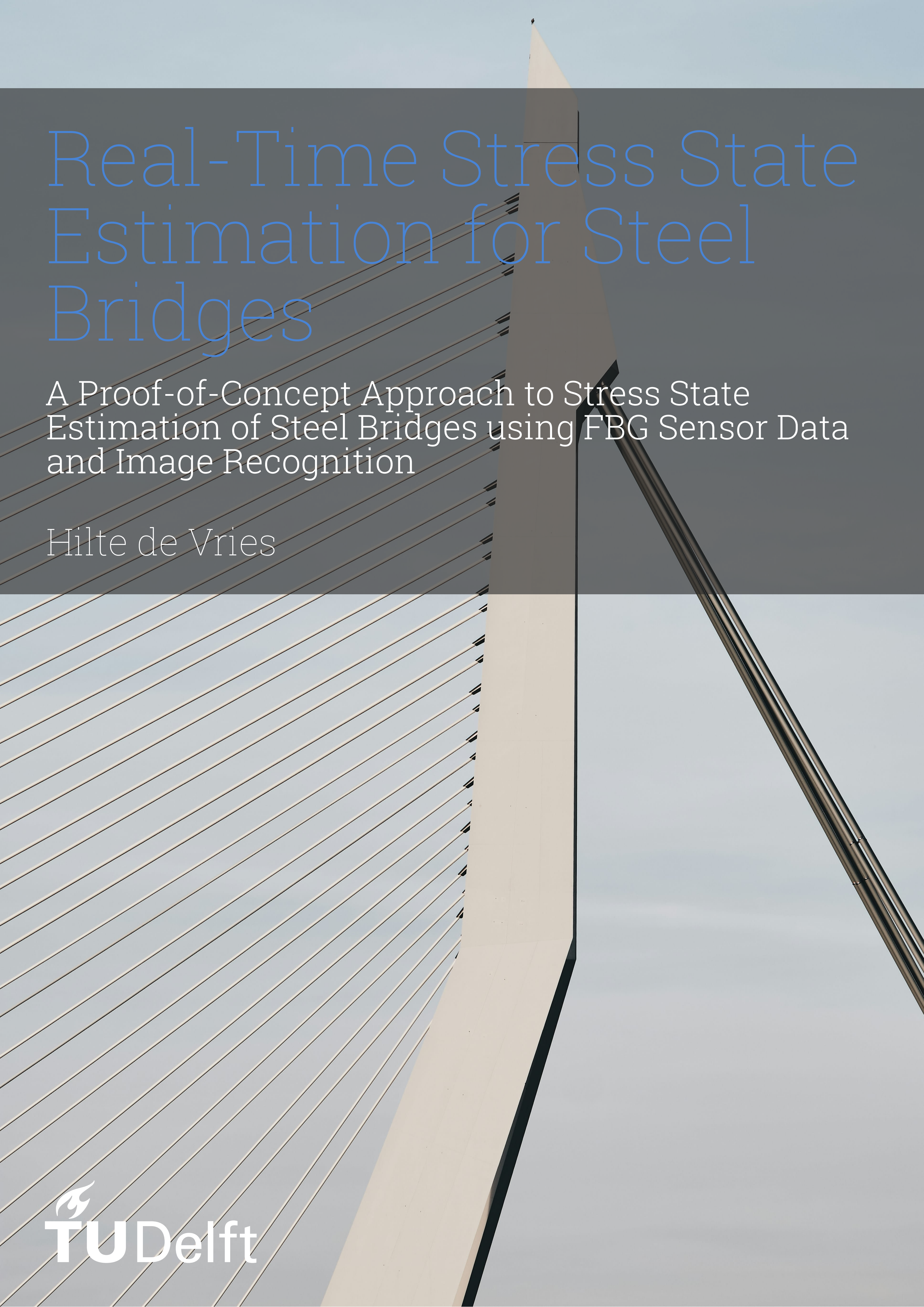


Real-Time Stress State Estimation for Steel Bridges

A Proof-of-Concept Approach to Stress State Estimation of Steel Bridges using FBG Sensor Data and Image Recognition

Hilte de Vries

Real-Time Stress State Estimation for Steel Bridges

A Proof-of-Concept Approach to Stress State Estimation of Steel Bridges using FBG Sensor Data and Image Recognition

by

Hilte de Vries

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday February 3, 2025 at 13:00.

Student number:	5128153	
Report number:	2024.MME.9015	
Project duration:	May 1, 2024 – December 20, 2024	
Thesis committee:	Dr.ir. Y. Pang,	TU Delft, Chair
	Ir. W. v.d. Bos,	TU Delft, Supervisor
	Dr.ir. J.H. den Besten,	TU Delft

Preface

This thesis presents a novel methodology that combines real-world measurements with computational modeling to estimate the stress states of steel bridges. The method was tested through an experimental setup on a 3-meter-long pedestrian bridge. By intelligently limiting the number of combinations, the methodology is made scalable to full-scale bridge applications. The primary goal of this approach is to predict the remaining fatigue life of a bridge using highly realistic load data, ultimately enabling bridges to remain operational for many more years than is currently achievable.

I completed this thesis as part of my Master's degree in Mechanical Engineering, specializing in Multi-Machine Engineering. With a personal interest in analytical solutions and programming, I developed this project in collaboration with Iv-Infra and their SBK department, which is responsible for the design and engineering of movable steel structures. The push for more accurate load information is critical in their work, ensuring the safe extension of the lifetime of bridges.

Working on this project has been both challenging and incredibly rewarding. I have gained valuable insights into the structural design of bridges and the associated engineering challenges. Developing a completely new methodology, executing the full functional pipeline, and testing its effectiveness has been a deeply gratifying experience. I am proud of what I have accomplished and excited to see the future possibilities within this field.

I would like to express my sincere gratitude to my TU Delft supervisor, Wouter van den Bos, for his expert guidance, continuous support, and constructive feedback. I greatly appreciated the creative freedom he provided, allowing me to explore and develop the project in line with my own interests.

My thanks also go to Garry Vandeberg and Jeremy Augustijn of Iv-Infra for their enthusiasm and valuable input throughout the project. Their expert opinions and guidance were essential in shaping the direction of the thesis, particularly in setting up the experimental framework and securing the company's interest in my research. In addition, I would like to extend my thanks to all my colleagues at the SBK department for creating a fantastic working environment and for the enjoyable moments shared over the past few months.

Lastly, I am immensely grateful to my family and friends for their unwavering support throughout my studies, and especially during the writing of my master's thesis. Their encouragement and genuine interest in my progress provided me with great motivation to deliver the best possible results.

Hilte de Vries
Sliedrecht, January 2025

Summary

This research presents a novel approach for real-time stress state estimation in steel bridges using Fiber Bragg Grating (FBG) sensors and image recognition techniques. The methodology involves creating a digital model of the bridge, comprising a global finite element model (FEM) and detailed sub-models of critical areas. A database of precomputed load cases is generated, and real-time sensor data is matched to this database using the developed fingerprinting method. Image recognition is employed to detect multiple load scenarios, enhancing the accuracy of stress estimations and ensuring linear scalability for multi-load situations. The accuracy of the developed model was tested using a scaled setup using a 3 meter long aluminium bridge, proving its effectiveness in real-world conditions. The results demonstrate the feasibility of this approach, with reasonable accuracy achieved in both single and multi-load scenarios. Future work should focus on improving model accuracy, enhancing image recognition algorithms, and optimizing computational performance for large-scale applications.

Samenvatting

Dit onderzoek presenteert een nieuwe benadering voor het schatten van de spanningsstatus in stalen bruggen in realtime, door gebruik te maken FBG sensoren en beeldherkenningstechnieken. De methodologie omvat het creëren van een digitaal model van de brug, bestaande uit een FEM model en gedetailleerde submodellen van kritieke gebieden. Een database van vooraf berekende belastingen wordt gegenereerd, en realtime sensordata wordt gekoppeld aan deze database met behulp van de ontwikkelde fingerprinting methode. Beeldherkenning wordt ingezet om het aantal belastingen te detecteren, waardoor de nauwkeurigheid van spanningsschattingen wordt verbeterd en lineaire schaalbaarheid voor meervoudige belasting situaties wordt gegarandeerd. De nauwkeurigheid van het ontwikkelde model werd getest met een proefopstelling met een 3 meter lange aluminium brug, waarmee de effectiviteit in realistische omstandigheden werd aangetoond. De resultaten tonen de haalbaarheid van deze benadering aan, met redelijke nauwkeurigheid in zowel enkel- als meervoudige belasting scenario's. Toekomstig onderzoek kan zich richten op het verbeteren van de modelnauwkeurigheid, het verbeteren van beeldherkenning algoritmes en het optimaliseren van de rekensnelheid voor grootschalige toepassingen.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
DAM	Data Acquisition Modality
DSS	Decision Support System
FBG	Fiber Bragg Grating
FEM	Finite Element Methods
GPs	Grid Points
HCF	High-Cycle Fatigue
KPIs	Key Performance Indicators
LEFM	Linear Elastic Fracture Mechanics
MAD	Median Absolute Deviation
RTSP	Real-Time Streaming Protocol
SCs	Sensor Configurations
SHM	Structural Health Monitoring
TCP	Transmission Control Protocol

Symbols

Symbol	Definition	Unit
a	Length coordinate of footstep	[mm]
a_{box}	Length of boundary box from initial camera location estimate along the length of the asset	[mm]
a_{cam_c}	a coordinate from initial camera location estimate of grid point c along the length of the asset	[mm]
a_{est}	Estimate of length coordinate of footstep	[mm]
a_{est_c}	Estimate of length coordinate of footstep c	[mm]
$a_{k,h}$	Length coordinate of footstep n located in area h at position k	[mm]
$a_{k,h_{\text{min}}}$	Length coordinate of footstep n located in optimal area h_{min} at position k	[mm]
$a_{k_c,h_{\text{min}_c}}$	Length coordinate of footstep c located in optimal area h_{min} at position k	[mm]
a_{lower}	Lower bound for length coordinate	[mm]
a_{min}	a coordinate at grid point n_{min}	[mm]
a_{min_c}	a coordinate at grid point c	[mm]
a_{upper}	Upper bound for length coordinate	[mm]
a_q	Length coordinate of grid point q	[mm]
b	Width coordinate of footstep	[mm]
a_{box}	Length of boundary box from initial camera location estimate along the width of the asset	[mm]
b_{cam_c}	b coordinate from initial camera location estimate of grid point c along the width of the asset	[mm]
b_{est}	Estimate of width coordinate of footstep	[mm]
a_{est_c}	Estimate of width coordinate of footstep c	[mm]

Symbol	Definition	Unit
$b_{k,h}$	Width coordinate of footstep n located in area h at position k	[mm]
$b_{k,h_{\text{min}}}$	Width coordinate of footstep n located in optimal area h_{min} at position k	[mm]
$b_{k_c,h_{\text{min}_c}}$	Width coordinate of footstep c located in optimal area h_{min} at position k	[mm]
b_{lower}	Lower bound for width coordinate	[mm]
b_{min}	b coordinate at grid point n_{min}	[mm]
b_{min_c}	b coordinate at grid point c	[mm]
b_{upper}	Upper bound for width coordinate	[mm]
b_r	Width coordinate of grid point r	[mm]
C	Set of combined grid points in sets C^- and C^+	[-]
C^-	Set of coupled grid points out of the set N_{GP} that are currently being optimized	[-]
C^+	Set of coupled grid points out of the set N_{GP} that are locked during the optimization of the grid point in set C^-	[-]
C_p	Empirical constant of Paris equation	$\left[\frac{\text{mm/cycle}}{\left(\text{MPa}\sqrt{\text{m}}\right)^m}\right]$
c	Selected grid point numbers out of set N_{GP} representing the currently active grid points during multi-load interpolation	[-]
D_s	Total accumulated fatigue damage in detail s	[-]
d_{diff}	Inaccuracy in distance estimation	[-]
$f_{x,s}$	Frequency of stress range x occurring in detail s	[-]
F	Force magnitude of load	[N]
F_{diff}	Inaccuracy in load estimation	[%]
F_{est}	Estimated force magnitude of load	[N]
F_{est_c}	Estimated force magnitude of load c	[N]
$F_{\text{k,h}_{\text{min}}}$	Force magnitude of grid point located in optimal area h_{min} at position k	[N]
$F_{\text{k}_c,\text{h}_{\text{min}_c}}$	Force magnitude of grid point located in optimal area h_{min} at position k around coupled grid point c	[N]
H	Set of quadrants surrounding grid point n_{min}	[-]
HS_{diff}	Average inaccuracy in hot spot stress estimation	[%]
h	Index for area number in quadrants H	[-]
h_c	Index for area number in quadrants H around coupled grid point c	[-]
h_{c^-}	Index for area number in quadrants H during the optimization of grid point c^-	[-]
h_{min}	Area number for which the MAD of the relative strain matrix is minimal	[-]
h_{min_c}	Area number of grid point c for which the MAD of the relative strain matrix is minimal	[-]
$h_{\text{min}_{c^-}}$	Area number of grid point c for which the MAD of the relative strain matrix is minimal during the optimization of grid point c^-	[-]
I	Set of sensors providing strain measurements	[-]
I^-	Set of redundant sensors not used during the fingerprinting algorithm, which are then used for error estimation	[-]
i	Index of strain sensors	[-]
j	Index of strain sensors	[-]
K	Set of grid points forming area h	[-]
k	Index for grid point in grid points K	[-]
k_c	Index for grid point in grid points K around coupled grid point c	[-]

Symbol	Definition	Unit
L_s	Theoretical fatigue life of detail s	[years]
l	Crack length	[mm]
$\mathrm{MAD}^{(\mathrm{h})}$	Median absolute deviation around the median for area h	[-]
$\mathrm{MAD}^{(\mathrm{h_{c-}})}$	Median absolute deviation around the median for area h during the optimization of grid point c^-	[-]
$\mathrm{MAD}^{(\mathrm{n})}$	Median absolute deviation around the median for grid point n	[-]
m	Empirical constant of Paris equation	[-]
N	Load cycles	[-]
N_a	Number of grid points in the length direction	[-]
N_b	Number of grid points in the width direction	[-]
N_{GP}	Set of grid points	[-]
N_{box}	Set of grid points falling inside of the camera initial location guess box	[-]
N_{total}	Total number of grid points	[-]
N_x	Stress cycles until theoretical fatigue limit for stress level x	[-]
n	Index representing the grid point	[-]
$n_{\mathrm{footsteps}}$	Number of footsteps that are present on the asset	[-]
$n_{\mathrm{footsteps_{max}}}$	Maximum number of footsteps that can be present on the asset	[-]
$n_{k,h}$	Grid point n located in area h at position k	[-]
$n_{k,h}$	Grid point n located in area h at position k	[-]
n_{loads}	Number of loads that are present on the asset	[-]
n_{min}	Best matching grid point, providing the minimum objective value	[-]
$n_{x,s}$	Stress cycles occurred during asset's lifetime for stress level x in detail location s	[-]
q	Index ranging from 1 to N_a, representing the q'th grid point in length direction	[-]
r	Index ranging from 1 to N_b, representing the r'th grid point in width direction	[-]
S	Set of analyzed detail locations on the asset	[-]
SG_L	Number of strain gauges located on the left under flange	[-]
SG_M	Number of strain gauges located on the cross girders	[-]
SG_R	Number of strain gauges located on the right under flange	[-]
t	Total operational period	[h]
$\tilde{X}^{(h)}$	Mean of relative strain matrix of grid points in area h	[-]
$\tilde{X}^{(h)}$	Mean of relative strain matrix of grid points in area h during the optimization of grid point c^-	[-]
$\tilde{X}^{(n)}$	Mean of relative strain matrix of grid point n	[-]
$X^{(h)}$	Relative strain matrix, comparing measurement data to grid points in area h	[-]
$X^{(n)}$	Relative strain matrix, comparing measurement data to grid point n	[-]
$X_{ij}^{(h)}$	Element i, j of the relative strain matrix, comparing measurement data to grid points in area h	[-]
$X_{ij}^{(h_{c^-})}$	Element i, j of the relative strain matrix, comparing measurement data to grid points in area h during the optimization of grid point c^-	[-]
$X_{ij}^{(n)}$	Element i, j of the relative strain matrix, comparing measurement data to grid point n	[-]
Z	Objective function	[-]
$\%\Delta\sigma$	Median stress estimation inaccuracy	[-]

Symbol	Definition	Unit
$\%\Delta\epsilon$	Median strain estimation inaccuracy	[-]
α	Scale factor for force and stress scaling of coupled grid point to the estimated values of the measurement data	[-]
$\alpha^{(c)}$	Scale factor for force and stress scaling of coupled grid point c to the estimated values of the measurement data	[-]
$\alpha^{(n)}$	Scale factor for force and stress scaling of coupled grid point n to the estimated values of the measurement data	[-]
$\beta_{k,h}$	Scale factor for grid point location k of area h	[-]
$\beta_{k,h_{\min}}$	Scale factor for grid point location k of optimal area $h_{\min}$	[-]
β_{k_c,h_c}	Scale factor for grid point location k of area h around coupled grid point c	[-]
$\beta_{k_c,h_{\min_c}}$	Scale factor for grid point location k of optimal area $h_{\min}$ around coupled grid point c	[-]
$\gamma^{(c)}$	Switch for turning grid points active and inactive, where the value $\gamma^{(c)}=0$ if grid point c has no load and $\gamma^{(c)}=1$ if it has a load	[-]
$\gamma^{(n)}$	Switch for turning grid points active and inactive, where the value $\gamma^{(n)}=0$ if grid point n has no load and $\gamma^{(n)}=1$ if it has a load	[-]
γ_{Mf}	Partial factor for fatigue strength	[-]
Δa	Difference between grid points in the length direction	[mm]
Δb	Difference between grid points in the width direction	[mm]
Δd	Euclidean distance between load location estimation and the actual location	[mm]
ΔK	Stress intensity factor range	$[\mathrm{MPa}\sqrt{\mathrm{m}}]$
ϵ_c	Cut-off strain for fingerprinting to occur	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
ϵ_i	Strain at measurement location i	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_i^{(n)}$	Strain at measurement location i for grid point n	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_i^{(n_{k,h})}$	Strain at measurement location i for grid point n located in area h at location k	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_i^{(n_{k,h_{\min}})}$	Strain at measurement location i for grid point n located in optimal area $h_{\min}$ at location k	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_i^{(n_{k_c,h_{\min_c}})}$	Strain at measurement location i for grid point n located in optimal area $h_{\min}$ at location k around coupled grid point c	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_i^{(n_{k_c,h_c})}$	Strain at measurement location i for grid point n located in area h at location k around coupled grid point c	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
ϵ_j	Strain at measurement location j	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_j^{(n)}$	Strain at measurement location j for grid point n	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_j^{(n_{k,h})}$	Strain at measurement location j for grid point n located in area h at location k	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_j^{(n_{k_c,h_c})}$	Strain at measurement location j for grid point n located in area h at location k around coupled grid point c	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
$\epsilon_{\max_i}$	Maximum strain at measurement location i	$[\mathrm{mm}\cdot\mathrm{mm}^{-1}]$
Σ	Solution space size	[-]
$\sigma_{0.4\mathrm{t}}$	Stress at 0.4 plate thickness from the supposed crack location of the weld	[MPa]
$\sigma_{1.0\mathrm{t}}$	Stress at 0.4 plate thickness from the supposed crack location of the weld	[MPa]
σ_{HS}	Hot spot stress at the supposed crack location of the weld	[MPa]
σ_s	Hot-spot stress at researched detail location s	[MPa]
$\sigma_{s,\mathrm{est}}$	Estimated hot-spot stress at researched detail location s	[MPa]

Symbol	Definition	Unit
$\sigma_s^{(n_{k,h_{\mathrm{min}}})}$	Hot-spot stress at researched detail location s as a result of load applied by grid point n located in optimal area h_{min} at location k	[MPa]
$\sigma_s^{(n_{k_c,h_{\mathrm{min}_c}})}$	Hot-spot stress at researched detail location s as a result of load applied by grid point n located in optimal area h_{min} at location k around coupled grid point c	[MPa]
$\sigma_{s,\mathrm{est}}$	Estimated hot-spot stress at researched detail location s	[MPa]

1

Introduction

One of the most prominent failure mechanisms for steel bridges is fatigue [37], [3]. Loads applied by for example vehicles, trains, pedestrians, wind, and temperature are typically below the yield strength of the material. However, the cyclic nature of the loads means that over time the structural integrity of the bridge decays, leading to fracture initiation, propagation, and eventually structural failure. Fatigue-related damage can be repaired and maintained by performing routine maintenance, but this maintenance is costly and often unnecessary given the poor understanding of fatigue status [35].

A large part of bridges in the Netherlands were built right after WW II, and with a designed lifetime of around 100 years, they are nearing their design limit. Iv [11] is an engineering firm in The Netherlands. They receive many projects for the possible extension of a bridge's theoratical lifetime. To safely extend the usage of the bridge, knowledge is required of the fatigue load throughout the bridge's operation to estimate the current fatigue state and predict the remaining lifetime. An example of a steel traffic bridge nearing the end of its lifetime is the Merwedebrug, shown in Figure 1.1 *a*). The bridge opened in 1961 [16] and has recently been closed for heavy traffic due to the discovery of small crack formation [16]. After repairs in 2016, as seen in Figure 1.1 *b*), the bridge opened again for all traffic. The closure of the bridge had a large effect on logistics and a total of 33 million euros was claimed in damages [26], in addition to the value lost from the large amounts of traffic jams caused by the closure. This calls for a smarter system to track loads and predict fatigue life to predict and/or prevent these costly closures.

Figure 1.1: a) Side view of the Merwedebrug, b) Bottom view of the Merwedebrug during repair work.

Determining the fatigue damage and remaining fatigue life is an active field of research. A review by X. W. Ye et al. [37] lists all of the used techniques to analyze the fatigue damage and remaining fatigue life. The classic analysis is split into fatigue life prediction and fracture mechanics. The fatigue life prediction of bridges is traditionally done using the stress life method, applicable for HCF (high-cycle fatigue). For this method an accurate prediction must be made of the relevant loads and can thus be combined with measured data to provide a more accurate prediction. Because the material characteristics, stress history and environment conditions are all uncertain in nature, the result of this analysis is always probabilistic, magnifying the need for SHM (structural health monitoring), even before reaching the theoretical stress life limit. When a crack has initiated in a detail the structural quality deteriorates, causing a reduction in the fatigue life. To account for this, fracture mechanics should be implemented. Fracture mechanics is a method of analyzing crack growth after initiation, by predicting and monitoring crack propagation. This is also a probabilistic analysis depending on the load conditions and material quality.

Ideally bridge management is a continuous process in the form of a digital model, where sensor data is used to estimate stress state and predict remaining fatigue life, while updating the digital model continuously to account for crack formation or other effects that influence the structural integrity over time. As suggested by Mousavi et al. [18], the main focus points of current research into digital models for bridge management are:

- **Monitoring**: Keeping track of the current conditions of the bridge [36]
- **Prediction**: Predicting current or future load cases based on data [5]
- **Simulation**: Simulating situations that may or may not occur in the future to analyze what were to happen in these scenarios [1]
- **Lifecycle management**: Assessing current fatigue state of the bridge using data collection [17]
- **Decision Support System (DSS)**: Using data to make the digital model decide on required maintenance [14]

Current research is often focused on one or a couple of these focus points such as crack detection [9], crack propagation [12] and structural fatigue damage [25]. While ideal workflows have been designed for a functional model of a steel bridge encompassing all of the relevant points [12], achieving the required data and knowledge on the asset is often too idealized. Different methods have been used to estimate the loads, such as data driven models [5], weight class distributions [29] or strain measurements at critical locations [25]. However, there are many critical details for which the fatigue life should be managed, which cannot all be measured. To limit the data flow, one should aim for accurate load estimations in real time with a minimum number of sensors. As discovered throughout this research, a key part of the fatigue life prediction is the estimation of loads from sensor data. The scalability of different load combinations is especially challenging to predict. This research focuses on developing a methodology for real-time stress state estimation in critical details using measurement data. The method is tested through a pipeline implemented in Python code. The resulting load estimations and the stress state of the bridge throughout its lifetime have various applications. In this study, the emphasis is placed on predicting the fatigue life of the structure.

To achieve real-time knowledge of the bridge's stress state the research will deal with the following research question: **How can real-time stress states of bridge structures be derived using a combination of sensor data and computational modeling?**

In addition to the main research question, several sub questions will be explored:

- What are the relevant methods for fatigue analysis?
- How can we measure loads on a bridge?
- What would a sensor and computational modeling setup look like?
- How can we optimize the number of sensors and the size of initial FEM computations to achieve an ideal balance between estimation accuracy and computational efficiency?
- How can the real-time stress state be used to predict fatigue life?

Chapter 2 outlines the background of fatigue analysis. Chapter 3 provides a detailed description of the situation of loads and fatigue checks of steel bridges. Following this in Chapter 4 the designed methodology will be presented that is used to estimate loads using the concept of "fingerprinting". Next in Chapter 7 the measurement setup will be shown and the captured data will be described. Chapter 8 will present the results from the analysis. Finally in Chapter 9 the research question will be answered and suggestions will be given on future research fields.

2
Fatigue Calculation Methods

During the design of steel bridges multiple failure mechanisms have to be taken into account. One of these is fatigue, which is due to the accumulated effect of many load cycles below the yield strength of the material. Ultimately, repeated smaller loads can cause failure. Under cyclic loading the material will ultimately initiate cracks, which will continue to grow until failure, unless the bridge structure is repaired. This chapter outlines the fatigue calculation methods employed in the research and explains the rationale for emphasizing real-time stress state estimation.

2.1. Stress-Life

The earliest stage of fatigue is classically described by the stress-life model. This describes high-cycle fatigue (HCF), that is fatigue accumulated by over 10,000 cycles of a process. It is mostly used for long fatigue life prediction with elastic stresses and strains [37]. No distinction is made between crack initiation and propagation in this method. Instead, the entire process deals with the total estimated life until failure.

As described by the standards, the coupling between maximum cycles and stress ranges is done using S-N curves, influenced by the detail category [22]. An S-N curve is determined by doing a series of cyclic load tests on a test piece. For different stress loads this piece will be cyclically loaded until failure occurs. This process is repeated for different stress ranges. The results are plotted to form the S-N curve for a specific material and detail category. An example of such a S-N curve is shown in Figure 2.1 for structural steel.

To determine the fatigue, the stress cycles are split into different discrete categories. Reading the maximum amount of cycles for that specific category N_x and comparing it to the applied cycles $n_{x,s}$ in detail s, provides a damage factor. Doing this procedure across the entire load spectrum gives a the total damage in a certain detail. This process is known as the Palmgren-Miner Rule. Here the total damage D_s of detail s

$$D_s = \sum_x \frac{n_{x,s}}{N_x} \tag{2.1}$$

is determined by summing damage factors over all relevant stress ranges x.

Often the most critical points in a steel structure are the welded details, where sharp angles occur. The sharp angles lead to singularities and with that unreasonably high stress values computationally. To obtain the stress cycles required to determine total stress-life fatigue damage using the Palmgren-Miner Rule, certain strategies have to be applied. Following the standard for steel bridge structures, the hot-spot stress method is used. This method approximates stresses at toe welds by applying a relationship from just outside of the weld towards the weld toe. Depending on the FEM model and type of hot-spot, different approximations of hot-spot stresses apply [21]. The application of this strategy will be further explained in Chapter 4.1.2.

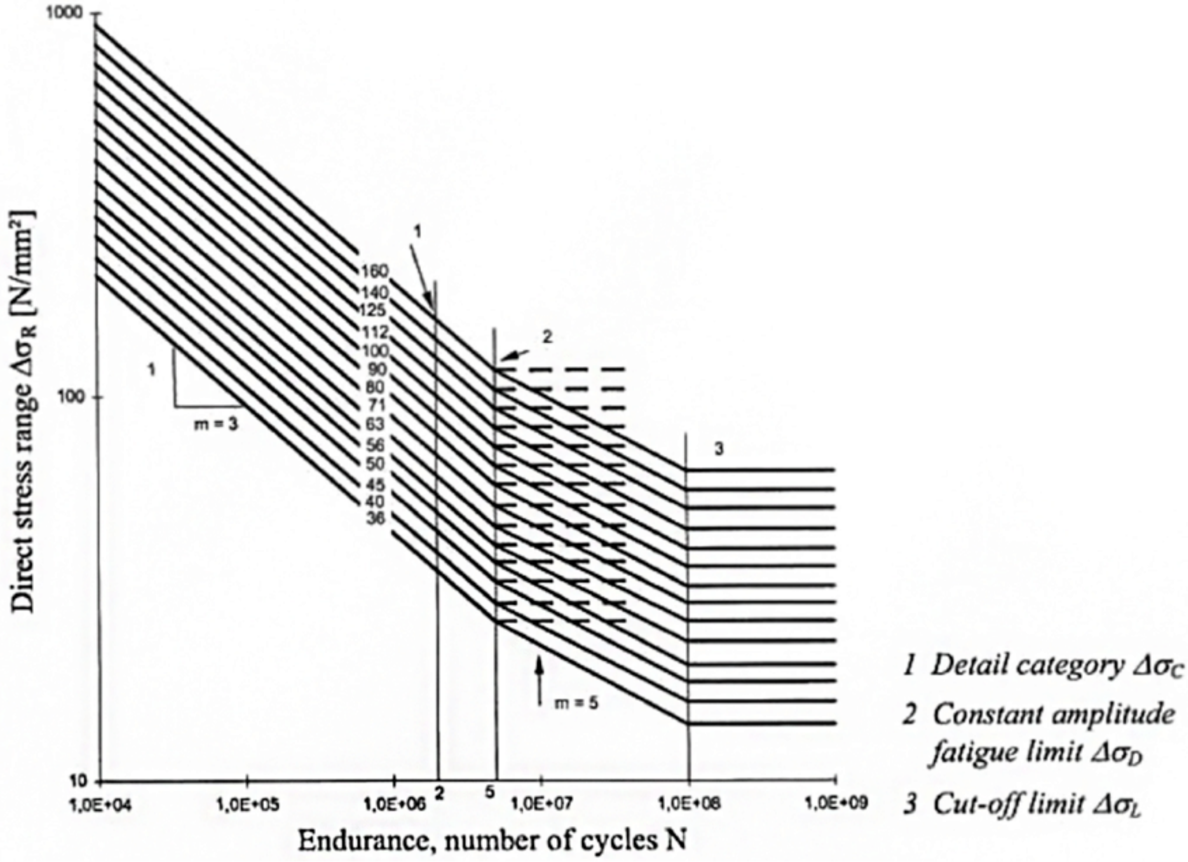


Figure 2.1: S-N curve of structural steel showing the load cycles until failure for different stress ranges [22].

2.2. Rainflow Counting

A widely used approach for determining the number of cycles $n_{x,s}$ of stress range x that have occurred in a specific detail s based on measurement data is rainflow counting. Rainflow counting, introduced by Tatsuo Endo in 1967, is a systematic method for evaluating fatigue damage from a load-time history. This chapter outlines the steps involved in implementing the rainflow counting method.

2.2.1. Hysteresis Filtering

The first step within rainflow counting is hysteresis filtering. This process functions to clean up the signal by filtering out extremely small fluctuations in the stress level. This reduces the number of iterations required in further steps of the rainflow counting. The process of hysteresis filtering starts by defining the gate, which is the minimum change in stress value that we want to consider in the further analysis. Depending on the material and process, this gate may take a different size. The gate should be chosen such that stress cycles that would contribute to fatigue damage are retained. For this reason a hysteresis filter matching the cut-off value for stress ranges is a suitable gate size.

When the gate size is determined, we can apply the hysteresis filtering process. The signal in Figure 2.2 a shows a stress profile which we want to filter. The hysteresis gate is placed on each measurement point to check if the next value falls within the hysteresis gate or not. Both positive and negative differences in stress are tested in this process. The hysteresis gates for which the next data point falls within the gate are marked in yellow (Figure 2.2 b). If this is not the case, the hysteresis gate is green. Then all of the points that fall within the hysteresis gate from the previous point are filtered out, resulting in the filtered stress profile seen in Figure 2.2 c.

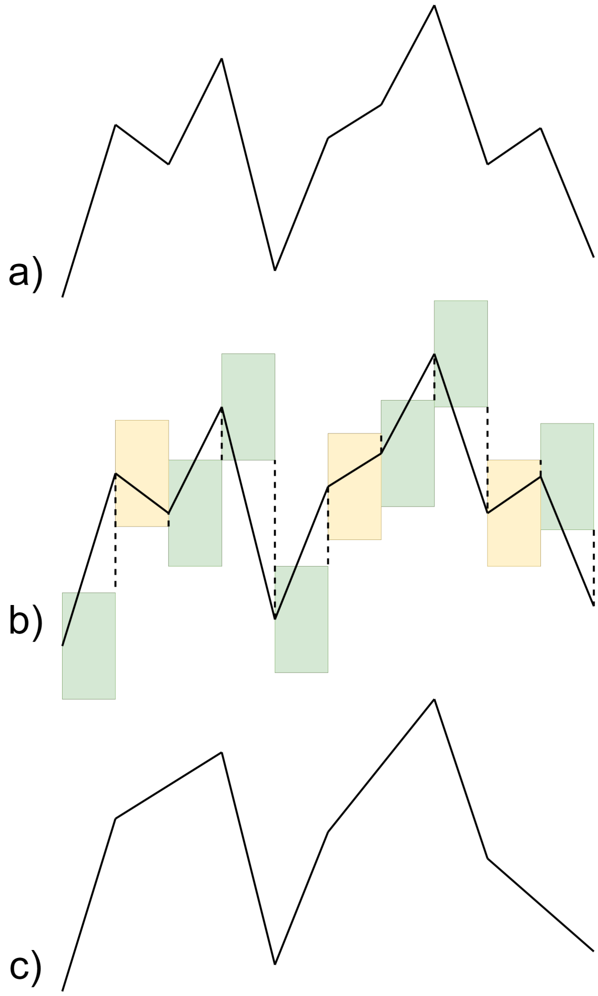


Figure 2.2: Process of hysteresis filtering. a) The original stress profile before hysteresis filtering is applied. b) The original stress profile with the hysteresis gate overlaid on each data point. c) The stress profile after hysteresis filtering has been applied.

