endl;
498                     }
499             }
500
501             //CREATE REST OF BRANCH PIPE, USING ROTATED AND DEFLECTED ELLIPSES
502             for (int i = 0; i <= (Nb2 + Nb3 - 1); i = i + 1)
503             {
504                     if (i >= Nb2)
505                     {
506                             R1r = Rb2 / sin(angv_r(Nb2 - 1));
507                             theta = angv_r(Nb2 - 1) - aphi2;
508                             dpos = ((0.5 - 0.45) / (M_PI / 2 - M_PI / 16))*(angv_r(Nb2 - 1) - M_PI / 16) + 0.45 - dpos_inc2;
509                             dLb = Lb3*(1 - pow(((double)i - (double)Nb2) / ((double)Nb3), 0.65));
510
511                     }
512                     else
513                     {
514                             R1r = Rb2 / sin(angv_r(i));
515                             theta = angv_r(i) - aphi2;
516                             dpos = ((0.5 - 0.45) / (M_PI / 2 - M_PI / 16))*(angv_r(i) - M_PI / 16) + 0.45 - dpos_inc2;
517                             dLb = Lb2*(1 - pow((double)i / ((double)Nb2), .8)) + Lb3;
518                     };
519
520
521                     d = dmax*(double)i / ((double)Nb2 + (double)Nb3);
522
523                     pos = dLb*move_pos_r.array() + pos0_r.array();
524
525
526                     XYZ = rot_transl_defl_elps(R1r, Rb2, d, dpos, theta, n, pos, spcr, aphi2);
527
528
529                     //FOR EACH NODES, WRITE COORDINATES ON FILE:
530                     for (int j = 0; j <= (n - 1); j = j + 1){
531
532                             NodeID = NodeID + 1;
533                             myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
534                             myfile << XYZ(j, 0) << setw(spc1) << XYZ(j, 1) << setw(spc1) << XYZ(j, 2) << endl;
535
536                     }
537             }
538
539             int nid7 = NodeID;
540             cout << "- Right brace complete." << endl;
541
542             //::::::::::::::::::::::::::::::::::::::::::
543             //:::HALF CIRCLES UNDER INTERSECTION::::::::
544             //::::::::::::::::::::::::::::::::::::::::::
545
546             int n_hc;
547             VectorXd t_hc, Y_hc, Z_hc;
548             double X_hc,beta_2;
549
550             n_hc = 2*n_end-1;                                       //n_hc - number of nodes on each half circle
551
552             t_hc = VectorXd::LinSpaced(n_hc, -M_PI, 0);             //angle vector
553             Y_hc = Rm*t_hc.array().cos();                           //coordinates of nodes on half circles
554             Z_hc = Rm*t_hc.array().sin();
555
556             beta_2 = L1 / (n_side + 1);                             //distance between each half circle (x-dir)
557
558             //FOR EACH HALF CIRCLE
559             for (int i = 1; i <= n_side+2; i = i + 1)
```

```
560         {
561                 X_hc = x_lim_r-(i-1)*beta_2;                    //increasing x-coordinate
562
563                 //FOR EACH NODES, WRITE COORDINATES ON FILE:
564                 for (int j = 1; j <= (n_hc - 2); j = j + 1){
565
566                         NodeID = NodeID + 1;
567                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
568                         myfile << X_hc << setw(spc1) << Y_hc(j) << setw(spc1) << Z_hc(j) << endl;
569                 }
570
571         }
572
573         int nid8 = NodeID;
574         cout << "- Bottom complete." << endl;
575
576         //:::::::::::::::::::::::::::::::::::::::::
577         //:::FILLING INTERSECTION::::::::::::::::::                //filling the "holes" in the intersections
578         //:::::::::::::::::::::::::::::::::::::::::
579         double qr, ql,qy, xsql_l, xsqr_l,ysq,beta3,xsql_r,xsqr_r;
580         VectorXd xsquare_l(n), ysquare(n), xsquare_r(n);
581
582         //square limits (defining a square geometry inside the intersection)
583         ql = 0.25; qr = 0.25; qy = 0.25;
584         xsql_l = Xi1_end + ql*abs(Xi1_end - Xi1_start);
585         xsql_r = Xi2_end + ql*abs(Xi2_end - Xi2_start);
586         xsqr_l = Xi1_start - qr*abs(Xi1_end - Xi1_start);
587         xsqr_r = Xi2_start - qr*abs(Xi2_end - Xi2_start);
588
589         ysq = ql*2*Rb1;
590         beta3 = abs(xsql_l - xsqr_l) / ((double)n / 4);
591
592         //square border node cooridinates
593         xsquare_l << xsqr_l*(MatrixXd::Ones(n / 8 + 1, 1)), VectorXd::LinSpaced(n / 4 - 1, xsqr_l-beta3, xsql_l+beta3),
594                 xsql_l*(MatrixXd::Ones(n / 4 + 1, 1)), VectorXd::LinSpaced(n / 4 - 1, xsql_l+beta3, xsqr_l-beta3),
595                 xsqr_l*(MatrixXd::Ones(n / 8 , 1));
596
597         xsquare_r << xsqr_r*(MatrixXd::Ones(n / 8 + 1, 1)), VectorXd::LinSpaced(n / 4 - 1, xsqr_r - beta3, xsql_r + beta3),
598                 xsql_r*(MatrixXd::Ones(n / 4 + 1, 1)), VectorXd::LinSpaced(n / 4 - 1, xsql_r + beta3, xsqr_r - beta3),
599                 xsqr_r*(MatrixXd::Ones(n / 8, 1));
600
601         ysquare << VectorXd::LinSpaced(n / 8 + 1, 0, ysq), ysq*(MatrixXd::Ones(n / 4 - 1, 1)),
602         VectorXd::LinSpaced(n / 4 + 1, ysq, -ysq),-ysq*(MatrixXd::Ones(n / 4 - 1, 1)),
603         VectorXd::LinSpaced(n / 8, -ysq, 0 - 0.5*beta3);
604
605
606         //create nodes from intersection to square (left side)
607         int nsq = n / 8;
608         VectorXd Xsq, Ysq, Zsq;
609         for (int i = 0; i <= n-1; i++)
610         {
611                 Xsq = VectorXd::LinSpaced(nsq, Xi1_temp(i), xsquare_l(i));
612                 Ysq = VectorXd::LinSpaced(nsq, Yi1_temp(i), ysquare(i));
613                 Zsq = (pow(Rm,2)-Ysq.array().pow(2)).array().pow(0.5);
614
615                 for (int j = 1; j <= (nsq - 1); j++){
616
617                         NodeID = NodeID + 1;
618                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
619                         myfile << Xsq(j) << setw(spc1) << Ysq(j) << setw(spc1) << Zsq(j) << endl;
620                 }
621         }
622         int nid9 = NodeID;
623
624         VectorXd Ysq2 = VectorXd::LinSpaced(n/4-1,ysq-0.5*beta3,-ysq+0.5*beta3);
625         VectorXd Xsq2 = VectorXd::LinSpaced(n / 4 - 1, xsqr_l - 0.75*beta3, xsql_l + 0.75*beta3);
626         VectorXd Zsq2 = (pow(Rm, 2) - Ysq2.array().pow(2)).array().pow(0.5);
627
628         //square fill (left side)
629         for (int i = 0; i <= (n / 4 - 2); i++){
630                 for (int j = 0; j <= (n / 4 - 2); j++){
631
632                         NodeID = NodeID + 1;
633                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
634                         myfile << Xsq2(j) << setw(spc1) << Ysq2(i) << setw(spc1) << Zsq2(i) << endl;
635                 };
636         }
637         int nid10 = NodeID;
638         cout << "- Left brace hole fill complete." << endl;
639
640         //create nodes from intersection to square (right side)
641         for (int i = 0; i <= n - 1; i++)
642         {
643                 Xsq = VectorXd::LinSpaced(nsq, Xi2_temp(i), xsquare_r(i));
644                 Ysq = VectorXd::LinSpaced(nsq, Yi2_temp(i), ysquare(i));
645                 Zsq = (pow(Rm, 2) - Ysq.array().pow(2)).array().pow(0.5);
646
647                 for (int j = 1; j <= (nsq - 1); j++){
648
649                         NodeID = NodeID + 1;
650                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