2.2.2. Peak-Valley Filtering

The second step of rainflow counting is peak-valley filtering. The relevant aspects of a stress profile for fatigue are the local maxima and minima, rather than the path between them. This step filters out all of the stress values for which the neighboring points are either both lower or higher. The resulting stress profile from the last step will be used again here, seen in Figure 2.3 a. After identifying all of the peaks and valleys the points are marked to be deleted as seen in Figure 2.3 b. These points are removed and the remaining stress profile is connected again (Figure 2.3 c).

2.2.3. Binning

The stress values of the profile are currently continuous. The Palgrem-Miner rule as seen in Equation 2.1 requires categorization of the stress ranges in the profile. For this reason, binning has to be applied as a method of discretization.

We choose the number of bins sufficiently high to avoid large rounding inaccuracies. The binning process starts with the continuous stress values in Figure 2.4 a. For each stress value we determine the bin index. In Figure 2.4 b the binning process is applied and the used bins visualized in green. The stress values now obtain the characteristic value of their respective bin, which is the average value of the bin boundaries.

2.2.4. Four-Point Cycle Counting

The last step within the rainflow counting method is to count the stress ranges. For this, the four-point counting method is used. This method iteratively goes through four-point combinations to determine if a full stress cycle has occurred within these points. If a full stress cycle occurred, the two center points are removed from the profile and the next iteration step starts. This method identifies all full cycles in the profile.

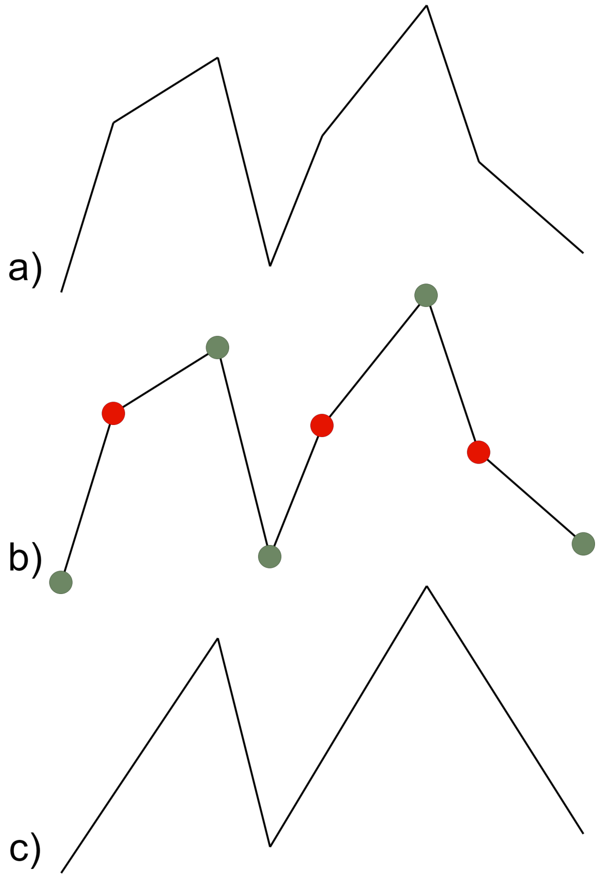


Figure 2.3: Process of peak-valley filtering. a) The original stress profile before peak-valley filtering is applied. b) The original stress profile with the peaks and valleys marked with green dots. The points to be filtered out are marked with red dots. c) The stress profile after hysteresis filtering has been applied.

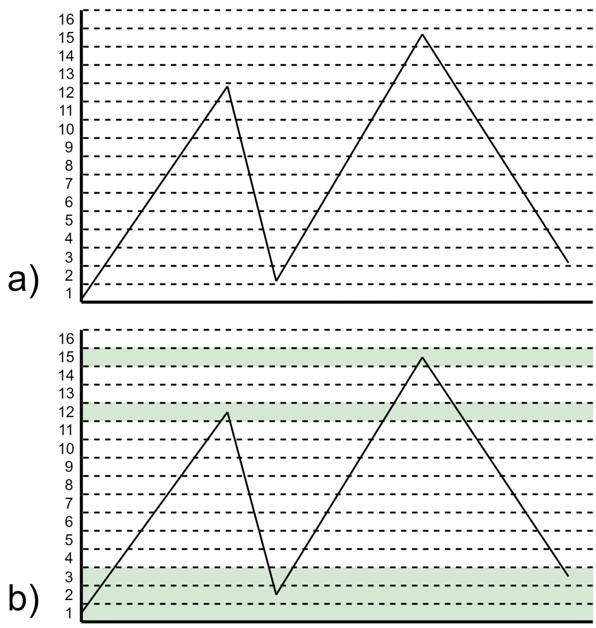


Figure 2.4: Process of stress discretization through binning. a) The original stress profile before binning is applied. b) The stress profile after binning is applied, with the used bins marked in green.

The four points that are used for the four-point counting are named A, B, C and D respectively. Figure 2.5 outlines the process on a stress profile:

- In step 2.5 a, the first four points are analyzed. Point B has a lower stress value than point D, and point C has a higher stress value than point A. This means that a full stress cycle has occurred between A and D. The stress range of this cycle is the stress difference between B and C, as marked in green. This stress-range value is stored and the points B and C are removed from the profile.
- In step 2.5 b, four new points A, B, C, and D are considered. This time point B has a higher stress value than point D, and point C has a higher stress value than point A. This means that no full stress cycle has occurred between points A and D.
- Because no stress cycle was previously found, step 2.5 c identifies the next combination of four points. This time point B has a lower stress value than point D, and point C has a lower stress value than point A. This means that no full stress cycle has occurred between points A and D.
- Step 2.5 d again identifies the next combination of four points. This time point B has a lower stress value than point D, and point C has a higher stress value than point A. This means that a full stress cycle has occurred with a stress range equal to the stress difference between B and C, as marked in green. Points B and C are removed.
- In step 2.5 e, four new points are considered. This time point B has a higher stress value than point D, and point C has a higher stress value than point A. This means that no full stress cycle has occurred.
- In step 2.5 f, new points are defined. This time point B has a higher stress value than point D, and point C has a lower stress value than point A. This means that a full stress cycle has occurred between points A and D, as marked in green.
- In step 2.5 g, there are not enough data points remaining to execute the four-step counting method. These remaining values will be counted as half cycles instead of full cycles.

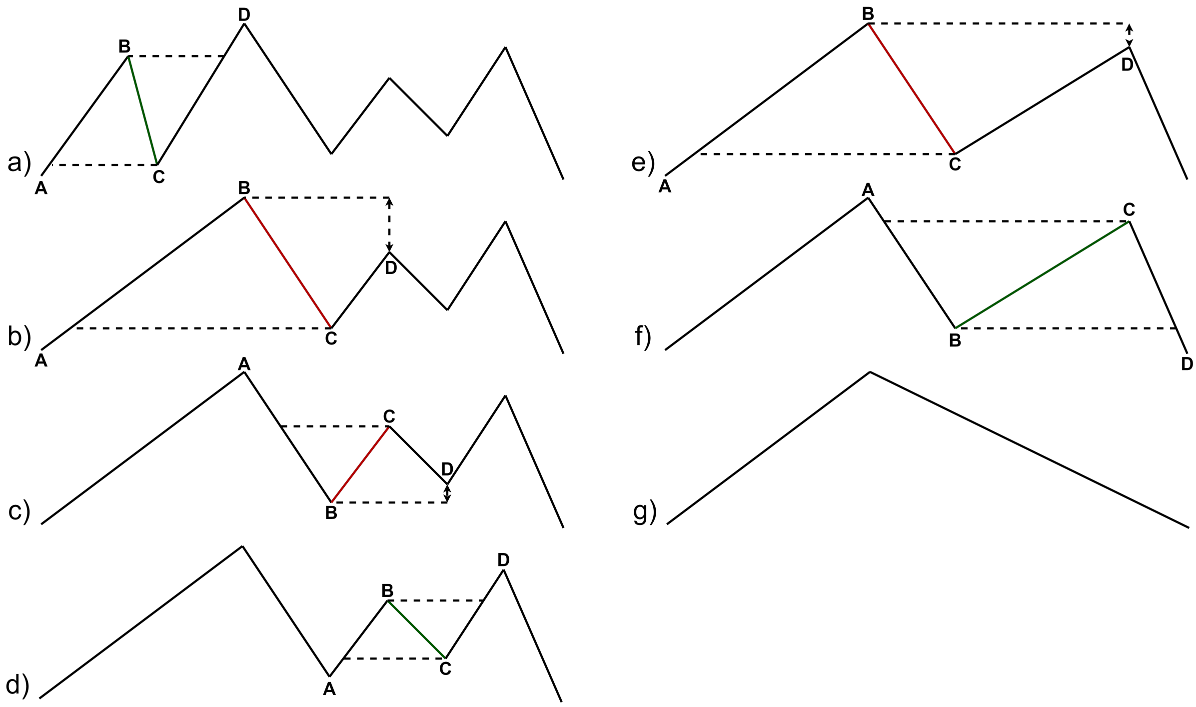


Figure 2.5: The four-point counting strategy as part of the rainflow counting method. The different figures show steps within the algorithm. If a full stress cycle is found between the stress values, the stress range is marked in green (a, d, f). If no full stress cycle is found it is marked in red instead (b, c, e). The remainder from this operation are counted as half cycles (g).

2.3. Obtaining Required Information for Fatigue Analysis

The method proposed in this research will determine real-time stress states in all researched details, necessary for fatigue analysis using rainflow counting and the stress-life approach. In Chapter 4 this methodology will be presented, focusing on real-time load and stress estimation.

Fatigue can occur in many parts of the structure, so the analysis must cover all critical points. However, placing sensors at every possible critical point is not practical, especially since many of these areas are hard to access. This means alternative methods are needed to ensure the fatigue analysis is accurate and complete.

To overcome this challenge, an alternative approach is employed: a limited number of strategically placed sensors are used to capture the relevant data. This data is then integrated with computational modeling techniques to derive loading present on the bridge. These loads can then be used to estimate stress values across the entire bridge structure. The critical element in determining accurate stress states is a precise understanding of the loads applied to the bridge. By knowing the number, location, and weight of the vehicles traveling on the bridge, it becomes possible to estimate the stress distribution throughout the structure using a FEM model and, from this, calculate the resulting fatigue damage. Consequently, the research strategy focuses on accurately identifying the load conditions affecting the bridge, which is vital for conducting precise and reliable fatigue analysis. A methodology for determining both the loads and the stress state of the bridge is thus developed.

3
Situation Description

To determine the stress state across a bridge, it is essential to first understand the load state of the structure. This chapter will outline the various types of loads that act on bridges and identify which of these will be analyzed in this research. It will also review how current standards describe load scenarios for designing bridges against fatigue. Finally, the chapter provides the mathematical framework for identifying live load locations and introduces an innovative method to linearize scaling effects efficiently.

3.1. Different Load Types Considered for Bridge Design

Steel bridges come in many different shapes and sizes. Many different types of loads occur on a full scale traffic or railway bridge, on a daily basis. In a full digital representation, all of these loads should be considered, or reasonably assumed to have no effect. The following are plausible load types for traffic and railway bridges, along with a consideration of whether these loads will be included in the scaled example discussed in this paper [20] [28]:

- Dead load: the self-weight of the structure. This load does not have a cyclic effect and will not cause fatigue damage.
- Live load: the load applied by the units using the structure, such as cars, trucks, trains, or pedestrians, depending on the asset's purpose. The live load will be taken into account during the analysis.
- Impact load: the dynamic factor that occurs when the vehicles and pedestrians move over the bridge. This impact load will be considered during the analysis.
- Wind loads: the forces exerted by wind on the bridge structure. These loads are typically only relevant for large bridges. Given that the test setup is indoor and relatively small, wind loading on the model will not be significant and can be safely ignored in the current analysis.
- Longitudinal forces: the force applied by vehicles when braking or accelerating. For pedestrians, these forces are also applied during normal loading as a portion of the forward motion from each footstep is transferred onto the bridge. This effect will be ignored in the current analysis.
- Centrifugal forces: the force applied by vehicles when traveling over a curved bridge. These forces are not applicable to this analysis.
- Hydraulic load: the forces exerted on a bridge when it is constructed over water. Since this analysis does not involve such a scenario, water load is not applicable.
- Thermal stresses: the effect of temperature variations causing the bridge to expand and contract. Sensor measurements will be adjusted for these temperature effects. However, the impact of thermal stresses on fatigue will be disregarded in this analysis, as the bridge is indoors.
- Seismic load: the impact of seismic activity on the bridge. However, since the bridge is not susceptible to this, it will be excluded from this analysis.
- Erection stresses: stresses due to construction. The construction period of bridges will not be considered, so this effect will be ignored.

3.1.1. Live Load Description According to Norm

The main load type considered during this study is the live load. According to current norms bridges have to be designed for heavy loads, in the form of trucks for traffic bridges and trains for railway bridges. In the case of a traffic bridge the present loads are simplified to a frequency of trucks as seen in Figure 3.1. The trucks are then further split into different truck categories, depending on the type of traffic that uses the road (Figure 3.2).

Traffic categories		N_{obs} per year and per slow lane
1	Roads and motorways with 2 or more lanes per direction with high flow rates of lorries	$2{,}0 \times 10^6$
2	Roads and motorways with medium flow rates of lorries	$0{,}5 \times 10^6$
3	Main roads with low flow rates of lorries	$0{,}125 \times 10^6$
4	Local roads with low flow rates of lorries	$0{,}05 \times 10^6$

Figure 3.1: Indicative number of expected heavy weight vehicles per year and per lane for heavy traffic. As presented in NEN-EN 1991:2003 en [19]

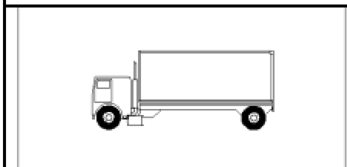
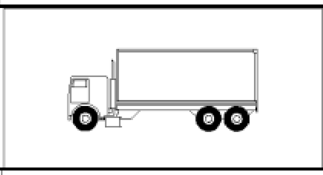
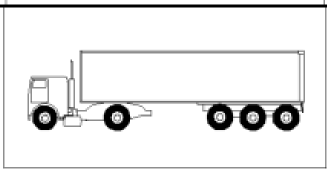
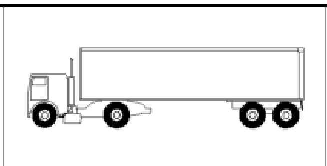
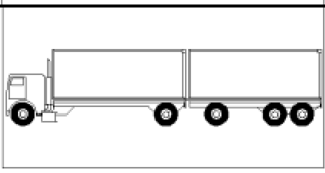
VEHICLE TYPE			TRAFFIC TYPE			
1	2	3	4	5	6	7
			Long distance	Medium distance	Local traffic	
LORRY	Axle spacing (m)	Equivalent axle loads (kN)	Lorry percentage	Lorry percentage	Lorry percentage	Wheel type
	4,5	70 130	20,0	40,0	80,0	A B
	4,20 1,30	70 120 120	5,0	10,0	5,0	A B B
	3,20 5,20 1,30 1,30	70 150 90 90 90	50,0	30,0	5,0	A B C C C
	3,40 6,00 1,80	70 140 90 90	15,0	15,0	5,0	A B B B
	4,80 3,60 4,40 1,30	70 130 90 80 80	10,0	5,0	5,0	A B C C C

Figure 3.2: Collection of equivalent trucks. As presented in NEN-EN 1991:2003 en [19]

The estimates for truck frequency and categories provide a guideline, but are very overgeneralized. The usage of each bridge is different, as is the traffic composition. Furthermore, the composition of traffic changes over time. For example, throughout the past years traffic has become heavier [34] and the prognosis is it will become even more frequent [27]. To better predict remaining fatigue life it is desired to get real-time data from measurements of the bridge and extrapolate to past and future.

3.2. Requirements for Measuring Loads

Various precise measurements can be made on a structure. When deformation and/or stresses on a number of locations are known, it is possible to derive the present load from these. Let us consider a simplified bridge model, where the only load is a vertical downwards live load. For this situation the amount of unknown parameters influencing the stress state of the bridge is three per load (a, b, F) as seen in Figure 3.3. In this example, a vertical load exerted by a car is modeled as an area load at each wheel's contact point on the bridge deck, represented as a single grid point.

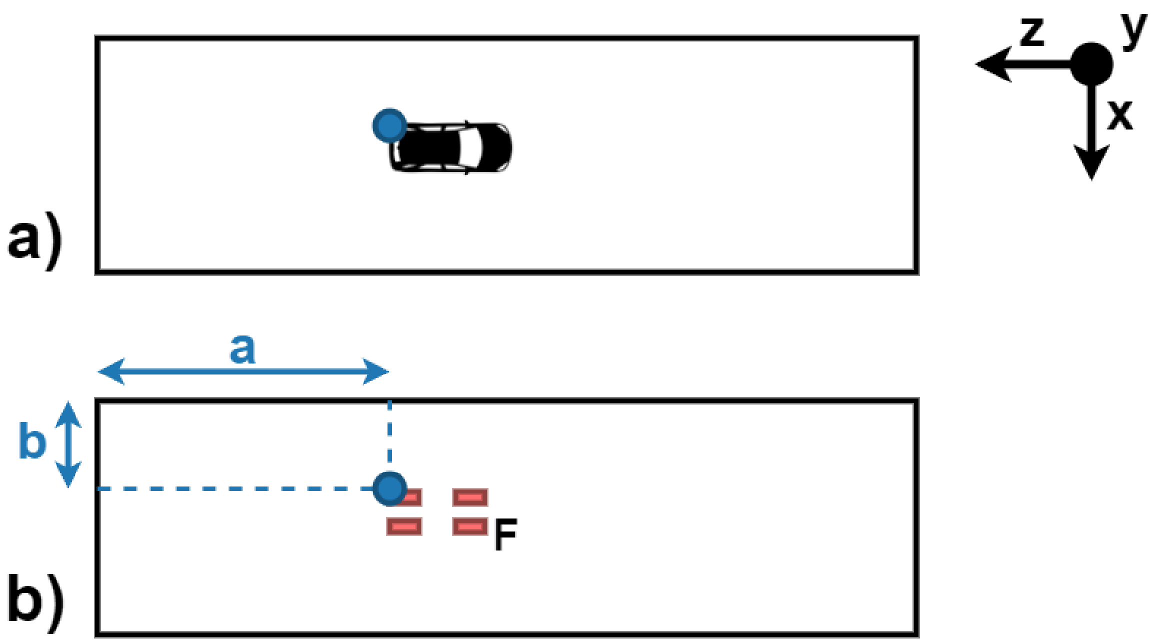


Figure 3.3: Top view of the asset showing the parameter description for a single live load example, where a is the length coordinate of the load, b is the width coordinate of the load and F is the magnitude of the load. These parameters describe the position and force magnitude of a load area, which in this case are the contact points of the wheels with the bridge deck, as marked in red. The top corner point with minimum a and b coordinates is defined as the grid point representing this load area, as marked with the blue dot.

As discussed in Chapter 2, the stress cycles are required to determine fatigue damage. This means that the stress state of the bridge has to be known at a frequency that captures the resulting cyclic behavior of the applied load. A frequency of 5 Hz is at least required to capture the passage of a truck [32]. To more precisely capture peaks, 10 Hz has previously been used for a road bridge [10]. It is desirable to keep the frequency as low as possible to minimize incoming data streams. To capture the stress state of the entire bridge, there would have to be an enormous amount of sensors, which are expensive to apply. A better strategy is to combine both digital information and limited sensor information to estimate the stress state in the entire bridge. An optimum has to be found between costs and accuracy.

Let us then consider an example (Figure 3.4) where a single car is present on a bridge with 8 attached strain sensors to capture the bridge's behavior. Not having a known number of loads or set locations, there are infinite combinations of loads that can cause these exact strain values. In all these solutions the strain values at the sensor locations are the exact same, but the stress state in the entire bridge is different for all but the real load situation.

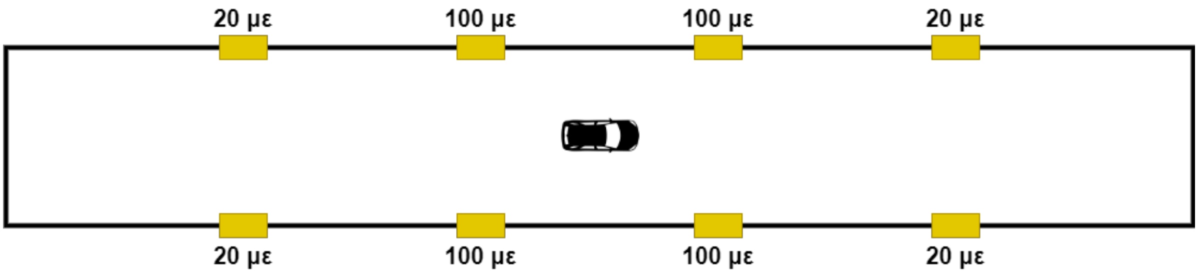


Figure 3.4: Example load of a single car on a bridge with respective strain sensor (yellow rectangles) measurement values.

It is impossible to compute each load situation in a digital model before implementation. However, the linearity of the material allows for linear scaling and superposition. This means that both magnitudes and different load situations can be computed independently and combined if necessary. This significantly reduces the required set of precomputed loads in a database. Assuming that a FEM model of the bridge is loaded with a single known load, representing that of a truck at a known location. This allows us to save the desired stress values of any location of the bridge. If we repeat this process for different locations we get a database of test loads and strain fields. The process of creating a grid of data points is further described in Chapter 4.3.

The load of a vehicle at some continuous location on the bridge can now be approximated as a load on the nearest grid point, linearly scaled for the impact of the vehicle. This reduces the solution space to a finite one. From a measurement of the sensor strain values, the first step in estimating the stress state of the bridge is determining the location of the loads. Without additional information, we do not even know the number of loads n_{loads} on the bridge. The total size Σ of the solution space for placing loads on the N_{GP} grid points is

$$\Sigma = \binom{N_{\text{GP}}}{1} + \binom{N_{\text{GP}}}{2} + \binom{N_{\text{GP}}}{3} + \ldots + \binom{N_{\text{GP}}}{N_{\text{GP}}}. \tag{3.1}$$

If there are 50 grid points, this would entail a bizarre amount of combinations of $\Sigma = 1.1 \times 10^{15}$. A couple of these combinations can be seen in Figure 3.5. Finding the optimum of this many possibilities is unrealistic within the allocated time window of $1/(10\ \text{Hz})$ to achieve a real-time stress state.

A strategy to find a quick solution for the load distribution, explored in this research, is to determine the number of loads n_{loads} using a camera with image recognition. Using a separately determined n_{loads} limits the solution space to a slightly more reasonable value of $50 \leq \binom{50}{n_{\text{loads}}} \leq 1.3 \times 10^{14}$ depending on the value of n_{loads}.

To further reduce the solution space we can also use the fact that image recognition can be used to estimate the location. We can then limit the searched combinations to the grid points within a certain boundary around the estimated location. If the boundary box contains for example 10 grid points per load, then the size of the solution space for placing loads on the N_{GP} grid points is reduced to $10^{n_{\text{loads}}}$. This still scales exponentially with the number of loads. Exponential scaling would quickly lead to the computations not being able to keep up with the real-time. For this reason a final strategy is used to optimize the location of each load independently while keeping the others fixed. This reduces the problem to n_{loads} optimizations with 10 possible solutions each. This strategy will be explained in more detail in Chapter 4.5.

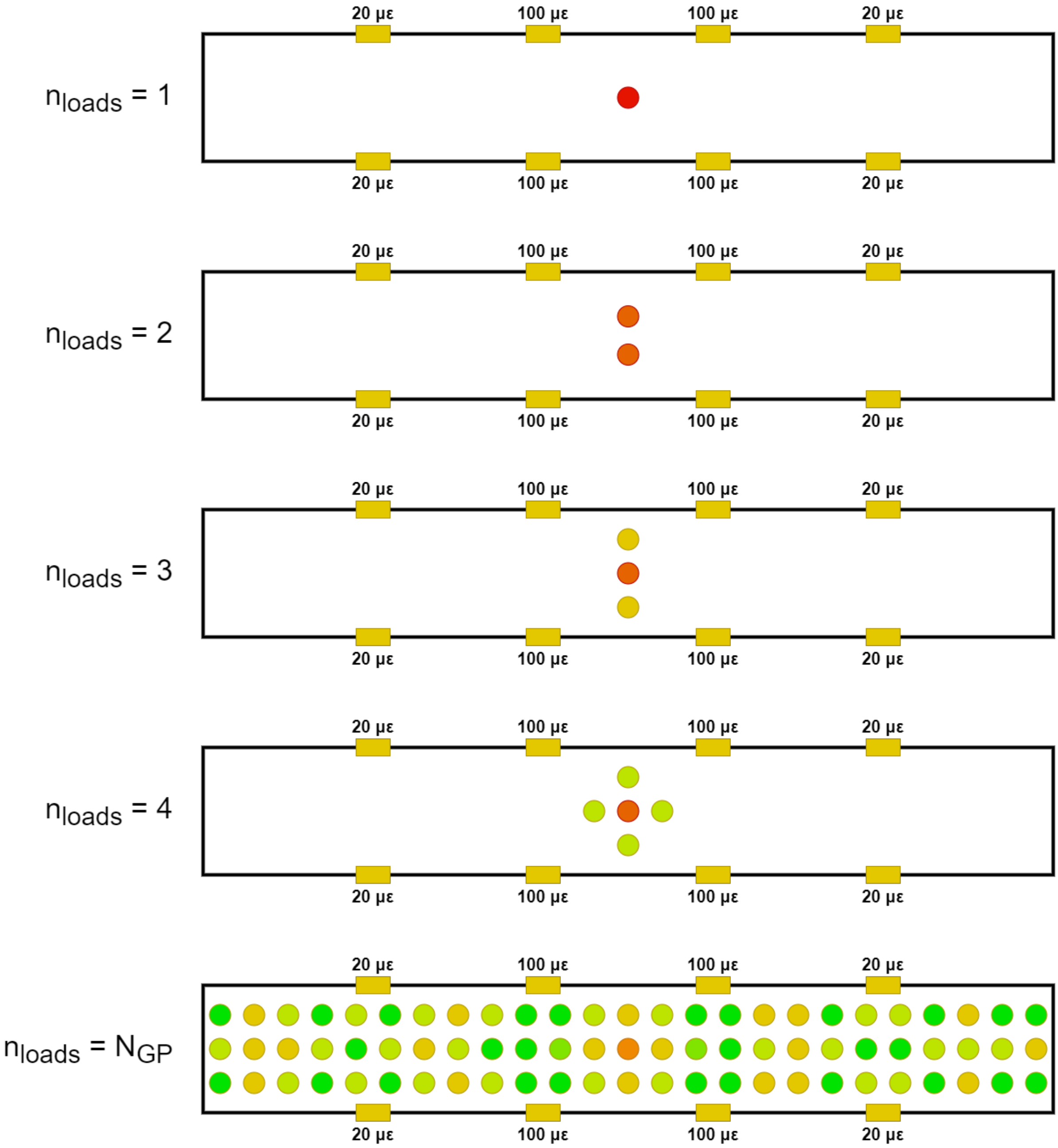


Figure 3.5: Options for different values of n_{loads} resulting in equal strain measurements in the sensors (yellow rectangles) by scaling the loads.

3.3. Proposed Pedestrian Bridge Model Setup

For easier access, and to exclude irrelevant data following from loads unrelated to traffic load as much as possible, we chose not to experiment on an existing bridge, but on a smaller asset. In addition, this has the advantage that the setup is portable to show the working principles at different locations. This alternative asset has a similar principle of application and the methodology developed and tested should be scalable to a full-scale traffic or railway bridge.

The chosen asset is a 3 m long aluminium pedestrian bridge. This is a relatively light object, and provides the necessary portability, as well as being relatively cheap to use for testing. The bridge can be seen in Figure 3.6.

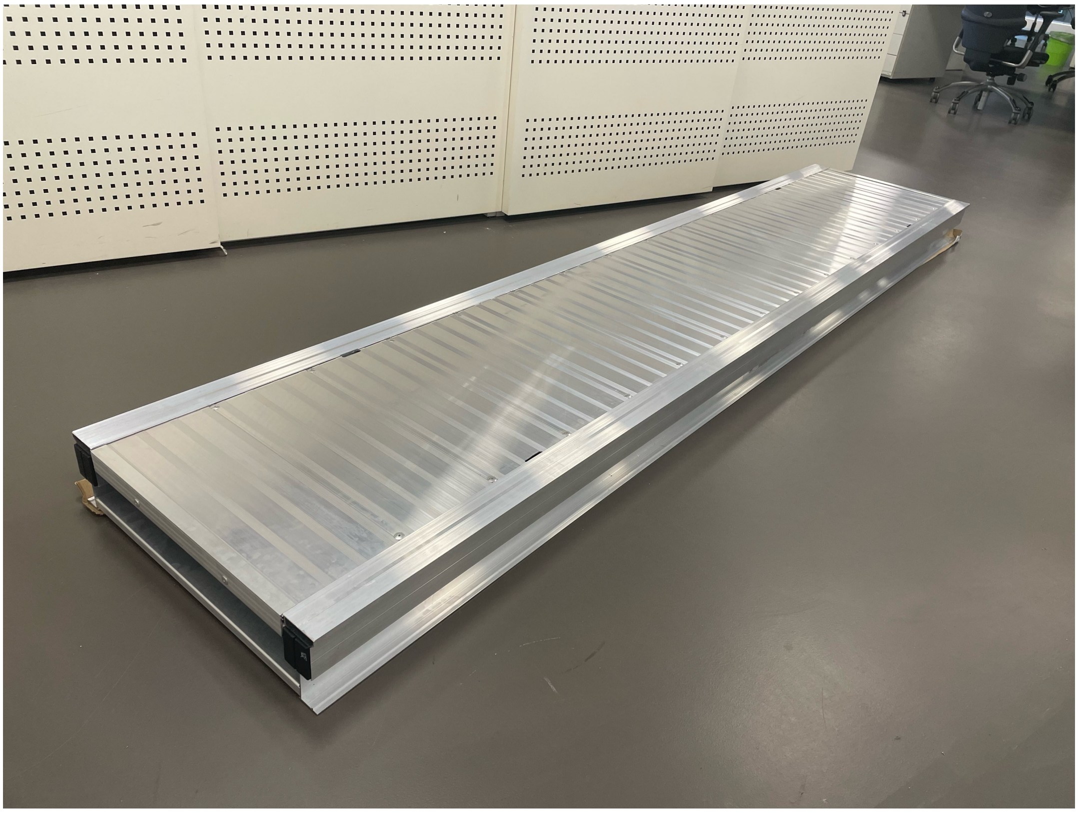


Figure 3.6: Physical asset used for data collection and testing.

3.3.1. Pedestrian Load Assumptions

For the loads of the pedestrians a rectangular area is considered the size of an average Dutch male footprint, as the vast majority of colleagues are Dutch males. For Dutch men the average shoe size is said to be 43 [2] [33]. This corresponds to a foot length of about $270 - 280$ mm [39] [15] [31]. With some margin for the actual shoe, a footprint length of 280 mm is considered. For European males this corresponds to a mean foot width of 105 mm [13]. This area is applied statically in the FEM model, but through time iterations will reflect dynamic behavior of loading and unloading points of the bridge. Based on this the following effects are disregarded: shoe types and the rolling motion of a footstep. Another assumption is that the longitudinal and lateral forces applied during walking loads are negligible compared to the vertical loads.

3.3.2. Structural Quality Assumptions

Material quality is never perfect, and depending on the individual asset can show local imperfections before operation. The material quality will be set to industry standard values in the FEM calculations. In addition to this, the stress distribution in a structure may also be influenced by its degradation. For instance, continuous use of a road deck can degrade the quality of the top layer, leading to cracks or potholes. These imperfections result in increased impact loads because of a higher dynamic factor, as well as increasing the effect of different axle loads [38]. Other effects that have to do with the degradation of the sub- or superstructure of the bridge are rusting, erosion and crack formation. These conditions will result in a weaker material that is more susceptible to fatigue. In this study, however, these effects will be disregarded, as there is insufficient time for them to occur. As a result, the bridge structure is assumed to follow linear deformation at all times.

4

Methodology for Load and Stress State Estimation

4.1. Designing a Digital Version of the Asset

The industry standard for modeling of steel structures is using FEM. For this analysis the FEM software package Ansys 2023 R2 is used. The FEM model will contain the global bridge model in a coarser model to capture the global behavior to applied loads. In addition, all researched details as mentioned in Chapter 4.1.2 are modeled in separate sub-models. These sub-models have a finer mesh to capture the local behavior around the welds sufficiently accurate.