651                         myfile << Xsq(j) << setw(spc1) << Ysq(j) << setw(spc1) << Zsq(j) << endl;
652                 }
653         }
654
```

```
655         int nid11 = NodeID;
656 
657         //square fill (right side)
658         Xsq2 = VectorXd::LinSpaced(n / 4 - 1, xsqr_r - 0.75*beta3, xsql_r + 0.75*beta3);
659 
660         for (int i = 0; i <= (n / 4 - 2); i++){
661                 for (int j = 0; j <= (n / 4 - 2); j++){
662 
663                         NodeID = NodeID + 1;
664                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
665                         myfile << Xsq2(j) << setw(spc1) << Ysq2(i) << setw(spc1) << Zsq2(i) << endl;
666                 };
667         }
668         int nid12 = NodeID;
669         cout << "- Right brace hole fill complete." << endl;
670 
671 
672         //:::::::::::::::::::::::::::::::::::::::::
673         //:::MAIN PIPE SIDE EXTENSION::::::::::::::            //extending chords at the outer ends with circles
674         //:::::::::::::::::::::::::::::::::::::::::
675         int n_s1l  = abs(x1 - x_lim_l) / beta + nl;
676         int n_s1r  = abs(x3 - x_lim_r) / beta + nr;
677         int n_s2   = 2 * n_end - 1 + (n_hc - 2);
678         VectorXd Xs_l = VectorXd::LinSpaced(n_s1l, x_lim_l - abs(x1 - x_lim_l)/n_s1l, x1);
679         VectorXd Xs_r = VectorXd::LinSpaced(n_s1r, x_lim_r + abs(x3 - x_lim_r)/n_s1r, x3);
680 
681         VectorXd t_s = VectorXd::LinSpaced(n_s2, 0, 2 * M_PI*(1-1/(double)n_s2));
682         VectorXd Ys  = Rm*t_s.array().cos();
683         VectorXd Zs  = Rm*t_s.array().sin();
684 
685         //LEFT SIDE
686         for (int i = 0; i <= n_s1l-1; i++)
687         for (int j = 0; j <= n_s2-1; j++)
688         {{
689                 NodeID = NodeID + 1;
690                 myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
691                 myfile << Xs_l(i) << setw(spc1) << Ys(j) << setw(spc1) << Zs(j) << endl;
692         }}
693         int nid13 = NodeID;
694         cout << "- Left chord complete." << endl;
695 
696 
697 
698         //RIGHT SIDE
699         for (int i = 0; i <= n_s1r - 1; i++)
700         for (int j = 0; j <= n_s2 - 1; j++)
701         {
702                 {
703                         NodeID = NodeID + 1;
704                         myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
705                         myfile << Xs_r(i) << setw(spc1) << Ys(j) << setw(spc1) << Zs(j) << endl;
706                 }
707         }
708         int nid14 = NodeID;
709         cout << "- Right chord complete." << endl;
710 
711 
712         //:::::::::::::::::::::::::::::::::::::::::
713         //:::BEAM NODES::::::::::::::::::::::::::::            //creating nodes at the center of each
714         //:::::::::::::::::::::::::::::::::::::::::
715         VectorXd beamNid_x(4),beamNid_z(4);
716 
717         beamNid_x << x1, x3, x4+dg, x5-dg;
718         beamNid_z << 0, 0, z4, z5;
719 
720 
721         for (int i = 0; i <= 3; i++){
722                 NodeID = NodeID + 1;
723                 myfile << "NODE" << setw(spc1 - 2) << NodeID << setw(spc1);
724                 myfile << beamNid_x(i) << setw(spc1) << 0.00 << setw(spc1) << beamNid_z(i);
725                 if (i == 0)     { myfile << setw(4) << " 1  1  1  1  1  1" << endl; }       //BC on one end
726                 else            { myfile << endl; }
727         }
728         int nid15 = NodeID;
729         cout << "- Beam nodes complete." << endl;
730 
731 u
```

Appendix B - Rainflow algorithm code

Below is the MATLAB code for the rainflow algorithm used for calculating the accumulated damage for each hotspots. It is used in the script *dmg_calc*.

```
1
2  function total_damage=rainflow(data,stress_yield,a_bar,m,plotting)
3
4  %This function reads a data set and material properties
5  %then performs a rainflow analysis and calculates the
6  %accumulated damage.
7
8  points   = length(data);                %numb. or data points
9  y        = data;                        %data
10 x        = linspace(1,points,points);   %time steps
11
12
13 if plotting == true;                    %plot initial data
14 plot(x,y,'b-x');
15 grid on;
16 hold on;
17 pause
18 end;
19 %::::::::::::::::::::::::::::
20
21 pv_count = 1;                           %"peak and valley" counter
22
23 y_PV(1) = y(1);                         %y_PV and x_PV are the vectors
24 x_PV(1) = x(1);                         %containing only the peaks and valleys
25
26 %PEAKS AND VALLEYS
27 for i=2:points-1                        %For each data point, except
28                                         %first and last.
29     s1  = y(i-1);
30     s2  = y(i);
31     s3  = y(i+1);
32
33     d1  = s2-s1;
34     d2  = s3-s2;
35
36     if abs(d1)>0                        %In case two consecutive values
37         delta1  = d1;                   %are the same (d1 or d2 = 0). We
38     end                                 %don't updata delta 1 and 2
39     if abs(d2)>0
40         delta2  = d2;
41     end
42
43     f   = delta1*delta2;                %Found a peak or valley if f is negative
44
45     if f<0
46         pv_count = pv_count + 1;        %count peaks/valleys
47         y_PV(pv_count) = s2;            %add point to new vectors
48         x_PV(pv_count) = x(i);
49     end
50 end
51 pv_count = pv_count + 1;                %we add the last points to
52 y_PV(pv_count) = y(points);             %to the vectors
53 x_PV(pv_count) = x(points);
54
55 %::::::::::::::::::::::::
56
57 %MAX MIN POINTS
58 y_max = max(y_PV);                      %it is suggested to change the
59 y_min = min(y_PV);                      %change the first and last points
60                                         %in the peaks and valeys to the min
61                                         %and max values of the data set.