4.1.1. Global FEM Model

The bridge consists of the main aluminium profiles, which are connected by the walking deck and six cross girders. The decks are welded to the profiles in addition to the existing rivets, to reduce vibrations. The bridge has a connection beam at either end of the bridge to raise the bottom flanges from the ground and allow for bending of the main girders. The FEM model of the global model can be seen in Figure 4.1

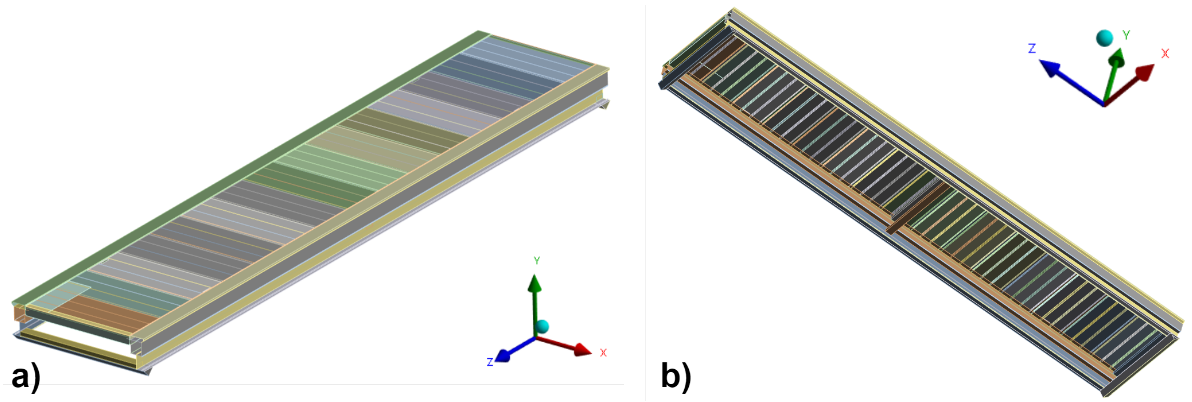


Figure 4.1: a) Top view of the bridge as modeled in Ansys 2023 R2, b) Bottom view of the bridge as modeled in Ansys 2023 R2.

Strain Sensors

Strain sensors are attached at multiple locations on the asset. The sensors are modeled at matching locations on the digital model. This is done in order to link the asset measurements to FEM results. The sensor positions are visualized in Figure 4.2. The used sensors are fiber Bragg grating (FBG) sensors. The cable length and desired amount of sensor points are determined in Chapter 6.3. Other considerations are the limited availability of different sensor configurations, budget and time frame. These strain sensors provide a single strain value ϵ_i in the length direction of the bridge for all sensors $i \in I$.

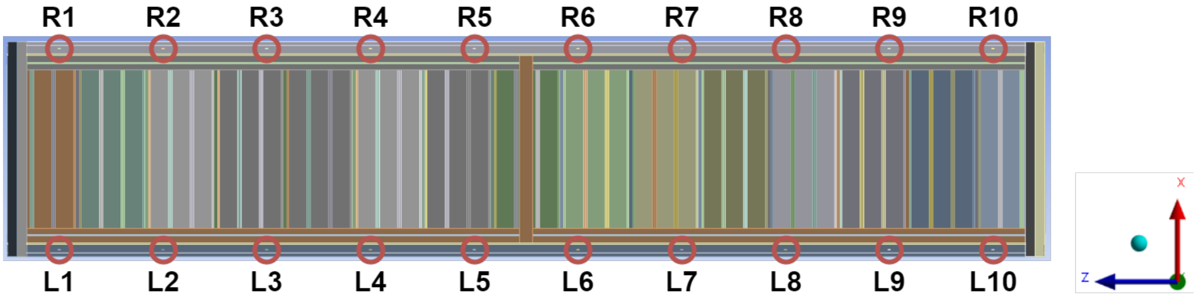


Figure 4.2: Bottom view of the bridge, showing the digital strain sensor measurement locations matching the physical sensor locations.

4.1.2. Detail Models

Each detail is modeled within a sub-model for more detailed material behavior around welds. An example sub-model is shown in Figure 4.3. The cut-off boundaries are chosen at a far enough distance from the weld to eliminate inaccuracies at the boundaries, but as close as possible to reduce the number of elements and thus computational time.

Figure 4.3: a) Global FEM model highlighting one sub-model (circled in red). b) Picture of the researched sub-model on the asset. c) Sub-model showing the adjusted weld plate thickness from the weld.

Adjustment for weld plate thickness

Welds are essentially added material, and thus provide extra stiffness to the plates. Plates are modeled with increased plate thickness to account for the increased stiffness. This is done according to the standard [24], with different recommendations based on the applicable weld type. An example of this is a tee joint single fillet weld, as seen in Figure 4.4 *h*, which appears on the cross girder connections to the aluminium profiles. This thickness adjustment is used on the detail shown in Figure 4.3 *c*.

Hot-spot calculation in detail

As discussed in Chapter 4.1.2, the area around a weld is most prone to fatigue damage, and to analyze this behavior the stress has to be determined at the weld toe. Due to the abrupt change in geometry at the welded connections, peaks will appear in the stress output of the FEM model at these points. To account for this the hot-spot stress method is used as suggested by the standard [21]. This calculation strategy entails combining values further away from the singularity and approximating the supposed value at the weld toe through extrapolation. For different types of hot-spots and mesh densities the hot-spot stress value is approximated using Table 4.1.

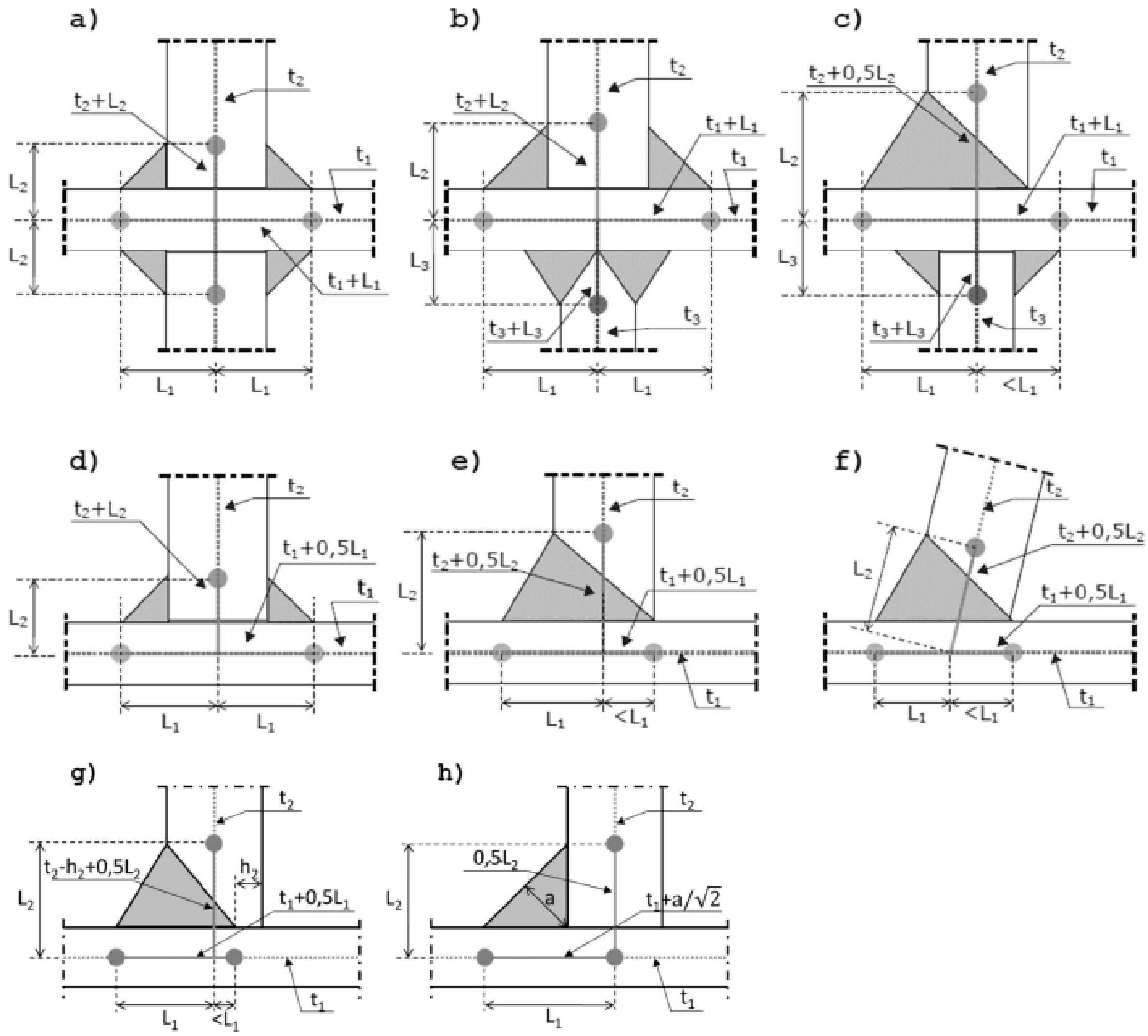


Figure 4.4: Plate thickness correction for different weld types [24].

Type of hot-spots point	*Linear extrapolation*		*Quadratic extrapolation*
	Fine mesh	*Coarse mesh*	*Fine mesh*
type "a"	0.4t and 1.0t	0.5t and 1.5t	0.4t, 0.9t and 1.4t
	$1.67\,\sigma_{0.4\mathrm{t}} - 0.67\,\sigma_{1.0\mathrm{t}}$	$1.5\,\sigma_{0.5\mathrm{t}} - 0.5\,\sigma_{1.5\mathrm{t}}$	$2.52\,\sigma_{0.4\mathrm{t}} - 2.24\,\sigma_{0.9\mathrm{t}} - 0.72\,\sigma_{1.4\mathrm{t}}$
type "b"	–	5 mm and 15 mm	4, 8 and 12 mm
	–	$1.5\,\sigma_{5\mathrm{mm}} - 0.5\,\sigma_{15\mathrm{mm}}$	$3\,\sigma_{4\mathrm{mm}} - 3\,\sigma_{8\mathrm{mm}} + \sigma_{12\mathrm{mm}}$

Table 4.1: Hot-spot stress formula for different types of hot-spots and mesh densities [21].

The type of hot-spots is of type “a”, with a linear extrapolation using a fine mesh. For this reason the hot-spot stresses are calculated

$$\sigma_{\mathrm{HS}} = 1.67\,\sigma_{0.4\mathrm{t}} - 0.67\,\sigma_{1.0\mathrm{t}} \tag{4.1}$$

by multiplying stress values obtained at a distance of 0.4 and 1.0 times the plate thickness from the weld toe with the linear extrapolation factors. These hot spot stresses are determined for each desired detail location $s \in S$. The locations are pre-identified as critical for fatigue analysis. The first 12 locations, positioned along the connections of the left profile and cross girders (DL1, DL2, DL3, DL4), are illustrated in Figure 4.5. An additional 12 locations, mirrored on the connections with the right profile (DR1, DR2, DR3, DR4), are also included. Therefore, this analysis involves calculating and estimating a total of 24 stress values, σ_s.

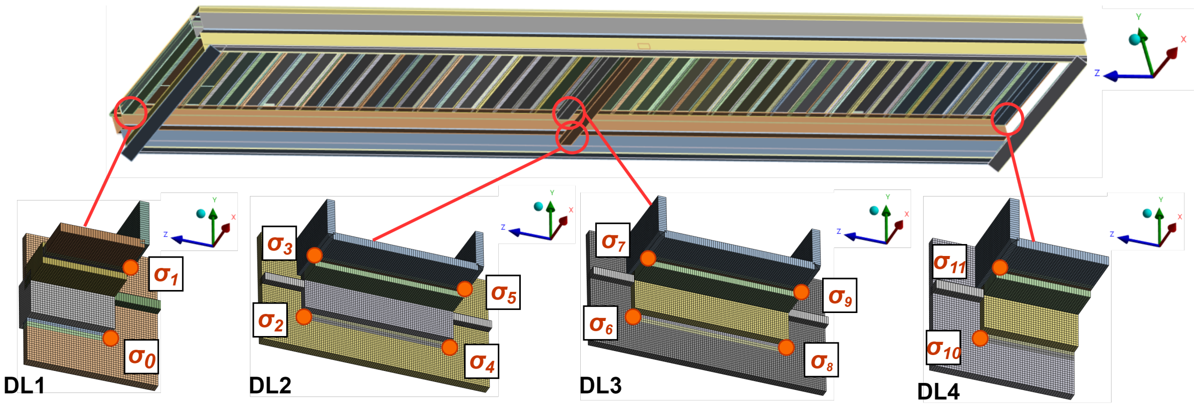


Figure 4.5: Global FEM model with 4 of the sub-model locations shown in detail (DL1, DL2, DL3, DL4). The locations where a hot-spot stress σ_s is calculated are marked with an orange dot.

4.2. Digital Representation of a Single Load Case

The first step is to consider a single load at a time. The load on the asset is a single footstep, represented as a rectangular load described using three parameters: a, the coordinate along the length of the bridge; b, the coordinate along the width of the bridge and F, the load applied by the footstep. This is further visualized in Figure 4.6. The length and width of the footstep are 280 mm and 105 mm, respectively, as described in Chapter 3.3.1. For this load case the values $\sigma_s\ \forall s \in S$ and $\epsilon_i\ \forall i \in I$ are stored.

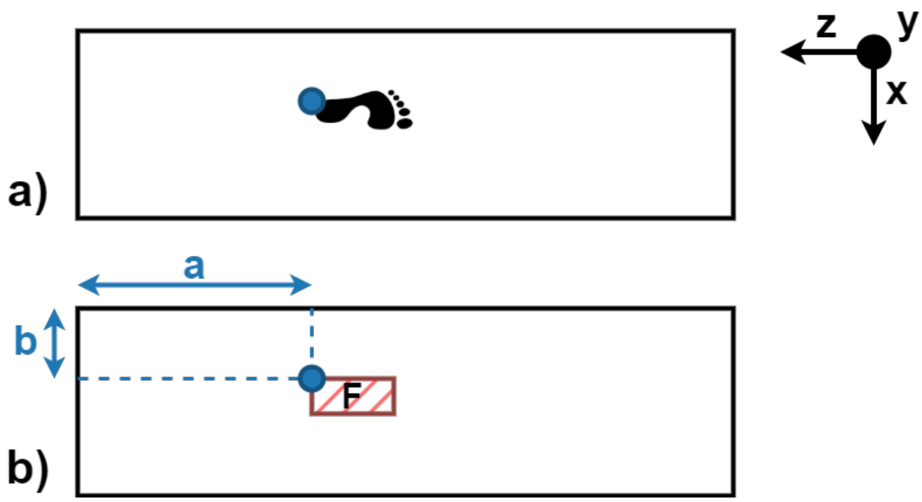


Figure 4.6: Top view of the asset showing the parameter description for a single load example, where a is the length coordinate of the load, b is the width coordinate of the load and F is the magnitude of the load. These parameters describe the load area representing the footstep, as marked in red. The top corner point with minimum a and b coordinates is defined as the grid point representing this load area, as marked with the blue dot.

4.3. Simulating a Database of Digital Load Cases

A database is set up that describes the asset's behavior for a set of test cases, which function as comparison for measurement values. The load of the test cases is kept at the same value, while only the parameters a and b are adjusted. A grid is set up with N_a grid points in the length direction and N_b grid points in the width direction. Given the maximum and minimum values of a and b, the distances between the grid points are

$$\Delta a = (a_{\text{upper}} - a_{\text{lower}})/(N_a - 1) \tag{4.2}$$

$$\Delta b = (b_{\text{upper}} - b_{\text{lower}})/(N_b - 1) \tag{4.3}$$

For each of the points in the grid the coordinates

$$a_q = a_{\text{lower}} + (q-1)\,\Delta a \quad \forall q \in 1, 2, \ldots, N_a \tag{4.4}$$

$$b_r = b_{\text{lower}} + (r-1)\,\Delta b \quad \forall r \in 1, 2, \ldots, N_b \tag{4.5}$$

are determined by their grid indices q and r in length and width direction of the bridge, respectively. Combined the coordinates are represented as (a_q, b_r). The FEM model is run for each of the grid points as described in Chapter 4.2, forming the database.

4.3.1. Resolution of Load Cases

By increasing the values of N_a and N_b, the resolution of the grid points increases. This makes the future predictions more accurate, but it comes at the cost of computational speed. As the amount of load cases to be computed in FEM

$$N_{\text{GP}} = N_a\, N_b \tag{4.6}$$

is equal to the product of the grid points in both directions. The minimum and maximum grid resolution used for the analysis are shown in Figure 4.7.

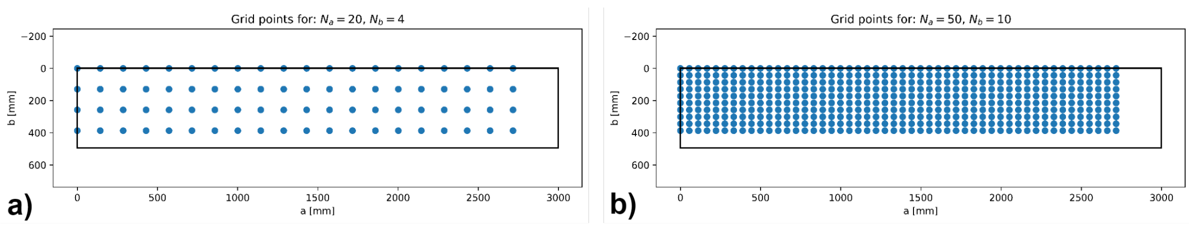


Figure 4.7: Visualization of simulated loads within the FEM model. a) Example grid configuration with a density of $N_a = 20$ and $N_b = 4$, resulting in a total resolution of $N_{\text{total}} = 80$, b) Example grid configuration with a density of $N_a = 50$ and $N_b = 10$, resulting in a total resolution of $N_{\text{total}} = 500$.

4.4. Estimating Stress State for Single Load Cases

The new measurements obtained in real-time from the asset must now be matched to the existing knowledge of the digital model of the asset. Multiple strategies were considered to match new measurements to the created database, including machine learning. However, due to the data-hungry nature of machine learning, it was decided to use an alternative. The strategy presented in this chapter provides way more accurate results within a fraction of the calculation time. Using the strain values from the measurements of the asset, the closest matching grid point is determined from the database, a process similar to matching the fingerprint of a person in a criminal investigation, hence the name “fingerprinting”. The matching process from the database is explained in Chapter 4.4.1. Next an interpolation is made between surrounding points as explained in Chapter 4.4.2. Afterwards the test load is scaled to match the measurement as explained in Chapter 4.4.3.

4.4.1. Coupling Closest Digital Load Case

When a new measurement is obtained from the asset, where the strain sensor data is $\epsilon_i \;\; \forall i \in I$, where I is the set of active strain sensors on the asset, the most relevant grid point has to be coupled. A strain ratio difference matrix $X^{(n)}$ is set up for the measurement point compared to grid point n, where each of the matrix elements

$$X_{ij}^{(n)} = \left| \frac{\epsilon_i^{(n)}}{\epsilon_j^{(n)}} - \frac{\epsilon_i}{\epsilon_j} \right| \quad \forall i, j \in I, \forall n \in N_{\text{GP}} \tag{4.7}$$

Here, $\epsilon_i^{(n)}$ is the strain on sensor i due to a load on grid point n. $X_{ij}^{(n)}$ are individually determined for each of the sensor combinations.

To reduce the effect of explosive outliers that arise from near-zero divisions, multiple mathematical operations were considered, such as average, median, median absolute deviation around the median, median absolute deviation around the average. The MAD (Median Absolute Deviation) [4] around the median preformed best for location predictions and will thus be used.

The median

$$\tilde{X}^{(n)} = \text{med}\left(X_{ij}^{(n)} \quad \forall i, j \in I\right) \quad \forall n \in N_{\text{GP}} \tag{4.8}$$

is taken of the difference matrix $X_{ij}^{(n)}$.

After this the MAD operation is performed

$$\text{MAD}^{(n)} = \text{med}\left(\left|X_{ij}^{(n)} - \tilde{X}^{(n)}\right| \quad \forall i, j \in I\right) \quad \forall n \in N_{\text{GP}} \tag{4.9}$$

on the median difference matrix.

To choose the grid point n_{min} that best matches the strain ratios of the measurement point from the asset, the minimal MAD value is determined

$$n_{\text{min}} = \underset{n}{\text{argmin}}\, \text{MAD}^{(n)} \tag{4.10}$$

by taking the minimum value of $\text{MAD}^{(n)}$ out of all of the grid points $n \in N_{\text{GP}}$. For a new measurement with a single load present on the bridge, as shown in Figure 4.8 *a*, the coupled grid point n_{min} represents the best matching grid point, illustrated in Figure 4.8 *b*.

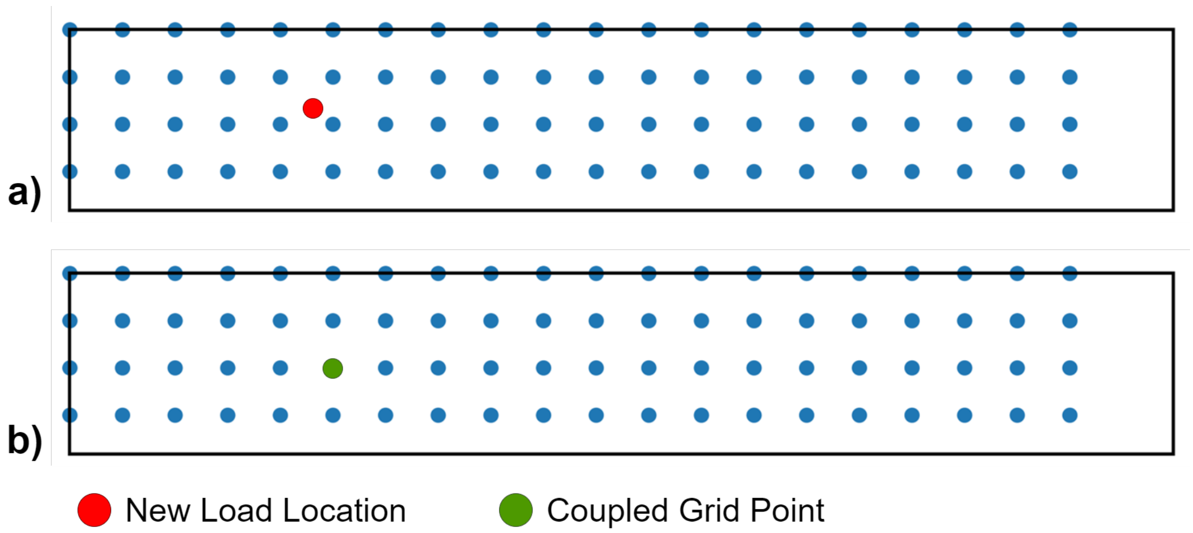


Figure 4.8: a) Example of a load location (in red) from a new measurement. b) Coupled grid point n_{min} (in green) representing the best matching grid point for the measurement..

4.4.2. Interpolation between Digital Load Cases

To achieve extra resolution from the determined database, while avoiding having to run more simulations, an interpolation is made from the coupled grid point $n_{\min}$ with the surrounding grid points. The actual location of the load is not known, meaning that it can be in any of the four quadrants around the coupled grid point.

The coordinates of the optimal grid point $n_{\min}$ are defined as $n_{\min}$: $\{a_{\min}, b_{\min}\}$. The surrounding eight grid points, visualized in Figure 4.9, are divided into the four areas formed by four grid points each. These are all at distances of Δa, Δb from each other as defined in Equations 4.4 and 4.5.

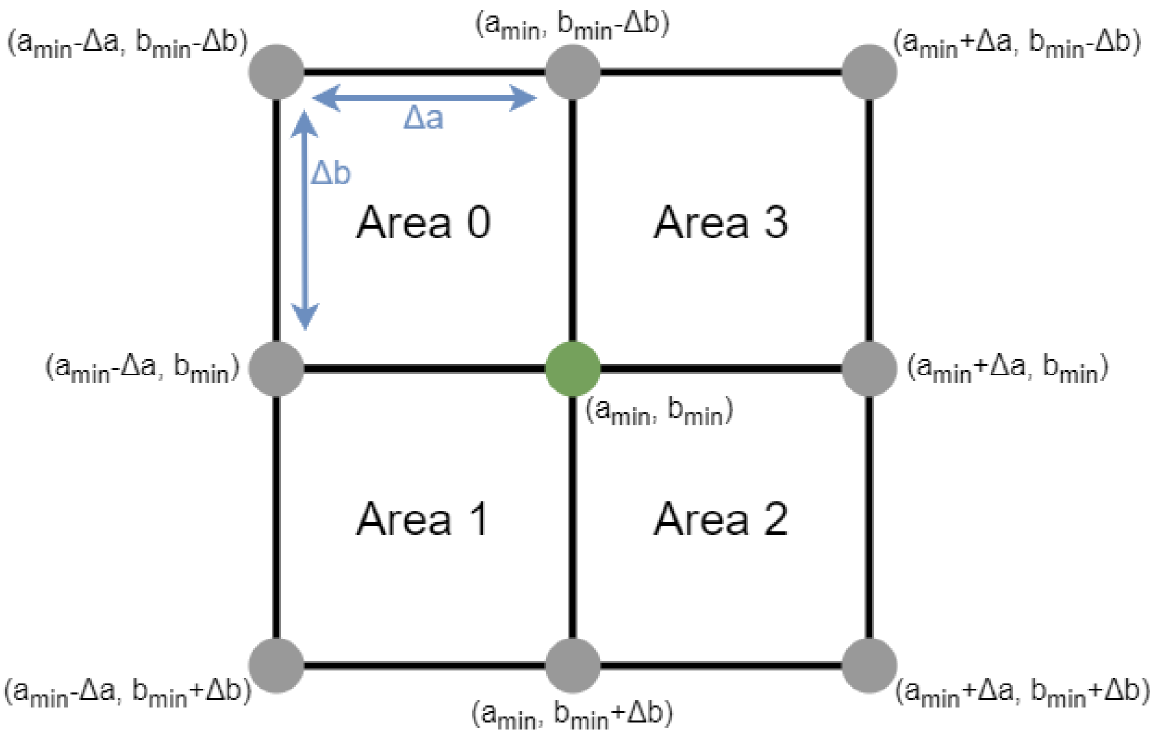


Figure 4.9: Minimal matching grid point (shown in green), with the surrounding grid points, forming four areas in which the measured load case could have been located.

For each area $h \in H$, where $H = \{0, 1, 2, 3\}$ is the set of quadrants, there are four grid points $k \in K$, where $K = \{0, 1, 2, 3\}$ is the set of grid points in the area. The coordinates of all of the grid points in their respective area are

$$a_{k,h} = a_{\min} + \Delta a \left(\left\lfloor \frac{k}{2} \right\rfloor + \left\lfloor \frac{h}{2} \right\rfloor - 1 \right) \quad \forall k \in K, \forall h \in H \tag{4.11}$$

$$b_{k,h} = b_{\min} + \Delta b \left(\frac{1 + (-1)^{\left\lfloor \frac{k-1}{2} \right\rfloor}}{2} + \frac{1 + (-1)^{\left\lfloor \frac{h-1}{2} \right\rfloor}}{2} - 1 \right) \quad \forall k \in K, \forall h \in H \tag{4.12}$$

Each of the coordinates obtained are one of the grid points from the database of N_{GP} grid points. We label the grid point with coordinates $(a_{k,h}, b_{k,h})$ as $n_{n,k}$.

We obtain a best estimate for the location of the load within one of the quadrants by scaling the strains calculated for loads on the corner grid points by factors $\beta_{k,h}$. For each quadrant h we find the optimal $\beta_{k,h}$ by minimizing

$$\text{MAD}^{(\text{h})} = \text{med} \left(\left| X_{ij}^{(h)} - \tilde{X}^{(h)} \right| \quad \forall i, j \in I \right), \quad \forall h \in H \tag{4.13}$$

subject to

$$\begin{cases} X_{ij}^{(h)} = \left| \frac{\sum_{k\in K} \beta_{k,h}\, \epsilon_i^{(n_{k,h})}}{\sum_{k\in K} \beta_{k,h}\, \epsilon_j^{(n_{k,h})}} - \frac{\epsilon_i}{\epsilon_j} \right| & \forall i,j \in I, \forall h \in H \\ \tilde{X}^{(h)} = \operatorname{med}\left(X_{ij}^{(h)} \quad \forall i,j \in I \right) & \forall h \in H \\ \beta_{k,h} \in \mathbb{R}_0^+ & \forall k \in K, \forall h \in H \\ \sum_{k\in K} \beta_{k,h} = 1 & \forall h \in H \end{cases} \tag{4.14}$$

Next, we find the overall best estimate location of the load by finding the optimal quadrant

$$h_{\text{min}} = \underset{h}{\operatorname{argmin}}\, \text{MAD}^{(h)} \tag{4.15}$$

Here quadrant h_{min} is the area in which the measured load case obtained from the asset is predicted to have occurred. For area h_{min} the scale factors

$$\beta_{k,h_{\text{min}}} = \{\beta_{0,h_{\text{min}}}, \beta_{1,h_{\text{min}}}, \beta_{2,h_{\text{min}}}, \beta_{3,h_{\text{min}}}\} \tag{4.16}$$

will then be as obtained during minimization. These represent the factors of influence of the four grid points to the total summed strain values.

An example is given in Figure 4.10 for $h_{\text{min}} = 2$ (Area 2, as seen in Figure 4.9). The predicted location of the load is shown, as determined from interpolation between the four surrounding grid points. The coordinates are

$$a_{\text{est}} = \sum_{k\in K} \beta_{k,h_{\text{min}}}\, a_{k,h_{\text{min}}} \tag{4.17}$$

$$b_{\text{est}} = \sum_{k\in K} \beta_{k,h_{\text{min}}}\, b_{k,h_{\text{min}}} \tag{4.18}$$

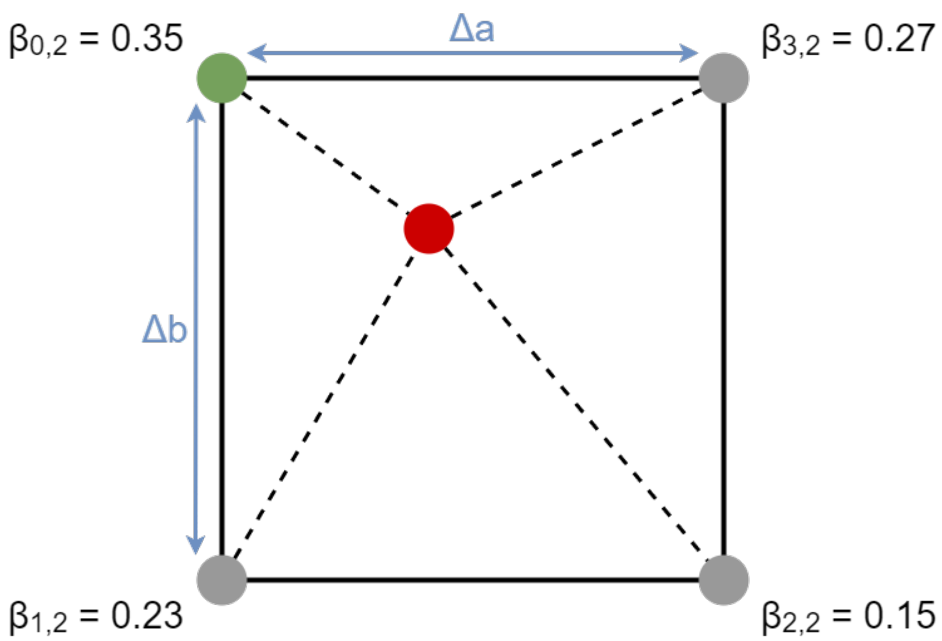


Figure 4.10: Example prediction using interpolation of four grid points in the case of $h_{min} = 2$. Where the predicted load location (shown in red), is determined by scaling the locations of the four surrounding grid points.

4.4.3. Scaling Coupled Load Case

Thus far the strain ratio between two sensors has been used to determine the predicted location of the load case obtained from the asset. To obtain the predicted load magnitude, the absolute strain will be considered. For this another scale factor, α, has to be determined. α will then represent the scale factor of the predicted load compared with the load that has been applied for the grid points in the database. Because all of the grid points are run with the same exact load, one scale factor α is sufficient.

We minimize

$$Z = \sum_{i \in I} \left| \sum_{k \in K} \left(\alpha\, \beta_{k,h_{\min}}\, \epsilon_i^{(n_{k,h_{\min}})} - \epsilon_i \right) \right| \tag{4.19}$$

subject to

$$\alpha \in \mathbb{R}_0^+ \tag{4.20}$$

Here the previously found scale factors $\beta_{k,h_{\min}}$ are used.

Scale factors $\beta_{k,h_{\min}}$ and α are then used to scale the grid point load to what is the estimated load

$$F_{\text{est}} = \alpha \sum_{k \in K} \beta_{k,h_{\min}}\, F_{k,h_{\min}} \tag{4.21}$$

applied to the asset. Here, $F_{k,h_{\min}}$ is the load applied to grid point $n_{k,h_{\min}}$ in the database. The same scale factors $\beta_{k,h_{\min}}$ and α are used to determine the estimated hot-spot stresses in all of the researched details

$$\sigma_{s,\text{est}} = \alpha \sum_{k \in K} \beta_{k,h_{\min}}\, \sigma_s^{(n_{k,h_{\min}})} \quad \forall s \in S \tag{4.22}$$

4.5. Estimating Stress State for Multiple Load Cases

For traffic bridges very often multiple loads will be present on the asset at once. This chapter will discuss the mathematical model used to predict the location and load of multi-load situations. Scalability is given special attention, such that the computation time does not become combinatorial explosive. To achieve this, cameras are used to determine the number of footsteps on the bridge as well as a rough estimate of their locations.

4.5.1. Footstep Image Recognition

Scaling is a significant problem for identifying multi-load situations. All options within the database are tested for a single load scenario. In cases involving multiple loads, every possible combination of grid points and load magnitudes would need to be evaluated to determine the optimal solution. This significantly increases the computational effort and complexity compared to single-load scenarios. Not knowing an estimate location or the number of footsteps $n_{\text{footsteps}}$ would mean a computational complexity in the order of $\mathcal{O}(|N_{\text{GP}}|^1) + \mathcal{O}(|N_{\text{GP}}|^2) + \mathcal{O}(|N_{\text{GP}}|^3) + \ldots + \mathcal{O}(|N_{\text{GP}}|^{n_{\text{footsteps}_{\max}}}) \approx \mathcal{O}(|N_{\text{GP}}|^{n_{\text{footsteps}_{\max}}})$. Cameras with image recognition are used to significantly limit computational speed by determining the number of footsteps $n_{\text{footsteps}}$ on the bridge, as well as providing an estimate of their location. This camera prediction is visualized in Figure 4.11 *b*, where the cameras predicted the number of footsteps, namely 3, and provided an estimate of their location bounded by the inaccuracy a_{box} and b_{box}. For the image recognition the shoe detection algorithm from a study about risk assessment of cane users is applied [8]. We explicitly consider the detection of feet floating in the air, thus providing no load onto the asset. The image recognition application will further be explained in Chapter 5.3.2.

4.5.2. Multi-Load Coupling

For the single load situation Equations 4.7, 4.8, 4.9 were used. However, the consideration of strain ratio from the single-load analysis does not work for the multi-load situation. It is impossible to find the location of more than one load, unless we also scale the loads correctly.

To couple the best matching grid points for multi loads all combinations of $\gamma^{(n)}$ should manually be calculated to certainly achieve the best estimate. However, the nature of this computation would be of order $\mathcal{O}(|N_{GP}|^{n_{\text{footsteps}}})$, becoming combinatorially explosive in computational time as the number of footsteps increases. This leads to computational times that would not be viable for large $n_{\text{footsteps}}$, and would cause issues especially for traffic bridges, where multiple loads are often present. For this reason the computational scaling is linearized by locking all but one of the footsteps to the grid points closest to the coordinates determined by the cameras, and searching for the optimal grid point one by one. This process is visualized in Figure 4.11. Selecting the points sequentially could mean that the guesses for the later footsteps are more accurate, meaning that running the process multiple times could result in better predictions. After testing the predictions, however, it was found that the predictions were already closer to the actual situation than the distance between grid points. Thus, for present setup, extra iterations are not required.

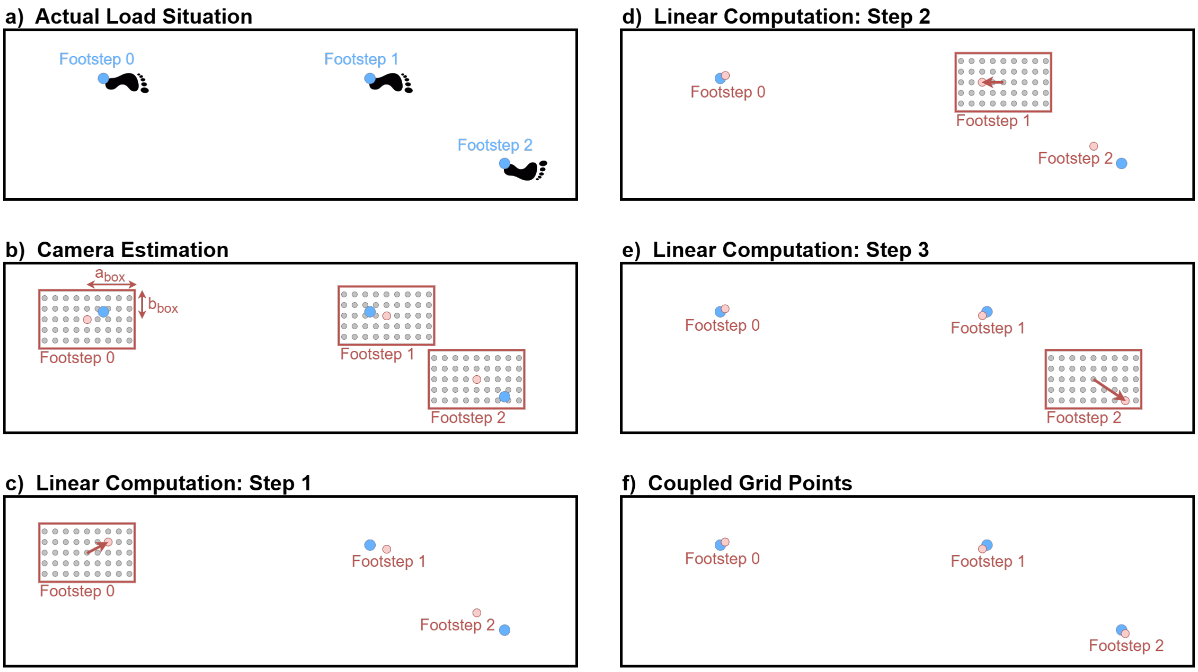


Figure 4.11: Linear computation strategy for coupling grid point in a multi-load scenario. a) Depiction of an unknown load situation at a random moment in the asset's lifetime, where $n_{\text{footsteps}} = 3$. b) Camera prediction location (red dots), with the inaccuracy bounding boxes, in which grid points are considered. c), d), e) illustrate steps 1, 2 and 3 respectively, in determining the closest matching grid point within their respective bounding boxes. f) Presents the resulting coupled grid points obtained from the linear computation.

In the multi-load algorithm we use an array $\gamma^{(n)} \quad \forall n \in N_{\text{GP}}$, where the value $\gamma^{(n)} = 0$ if grid point n has no load and $\gamma^{(n)} = 1$ if it has a load. In a scenario with $n_{\text{footsteps}}$ loads, we perform $n_{\text{footsteps}}$ steps of the following type. The initial guess for the grid point index for each of the loads is based on the camera image. All grid points indices containing a load are stored in array C. During each iteration, the grid point index of one of the loads (subset C^-) is optimized, while those of the other loads (subset C^+) are kept fixed.

In each iteration, we minimize

$$Z = \sum_{i \in I} \left| \sum_{n \in N_{\text{GP}}} \left(\alpha^{(n)} \gamma^{(n)} \epsilon_i^{(n)} - \epsilon_i \right) \right| \tag{4.23}$$

by finding a new grid point index n for the load we are optimizing, and by finding best estimates for $\alpha^{(n)} \quad \forall n \in C$. We use the following constraints:

$$\begin{cases} \sum_{n \in N_{\text{GP}}} \gamma^{(n)} = n_{\text{footsteps}} & \\ \gamma^{(c)} = 1 & \forall c \in C^+ \\ \gamma^{(n)} \in \{0, 1\} & \forall n \in N_{\text{box}} \\ \alpha^{(n)} \in \mathbb{R}_0^+ & \forall n \in C \end{cases} \tag{4.24}$$

The force scale factors $\alpha^{(n)}$ are decision variables, which scale the individual grid points to the predicted force, making the absolute strain mathematically comparable. Each iteration the coupled grid points set $\{\gamma^{(n)}\}$ gets updated with the newly obtained optimal grid point.

4.5.3. Multi-Load Interpolation

Interpolation as done for the single-load algorithm has a similar scaling problem as coupling. The computation time would be in the order of $\mathcal{O}(|H|^{n_{\text{footsteps}}})$. The same approach is used to linearize the computation as done for coupling, by locking all but one of the areas around the coupled grid points. As an initial guess for the continuous location of each load, we assume that it lies in the quadrant towards it leans in the camera image compared to those of the coupled grid point:

$$h_{\text{min}_c} = \begin{cases} 0 & \text{if } a_{\text{cam}_c} \leq a_{\text{min}_c} \text{ and } b_{\text{cam}_c} \leq b_{\text{min}_c} \\ 1 & \text{if } a_{\text{cam}_c} \leq a_{\text{min}_c} \text{ and } b_{\text{cam}_c} \geq b_{\text{min}_c} \\ 2 & \text{if } a_{\text{cam}_c} \geq a_{\text{min}_c} \text{ and } b_{\text{cam}_c} \geq b_{\text{min}_c} \\ 3 & \text{if } a_{\text{cam}_c} \geq a_{\text{min}_c} \text{ and } b_{\text{cam}_c} \leq b_{\text{min}_c} \end{cases} \quad \forall c \in C \tag{4.25}$$

During each iteration there will be a set of locked grid points C^+, of size $n_{\text{footsteps}} - 1$ representing all but the currently optimized point, which itself falls in the set C^-. In each iteration, we first minimize

$$\text{MAD}^{(h_{c^-})} = \text{med}\left(\left|X_{ij}^{(h_{c^-})} - \tilde{X}^{(h_{c^-})}\right| \quad \forall i, j \in I\right), \quad \forall h_{c^-} \in H \tag{4.26}$$

with

$$\begin{cases} X_{ij}^{(h_{c^-})} = \left| \frac{\sum_{k\in K}\sum_{c\in C} \alpha^{(c)}\, \beta^{(k_c,h_c)}\, \epsilon_i^{(n_{k_c,h_c})}}{\sum_{k\in K}\sum_{c\in C} \alpha^{(c)}\, \beta^{(k_c,h_c)}\, \epsilon_j^{(n_{k_c,h_c})}} - \frac{\epsilon_i}{\epsilon_j} \right| & \forall i, j \in I, \forall h_{c^-} \in H \\ \tilde{X}^{(h_{c^-})} = \text{med}\left(X_{ij}^{(h_{c^-})} \quad \forall i, j \in I\right) & \\ h_c = h_{\text{min}_c} & \forall c \in C^+ \\ \beta^{(k_c,h_c)} \in \mathbb{R}_0^+ & \forall k_c \in K, \forall h_c \in H \\ \sum_{k_c} \beta^{(k_c,h_c)} = 1 & \forall h_c \in H \end{cases} \tag{4.27}$$

to find the optimal values of $\beta^{(k_c,h_c)} \quad \forall c \in C$. In this step, the values of $\alpha^{(c)} \quad \forall c \in C$ are kept fixed at the values determined in the coupling phase.

Next, each iteration finds the optimal quadrant h_{c^-} of the load that is optimized as

$$h_{\text{min}_{c^-}} = \underset{h_{c^-}}{\text{argmin}}\, \text{MAD}^{(h_{c^-})} \tag{4.28}$$

This value is then inserted back into the array of optimal area values $\{h_c\}$ for the next iteration. After all of the iterations have been executed, the areas list $\{h_{\text{min}_c}\}$ and their respective interpolated scale factors $\beta_{k_c,h_{\text{min}_c}} \quad \forall c \in C$ have been found.

The estimated load locations for each footstep

$$a_{\text{est}_c} = \sum_{k\in K} \beta_{k_c,h_{\text{min}_c}}\, a_{k_c,h_{\text{min}_c}} \quad \forall c \in C \tag{4.29}$$

$$b_{\text{est}_c} = \sum_{k\in K} \beta_{k_c,h_{\text{min}_c}}\, b_{k_c,h_{\text{min}_c}} \quad \forall c \in C \tag{4.30}$$

follow from the summation of the location of the grid points in the optimal area by their scale factors $\beta_{k_c,h_{\text{min}_c}}$.

4.5.4. Multi-Load Scaling

The load factor $\alpha^{(n)}$, as determined during the coupling phase, provides an initial guess for the simplified locations. However, with the interpolation a more precise location has been determined. For this reason it is desirable to redetermine the scale factor $\alpha^{(c)}$ to better match the newly predicted load locations. We minimize

$$Z = \sum_{i \in I} \left| \sum_{c \in C} \sum_{k \in K} \alpha^{(c)} \beta_{k_c, h_{\min_c}} \epsilon_i^{\left(n_{k_c, h_{\min_c}}\right)} - \epsilon_i \right| \tag{4.31}$$

subject to

$$\alpha^{(c)} \in \mathbb{R}_0^+ \tag{4.32}$$

for global force scale factors $\alpha^{(c)}$. Here the previously found scale factors $\beta_{k_c, h_{\min_c}}$ are used.

The estimated loads for each of the footsteps follow

$$F_{\text{est}_c} = \alpha^{(c)} \sum_{k \in K} \beta_{k_c, h_{\min_c}} F_{k_c, h_{\min_c}} \quad \forall c \in C \tag{4.33}$$

where scale factors $\beta_{k_c, h_{\min_c}}$ and $\alpha^{(c)}$ are used to scale the force used for the database computation.

To determine the estimated hot-spot stresses in all of the researched details

$$\sigma_{s,\text{est}} = \sum_{c \in C} \alpha^{(c)} \sum_{k_c \in K} \beta_{k_c, h_{\min_c}} \sigma_s^{\left(n_{k_c, h_{\min_c}}\right)} \quad \forall s \in S \tag{4.34}$$

the same scale factors $\beta_{k_c, h_{\min_c}}$ and $\alpha^{(c)}$ are used. These values provide the live stress state of the desired details and the load description on the asset.

4.6. Fingerprinting Analysis over Time

The estimated stress values in each detail $\sigma_{s,\text{est}}$ are determined for a specific value measurement in time. The fingerprinting analysis is run at 10 Hz to capture the load behavior over time. By running the analysis over a period of time provides the stress-time graph in all of the details. This enables the calculation of fatigue damage using rainflow counting and Palmgren-Miner's rule. Chapter 7.5 provides a numerical example of the complete fingerprinting methodology described in this chapter.

5

Developing Real-Time Assessment Pipeline

In order to achieve a functional test-setup the methodology described in Chapter 4 will be turned into a pipeline of Python code. This pipeline should entail all of the relevant functionalities in order to achieve load and stress state estimation, as well as provide necessary information for fatigue life prediction. Chapter 5.1 will describe the entire pipeline and the structure of the system. Chapter 5.2 will explain the input data of both FBG sensors and cameras and how their real-time data transfer is achieved. In Chapter 5.3 all relevant data analysis functionalities will be thoroughly explained. Finally, Chapter 7.7 will describe the output of the pipeline.

5.1. Pipeline Structure

To provide a broader perspective, let us examine the global application framework first. The desired pipeline should have an input of sensor data, and through data analysis output all necessary information to predict fatigue life. A simplified diagram of the application is shown in Figure 5.1.

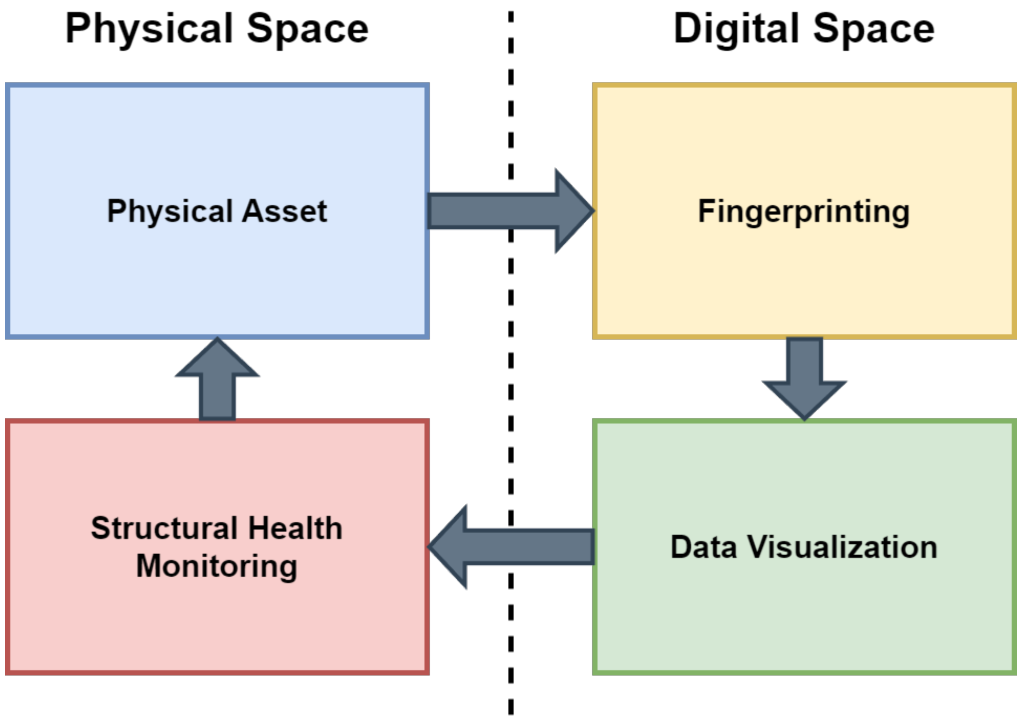


Figure 5.1: Simplified block diagram showing the functionality of the fingerprinting methodology in a broader desired application.

A split is made here between the Physical Space and the Digital Space. The physical space entails the physical asset, including all of the connected sensor equipment, as well as the loads applied to the asset. The digital space consists of the written fingerprinting methodology in its broadest form, including all data handling and processing steps, as well as a visualization of the data. In addition, a block for structural health monitoring exists as a loop to determine if fatigue damage has occurred and matches the simulated results. If there is fatigue damage, then the digital model could be adjusted to account for the damages, and more accurately predict future fatigue life.

To provide a deeper understanding of the meaning behind each block and the steps executed in the process—along with those currently excluded but worth exploring in future research—a more detailed block diagram is presented in Figure 5.2. The diagram again highlights the division between the physical and digital spaces. The blue blocks represent the aspects related to data handling.

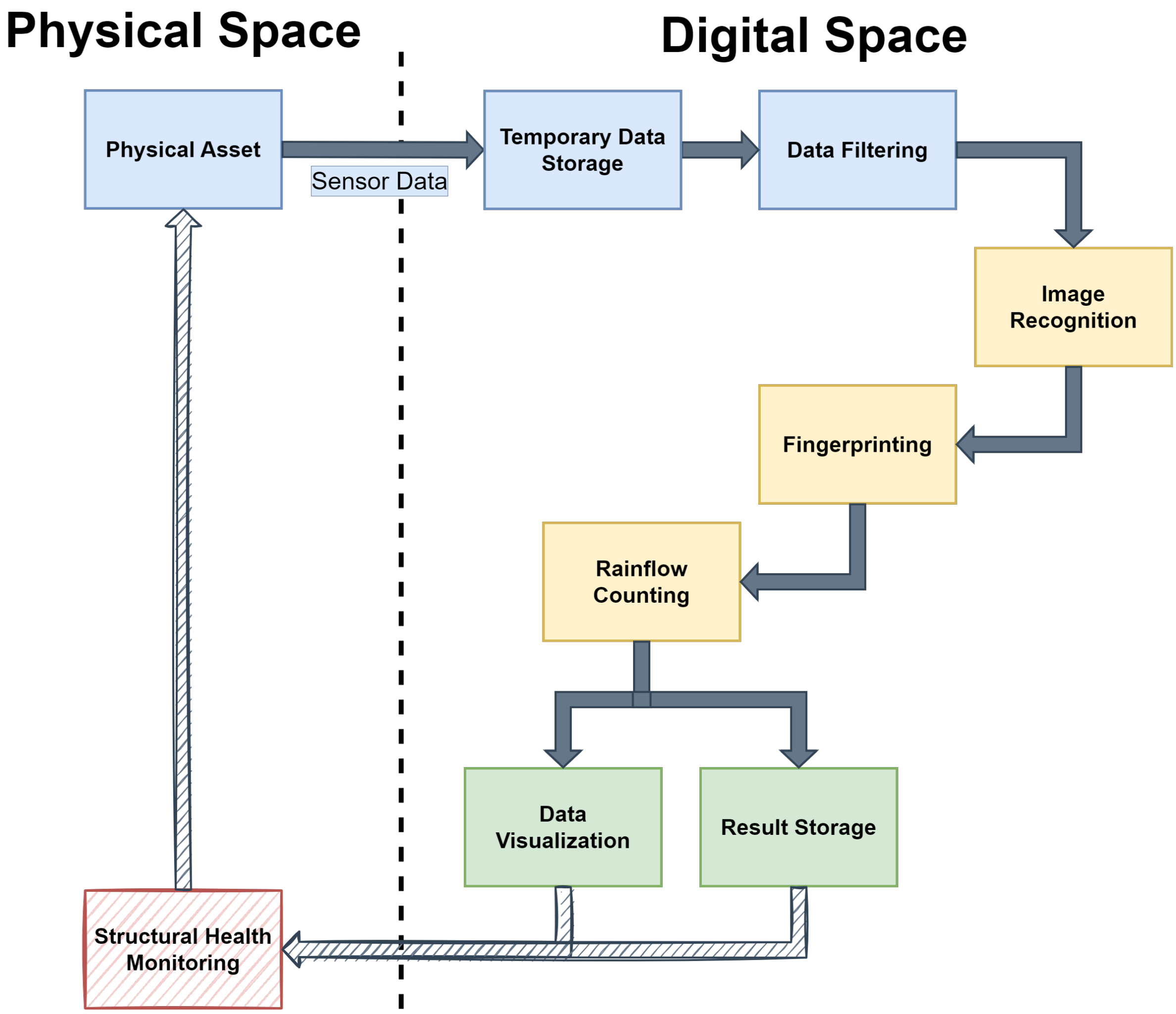


Figure 5.2: Expanded block diagram showing the real-time assessment pipeline and the interaction between physical and digital space.

In the physical space, sensors connected to the asset continuously provide strain data and camera images. This data is transmitted from the physical space into the digital space, where it is first stored in temporary data storage. If any one of the strain values exceeds a predetermined cut-off threshold, we interpret this as a relevant load on the asset. The strain data are then selected for fingerprinting analysis.

The data analysis process, depicted in yellow blocks, consists of three primary steps within the broader fingerprinting framework. The first step is image recognition, where the number of loads on the asset is determined from associated camera images. This information, combined with the strain data, serves as the input for the fingerprinting algorithm in the second step. The algorithm estimates the stress values in each detail of the structure. Finally, in the third step, rainflow counting is applied to assess fatigue damage over a time window of accumulated stress values, resulting in a comprehensive stress profile.

Following the analysis, the gathered results are stored in the green blocks, which consists of two main components. The first is data visualization, providing an accessible way to assess the asset's condition and analyze the effectiveness of the fingerprinting process. The second is results storage, where a broad range of valuable information is archived. This includes data on loads, frequencies, stress states, and fatigue damage, which could support future analyses and offer essential insights into the asset's long-term performance and durability.

The red block represents the structural health monitoring component, which is designed to create a closed-loop system for the entire process. However, to simplify the scope of this research, this step is not implemented at this stage. In future studies, this aspect could be investigated further to enhance the system's capabilities. Incorporating this component would enable continuous feedback and integration of the results back into the monitoring and decision-making processes, offering a more robust and adaptive approach to asset management.

5.1.1. Application Strategy

The application involves an integrated approach that combines sensor connectivity, real-time data analysis, and efficient data storage. This means that every component of the pipeline (data acquisition, processing, and storage) must work seamlessly together to ensure the system's reliability and responsiveness. Given the real-time requirements, careful attention is placed on minimizing latency, optimizing data flow, and maintaining synchronization between various stages of the pipeline.

The system interfaces with a network of sensors to collect high-frequency data in real time. Ensuring robust and consistent communication with the sensors is critical, especially in environments prone to noise or disruptions. The application establishes a direct connection to the sensors using Ethernet connections, allowing for the continuous transfer of data streams. This connectivity layer is designed to handle potential data loss or delays, employing mechanisms such as error correction and buffer management to ensure the fidelity of incoming data.

The real-time data analysis component is at the heart of the application. It processes the incoming data stream at the required frequency, extracting meaningful insights and making decisions without delay. Python was chosen as the primary programming language for this aspect of the application. This decision was guided by previous experience with Python, as well as its rich ecosystem of libraries and tools for numerical computation, data manipulation, and machine learning. Libraries such as NumPy, Pandas, and SciPy provide a robust foundation for handling sensor data, while frameworks like Dash facilitate real-time visualization of analysis results. Python's versatility also enables rapid prototyping and iterative development, which are essential for fine-tuning the application's algorithms.

In parallel with real-time analysis, the system incorporates a data storage layer to ensure that all sensor readings and analysis results are logged for future reference. This storage solution is optimized for both high write speeds and efficient data retrieval. This archival capability not only supports post-analysis but also provides a backup mechanism to safeguard against data loss.

By integrating these components into a unified pipeline, the application achieves its goal of delivering accurate, real-time insights based on live sensor data. Python's flexibility and ease of use play a crucial role in bridging the gap between the hardware (sensors) and software (analysis and storage), ensuring the system's robustness and scalability for future expansions or modifications.

5.1.2. Folder Structure

The folder structure for the application is designed to align with the overall application strategy. As shown in Figure 5.3, the main folder is named "Real-Time Assessment" and contains several subfolders along with a few key files.

One of the files is a "README" file, which provides an overview of how the tool works. It includes step-by-step instructions on how to make necessary adjustments and ensure the application operates correctly. Another important file is the main Python script, which acts as the starting point for the real-time assessment application. This script gathers input from the user, initiates all core functions, and manages time synchronization to ensure smooth operation between the different sensors. Additionally, there is a separate Python script that contains all the supporting functions required for specific tasks and analysis.

The subfolders in this structure correspond to the yellow blocks in the block diagram shown in Figure 5.2. These include the Image Recognition, Fingerprinting, and Rainflow Counting subfolders. Each subfolder is dedicated to a specific function, and their roles and purposes will be detailed in Chapter 5.3.

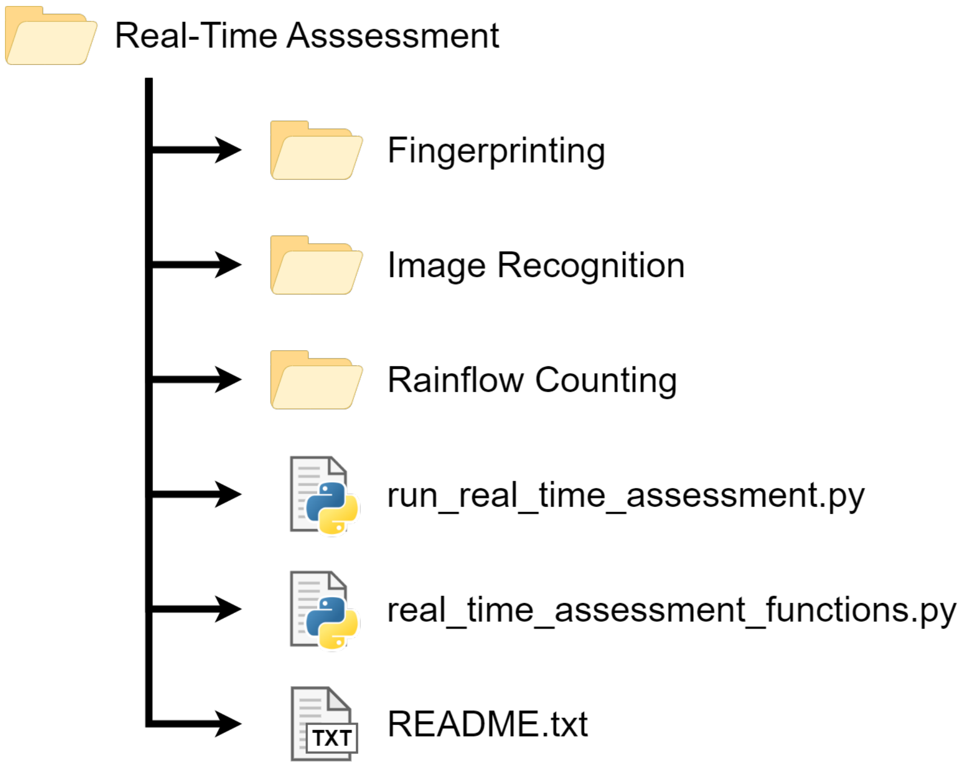


Figure 5.3: Folder structure of the Real-Time Assessment pipeline.

5.1.3. Main Run Loop Logic

The main script of the Real-Time Assessment pipeline is the Python file run_real_time_assessment.py. This file serves as the system's core, managing the reading of sensor data, synchronizing data to the current iteration time, initiating data analysis functions, and producing results. It also handles visualization through a dashboard. Establishing clear and logical operations within this script is essential for the pipeline's functionality. The overall logic is represented in the block diagram in Figure 5.4.

In the diagram:

- Blue represents input data, including input parameters, strain sensor data, and camera images.
- Yellow shows the different stages of data analysis.
- Green indicates the output of the analysis, in this case, the counted cycles.

While the script includes many additional outputs and processes, the explanation here focuses on its core functionality for simplicity.

Sensor data streams continuously and is time-stamped as it is received. When an iteration begins, the system selects the sensor data closest to the iteration time and discards older data. At the early stages of data collection, there may not be enough information to effectively perform rainflow counting, as this method requires a representative dataset to provide meaningful results. Additionally, rainflow counting is computationally intensive, making it inefficient to execute often with limited data. Therefore, the rainflow counting process is postponed until a sufficient amount of data has been gathered, ensuring that the analysis is both representative and computationally justifiable.

If any strain values $\epsilon_i, , , \forall i \in I$ exceed a predefined threshold ϵ_c, the applied load is deemed significant enough to initiate the fingerprinting algorithm. The process begins by identifying the camera image closest to the current iteration time t. Image recognition is then performed on this image to determine the number of footsteps present on the bridge. Based on this information, the fingerprinting algorithm calculates the stress values at critical locations on the bridge. These calculated stress values are subsequently stored, and the system advances to the next iteration.

Once enough data is accumulated, the rainflow counting operation begins, producing an output of counted cycles. Afterward, the process continues with subsequent iterations as described.

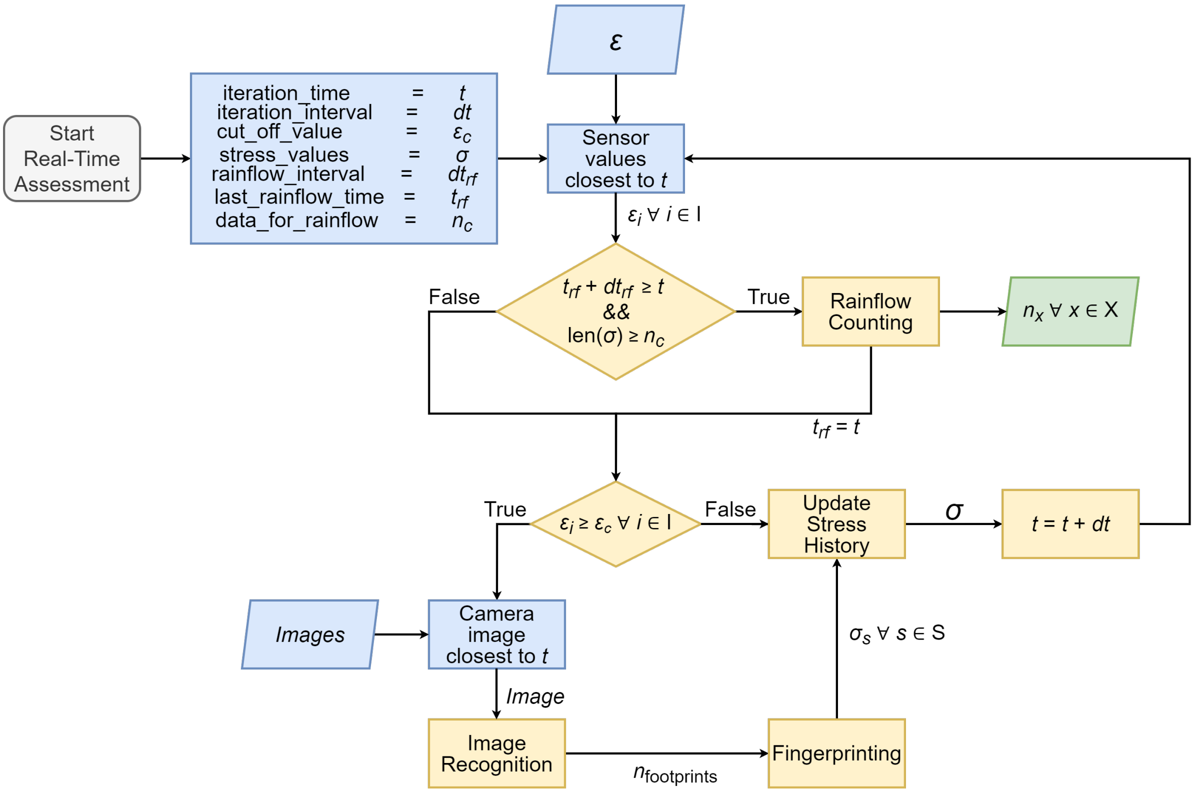


Figure 5.4: Logic of the main operational functionality of the Real-Time Assessment source code. The blue blocks represent the input data, the yellow blocks represent the data analysis stages, and the green block represents the output.

5.2. Data Input

The data input to the real-time assessment pipeline is provided by strain sensors and a camera. This chapter will describe the used sensors, how they work and what type of data they provide.

5.2.1. FBG Sensor Data

The strain sensors utilized in this application are Fiber Bragg Grating (FBG) strain sensors produced by FBGS [7]. These sensors were selected for their ability to achieve high-frequency measurements and their suitability for managing a large number of sensor locations, even on full-scale traffic and railway bridges. Constructed from fiberglass cables, FBG sensors operate based on variations in the refractive index at specific points along the cable. These variations result in shifts in the wavelength of light traveling through the cable, enabling precise strain measurements.

The wavelength shift depends on how much the sensor stretches, which happens when the material to which the sensor is attached deforms. As the material stretches, the sensor stretches too, causing a change in the wavelength. By measuring this change, the strain on the material can be calculated. This process is illustrated in Figure 5.5.