62
63 if y_PV(2)>y_PV(1)                      %if first peak or valley is a peak
64     y_PV(1) = y_min;                    %we change first to point to min point
65 else
66     y_PV(1) = y_max;                    %else max point
67 end
68
69 if y_PV(pv_count-1)>y_PV(pv_count)      %same goes for last point
70     y_PV(pv_count) = y_min;
71 else
72     y_PV(pv_count) = y_max;
73 end
74
75 if plotting == true;  clf('reset'); plot(x_PV,y_PV,'r-'); grid on; hold on; pause; end;
76
77 %::::::::::::::::::::::::::::::::::::::
78
79
80 %CYCLES
81 points_3     = false;            %true if only 3 points left (special case)
82 rounds       = 0;                %counter for amount of rounds removing cycles
83 Dmg          = 0;                %damage for hotspot
84
85 while pv_count >2                %CHECK A NEW ROUND OF CYCLE UNTIL 2 POINTS ARE LEFT
86     rounds = rounds + 1;         %counting rounds of removing cycles
```

```
87
88      y_PV_new     = y_PV;          %SAVING THE LAST POINTS AS A NEW VECTOR
89      x_PV_new     = x_PV;          %for every cycle search
90
91      cycles       = 0;             %counter for amount of cycles per round
92      r_id         = 1;             %counter for points (ids) to remove
93      remove_ids   = [];            %vector containing points (ids) to remove
94
95      if pv_count == 3              %checking if only 3 points left
96          points_3     = true;
97          i_max        = 1;         %only one iteration if 3 points
98      else
99          points_3     = false;
100         i_max        = pv_count-3;
101     end
102
103 for i=1:i_max                     %for each points in peaks and valleys
104
105     s1 = y_PV_new(i);             %save 4 stress values
106     s2 = y_PV_new(i+1);
107     s3 = y_PV_new(i+2);
108
109     if points_3 == false
110     s4 = y_PV_new(i+3);           %if there are only 3 points left, leave out s4
111     end
112
113
114     d21 = s2-s1;                  %3 stress differences
115     d32 = s3-s2;
116     if points_3 == false          %if there are only 3 points left, leave out d43
117     d43 = s4-s3;
118     end
119
120     if points_3 == true           %special case if only 3 points left
121         cyc_amp      = min([abs(s2-s1),abs(s3-s2)])/4;
122         cyc_mean     = s2 - sign(d21)*cyc_amp;
123         points_3     = true;
124     end
125
126     %checking if its a full cycle
127     if (abs(d43)>=abs(d32) && abs(d21)>=abs(d32)) || points_3==true
128
129         cycles       = cycles + 1;                 %counting cycles per round
130
131         if points_3 == false
132             cyc_amp      = abs(s3-s2)/2;           %cycles amplitude
133             cyc_mean     = min([s2,s3])+cyc_amp;   %cycles mean stress
134         end
135
136         cyc_amp_mod = cyc_amp/(1-abs(cyc_mean)/stress_yield);%Mod. amplitude (not used)
137         delta_dmg    = (1/a_bar)*cyc_amp^m;                   %Damage for cycle
138         Dmg          = Dmg + delta_dmg*1e8;                   %Adding damage
139                                                               %(1e8 is a scaling factor)
140
141
142         cyc_amp_M(rounds,cycles)      = cyc_amp;              %saving cycles amplitude
143         cyc_mean_M(rounds,cycles)     = cyc_mean;             %saving cycles mean stress
144
145         if plotting == true;
146         %ploting a line at each mean stress
147         plot([x_PV_new(i+1),x_PV_new(i+2)],[cyc_mean,cyc_mean],'black-')
148
149         %calculating the time value (x) at the center of the cycle
150         x_mid = x_PV_new(i+1)+(x_PV_new(i+2)-x_PV_new(i+1))/2;
151
152         %plotting a vertical line from min stress to mean stress
153         plot([x_mid,x_mid],[0,cyc_mean],'color',[0.2 0.3 0.05],'marker','*')
154         end;
155
156         if points_3 == false
157             remove_ids([r_id,r_id+1]) = [i+1,i+2];  %adding points to remove to remove_ids
                    vector
158             r_id=r_id+2;                            %increasing counter with 2 (removing 2
                    points)
159         else
160             remove_ids(r_id) = [i+1];
161         end
162     end
163 end
164
165     y_PV(remove_ids)=[];                %remove stress values
166     x_PV(remove_ids)=[];                %remove time values
167
168     pv_count = length(y_PV);            %setting the new amount of points
169
170     if plotting == true;
171     pause
172     clf('reset')                        %reset plot
173     grid on;
174     hold on
175     plot(x_PV,y_PV,'red-*')             %plotting new cycles
176     disp(['Total damage:',num2str(Dmg)])%display total damage for a round
177     end
178 end
179
```

```
180 disp(['TOTAL damage:',num2str(Dmg)])      %display total for hotspot
181
182 total_damage = Dmg;                      %return total damage for hotspot
```