One sensor cable can have multiple sensor locations along its length, allowing it to provide strain data from several points. The strain measured at each location is reported in the unit microstrain ($\mu\epsilon$). The measurement frequency can be adjusted using the FBGS software, and the required frequency of 10 Hz is easily achievable.

For this application, a slightly higher measurement frequency of 15 Hz was chosen. This helps to reduce synchronization offsets between the strain data from the sensors and the images captured by the camera. Figure 5.6 shows an example of the measurement plots provided at 10 sensor locations, when a person steps onto the bridge, stands still for some time, and then steps off.

The strain values are transmitted live using the Transmission Control Protocol (TCP). This protocol sends the data over the internet, making it accessible to any device connected to the specified IP address and port number. The data is streamed at the same frequency as the measurements, 15 Hz.

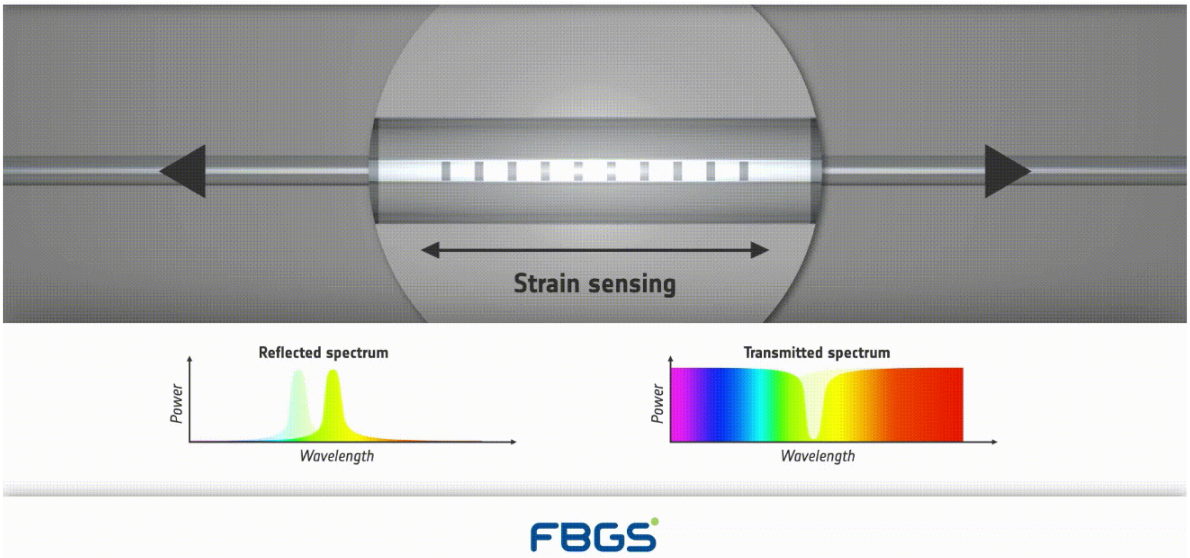


Figure 5.5: Visual representation of the functioning principle of FBG strain sensors. Obtained from FBGS [6].

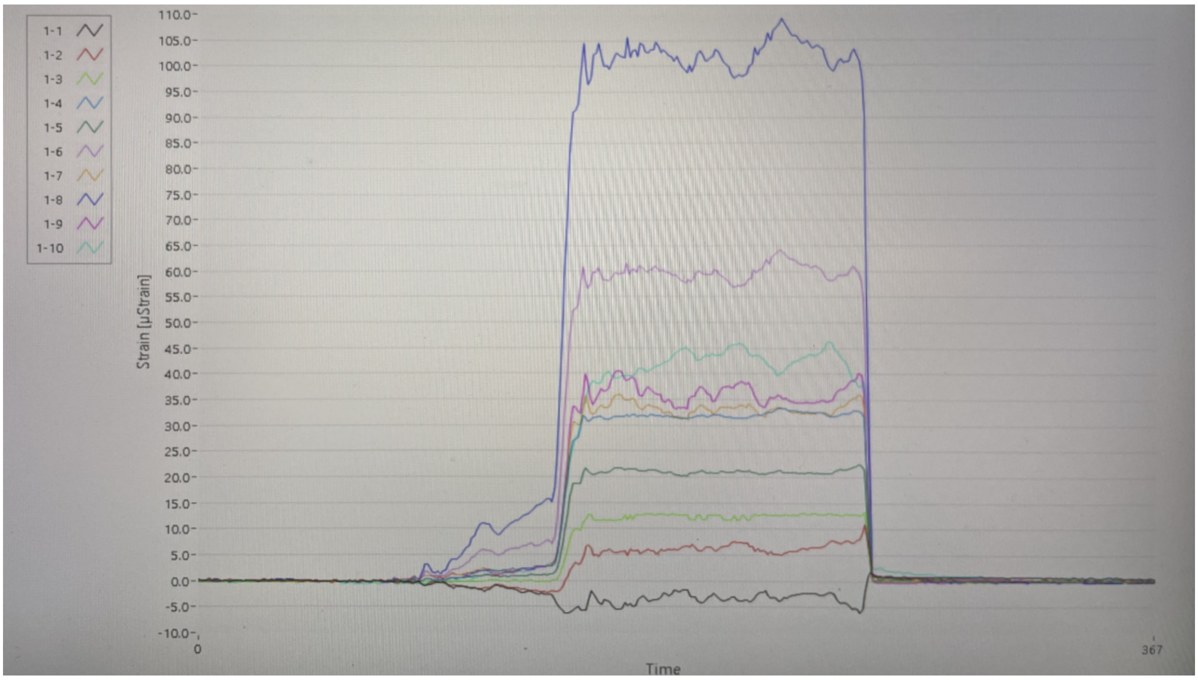


Figure 5.6: Example strain graph from 10 of the measurement points along the FBG strip as a result of loading the bridge by stepping onto the bridge, and stepping off again after a couple of seconds.

The data transmitted from the sensors is a string of bytes with a variable length. It includes several key pieces of information, such as the measurement timestamp, the wavelengths at each measurement point, and the corresponding strain values. To extract the strain information, the string must first be converted into a readable format. The source code used for reading this sensor data is provided in Appendix B.

Once the strain values are read, they are processed in two steps:

- The raw strain values are processed using a function detailed in Appendix D. This prepares the data in the desired format for further analysis.
- The effect of temperature variation is accounted for by applying a moving average. This step removes the influence of temperature changes, which can cause slight drifts in the strain measurements due to the expansion and contraction of the bridge material. The function used for this step is shown in Appendix E.

Once the conversion is complete, the strain values at each measurement point are obtained at 15 Hz. These values are then used as input for the subsequent stages of data analysis.

5.2.2. Camera Data

The camera utilized for this analysis is the Axis Q6055-E, a device readily available and equipped with specifications well-suited for the intended tasks. One of its key features is the ability to stream and access images in real time through the Real-Time Streaming Protocol (RTSP). This capability eliminates the need to store video files, significantly reducing the storage space requirements. Moreover, specific images can be captured on demand with precise timing, ensuring high accuracy in data collection.

Once RTSP streaming is enabled via the camera's online settings interface, the live stream can be accessed through an RTSP streaming link. This setup allows seamless integration with the real-time assessment pipeline.

The source code for capturing camera images can be seen in Appendix C. The function defined in this code uses the RTSP streaming link to access the video stream and capture images as often as possible to have the most synchronized image to the sensor data.

The captured images are subsequently used in the data analysis process, particularly for image recognition tasks. This step plays a vital role in identifying the number and locations of loads on the asset, which is essential for the subsequent phases of analysis and decision-making within the pipeline.

5.3. Data Analysis

The next step in setting up the real-time assessment pipeline is to analyze the incoming data. The data analysis will be executed with three major functions at the core. These functions are image recognition, fingerprinting and rainflow counting. This chapter will further discuss their implementation and how they work within the real-time assessment pipeline.

5.3.1. Time Control

The time control logic is fundamental to the smooth operation of the system, as it ensures that the current iteration time is consistently aligned with the actual current time. This is achieved through a real-time comparison, where if the process is found to be lagging behind the actual time, corrective measures are immediately implemented. These adjustments can include pausing the data analysis briefly (putting the process to sleep) or, if necessary, allowing the process to catch up by proceeding without delay.

This mechanism is critical for maintaining the system's intended run frequency of 10 Hz, and is necessary to ensure continuous and accurate synchronization across multiple components. Specifically, it ensures that sensor data, recorded at 15 Hz, camera images, captured at 30 frames per second (fps), and iteration times are harmonized seamlessly. The underlying source code that drives this time control logic is provided in Appendix F, offering further insight into its detailed implementation and functionality.

5.3.2. Implementation of Image Recognition

The image recognition system uses images from the camera stream to estimate the number of footsteps visible in the image and provide a first estimate of their locations on the bridge. Developing a custom image recognition algorithm specifically for this application would require significant time and data. While a tailored solution could offer improved accuracy, the time constraints of this project made it impractical to train such an algorithm. Instead, an existing pre-trained image recognition algorithm for shoe detection was used.

The selected algorithm originates from a study by Fernandez et al. at the Graduate School of Life Science and Systems Engineering, Kyushu Institute of Technology, Japan [8]. This study utilized image recognition to detect shoes from a cane-mounted camera designed to monitor elderly individuals. The purpose was to identify shoe locations and predict potential falls, enabling timely interventions to prevent accidents.

The algorithm is based on the MobileNetV2 architecture, known for its excellent balance between accuracy and efficiency. MobileNetV2 is a state-of-the-art model for object detection that performs well with limited computational resources [30]. In the study, it achieved a mean average precision (mAP) of 60.2% in real-time at 25 frames per second using a low-cost device. Its real-time processing capability makes it particularly suitable for the Real-Time Assessment method.

While the algorithm's real-time functionality is highly beneficial, it does have some notable limitations:

- It struggles with accurately identifying shoes when there are many present in a single frame.
- It can incorrectly detect shoes when multiple are close together, leading to errors in counting and localization.

An example from the study demonstrating the shoe detection algorithm is shown in Figure 5.7. The figure highlights both the strengths and weaknesses of the algorithm. Correctly detected shoes are marked, but errors are visible in scenarios with numerous shoes or when shoes are clustered closely together.

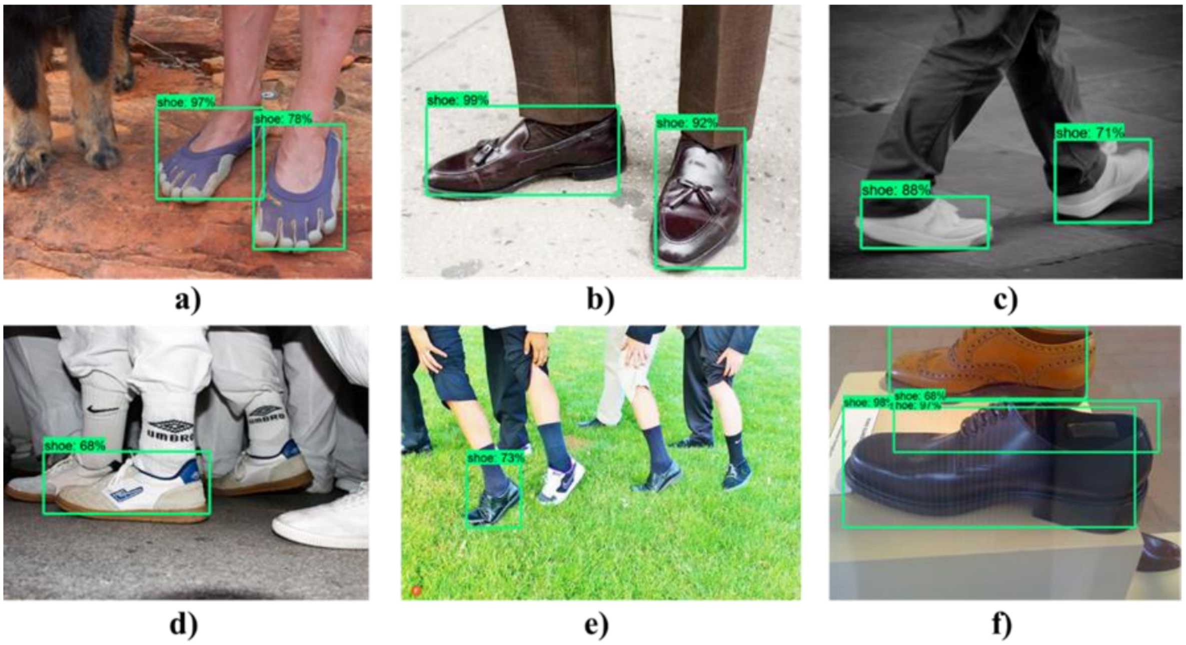


Figure 5.7: Examples of shoe detections: correct recognitions are shown in a), b), and c), while detection errors are illustrated in d), e), and f). Images sourced from Fernandez et al. [8].

The source code for the image recognition system is provided in Appendix G. The trained TensorFlow model is loaded and applied to the image. Based on the detected footsteps in the image, the coordinates of each recognized footstep location on the bridge are then determined.

5.3.3. Implementation of Fingerprinting

The next step within the data analysis is to use the sensor data and coupled image at almost the exact same time step as input for the fingerprinting algorithm. This methodology is implemented in the pipeline as Python scripts with the functionality as described in Chapter 4.5.

The folder structure shown in Figure 5.3 is expanded to include the sub-folder structure of the Fingerprinting folder. This is illustrated in Figure 5.8. The folder consists of two main parts:

- Import Files: This section contains the .csv file with the database as generated by running the FEM model with a unity load at each grid point.
- prepare_fingerprints.py: This Python file includes the functions responsible for executing all operations related to the fingerprinting process.

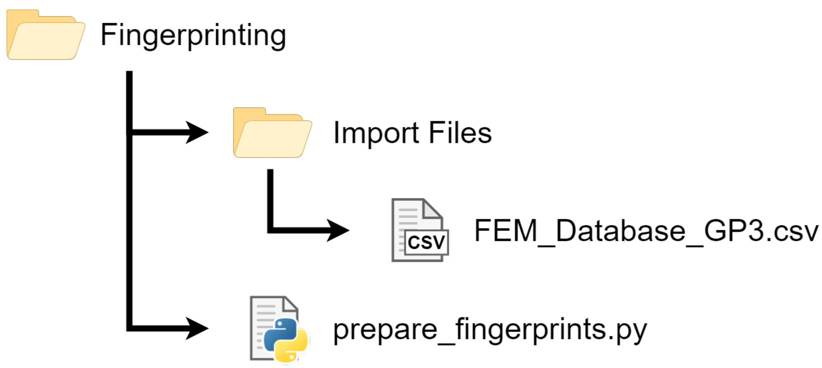


Figure 5.8: Folder structure of the Fingerprinting sub-folder.

The source code of the prepare_fingerprints.py script is shown in Appendix H. Now follows an explanation on the way the file is structured, the input, the functions used to obtain the best estimate for stress and load estimation and the output.

This code is designed to determine the unique characteristics (or fingerprint) of a measurement point by coupling the closest grid points, interpolating between grid points and applying scaling of the coupled grid points.

The input to the code is a row of data consisting of the estimated positions from image recognition on the coordinates of the loads and the strain values from all of the sensors. In addition the file imports the database of grid points that was obtained by executing each of the grid points in FEM to obtain the stress and strain values for different load locations under the same magnitude.

The core of the script is the main function, determine_fingerprint_of_row. Based on how accurate the image recognition is determined to be the grid points are filtered for each of the footstep locations. This limits the number of grid points that have to be analyzed during later steps. To avoid division by zero during further computations all of the strain values that are exactly zero are changed to a negligibly small value. This provides more robust computations.

Implementation of Coupling Grid Points

The first helper function is used to identify coupled grid points and calculate the scale factors for these points. It iteratively runs all linear combinations of grid points from the filtered dictionary on the first objective function, which aims to minimize the summed absolute strain difference between the scaled strain values from the grid points and the measured strain values. This minimization is carried out using Scipy.Minimize with the "COYBLA" method. Alternative methods were tested but either provided worse results or were less robust, as they produced optimization errors at inconsistent iterations. The following code provides the implementation of Equation 4.23.

```
# Define the objective function using NumPy for faster computation
def objective(alpha):
    alpha = np.expand_dims(alpha, axis=-1)
    eps_FP_sum = np.array(np.sum(alpha * eps_FP_list, axis = 0))
    return np.sum(np.abs(eps_FP_sum - eps_array))

# Set initial guess for alpha value
initial_alpha = np.full(len(eps_FP_list),1)

```

```
# Set bounds to ensure alpha values are non-negative
bounds = [(0, 10) for _ in range(len(initial_alpha))]

# Use minimize with the vectorized objective function and bounds
result = minimize(objective, initial_alpha, method='COBYLA', bounds=bounds,)
```

Implementation of Interpolating Grid Points

The second helper function interpolates between the four grid points within each of the quadrants surrounding the coupled grid points and finds the linearly computed combination of quadrants or areas that together sum to the least difference in absolute strain from the measured strain values, as described in Chapter 4.5.3. The main part here again is a minimization using Scipy.Minimize, this time using the method “SLSQP”, which provides both bounds and constraints for the minimization process. The function now uses relative difference to more precisely determine the location. After all of the linear combinations are tried, the optimal global force scale factors $\alpha^{(c)}$ and local relative scale factors $\beta_{k_c,h_{\min_c}}$ are found. The following code provides the implementation of Equation 4.31.

```
# Defining the objective function to find the minimal difference area and with that the betas
def objective(betas, gp_points_strain, relative_relationships_measurement, alpha):

    # Split betas in lengths of 4 for each of the coupled grid points
    beta_part = [betas[b:b+4] for b in range(0, len(betas), 4)]
    beta_part = np.array(beta_part)

    # Restructure betas to be multipliable
    beta_part = np.expand_dims(beta_part, axis=-1)
    alpha = np.expand_dims(alpha, axis=-1)
    alpha = np.expand_dims(alpha, axis=-1)

    # Tranforming grid point strain to array
    gp_points_strain = np.array(gp_points_strain)

    # Determine relative square points strain after scaling and summation
    scaled_points_strain = np.sum(beta_part * alpha * gp_points_strain, axis=(0,1))
    relative_square_points_strain = scaled_points_strain[:, np.newaxis] /
        scaled_points_strain[np.newaxis, :]

    # Determine the MAD values per square grid point
    diff_nested = np.abs(relative_square_points_strain - relative_relationships_measurement)
    median_diff = np.median(diff_nested)
    median_absolute_deviation = np.median(np.abs(diff_nested - median_diff))
    return median_absolute_deviation

# Defining the constraint
def constraint_sum(betas):
    constraints = []
    for g in G:
        beta_part = betas[g*4:(g+1)*4]
        constraints.append(np.sum(beta_part) - 1)
    return np.array(constraints)

# Define the constraints dictionary
constraints = {'type': 'eq', 'fun': constraint_sum}

# Defining the bounds on the beta values to be between 0 and 1
bounds = [(0, 1) for _ in range(4*len(active_fingerprints))]

# Check if 4 points surround the grid points. Does not apply to grid points at the boundaries
     of the grid, such as (0, 0)
if all(len(item) == 4 for item in active_areas_data):

    gp_points_strain = []

    for g in G:
        # Filter the strain values from the dataframe
        points_strain = active_areas_data[g].filter(like=strain_handle).filter(regex=
            disabled_strain_pattern).values
        points_strain[(points_strain >= 0) & (abs(points_strain) < infinitely_small_value)] =
             infinitely_small_value
        points_strain[(points_strain < 0) & (abs(points_strain) < infinitely_small_value)] =
            -infinitely_small_value
```

```
        gp_points_strain.append(points_strain)

    # Define intial guess for beta values
    initial_betas = np.full(4*len(active_fingerprints), 0.25)

    # Run the minimization function to obtain the optimal area and betas
    result = minimize(objective, initial_betas, args=(gp_points_strain,
        relative_relationships_measurement, alpha),
                    method='SLSQP', bounds=bounds, constraints=constraints,)
```

Implementation of Scaling Grid Points

The third helper function recalculates the best estimate for the force scale factors ($\alpha^{(c)}$) after the interpolation step. This process is similar to the coupling step but now uses four grid points per footstep location. The minimization is again carried out using the "COYBLA" method.

The implementation follows the logic of Equation 4.31, and the corresponding code provides the necessary steps to achieve this refinement.

```
# Defining the objective function to determine optimal scale factors alpha
def objective(alpha):
    alpha = np.expand_dims(alpha, axis=-1)
    alpha = np.expand_dims(alpha, axis=-1)
    return np.sum(np.abs(np.sum(alpha * interpolated_betas * min_areas_data_filtered, axis =
        (0,1)) - measurement_row_filtered))

# Set initial guess for alpha values
initial_alpha = np.full(len(interpolated_betas),1)

# Set bounds to ensure alpha values are non-negative
bounds = [(0, 10) for _ in range(len(interpolated_betas))]

# Use minimize with the vectorized objective function and bounds
result = minimize(objective, initial_alpha, method='TNC', bounds=bounds,)
```

Implementation of Fingerprinting through Time

The analysis thus far has focused on single moments in time and considered all the hot-spot stress details collectively. The fingerprinting algorithm will be executed repeatedly for each iteration step until a sufficient number of data points have been collected to proceed with the rainflow counting algorithm. Subsequently, the next chapter will delve into the rainflow counting analysis, which is conducted over the analyzed time period and applied to each detail individually.

5.3.4. Implementation of Rainflow Counting

The next key part of the Real-Time Assessment method is the Rainflow Counting functionality. The implementation is based on the methodology explained in Chapter 2.2. In this version of the code, hysteresis filtering is skipped since it was determined to be unnecessary, because the code already runs efficiently without it.

Implementation of Peak-Valley Filtering

The first step in the process is peak-valley filtering. During this step, each stress value is evaluated to identify whether it is:

- A peak (the current stress value is higher than the surrounding values).
- A valley (the current stress value is lower than the surrounding values).
- Neither, in which case the stress value is removed from the dataset.

This filtering step prepares the data for further rainflow counting analysis. The following code shows how the peak-valley filtering was implemented.

```
# Step 1: Apply peak-valley filtering to retain significant stress points.
def apply_peakvalley_filter(df):
    """
    Identifies and retains only the peaks and valleys in the stress data.

    Parameters:
    - df (DataFrame): DataFrame containing the stress values.
```

```

    Returns:
    - DataFrame: Filtered DataFrame with only peaks and valleys.
    """
    logger.info("Applying␣peak-valley␣filtering.")
    stress = df.values
    peakvalley_drop = np.zeros(len(df), dtype=bool)

    # Identify peaks and valleys by checking neighboring values
    for i in range(1, len(df) - 1):
        if stress[i] > stress[i - 1] and stress[i] > stress[i + 1]:
            continue  # Peak
        elif stress[i] < stress[i - 1] and stress[i] < stress[i + 1]:
            continue  # Valley
        else:
            peakvalley_drop[i] = True  # Not a peak or valley

    # Filter out points that are neither peaks nor valleys
    df = df[~pd.Series(peakvalley_drop)].reset_index(drop=True)
    return df
```

Implementation of Binning

The next step in the Rainflow Counting process is binning. This step involves creating bins based on the range of maximum and minimum stress values and defining the desired number of bins. Each stress value is then assigned to the bin whose average stress value is closest to the given stress value. This binning process discretizes the continuous stress data into corresponding bin values, simplifying the data for subsequent analysis. The following code snippet illustrates the implementation of peak-valley filtering, which precedes the binning process.

```
# Step 2: Discretize stress values into bins.
def apply_binning(df, n_bins, maximum_stress, minimum_stress):
    """
    Bins stress values into discrete intervals for analysis.

    Parameters:
    - df (DataFrame): DataFrame containing the stress values.
    - n_bins (int): Number of bins.
    - maximum_stress (float): Maximum stress value.
    - minimum_stress (float): Minimum stress value.

    Returns:
    - list: Binned stress values.
    - list: List of bin ranges and metadata.
    """
    logger.info("Applying␣stress␣value␣binning.")
    stress = df.values
    stress_range = abs(maximum_stress - minimum_stress)
    bin_size = stress_range / n_bins
    bins = []

    # Generate bin ranges and metadata (start, end, average value, bin index)
    start_value = minimum_stress
    for i in range(n_bins):
        end_value = start_value + bin_size
        avg_value = (end_value + start_value) / 2
        bins.append((start_value, end_value, avg_value, i))
        start_value = end_value

    # Map each stress value to the nearest bin's average value
    bin_stress_values = [min(bins, key=lambda b: abs(b[2] - s))[2] for s in stress]
    return bin_stress_values, bins

# Discretize the filtered stress data
bin_stress_values, bins = apply_binning(df, n_bins, maximum_stress, minimum_stress)
```

Implementation of Four-Point Counting

The main step in rainflow counting is the four-point counting method. This method is implemented by continuously looping through the stress profile until there are fewer than 4 stress values remaining. A new stress cycle is identified if the following conditions are met:

1. **Components:**
   - $S_{\text{inner}} = |S_2 - S_3|$: Difference between the two inner points.
   - $S_{\text{outer}} = |S_1 - S_4|$: Difference between the two outer points.
   - **Condition:** $S_{\text{inner}} \leq S_{\text{outer}}$, ensuring that the points form a valid cycle shape, where the outer difference encompasses the inner one.
2. **First Cycle Condition ($S_1 > S_4$):**
   - The cycle forms a descending slope followed by an ascending slope.
   - Additional checks:
     (a) $S_1 \geq S_3$: The peak is at least as high as the descending midpoint.
     (b) $S_4 \leq S_2$: The trough is at most as low as the ascending midpoint.
3. **Second Cycle Condition ($S_1 < S_4$):**
   - The cycle forms an ascending slope followed by a descending slope.
   - Additional checks:
     (a) $S_1 \leq S_3$: The trough is at most as low as the ascending midpoint.
     (b) $S_4 \geq S_2$: The peak is at least as high as the descending midpoint.

The following code provides the implementation of the four point counting method:

```python
# Step 3: Identify full cycles using four-point counting.
def apply_fourpointcounting(bin_stress_values):
    """
    Applies four-point counting to detect full stress cycles.

    Parameters:
    - bin_stress_values (list): List of binned stress values.

    Returns:
    - list: List of identified full stress cycles.
    - list: Residual stress values.
    """
    logger.info("Performing four-point cycle counting.")
    stress = np.array(bin_stress_values)
    rainflow_cycles = []

    while True:
        # Look for a four-point cycle in the data
        for n in range(len(stress) - 3):
            S1, S2, S3, S4 = stress[n:n + 4]
            S_inner = abs(S2 - S3)
            S_outer = abs(S1 - S4)

            # Check the cycle conditions
            if (S1 > S4 and S_inner <= S_outer and S1 >= S3 and S4 <= S2) or \
               (S1 < S4 and S_inner <= S_outer and S1 <= S3 and S4 >= S2):
                rainflow_cycles.append((S2, S3))
                # Remove the identified cycle from the data
                stress = np.concatenate((stress[:n + 1], stress[n + 3:]))
                break
        else:
            break

    return rainflow_cycles, stress.tolist()
```

Implementation of Exporting Rainflow Cycles

The final step in rainflow counting is to export the collected cycles. All of the full stress cycles are grouped by their absolute stress range. The residue from the stress profile is counted as half cycles. The following code provides the implementation to process the rainflow counting results:

```python
# Step 4: Export the results to a DataFrame.
def export_rainflow(rainflow_cycles, residue):
    """
    Converts the rainflow counting results into a DataFrame.

    Parameters:
    - rainflow_cycles (list): Full cycles from the analysis.
    - residue (list): Remaining stress points not part of a full cycle.

    Returns:
    - DataFrame: Rainflow counting summary.
    """
    logger.info("Exporting rainflow counting results.")

    # Process full cycles (absolute stress differences)
    full_cycles = [abs(c[0] - c[1]) for c in rainflow_cycles]
    df_full = pd.DataFrame({full_cycles_column_name: full_cycles})
    df_full = df_full.groupby(full_cycles_column_name).size().reset_index(name=frequency_col_name)

    # Process half cycles (absolute stress differences in residue)
    half_cycles = [abs(residue[i + 1] - residue[i]) for i in range(len(residue) - 1)]
    df_half = pd.DataFrame({half_cycles_column_name: half_cycles})
    df_half = df_half.groupby(half_cycles_column_name).size().reset_index(name=frequency_col_name)
    df_half[frequency_col_name] *= 0.5  # Adjust frequency for half cycles

    # Combine full and half cycle results into one DataFrame
    df_combined = pd.concat([
        df_full.rename(columns={full_cycles_column_name: stress_cycles_column_name}),
        df_half.rename(columns={half_cycles_column_name: stress_cycles_column_name})
    ])
    df_combined = df_combined.groupby(stress_cycles_column_name)[frequency_col_name].sum().reset_index()

    return df_combined
```

6

Optimizing Estimation Performance

This chapter focuses on evaluating the system's performance based on two main Key Performance Indicators (KPIs): computational time and model accuracy. Computational time is determined by both the database generation time and the average iteration speed of the Real-Time Assessment pipeline, while model accuracy is measured by the precision of location, load, and hot spot stress estimations. The performance of the real-time assessment method is influenced by factors such as the resolution of the generated FEM database and the number of strain sensors deployed. The chapter will explore strategies for determining the optimal configuration of strain sensors and grid points, aiming to balance computational accuracy with required processing speed for efficient real-time assessment.

6.1. Database Resolutions Tested

The database resolution depends on the number of grid points over the length of the bridge, N_a, and the width of the bridge, N_b. Four different densities are tested: $N_a, N_b = (20, 4), (30, 6), (40, 8), (50, 10)$, leading to total grid points $N_{\text{GP}} = 80, 180, 320, 500$; respectively referred to as GP1, GP2, GP3, and GP4.

For each of these configurations, FEM calculations need to be performed at each grid point, with each calculation taking around six minutes to complete for the full FEM model and all detail models. Figure 6.1 illustrates the total generation time required for these configurations when calculated on a laptop. Because the FEM model is not influenced by the number of grid points, the database generation time increases linearly with the total number of grid points.

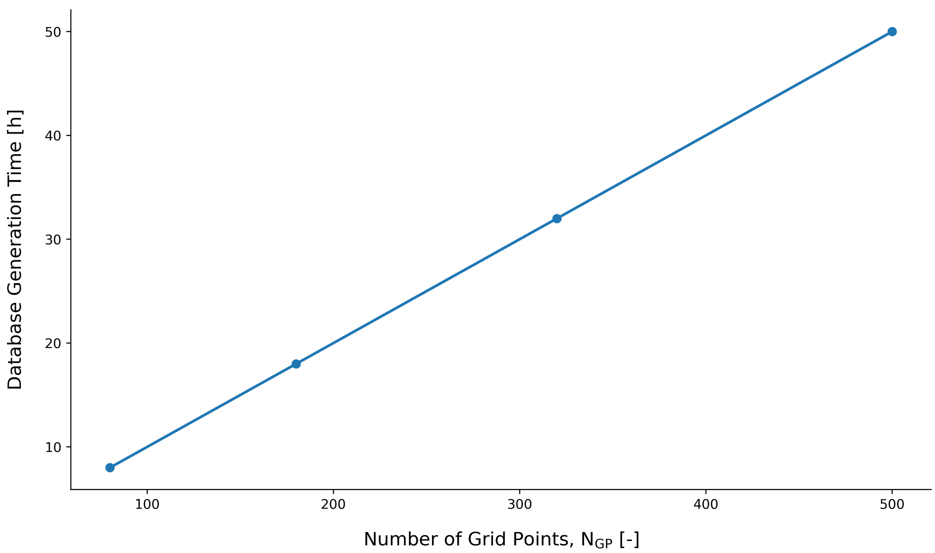


Figure 6.1: Total calculation time of digital database for different amounts of grid points N_{GP}.

6.2. Sensor Configurations Tested

Five different sensor configurations (SCs) were tested to determine the optimal performance. Each configuration placed sensors along the bottom left and right flanges, aligned along the centerline of the flanges, but the number and placement of sensors varies. The sensor configurations consist of the following number of measurement points:

- SC1: 20 measurement locations
- SC2: 16 measurement locations
- SC3: 12 measurement locations
- SC4: 8 measurement locations
- SC5: 4 measurement locations

The more sensors in a configuration, the more detailed the information available, which is crucial for distinguishing different load locations, especially in multi-load situations. Having enough sensors is important to accurately identify these loads. However, there is no definitive answer to how many sensors are necessary. Therefore, the sensor configurations will be tested across a large number of test loads to assess and determine the overall performance of each configuration.

6.3. Optimizing the Database Resolution and Sensor Configuration

Both the database resolution and sensor configurations are taken into account for optimization. The various database resolutions and sensor configurations that were tested are illustrated in Figure 6.2.

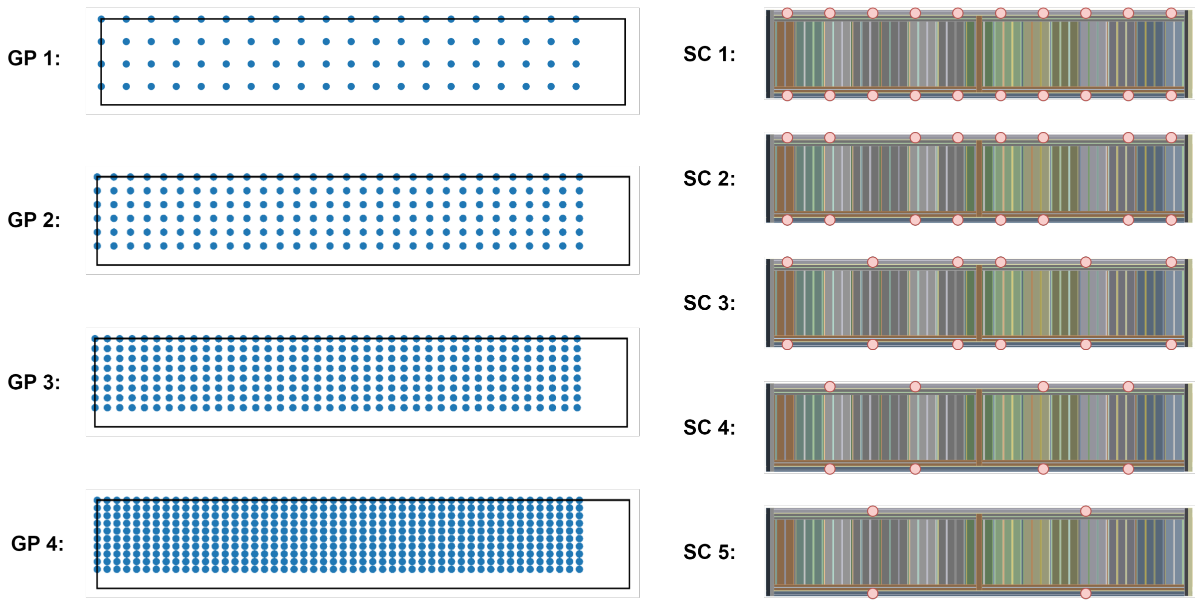


Figure 6.2: On the left are the tested grid point resolutions. On the right are the tested sensor configurations (SCs), where the sensor locations are marked by red dots in their respective configuration.

A total of 200 test loads are simulated in the FEM model, consisting of 50 loads for each of the four footstep scenarios ($n_{\text{footsteps}} = 1, 2, 3, 4$), with randomly generated parameters for location (a, b) and force (F). The multi-load fingerprinting method, described in Chapter 4.5, is applied to these digitally generated strain values to assess the average computational speed and the median hot-spot estimation inaccuracy. All possible combinations of database resolution and sensor configurations are evaluated in this analysis.

The objective is to minimize computational time per iteration while maximizing the accuracy of hot spot estimation. However, these goals often conflict, as achieving higher accuracy typically requires increased computational resources. The laptop used in this study was unable to meet the target operational frequency of 10 Hz, highlighting the need for greater computational power. The selection of grid points and sensors can be adjusted to improve accuracy, depending on the computational resources available. In practical applications, the desired operational frequency becomes the primary criterion for determining the appropriate balance between computational efficiency and accuracy.

The results, displayed in Figure 6.3, show the performance of different combinations of grid points (GPs) and sensor configurations (SCs). SC5 is the worst performer in terms of hot spot estimation accuracy due to insufficient sensor coverage, which prevents proper identification of load locations and magnitudes. SC1, SC2, and SC3 yield similar estimation results, but SC3 achieves lower inaccuracy at nearly twice the speed of SC1. The optimal configurations are found to be either SC3 or SC4. Given that SC3 offers significantly better accuracy at only a slight decrease in computational speed, it was chosen. When paired with SC3, GP2 and GP3 provide a good balance of performance and speed, with GP3 selected to maximize estimation accuracy.

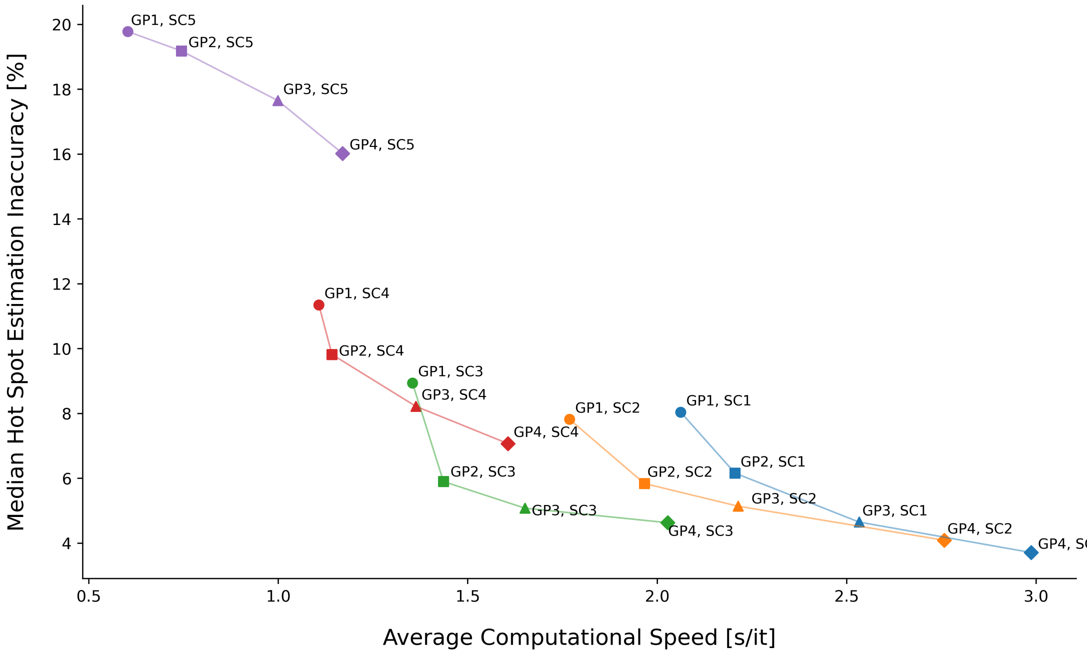


Figure 6.3: Computation speed and hot-spot estimation inaccuracy of different combinations of GPs (represented by different shaped indicators) and SCs (represented by the different colors).

7

Data Collection

To evaluate the effectiveness of the proposed methodology in a real-life scenario, this chapter introduces the sensor setup designed to collect real-time data from the asset. It provides a detailed description of the sensor equipment employed and explains how the various components of the experimental setup are interconnected. Additionally, this chapter outlines the approach used to establish a real-time connection and perform data analysis directly from the measurement system. Finally, it discusses the nature of the collected data, including the recorded data points and the overall dataset characteristics, providing a comprehensive overview of the experimental framework.

7.1. Measurement Setup

The sensor setup consists of a FBG strain sensor strip and a camera. As determined the desired sensor configuration is SC3, which consists of a total of 12 sensors. The availability of the FBG was limited, meaning that a slightly sub optimal strip is used. The used FBG sensor is visualized in Figure 7.1, which is a 20 meter long cable with a total of 20 measurement points at 1 meter intervals.

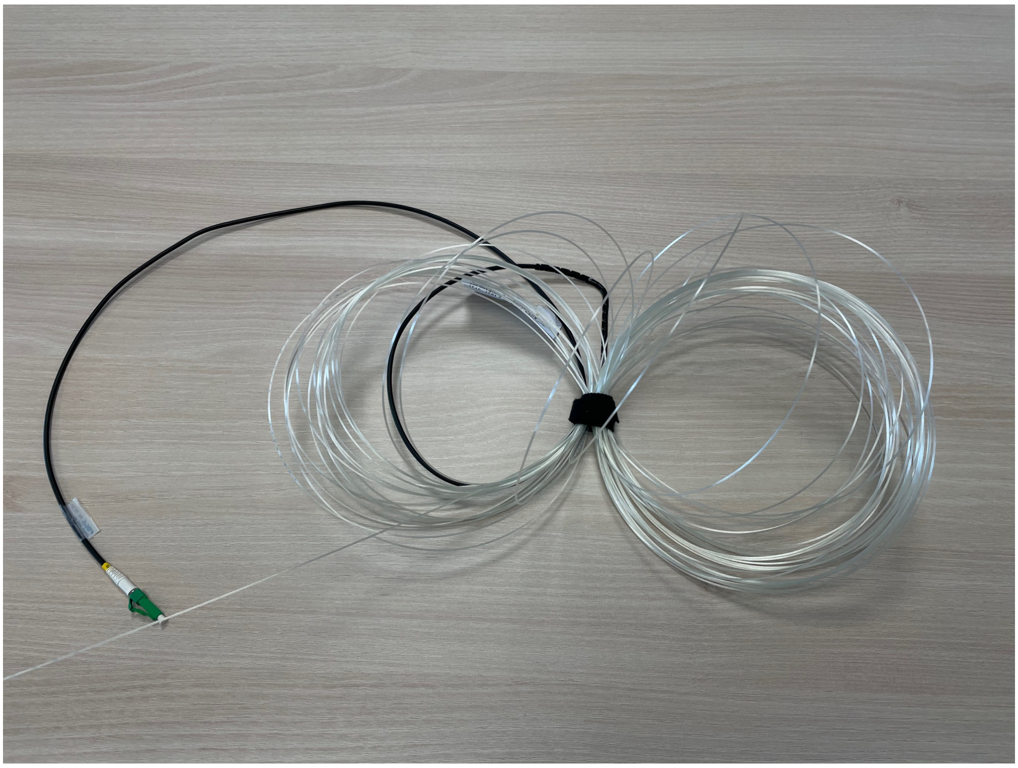


Figure 7.1: Picture of the used FBG sensor before installation.

To securely attach the FBG cable to the bridge deck, a special type of structural adhesive is used. This adhesive ensures that the cable is firmly bonded, allowing it to accurately sense and transmit every strain experienced by the bridge deck. The process of connecting the FBG sensor to the bridge with structural glue involves the following steps:

- Marking the desired sensor locations with a marking at 5 cm distance on either side for the 10 cm glue length.
- Sanding the desired sensor locations up to 5 cm on both sides of the marking, such that the glue sticks to the material.
- Drawing a line of glue along a 10 cm path in line with the desired measurement direction. The line is done in an up-and-down motion to cover enough space for the line to fall in.
- Laying down one of the marked measurement points of the FBG sensor centered in the glue line.
- Taping up the FBG strip to the asset on either side using duct tape, such that it does not move out of position.
- Pressing down on the sensor location to press out all of the air underneath the sensor strip.
- Laying down a line of glue on top of the sensor strip to fully coat the strip in glue.
- Waiting a day for the glue to fully dry.

The instrumented FBG sensor strip is shown in Figure 7.2. Due to the cable's length and the specific positions required for the sensors, the FBG cable had to be routed back and forth across the left and right bottom flanges. This arrangement ensured a neat and secure fit while allowing the excess cable to be properly attached to the underside of the bridge. By securing the excess cable in this way, the risk of it flapping around was reduced, contributing to a cleaner and more functional setup.

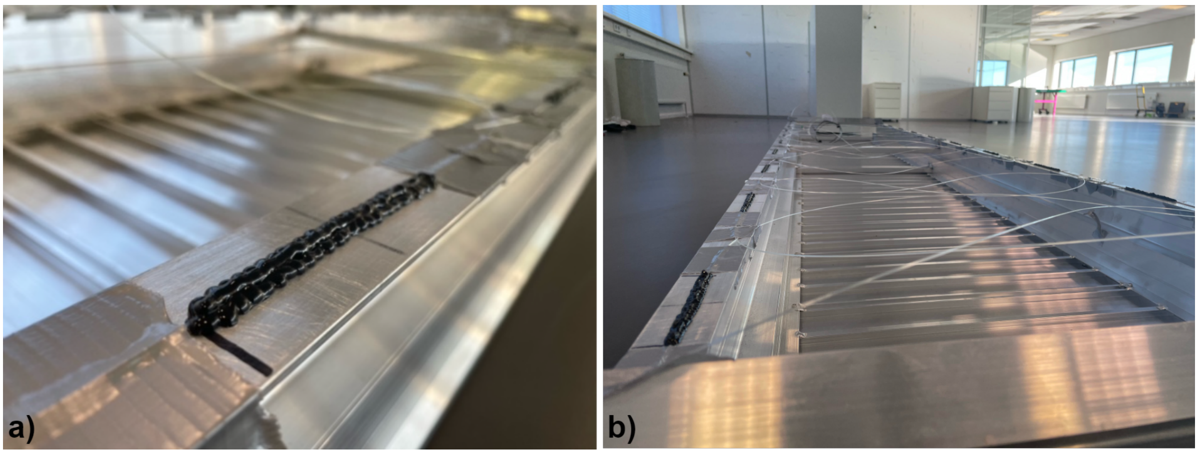


Figure 7.2: a) Detail view of one of the FBG sensor locations connected to the asset. b) Visualization of the instrumented FBG sensor strip over the entire bridge

The camera used for the testing is an Axis Q6055-E, shown in Figure 7.3. The camera has the functionality of providing a live stream of the camera image. This live stream can then be accessed by the Python script to capture an image at any desired time frame. This has the added benefit of not requiring data storage beyond the direct images captured from the live stream. The camera is positioned at feet height, halfway down the width of the bridge, viewing the bridge from a direction perpendicular to the walking direction. A single camera can be used to estimate a in this orientation. A more complex setup of multiple cameras can be used to more precisely estimate the location of footsteps in two dimensions.

The sensors and the laptop running the computational script are all connected to the same router, enabling high-bandwidth data transfer with low latency. The general data flow is illustrated in Figure 7.4. This setup is also adaptable for use in external locations. The required equipment, including a router, can be deployed on-site to transfer data either via Ethernet to a local computer or, ideally, over the internet to a cloud-based computer for processing.

Figure 7.3: Axis Q6055-E Camera, as used in the experimental setup.

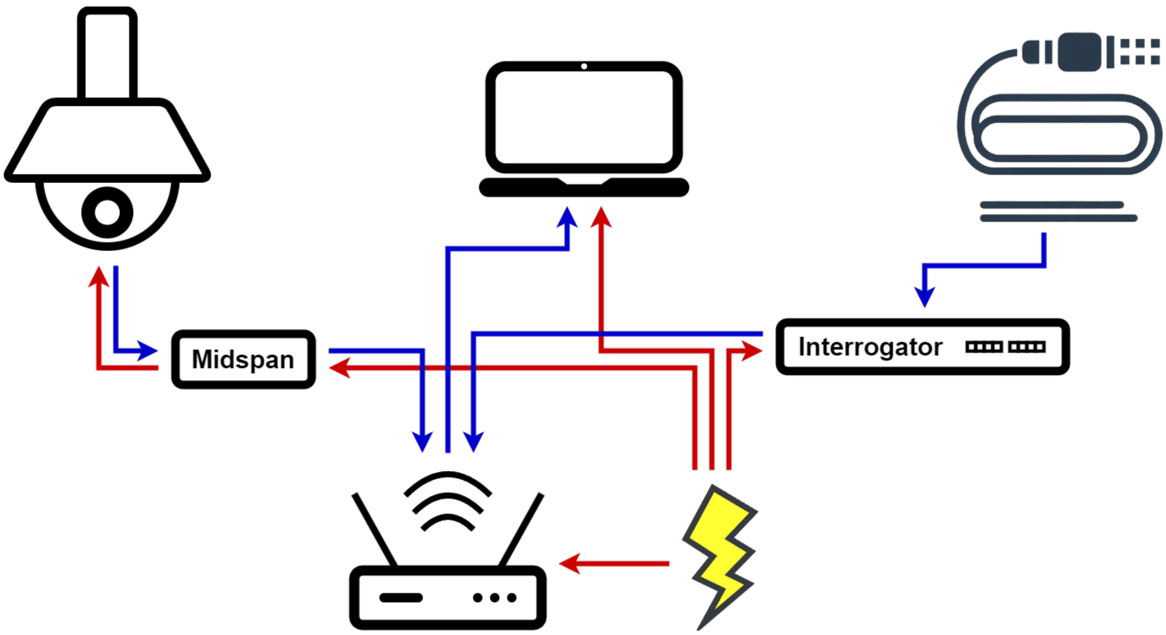


Figure 7.4: Diagram of power- (in red) and data flow (in blue) within the test setup.

7.1.1. Risk Analysis

The measurement setup involves high-value equipment, making it essential to manage risks associated with improper installation. One potential issue is the application of the strain sensors. The optical strain sensors, as shown in Figure 7.1, are custom-made to specific lengths and can only be installed on the bridge once. While the sensors themselves are not the most costly component, their replacement and re-installation can lead to significant delays due to delivery times.

In the context of traffic or railway bridges, this risk requires careful consideration, as replacing sensors could necessitate a temporary bridge closure. Moreover, traffic bridges may present additional challenges, such as sensors being located in areas that are difficult to access compared to the controlled environment of the test setup. To mitigate these risks, installing redundant cables could be beneficial, even though it involves a higher initial investment. Determining the number of sensors and their exact locations in advance is crucial to ensure that the sensors can be installed correctly in a single attempt.

Another risk is the potential for the FBG sensor to break during installation or handling. The material's susceptibility to snapping when excessively bent became evident during the testing phase. The cable snapped during testing, leading to delays in acquiring some of the desired data. While the cable can be repaired by welding it back together, this process requires specialized equipment. To minimize this risk, it is strongly recommended to preplan the exact routing of the cable on the asset, including its endpoint leading to the Interrogator device. A well-defined cable route without loose or unsupported sections will significantly reduce the likelihood of breakage during installation or operation.

7.2. Achieving Real-Time Data Transfer

The device executing the optimization script is connected to a router, as illustrated in Figure 7.4. Both the camera and sensors transmit data via an Ethernet connection. Data transfer delays are not problematic, even if the computer is located remotely or operates in the cloud, as long as the exact recording time of the sensor data is accurately logged for synchronization purposes. The primary requirement is that the Ethernet cables and router have adequate speed to handle the data transfer, which is easily achievable with modern equipment.

7.3. Achieving Real-Time Data Analysis

The Real-Time Assessment script must process all input data into the desired outputs in real time. Failure to do so could result in the analysis lagging behind, potentially providing outdated information for months or even years. While the system could theoretically catch up during less busy periods, such as nights or weekends, but relying on this would compromise the robustness of the system design.

In the experimental setup, the laptop used for testing was unable to perform the real-time assessment pipeline continuously without requiring pauses between applied loads to complete calculations. A significant portion of computational resources was dedicated to image recognition, which is inherently more optimized for cars than shoes. Advancements in this field are likely to improve performance for future iterations of this application.

For a full-scale implementation, the computational demands do not increase significantly, as the heavy calculations related to the large FEM model are precomputed. The primary factors influencing calculation time are the number of grid points, sensors, and loads. For real-world applications, deploying more capable computational hardware, such as modern cloud computing systems, will likely resolve these limitations. The required processing speeds are expected to be well within the capabilities of current technologies.

7.4. Obtained Data Description

For the experimental evaluation of the real-time assessment pipeline's performance, the following methodology was implemented. A total of 50 locations were randomly generated, each with unique random a and b coordinates. To standardize the testing, the same force magnitude F was used for all loads, corresponding to the weight of the test load applied to the asset during the experiment. The generated single-load scenarios are detailed in Table 7.1.

All of these randomly generated loads were analyzed using the FEM model to calculate the hot spot stresses at critical details and the strains at the locations corresponding to the sensor measurement points. These FEM results serve as a baseline for comparison with the experimental data and are referred to as Data Acquisition Modality 1 (DAM1).

The strain values from the FEM model, at the locations specified by SC3, are then fed into the Real-Time Assessment pipeline. This provides estimations of stress states and loads based on the digital sensor measurements, which is termed the Simulation Estimation or DAM2.

Table 7.1: Randomly generated load locations for experimental testing.

#	a [mm]	b [mm]	F [N]
DP 1	1495	300	-834
DP 2	2654	217	-834
DP 3	1334	125	-834
DP 4	459	154	-834
DP 5	498	133	-834
DP 6	2347	254	-834
DP 7	1502	198	-834
DP 8	557	227	-834
DP 9	300	25	-834
DP 10	1661	3	-834
DP 11	1147	12	-834
DP 12	2162	201	-834
DP 13	1101	148	-834
DP 14	1431	26	-834
DP 15	385	279	-834
DP 16	1571	124	-834
DP 17	1624	195	-834
DP 18	2411	113	-834
DP 19	1878	349	-834
DP 20	1456	94	-834
DP 21	718	100	-834
DP 22	353	2	-834
DP 23	519	307	-834
DP 24	984	203	-834
DP 25	853	187	-834

(a) First 25 rows.

#	a [mm]	b [mm]	F [N]
DP 26	151	225	-834
DP 27	786	343	-834
DP 28	2621	228	-834
DP 29	1022	262	-834
DP 30	699	238	-834
DP 31	1648	21	-834
DP 32	1459	108	-834
DP 33	1624	285	-834
DP 34	383	213	-834
DP 35	1181	194	-834
DP 36	1648	135	-834
DP 37	857	42	-834
DP 38	1687	94	-834
DP 39	236	346	-834
DP 40	258	259	-834
DP 41	214	159	-834
DP 42	349	179	-834
DP 43	352	33	-834
DP 44	1873	304	-834
DP 45	1173	313	-834
DP 46	2561	46	-834
DP 47	1603	70	-834
DP 48	1270	167	-834
DP 49	289	285	-834
DP 50	1801	10	-834

(b) Last 25 rows.

For the experimental analysis, the same 50 locations listed in Table 7.1 were marked on the bridge using tape. These marked locations are illustrated in Figure 7.5. A person sequentially applied a load at each marked location while the Real-Time Assessment pipeline was fully operational. The strain data collected by the FBG sensor strip during this process is referred to as Experimental Measurement or DAM3.

Next, the data from the SC3 sensors was used as input to the Real-Time Assessment pipeline to estimate loads, stress states, and, critically for this analysis, the strain values at eight additional sensor locations excluded during the fingerprinting process. This approach is referred to as the Experimental Estimation or DAM4.

This methodology thus defines four distinct Data Acquisition Modalities (DAMs) used in the experimental analysis:

- DAM1: Simulation calculation (FEM)
- DAM2: Fingerprinting, with input strain values from simulation
- DAM3: Experimental measurement (sensors)
- DAM4: Fingerprinting, with input strain values from experiment

The 8 sensors that were left out for the fingerprinting by choosing SC3 are now used to compare errors of different DAMs. These 8 sensors are L2, L4, L7, L9, R2, R4, R7, R9 determined by their location on the left or right flange and order within the sensor strip, as seen in Figure 4.2.

During the experimental analysis of DAM3 and DAM4, the bridge was loaded at the specified load locations. The process of applying a load to the bridge, remaining stationary, and then stepping off takes a certain amount of time. As a result, multiple estimations from the fingerprinting algorithm were generated while the load was applied, considering both the time the load remained in place and the cut-off value for strain readings.

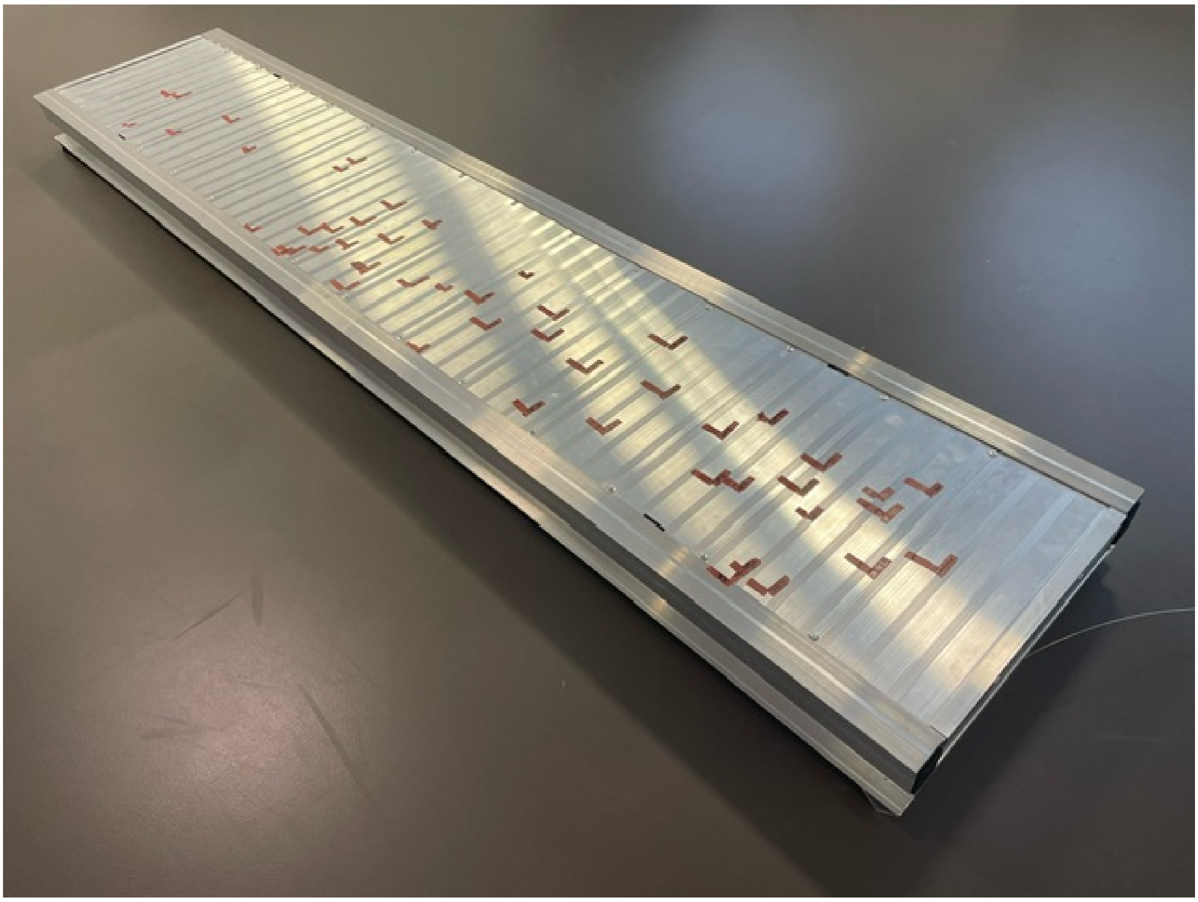


Figure 7.5: Marked locations on the asset for single-load controlled testing.

The data used for estimation was selected based on the clear identification of the load being fully applied, as verified by the image recognition process. The data points that met this criterion were then averaged to provide the estimated load locations, stresses, and strain values in the eight redundant sensors.

In total, the following data was collected and analyzed:

- 50 single load footsteps
- 1203 optimized data points
- 120 seconds of recorded data

The results from the single load location tests for DAM1, DAM2, DAM3, and DAM4 are provided in Appendix J.

7.5. Example Analysis

An example is provided for one of the 50 single load footsteps with the parameters: $a = 1495$ mm, $b = 300$ mm, $F = 834$ N. All the steps involved in obtaining the estimations for this example are presented to demonstrate the working principle of the developed fingerprinting algorithm.

7.5.1. DAM1 Example

DAM1 consists of data generated by the FEM model for a specific load case. The single load footstep with the parameters: $a = 1495$ mm, $b = 300$ mm, $F = 834$ N is considered. This load is applied to the FEM model as seen in Figure 7.6.

The FEM model is solved for this load scenario. The global FEM model provides the strain values $\epsilon_i \ \forall i \in I$ in the z-direction at the same locations where the sensors are positioned on the asset, as shown in Figure 7.7 a.

Additionally, the sub-models are solved to obtain the hot-spot values at $0.4t$ and $1.0t$ from the weld toe. These values are used to calculate the hot-spot stresses $\sigma_s \ \forall s \in S$ at the weld toe using Equation 4.1, as illustrated in Figure 7.7 b.

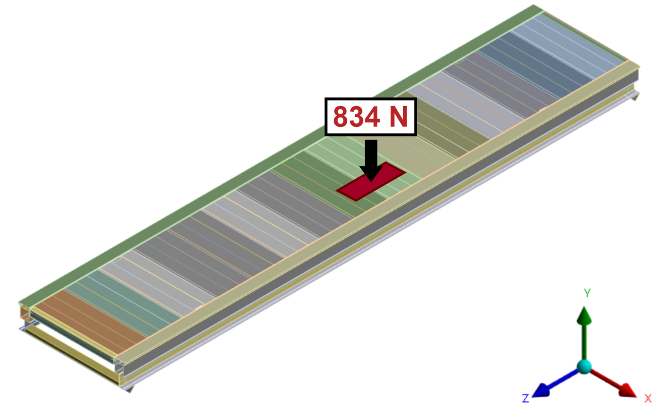


Figure 7.6: Single load footstep with the parameters: $a = 1495$ mm, $b = 300$ mm, $F = 834$ N, as applied to the global FEM model.

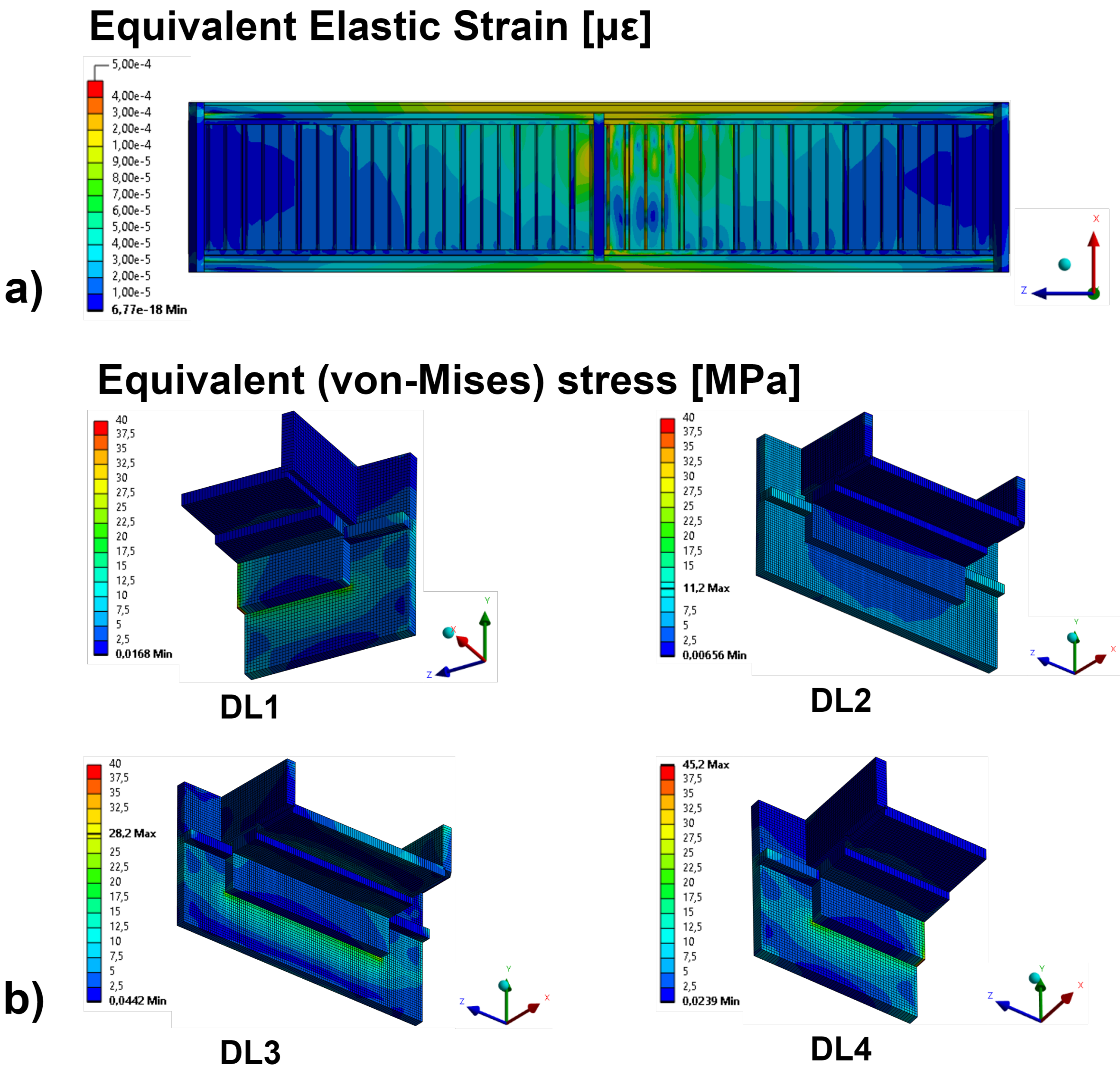


Figure 7.7: a) Bottom view of the global FEM model, showing the strain in the solved structure for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N. b) Sub-models of the FEM model, showing the stress in the solved structure for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

The FEM model output for the specified load case includes both strain and stress values. The strain results are summarized in Table 7.2, while the stress results are detailed in Table 7.3.

a [mm]	1495
b [mm]	300
F [N]	-834
ϵ_0 (Sensor L1) [$\mu\epsilon$]	11.0
ϵ_1 (Sensor L2) [$\mu\epsilon$]	27.6
ϵ_2 (Sensor L3) [$\mu\epsilon$]	44.1
ϵ_3 (Sensor L4) [$\mu\epsilon$]	59.8
ϵ_4 (Sensor L5) [$\mu\epsilon$]	73.9
ϵ_5 (Sensor L6) [$\mu\epsilon$]	81.0
ϵ_6 (Sensor L7) [$\mu\epsilon$]	69.9
ϵ_7 (Sensor L8) [$\mu\epsilon$]	51.2
ϵ_8 (Sensor L9) [$\mu\epsilon$]	32.2
ϵ_9 (Sensor L10) [$\mu\epsilon$]	13.6
ϵ_{10} (Sensor R1) [$\mu\epsilon$]	24.1
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	47.2
ϵ_{12} (Sensor R3) [$\mu\epsilon$]	76.7
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	110.8
ϵ_{14} (Sensor R5) [$\mu\epsilon$]	144.0
ϵ_{15} (Sensor R6) [$\mu\epsilon$]	162.6
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	138.0
ϵ_{17} (Sensor R8) [$\mu\epsilon$]	95.6
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	58.4
ϵ_{19} (Sensor R10) [$\mu\epsilon$]	29.0

Table 7.2: Load parameters for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N, and resulting strain values from DAM1.

σ_0 [MPa]	22.43
σ_1 [MPa]	2.12
σ_2 [MPa]	0.89
σ_3 [MPa]	-1.30
σ_4 [MPa]	1.71
σ_5 [MPa]	-1.12
σ_6 [MPa]	-21.97
σ_7 [MPa]	-1.80
σ_8 [MPa]	-25.29
σ_9 [MPa]	-0.29
σ_{10} [MPa]	23.30
σ_{11} [MPa]	2.31
σ_{12} [MPa]	-23.32
σ_{13} [MPa]	-2.71
σ_{14} [MPa]	7.03
σ_{15} [MPa]	-1.06
σ_{16} [MPa]	6.42
σ_{17} [MPa]	-0.81
σ_{18} [MPa]	1.61
σ_{19} [MPa]	2.74
σ_{20} [MPa]	-4.77
σ_{21} [MPa]	6.69
σ_{22} [MPa]	-19.89
σ_{23} [MPa]	-2.44

Table 7.3: Resulting stress values from DAM1 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

7.5.2. DAM2 Example

DAM2 involves executing the fingerprinting algorithm using the strain values obtained from DAM1. Since SC3 was selected as the sensor configuration, the strain data is limited to the active sensor locations in SC3. This excludes strain values $\epsilon_1, \epsilon_3, \epsilon_6, \epsilon_8, \epsilon_{11}, \epsilon_{13}, \epsilon_{16}, \epsilon_{18}$ from Table 7.2. The remaining active sensor data used as input for the fingerprinting algorithm is provided in Table 7.4.

Fingerprinting continues only if at least one of these strain values exceeds the cut-off magnitude $\epsilon_c = 10\ \mu\epsilon$, which is the case for a given input. As DAM2 focuses on a single load case, no image recognition is applied and $n_{\text{footsteps}}$ is set to 1.

ϵ_0 (Sensor L1) [$\mu\epsilon$]	11.0
ϵ_2 (Sensor L3) [$\mu\epsilon$]	44.1
ϵ_4 (Sensor L5) [$\mu\epsilon$]	73.9
ϵ_5 (Sensor L6) [$\mu\epsilon$]	81.0
ϵ_7 (Sensor L8) [$\mu\epsilon$]	51.2
ϵ_9 (Sensor L10) [$\mu\epsilon$]	13.6
ϵ_{10} (Sensor R1) [$\mu\epsilon$]	24.1
ϵ_{12} (Sensor R3) [$\mu\epsilon$]	76.7
ϵ_{14} (Sensor R5) [$\mu\epsilon$]	144.0
ϵ_{15} (Sensor R6) [$\mu\epsilon$]	162.6
ϵ_{17} (Sensor R8) [$\mu\epsilon$]	95.6
ϵ_{19} (Sensor R10) [$\mu\epsilon$]	29.0

Table 7.4: Input strain data used for DAM2 example as obtained from DAM1.

The first step of the fingerprinting algorithm is the **coupling** of the best matching grid point $n_{\min}$. In this example, all points in the GP3 database are considered since no initial position guess is made, as the camera data is not utilized. For each grid point in the database, the MAD is calculated using Equation 4.23. The MAD values calculated for all grid points are shown in Figure 7.8, where the grid point color represent the MAD value. The lower the MAD, the better the match. The best matching grid point $n_{\min}$ is highlighted.

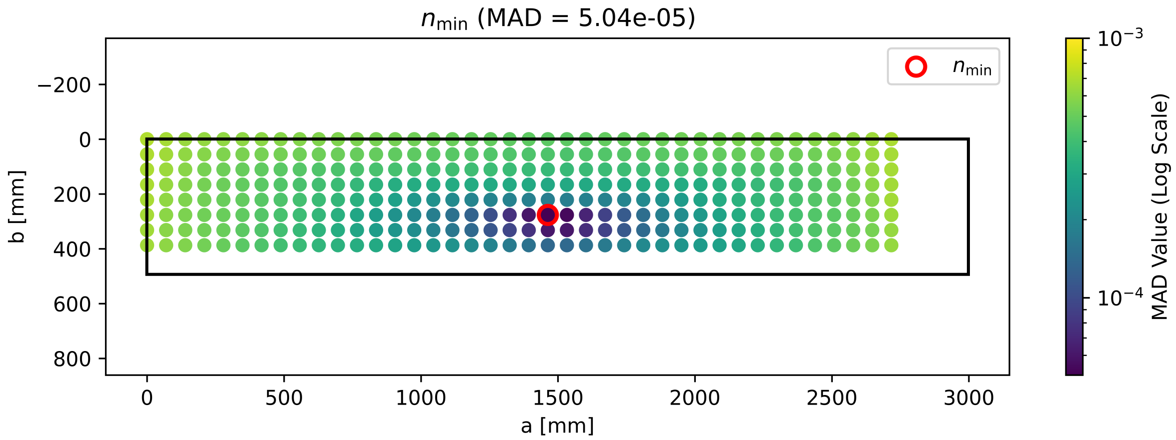


Figure 7.8: Example result for coupling step. Showing the best matching grid point $n_{\min}$ with the minimum MAD out of the grid points.

The second step of the fingerprinting algorithm involves **interpolation** between the optimal grid point $n_{\min}$, and the surrounding grid points. For each surrounding area $h \ \forall h \in H$, the $\text{MAD}^{(h)}$ is computed using Equation 4.31. The results are shown in Figure 7.9. The area with the lowest $\text{MAD}^{(h)}$, identified as $h_{\min}$ the bottom-right quadrant in this case.

The scale factors for each grid point within $h_{\min}$, $\beta_k, h_{\min} \ \forall k \in K$, 0.31, 0.24, 0.19, and 0.26, respectively. Using these scale factors, the load location coordinates $\{a_{\text{est}}, b_{\text{est}}\}$ are estimated with Equations 4.29 and 4.30. The estimated coordinates are 1494.63, 300.23, which closely match the actual load location of $a, b = 1495, 300$. This estimation is visualized in Figure 7.10.

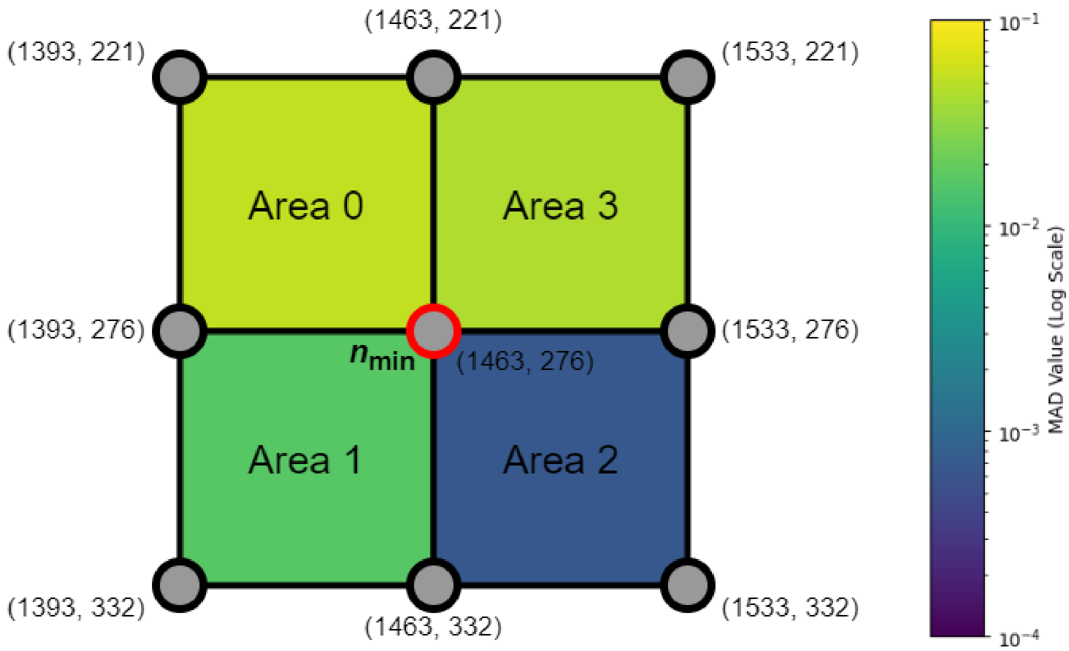


Figure 7.9: Example of the area location during the interpolation step. The $\text{MAD}^{(h)}$ values for each area are represented by the corresponding area color. The area with the lowest $\text{MAD}^{(h)}$, identified as $h_{\min}$, is Area 2.

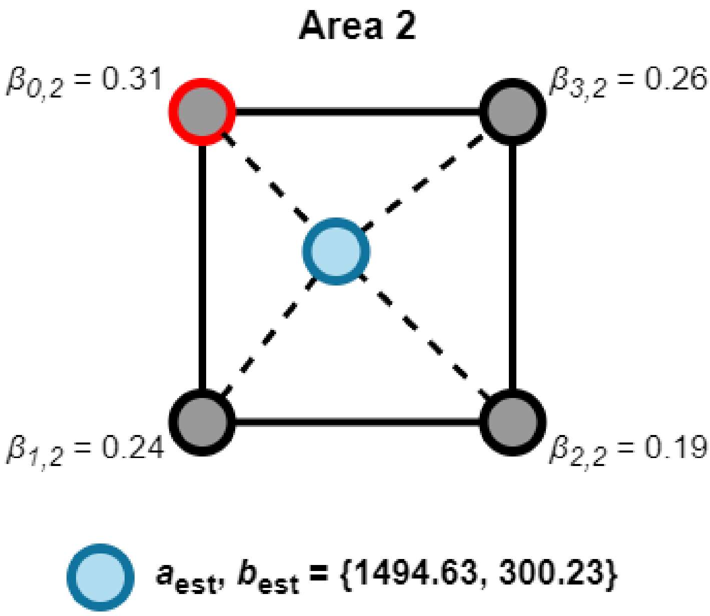


Figure 7.10: Example interpolation between grid points in optimal area h_{min} around the optimal grid point n_{min}. Showing the scale factors of each grid point.

The third step of the fingerprinting algorithm involves **scaling** the force magnitude, performed using Equation 4.5.4. The scale factor α is calculated as 1.043, resulting in an estimated force magnitude F_{est} of 834.03 N, as determined by Equation 4.33 (Figure 7.11). This brings the estimated parameters to $a_{\mathrm{est}}, b_{\mathrm{est}}, F_{\mathrm{est}} = 1494.63, 300.23, 834.03$, closely matching the initial single load case $a, b, F = 1495, 300, 834$.

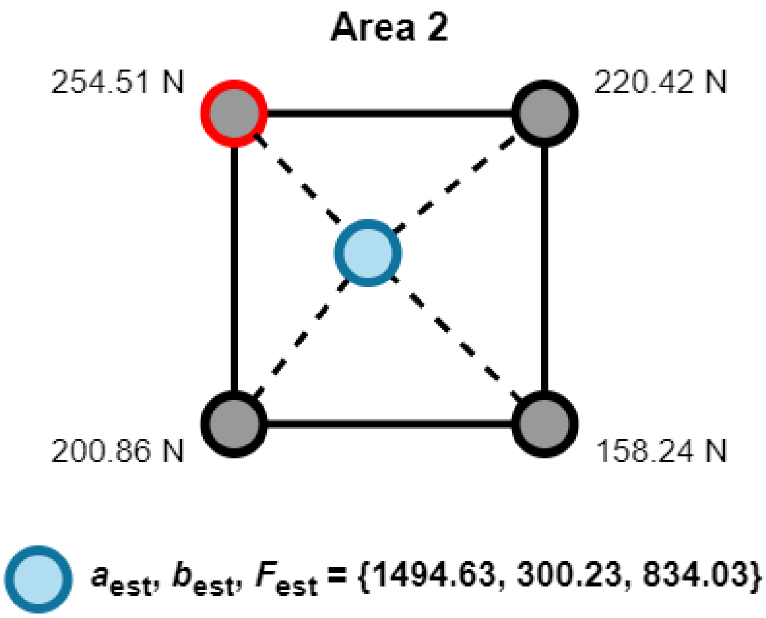


Figure 7.11: Example interpolation between grid points in optimal area h_{min} around the optimal grid point n_{min}. Showing the scaled force magnitude in each grid point.

The strain values in the extra sensors L2, L4, L7, L9, R2, R4, R7, R9, as well as the hot-spot stress in all details $\sigma_s \ \ \forall s \in S$ are determined according to Equation 4.34. The output from DAM2 is shown in Table 7.5 and Table 7.6.

a_{est} [mm]	1494.63
b_{est} [mm]	300.23
F_{est} [N]	-834.03
ϵ_1 (Sensor L2) [$\mu\epsilon$]	27.6
ϵ_3 (Sensor L4) [$\mu\epsilon$]	59.8
ϵ_6 (Sensor L7) [$\mu\epsilon$]	69.9
ϵ_8 (Sensor L9) [$\mu\epsilon$]	32.2
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	47.3
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	110.9
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	137.9
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	58.4

Table 7.5: Output parameters and strains from DAM2 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

σ_0 [MPa]	22.51
σ_1 [MPa]	2.13
σ_2 [MPa]	0.84
σ_3 [MPa]	-1.30
σ_4 [MPa]	1.66
σ_5 [MPa]	-1.12
σ_6 [MPa]	-22.08
σ_7 [MPa]	-1.77
σ_8 [MPa]	-25.00
σ_9 [MPa]	-0.32
σ_{10} [MPa]	23.39
σ_{11} [MPa]	2.32
σ_{12} [MPa]	-23.44
σ_{13} [MPa]	-2.73
σ_{14} [MPa]	6.91
σ_{15} [MPa]	-1.07
σ_{16} [MPa]	6.32
σ_{17} [MPa]	-0.82
σ_{18} [MPa]	1.32
σ_{19} [MPa]	2.94
σ_{20} [MPa]	-3.97
σ_{21} [MPa]	6.32
σ_{22} [MPa]	-20.07
σ_{23} [MPa]	-2.46

Table 7.6: Output hot-spot stresses from DAM2 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

7.5.3. DAM3 Example

DAM3 involves experimental data collected from the FBG sensors on the asset, under the same load parameters as the example: $a = 1495$ mm, $b = 300$ mm, $F = 834$ N. A single iteration of this experimental data, recorded at the timestamp 2024-11-8 13:28:01.339193 (see Figure 7.13), is analyzed. While the strain values from the sensors used in DAM4's fingerprinting analysis were not stored, additional sensor data from locations L2, L4, L7, L9, R2, R4, R7, and R9 at this timestamp are listed in Table 7.12.

The strain values differ significantly from those obtained from the FEM model in DAM1. These discrepancies may be attributed to various factors: slight deviations in the actual load location on the asset during DAM3, inaccuracies in sensor placement or performance, or differences in behavior between the FEM model and the physical asset.

The most plausible explanation for such notable inaccuracies, particularly near the center of the bridge, is a mismatch between the FEM model and the real-world behavior of the asset. This indicates that the FEM model is not fully accurate. While more thorough validation of the FEM model is needed, time constraints prevented such refinements during this study.

ϵ_1 (Sensor L2) [$\mu\epsilon$]	30.58
ϵ_3 (Sensor L4) [$\mu\epsilon$]	67.63
ϵ_6 (Sensor L7) [$\mu\epsilon$]	86.98
ϵ_8 (Sensor L9) [$\mu\epsilon$]	36.59
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	38.53
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	129.72
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	169.81
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	57.92

Figure 7.12: Experimental sensor data from DAM3 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

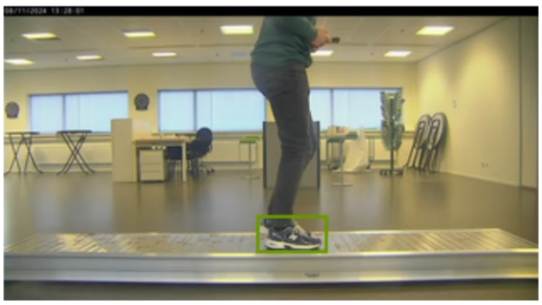


Figure 7.13: Experimental setup image from DAM3 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

7.5.4. DAM4 Example

DAM4 consists of executing the fingerprinting algorithm based on the input measurement data from DAM3. As mentioned, the input has not been saved, but the cut-off strain was reached at this iteration. Thus a camera image was coupled and the image recognition correctly recognized the shoe in this image as seen in Figure 7.13. With the knowledge that $n_{\text{footsteps}} = 1$, the fingerprinting algorithm is initiated.

The first step of the fingerprinting algorithm is the **coupling** of the best matching grid point n_{min}. In this example, only the the grid points within the boundary boxes from the initial camera location guess are considered. For each grid point within this box, the MAD is calculated using Equation 4.23. The best matching grid point n_{min} from this minimization is shown in Figure 7.14.

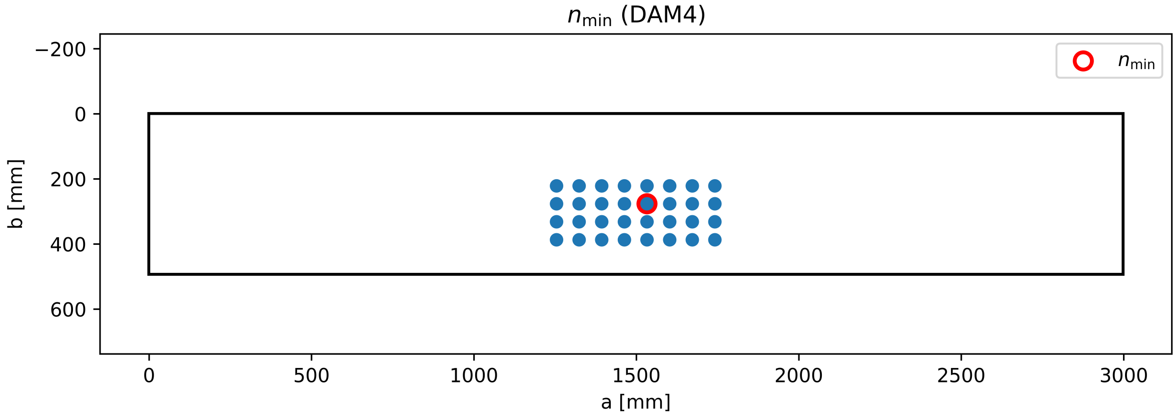


Figure 7.14: Example result for coupling step. Showing the best matching grid point n_{min} for DAM4.

The second step of the fingerprinting algorithm involves **interpolation** between the optimal grid point n_{min}, and the surrounding grid points. For each surrounding area $h \;\; \forall h \in H$, $\text{MAD}^{(h)}$ is computed using Equation 4.31. The area with the lowest $\text{MAD}^{(h)}$, identified as h_{min} the bottom-left quadrant in this case, as shown in Figure 7.15.

The scale factors $\beta_{k,h_{\text{min}}} \;\; \forall k \in K$ for each grid point within h_{min}, are 0.14, 0.41, 0.07, and 0.38, respectively. Using these scale factors, the load location coordinates $\{a_{\text{est}}, b_{\text{est}}\}$ are estimated with Equations 4.29 and 4.30. The estimated coordinates are 1494.05, 302.84, which closely match the actual load location of $a, b = 1495, 300$. This estimation is visualized in Figure 7.16.

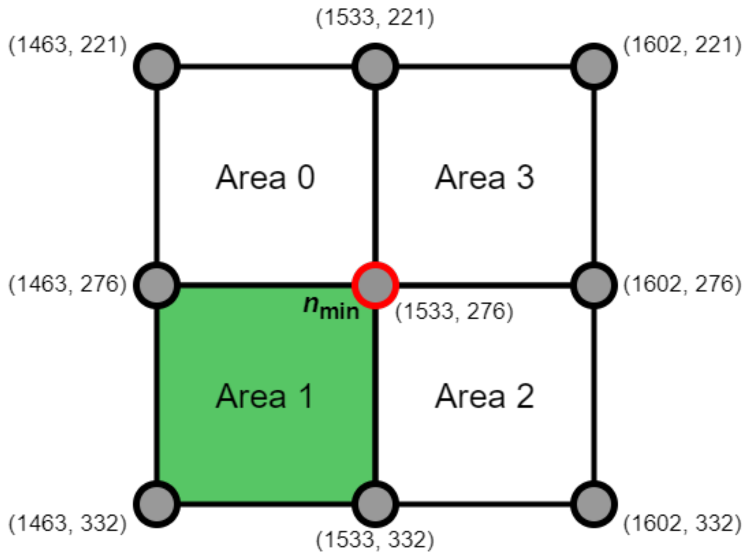


Figure 7.15: Example of the area location during the interpolation step from DAM4. The area with the lowest $\text{MAD}^{(h)}$, identified as h_{min}, is Area 1 (marked in green).

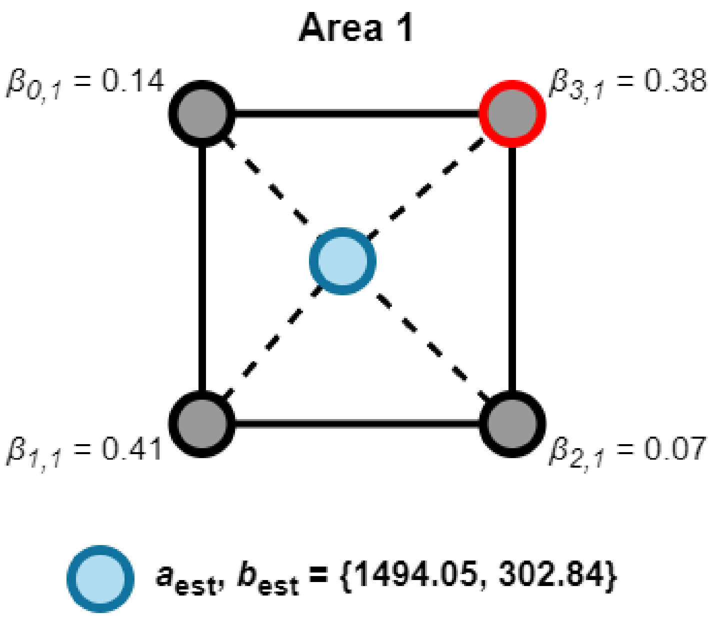


Figure 7.16: Example interpolation for DAM4 between grid points in optimal area h_{min} around the optimal grid point n_{min}. Showing the scale factors of each grid point.

The third step of the fingerprinting algorithm involves **scaling** the force magnitude, performed using Equation 4.5.4. The scale factor α is calculated as 1.25, resulting in an estimated force magnitude F_{est} of 1002.47 N, as determined by Equation 4.33 (Figure 7.17). This brings the estimated parameters to $a_{\mathrm{est}}, b_{\mathrm{est}}, F_{\mathrm{est}} = 1494.05, 302.84, 1002.47$, closely matching the initial single load case $a, b, F = 1495, 300, 834$ for the coordinates. However, a notable discrepancy is observed in the load magnitude. This difference is likely due to inaccuracies in the FEM model, which caused deviations between the strain values used in the database and those experimentally measured.

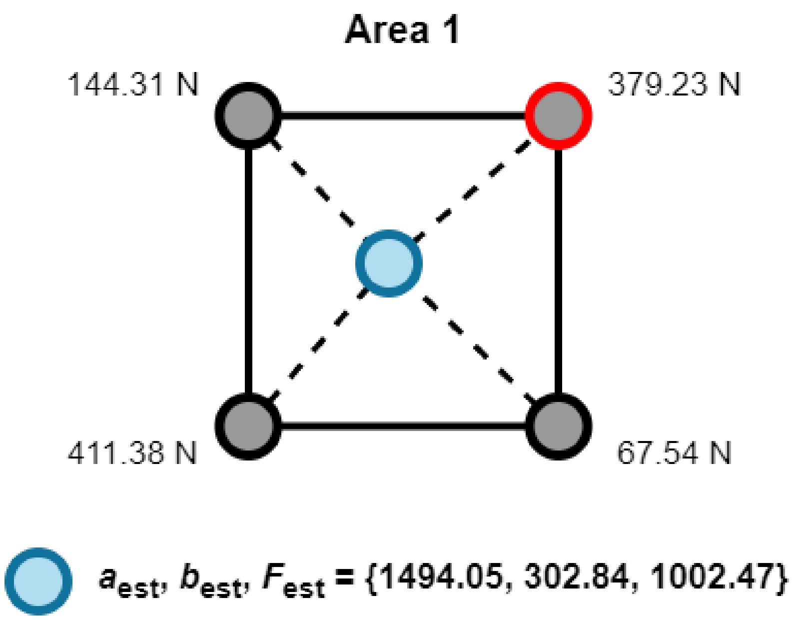


Figure 7.17: Example interpolation for DAM4 between grid points in optimal area h_{min} around the optimal grid point n_{min}. Showing the scaled force magnitude in each grid point.

The strain values in the extra sensors L2, L4, L7, L9, R2, R4, R7, R9, as well as the hot-spot stress in all details $\sigma_s \;\; \forall s \in S$ are determined according to Equation 4.34. The output from DAM4 is shown in Table 7.7 and Table 7.8.

a_{est} [mm]	1494.05
b_{est} [mm]	302.84
F_{est} [N]	-1002.47
ϵ_1 (Sensor L2) [$\mu\epsilon$]	33.1
ϵ_3 (Sensor L4) [$\mu\epsilon$]	71.7
ϵ_6 (Sensor L7) [$\mu\epsilon$]	83.0
ϵ_8 (Sensor L9) [$\mu\epsilon$]	38.3
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	58.0
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	136.1
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	168.1
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	71.1

Table 7.7: Output parameters and strains from DAM4 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

σ_0 [MPa]	27.74
σ_1 [MPa]	2.64
σ_2 [MPa]	0.89
σ_3 [MPa]	-1.57
σ_4 [MPa]	1.88
σ_5 [MPa]	-1.35
σ_6 [MPa]	-26.50
σ_7 [MPa]	-2.26
σ_8 [MPa]	-29.97
σ_9 [MPa]	-0.53
σ_{10} [MPa]	28.80
σ_{11} [MPa]	2.87
σ_{12} [MPa]	-28.95
σ_{13} [MPa]	-3.36
σ_{14} [MPa]	8.29
σ_{15} [MPa]	-1.30
σ_{16} [MPa]	7.60
σ_{17} [MPa]	-1.00
σ_{18} [MPa]	1.76
σ_{19} [MPa]	3.55
σ_{20} [MPa]	-4.55
σ_{21} [MPa]	7.59
σ_{22} [MPa]	-24.84
σ_{23} [MPa]	-3.04

Table 7.8: Output hot-spot stresses from DAM4 for the single load case $a = 1495$ mm, $b = 300$ mm, $F = 834$ N.

7.6. Multi-Load Example

The methodology developed in this research has been carefully designed to ensure scalability for scenarios involving multiple simultaneous loads. This chapter presents a detailed data example of a multi-load situation, as determined through simulation studies. For this analysis, only DAM1 and DAM2 are considered, as this specific load configuration was not experimentally applied to the physical asset.

7.6.1. Multi-Load DAM1

DAM1 consists of data generated by the FEM model for a specific load case. The multi-load scenario where $n_{\text{footsteps}} = 3$ with the parameters: $a_0 = 2654$ mm, $b_0 = 217$ mm, $F_0 = 1102$ N; $a_1 = 349$ mm, $b_1 = 179$ mm, $F_1 = 1486$ N; $a_2 = 1173$ mm, $b_2 = 313$ mm, $F_2 = 791$ N is considered. This multi-load scenario is applied to the FEM model as seen in Figure 7.18.

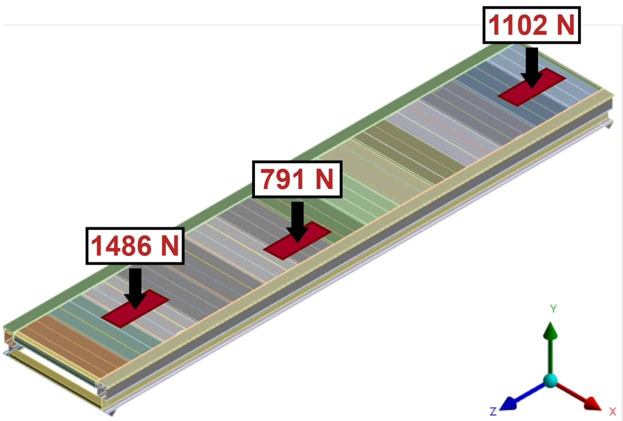


Figure 7.18: Multi-load scenario as applied to the global FEM model.

The FEM model is solved for this load scenario. The global FEM model provides the strain values $\epsilon_i \ \forall i \in I$ in the z-direction at the same locations where the sensors are positioned on the asset, as shown in Figure 7.19 a.

Additionally, the sub-models are solved to obtain the hot-spot values at $0.4t$ and $1.0t$ from the weld toe. These values are used to calculate the hot-spot stresses $\sigma_s \ \forall s \in S$ at the weld toe using Equation 4.1, as illustrated in Figure 7.19 b.

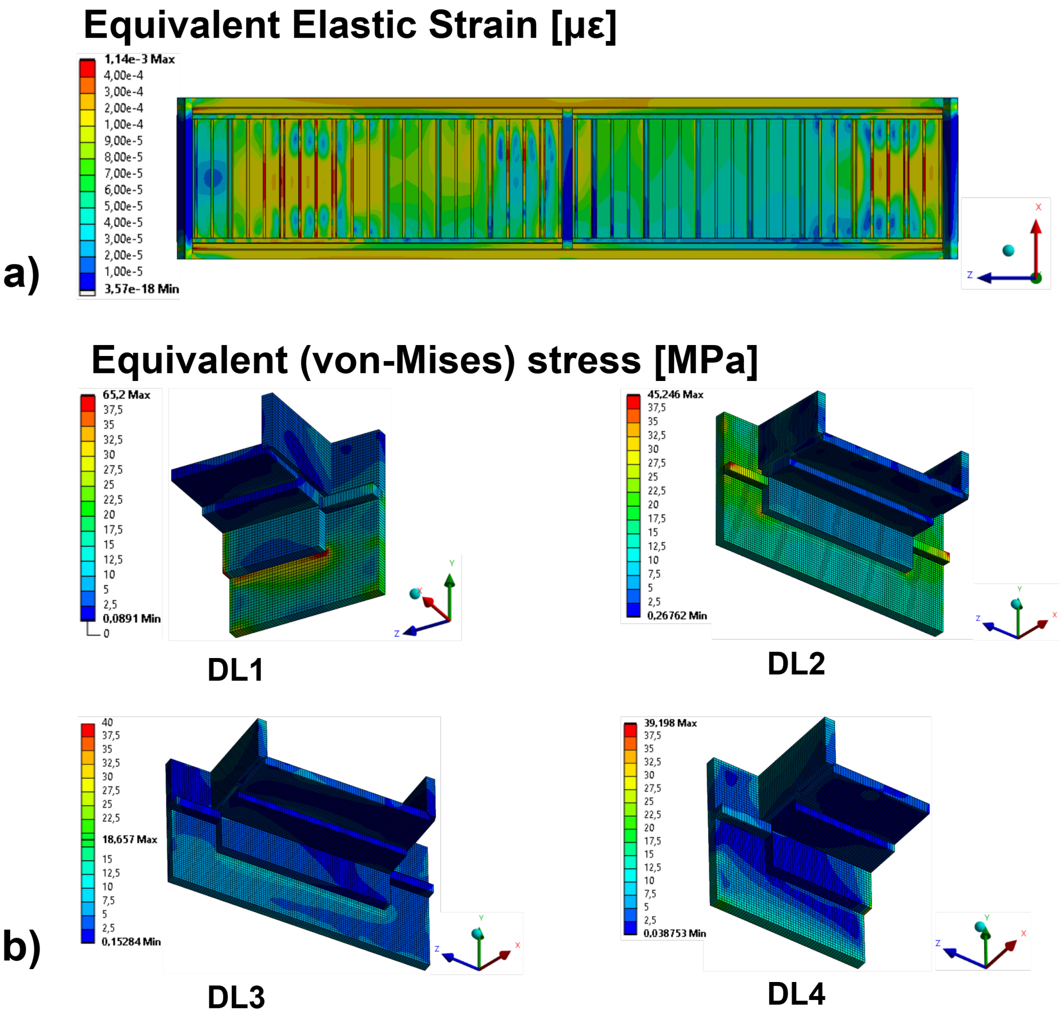


Figure 7.19: a) Bottom view of the global FEM model, showing the strain in the solved structure for a multi-load scenario where $n_{\text{footsteps}} = 3$. b) Sub-models of the FEM model, showing the stress in the solved structure for a multi-load scenario where $n_{\text{footsteps}} = 3$

The FEM model output for the specified load case includes both strain and stress values. The strain results are summarized in Table 7.9, while the stress results are detailed in Table 7.10.

7.6.2. DAM2 Example

DAM2 involves executing the fingerprinting algorithm using the strain values obtained from DAM1. Since SC3 was selected as the sensor configuration, the strain data is limited to the active sensor locations in SC3. This excludes strain values $\epsilon_1, \epsilon_3, \epsilon_6, \epsilon_8, \epsilon_{11}, \epsilon_{13}, \epsilon_{16}, \epsilon_{18}$ from Table 7.9. The remaining active sensor data used as input for the fingerprinting algorithm is provided in Table 7.11.

Fingerprinting continues only if at least one of these strain values exceeds the cut-off magnitude $\epsilon_c = 10\ \mu\epsilon$, which is the case for a given input. No image recognition is applied during DAM2, the number of footsteps determined is set at $n_{\text{footsteps}}$=3. A significant inaccuracy in the camera is considered. For this example the coordinates as determined by the camera are set to be: $a_{cam_0} = 2709$ mm, $b_{cam_0} = 236$ mm; $a_{cam_1} = 529$ mm, $b_{cam_1} = 80$ mm; $a_{cam_2} = 1146$ mm, $b_{cam_2} = 319$ mm. With a boundary box for considered grid points at $a_{box}, b_{box} = \{300, 100\}$ around the coordinates as determined by the camera.

a_0, a_1, a_2 [mm]	2654, 349, 1173
b_0, b_1, b_2 [mm]	217, 179, 313
F_0, F_1, F_2 [N]	-1102, -1486, -791
ϵ_0 (Sensor L1) [$\mu\epsilon$]	81.2
ϵ_1 (Sensor L2) [$\mu\epsilon$]	164.1
ϵ_2 (Sensor L3) [$\mu\epsilon$]	187.0
ϵ_3 (Sensor L4) [$\mu\epsilon$]	183.3
ϵ_4 (Sensor L5) [$\mu\epsilon$]	169.4
ϵ_5 (Sensor L6) [$\mu\epsilon$]	149.3
ϵ_6 (Sensor L7) [$\mu\epsilon$]	132.9
ϵ_7 (Sensor L8) [$\mu\epsilon$]	111.5
ϵ_8 (Sensor L9) [$\mu\epsilon$]	94.2
ϵ_9 (Sensor L10) [$\mu\epsilon$]	63.9
ϵ_{10} (Sensor R1) [$\mu\epsilon$]	92.0
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	180.7
ϵ_{12} (Sensor R3) [$\mu\epsilon$]	224.7
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	250.5
ϵ_{14} (Sensor R5) [$\mu\epsilon$]	250.5
ϵ_{15} (Sensor R6) [$\mu\epsilon$]	217.5
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	183.7
ϵ_{17} (Sensor R8) [$\mu\epsilon$]	146.9
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	120.3
ϵ_{19} (Sensor R10) [$\mu\epsilon$]	83.2

Table 7.9: Load parameters for a multi-load scenario where $n_{\text{footsteps}} = 3$, and resulting strain values from DAM1.

σ_0 [MPa]	47.05
σ_1 [MPa]	4.09
σ_2 [MPa]	25.36
σ_3 [MPa]	0.88
σ_4 [MPa]	18.47
σ_5 [MPa]	0.17
σ_6 [MPa]	-12.06
σ_7 [MPa]	0.86
σ_8 [MPa]	-10.64
σ_9 [MPa]	-1.51
σ_{10} [MPa]	0.62
σ_{11} [MPa]	-0.05
σ_{12} [MPa]	-2.10
σ_{13} [MPa]	-1.00
σ_{14} [MPa]	27.76
σ_{15} [MPa]	1.22
σ_{16} [MPa]	23.70
σ_{17} [MPa]	0.33
σ_{18} [MPa]	12.99
σ_{19} [MPa]	7.52
σ_{20} [MPa]	18.77
σ_{21} [MPa]	2.73
σ_{22} [MPa]	-38.30
σ_{23} [MPa]	-4.13

Table 7.10: Resulting stress values from DAM1 for a multi-load scenario where $n_{\text{footsteps}} = 3$.

ϵ_0 (Sensor L1) [$\mu\epsilon$]	81.2
ϵ_2 (Sensor L3) [$\mu\epsilon$]	187.0
ϵ_4 (Sensor L5) [$\mu\epsilon$]	169.4
ϵ_5 (Sensor L6) [$\mu\epsilon$]	149.3
ϵ_7 (Sensor L8) [$\mu\epsilon$]	111.5
ϵ_9 (Sensor L10) [$\mu\epsilon$]	63.9
ϵ_{10} (Sensor R1) [$\mu\epsilon$]	92.0
ϵ_{12} (Sensor R3) [$\mu\epsilon$]	224.7
ϵ_{14} (Sensor R5) [$\mu\epsilon$]	250.5
ϵ_{15} (Sensor R6) [$\mu\epsilon$]	217.5
ϵ_{17} (Sensor R8) [$\mu\epsilon$]	146.9
ϵ_{19} (Sensor R10) [$\mu\epsilon$]	83.2

Table 7.11: Input strain data used for DAM2 example as obtained from DAM1.

The first step of the fingerprinting algorithm is the **coupling** of the best matching grid point n_{min_c} $\forall c \in C$ for all of the loads in the multi-load scenario iteratively. Following the strategy as illustrated in Figure 4.11.

Footstep 1: The first iteration considers footstep 1 ($a_{cam_0} = 2709$ mm, $b_{cam_0} = 236$ mm) at all grid points of the subset within the boundary box. The other two footstep locations are locked at the grid point closest to the camera estimation. For each grid point in the subset of grid points, the MAD is calculated using Equation 4.23. The MAD values calculated for all grid points are shown in Figure 7.20 *c*, where the grid point color represent the MAD value. The lower the MAD, the better the match. The best matching grid point n_{min_0} is highlighted.

Footstep 2: The second iteration considers footstep 2 ($a_{cam_1} = 529$ mm, $b_{cam_1} = 80$ mm) at all grid points of the subset within the boundary box. The other two footstep locations are locked at the previously determined optimal (n_{min_0}), and the grid point closest to the camera estimation (n_{min_2}). For each grid point in the subset of grid points, the MAD is calculated using Equation 4.23. The MAD values calculated for all grid points are shown in Figure 7.20 *d*, where the grid point color represent the MAD value. The lower the MAD, the better the match. The best matching grid point $n_{\min_1}$ is highlighted.

Footstep 3: The third iteration considers footstep 3 (a_{cam_2} = 1146 mm, b_{cam_2} = 319 mm) at all grid points of the subset within the boundary box. The other two footstep locations are locked at the previously determined optimal $n_{\min_0}$ and $n_{\min_1}$. For each grid point in the subset of grid points, the MAD is calculated using Equation 4.23. The MAD values calculated for all grid points are shown in Figure 7.20 e, where the grid point color represent the MAD value. The lower the MAD, the better the match. The best matching grid point $n_{\min_2}$ is highlighted.

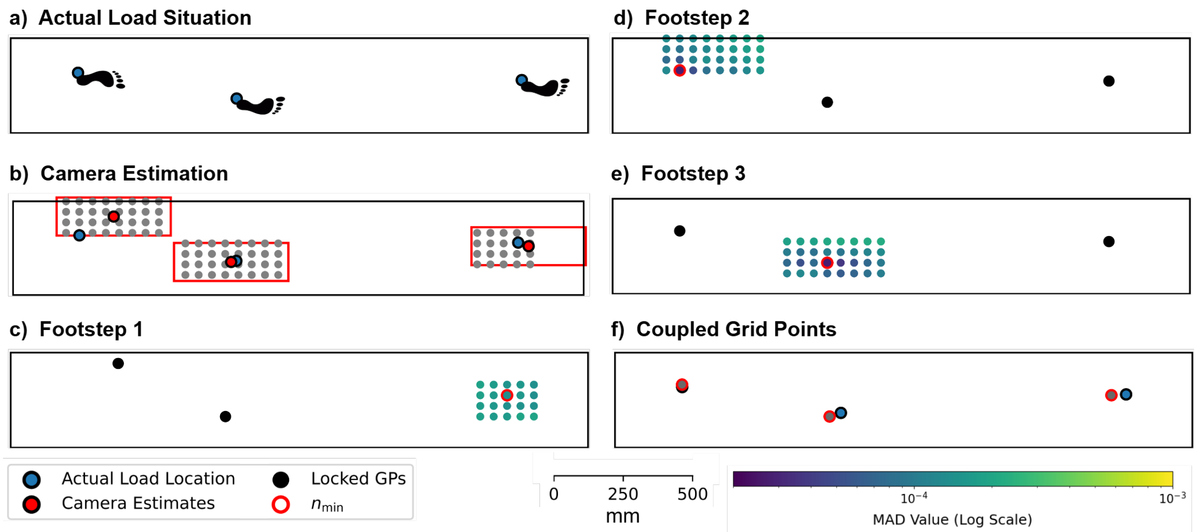


Figure 7.20: Example result for coupling step for $n_{\text{footsteps}} = 3$. Showing the coupled grid points $n_{\min_c}$ $\forall c \in C$ with the minimum MAD out of the considered subset of grid points.

The second step of the fingerprinting algorithm involves **interpolation** between the optimal grid points $n_{\min_c}$ $\forall c \in C$, and their surrounding grid points. For each surrounding area h $\forall h \in H$, the MAD$^{(h)}$ is computed using Equation 4.31. The results are shown in Figure 7.21. The area with the lowest MAD$^{(h)}$, identified as $h_{\min}$.

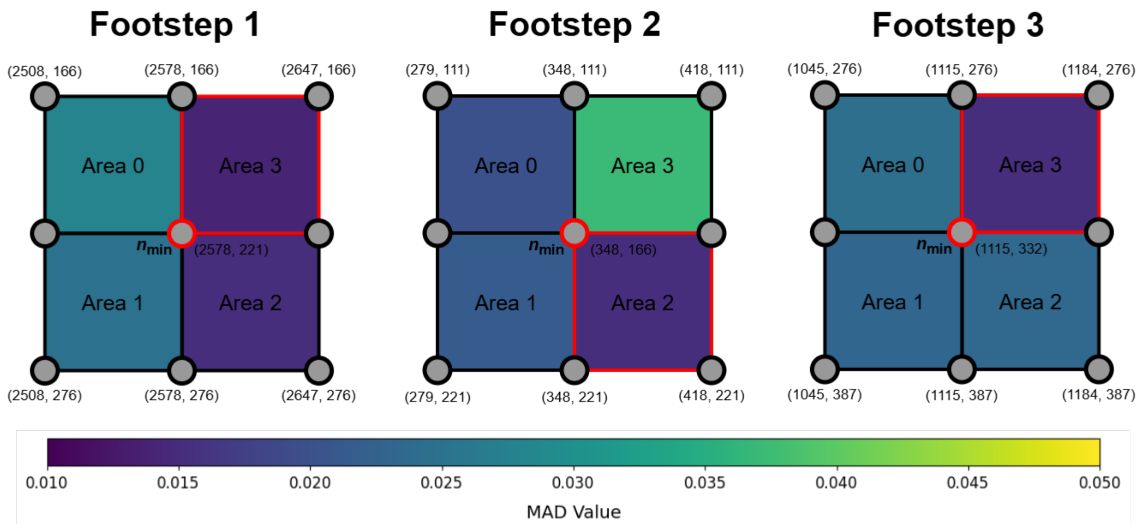


Figure 7.21: Determining optimal area h for each footstep $n_{\min_c}$ $\forall c \in C$. The MAD$^{(h)}$ values for each area are represented by the corresponding area color. The area with the lowest MAD$^{(h)}$, identified as $h_{\min}$ is marked in red.

Using these scale factors, the load location coordinates $\{a_{\text{est}_c}, b_{\text{est}_c}\}$ are estimated for all $c \in C$ with Equations 4.29 and 4.30. The estimated coordinates are $a_{\text{est}_0} = 2615.50$ mm, $b_{\text{est}_0} = 193.60$ mm; $a_{\text{est}_1} = 372.38$ mm, $b_{\text{est}_1} = 170.07$ mm; $a_{\text{est}_2} = 1156.17$ mm, $b_{\text{est}_2} = 326.65$ mm, which closely match the actual load locations of $a_0 = 2654$ mm, $b_0 = 217$ mm; $a_1 = 349$ mm, $b_1 = 179$ mm; $a_2 = 1173$ mm, $b_2 = 313$ mm. This estimation is visualized in Figure 7.22.

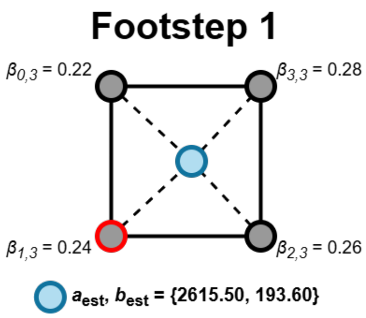


Figure 7.22: Example interpolation between grid points in optimal area h_{min} around the optimal grid points n_{min_c} $\forall c \in C$. Showing the scale factors of each grid point.

The third step of the fingerprinting algorithm involves **scaling** the force magnitude, performed using Equation 4.5.4. The scale factors α_c $\forall c \in C$ are determined using Equation 4.33, resulting in estimated force magnitudes $F_{\text{est}_c} = 996.21$ N, 1448.30 N, 766.60 N, This result is visualized in Figure 7.23.

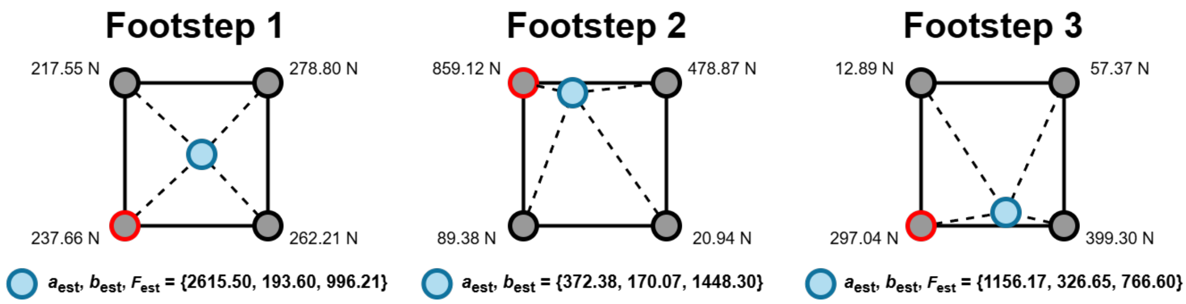


Figure 7.23: Example interpolation between grid points in optimal areas h_{min_c} around the optimal grid point n_{min_c}. Showing the scaled force magnitude in each grid point.

The strain values in the extra sensors L2, L4, L7, L9, R2, R4, R7, R9, as well as the hot-spot stress in all details σ_s $\forall s \in S$ are determined according to Equation 4.34. The combined output from DAM2 for the multi-load example is shown in Table 7.12 and Table 7.13. The estimation overall was very successful.

$a_{\text{est}_0}, a_{\text{est}_1}, a_{\text{est}_2}$ [mm]	2615.50, 193.60, 372.38
$b_{\text{est}_0}, b_{\text{est}_1}, b_{\text{est}_2}$ [mm]	193.60, 170.07, 326.65
$F_{\text{est}_0}, F_{\text{est}_1}, F_{\text{est}_2}$ [N]	-996, -1448, -767
ϵ_1 (Sensor L2) [$\mu\epsilon$]	163.8
ϵ_3 (Sensor L4) [$\mu\epsilon$]	184.6
ϵ_6 (Sensor L7) [$\mu\epsilon$]	134.3
ϵ_8 (Sensor L9) [$\mu\epsilon$]	98.9
ϵ_{11} (Sensor R2) [$\mu\epsilon$]	176.1
ϵ_{13} (Sensor R4) [$\mu\epsilon$]	252.3
ϵ_{16} (Sensor R7) [$\mu\epsilon$]	183.8
ϵ_{18} (Sensor R9) [$\mu\epsilon$]	117.9

Table 7.12: Output parameters and strains from DAM2 for the multi load case example.

σ_0 [MPa]	48.04
σ_1 [MPa]	4.22
σ_2 [MPa]	24.08
σ_3 [MPa]	0.89
σ_4 [MPa]	18.22
σ_5 [MPa]	0.19
σ_6 [MPa]	-8.84
σ_7 [MPa]	0.78
σ_8 [MPa]	-8.24
σ_9 [MPa]	-1.30
σ_{10} [MPa]	12.82
σ_{11} [MPa]	1.38
σ_{12} [MPa]	-3.73
σ_{13} [MPa]	-1.19
σ_{14} [MPa]	26.52
σ_{15} [MPa]	1.03
σ_{16} [MPa]	22.62
σ_{17} [MPa]	0.16
σ_{18} [MPa]	13.65
σ_{19} [MPa]	6.74
σ_{20} [MPa]	18.16
σ_{21} [MPa]	2.47
σ_{22} [MPa]	-30.36
σ_{23} [MPa]	-3.38

Table 7.13: Output hot-spot stresses from DAM2 for the multi load case example.

7.7. Experimental Output

This chapter explores the various results obtained from the Real-Time Assessment pipeline as applied to DAM4 for all of the 50 single load locations combined. It also illustrates what the stored output might resemble during operation. The results encompass several key aspects, including computational speed, load estimations, and stress range frequencies. Additionally, it introduces a dashboard designed to provide a clear and accessible overview of the system's real-time status. This dashboard serves as an intuitive interface, offering a snapshot of the ongoing analysis and highlighting critical metrics to ensure efficient monitoring and understanding of the pipeline's performance during each iteration.

7.7.1. Computational Speed

The desired iteration speed was set at 10 Hz. During the test of all 50 load locations in DAM3 and DAM4, lasting approximately 64 minutes, a total of 38,574 iterations were performed. Of these, 1,203 iterations exceeded the cut-off threshold in at least one strain sensor, triggering the execution of the fingerprinting algorithm.

For the iterations exceeding the cut-off threshold, the average speed was approximately 2.55 seconds per iteration, corresponding to a frequency of 0.39 Hz. This included all components of the analysis—image recognition, fingerprinting, and rainflow counting—but fell short of the target 10 Hz. The most significant contributor to this slowdown was the image recognition process, which requires substantial optimization. Running at a lower frequency introduces delays, necessitating the storage of sensor data and camera images, which is undesirable. Prolonged delays could lead to storage limitations or an inability to process data in real-time, causing information to become outdated. To mitigate this, transitioning the analysis to a faster computing device or leveraging cloud computing could offer substantial performance improvements.

For iterations where the cut-off threshold was not reached, the average speed was approximately 0.086 seconds per iteration, equating to a frequency of 11.6 Hz, which surpasses the desired 10 Hz. However, the iteration speed is capped at 10 Hz unless the process is actively compensating for delays from above-threshold calculations.

7.7.2. Estimation Results

The results from each estimation of load location(s) and magnitude(s) are optionally stored for analysis. For every measurement where a strain value exceeds the cut-off value, the pipeline provides an estimation. The saved data includes the timestamp of the measurement, the estimated coordinates (a_{est_c} and b_{est_c}), and the estimated force magnitude for each load (F_{est_c}).

If redundant strain sensors are present—those intentionally excluded from the fingerprinting algorithm (as discussed further in Chapter 8.2)—their strain values are also estimated. The saved results include both the actual strain values measured by these redundant sensors and the corresponding estimated strain values. This comprehensive data storage ensures detailed tracking and validation of the pipeline's performance. One such example output is shown in Table 7.14, which uses the same load case as Chapter 7.5.

Timestamp	8-11-2024 13:28:01
a_{est_0} [mm]	1425.05
b_{est_0} [mm]	302.84
F_{est_0} [N]	-1002.47
ϵ_0 [µε]	30.58
ϵ_2 [µε]	67.63
ϵ_5 [µε]	86.98
ϵ_7 [µε]	36.59
ϵ_{10} [µε]	38.53
ϵ_{12} [µε]	129.72
ϵ_{15} [µε]	169.81
ϵ_{18} [µε]	57.92
$\epsilon_{0_{est}}$ [µε]	33.1
$\epsilon_{2_{est}}$ [µε]	71.7
$\epsilon_{5_{est}}$ [µε]	83.0
$\epsilon_{7_{est}}$ [µε]	38.3
$\epsilon_{10_{est}}$ [µε]	58.0
$\epsilon_{12_{est}}$ [µε]	136.1
$\epsilon_{15_{est}}$ [µε]	168.1
$\epsilon_{18_{est}}$ [µε]	71.1

Table 7.14: An example of an optional output includes the load and strain estimations alongside the measured strain values for a specific applied load. In this case, the load parameters are $a = 1495$, $b = 300$ and $F = -834$.

7.7.3. Coupled Grid Point Frequency

Another important output is a frequency counter that tracks how often each grid point was coupled. For every grid point, the total count is recorded, as illustrated in Figure 7.24. This output provides valuable insights into the common footstep locations and highlights the frequency of usage for specific parts of the asset, offering a clear understanding of usage patterns and areas of concentrated activity.

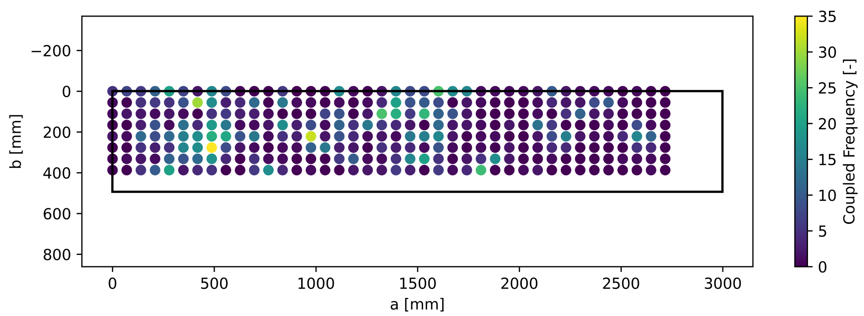


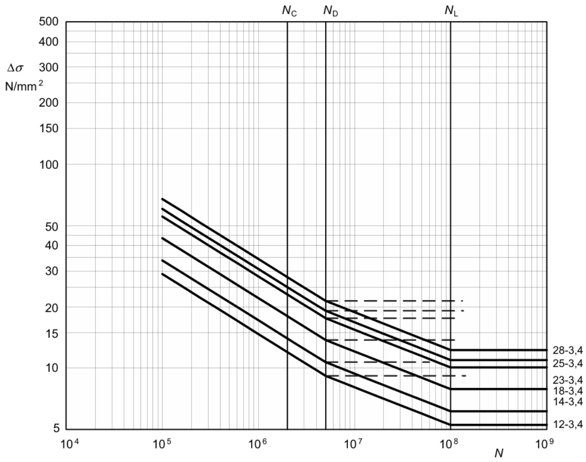
Figure 7.24: A top-down view of the asset showing all grid points. The grid points are color-coded to represent the frequency of their coupling during experimental testing.

7.7.4. Frequency per Stress Range

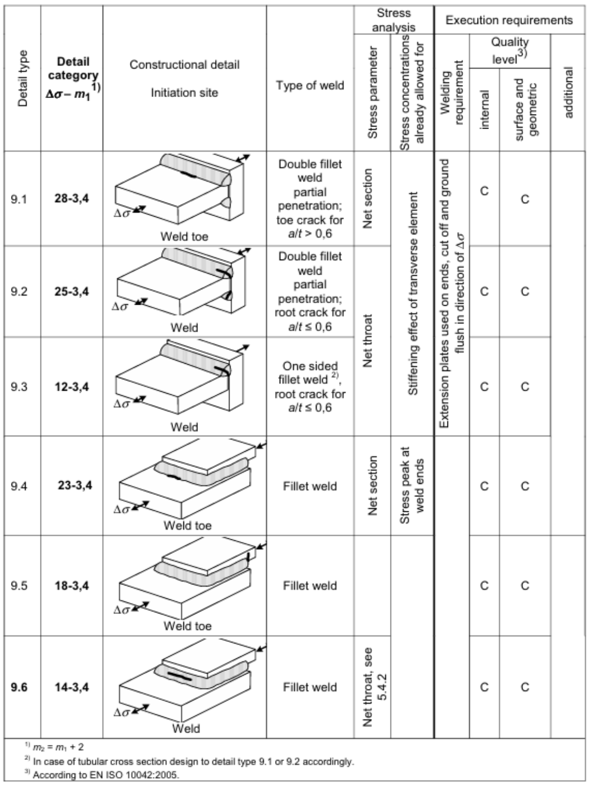
The frequency per stress range in each of the researched details, as obtained from rainflow counting, is also stored. For each detail the frequency $f_s \ \forall s \in S$ over all of the stress ranges is shown in Appendix K.

7.7.5. Cumulative Damage

We now have the frequency of stress ranges that occurred in each of the details $s \in S$ as shown in Appendix K. The next step is to determine the fatigue damage from the stress cycles in each stress range. In order to calculate the fatigue damage we need the theoretical maximum number of cycles for each stress range. For this we need the S-N curve of the material aluminium and the detail category to obtain the specific detail S-N curve. The S-N curve for aluminium for different weld categories is shown in Figure 7.25a. The researched details are all single sided fillet welds as seen in Figure 4.3 *b*. This specific detail type falls under the category 12-3,4, as shown in Figure 7.25b. The endurance limit for each of the researched stress ranges is obtained from Figure 7.25a and listed in Appendix L.



(a) S-N Curves of aluminium alloys for different detail categories. [23]



Detail type	Detail category $\Delta\sigma - m_1$ [1)]	Constructional detail / Initiation site	Type of weld	Stress analysis: Stress parameter	Stress analysis: Stress concentrations already allowed for	Execution requirements: Welding requirement	Quality level [3)]: internal	Quality level [3)]: surface and geometric	additional
9.1	**28-3,4**	Weld toe	Double fillet weld partial penetration; toe crack for $a/t > 0,6$	Net section	Stiffening effect of transverse element	Extension plates used on ends, cut off and ground flush in direction of $\Delta\sigma$	C	C	
9.2	**25-3,4**	Weld	Double fillet weld partial penetration; root crack for $a/t \leq 0,6$	Net throat			C	C	
9.3	**12-3,4**	Weld	One sided fillet weld [2)], root crack for $a/t \leq 0,6$				C	C	
9.4	**23-3,4**	Weld toe	Fillet weld	Net section	Stress peak at weld ends		C	C	
9.5	**18-3,4**	Weld toe	Fillet weld				C	C	
9.6	**14-3,4**	Weld	Fillet weld	Net throat, see 5.4.2			C	C	

[1)] $m_2 = m_1 + 2$
[2)] In case of tubular cross section design to detail type 9.1 or 9.2 accordingly.
[3)] According to EN ISO 10042:2005.

(b) Detail categories for different weld types of aluminium alloys. [23]

Figure 7.25: Comparison of S-N curves and aluminium detail categories.

By combining Appendix K and Table L, we can determine fatigue damages using the Palmgren-Miner rule as expressed in Equation 2.1. In this scenario, the maximum total fatigue damage for any detail $s \in S$ is determined to be $D_6 = 8.66 \cdot 10^{-5}$. The applied loads during testing (collected over approximately 64 minutes, $t = 1.07$ hours) are assumed to be representative, and an operational period of 8 hours per day (t_{op}) is considered. Additionally a safety factor γ_{Mf} is set at 1, as recommended in the standard [23]. Then the theoretical fatigue life of the most critical detail—and by extension, the bridge—is given by: $L = L_s = \frac{t}{365\, t_{\mathrm{op}}\, \gamma_{Mf}\, D_6} \approx 4.8$ years.

7.8. Extrapolation of Trends

Extrapolating trends within load conditions enables more accurate fatigue life predictions. When the fingerprinting algorithm is applied from the time of a bridge's commissioning, stress ranges can be estimated for the entire past lifespan and projected into the future. However, if the algorithm is implemented during the operational lifetime of the bridge, the past stress cycles are unknown and must also be extrapolated.

When the real-time assessment pipeline is applied for a limited portion of the bridge's operational lifetime, such as a few months or a year, the observed trends during this period significantly influence fatigue life predictions when extrapolating the collected data. These trends may not fully capture long-term variations, potentially leading to inaccuracies. Additionally, the predicted fatigue life is further impacted by the selection of safety factors, which account for uncertainties, and by trends in stress cycles and applied loads. These factors must be carefully considered to ensure reliable and conservative fatigue life estimations.

In this research, it is assumed that stress cycles exhibit similar behavior over time. For real-world applications, trends in frequency distributions from rainflow counting bins can be analyzed and used to refine both past and future estimations, improving the reliability of fatigue life predictions. This approach allows for a more comprehensive understanding of stress patterns and their long-term effects on structural integrity.

7.9. Dashboard

Additionally, results are clearly displayed on the dashboard, which automatically opens in a local browser to showcase the Real-Time Assessment pipeline in action. Both the current time and iteration time of the real-time assessment are visible, allowing users to determine how far behind the assessment is or if it is running in real-time.

The dashboard features the camera image corresponding to the current iteration time in the top-right corner. If any of the strain sensors have reached the cut-off value, the image also includes an overlay from the image recognition system, indicating the detected shoe locations with bounding boxes placed at those recognized spots.

The top-left section of the dashboard displays a top-down view of the bridge. If the cut-off value has not been reached during the iteration, no footstep loads are shown in this view. However, if the cut-off is exceeded, the estimated load locations and their corresponding force magnitudes (represented by color coding) are displayed in this window, as predicted by the fingerprinting algorithm. It also includes visualizations of various researched details, with color coding to indicate the estimated stress values in each detail during the specific iteration

The bottom-left section features a graph that shows the frequency of stress ranges occurring in a particular detail. Users can manually select any of the researched details to view this graph, which helps identify the most common stress types, indicative of different load conditions.

Finally, the bottom-right view displays a cumulative stress plot for all researched details, spanning the entire lifetime of the asset. This graph provides insight into the progression of cumulative damage across different details, offering an easy way to identify areas that may require maintenance based on their fatigue damage.

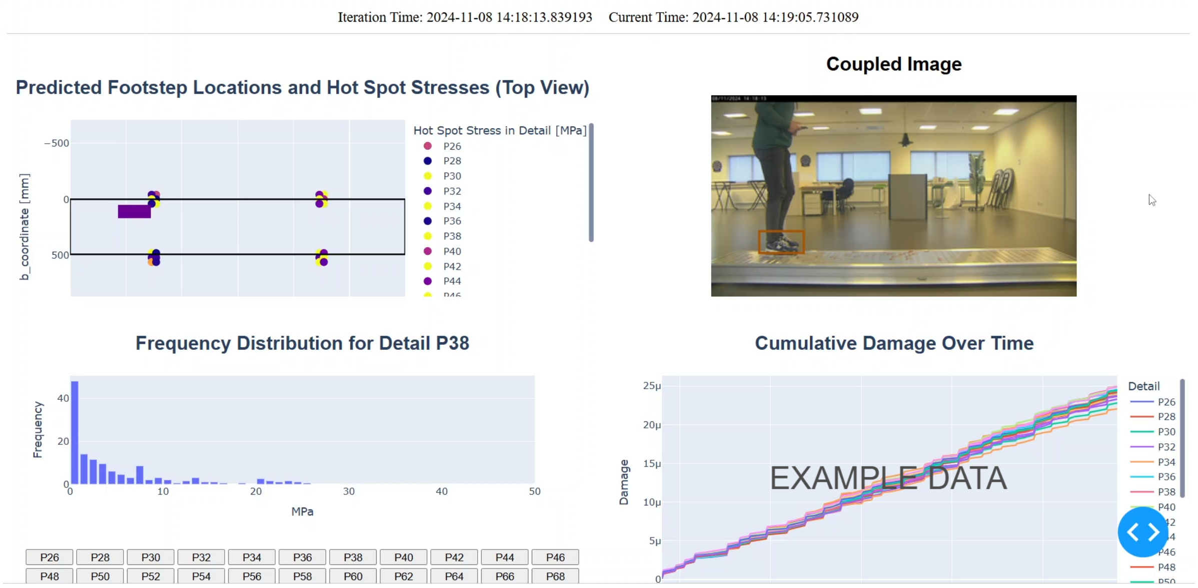


Figure 7.26: Screen capture of the dashboard during asset testing, displaying multiple useful views of the current state of the asset.

8

Results

This chapter will present the results of the research, focusing on the performance of the digital model in terms of load and stress state estimation accuracy. The results are discussed across three stages of testing. Chapter 8.1 will present the results from the simulation model, demonstrating the isolated accuracy of the fingerprinting algorithm and its potential to estimate stress states. Chapter 8.2 will cover the results from the experimental testing, where known locations on the bridge are loaded with a known force to assess the accuracy of the developed pipeline.

8.1. Digital model analysis

A new set of randomly generated digital load combinations is computed to test the methodology for coupling, interpolating, and scaling FEM data, as presented in Chapter 4, also known as the fingerprinting algorithm. This load set consists of four parts that are combined into a larger set. The subsets and the combined set generated for this purpose are outlined in Table 8.1.

$n_{\text{footsteps}}$	**Data points**
1	50
2	50
3	50
4	50
1, 2, 3, 4	200

Table 8.1: Subsets of data points used for model testing.

All the loads in this set will be individually computed in the FEM model to calculate the hot spot stresses within the detail models. The output will also include the strains at the sensor locations from SC3. By feeding these strain values into the fingerprinting model, the output will be the estimated load situation and the corresponding hot spot stresses. This enables a comparison between the estimation results from DAM2 and the FEM runs of DAM1, isolating the inaccuracy caused solely by the fingerprinting algorithm. This approach highlights the maximum potential accuracy, assuming that the provided sensor data perfectly matches the data in the FEM database.

The estimation inaccuracies between DAM1 and DAM2 are evaluated through location and hot spot stress estimation errors. Location estimation inaccuracy is quantified by the Euclidean distance error over each of the loads with

$$\Delta d = \sum_{c \in C} \sqrt{(a_c - a_{est_c})^2 + (b_c - b_{est_c})^2} \tag{8.1}$$

for each of the 200 load situations. Then Δd is averaged over the 200 load situations. These results are presented in Figure 8.1, showing a clear trend of increasing location estimation error as $n_{\text{footsteps}}$ increases. This indicates the increasing difficulty for the fingerprinting algorithm to distinguish multiple simultaneous loads on the bridge.

For the chosen grid resolution GP3 and $n_{\text{footsteps}}$, the average location inaccuracy is only 3.26 mm. Considering the bridge's total length of 3000 mm, this represents an extremely precise estimation. However, the accuracy declines as the

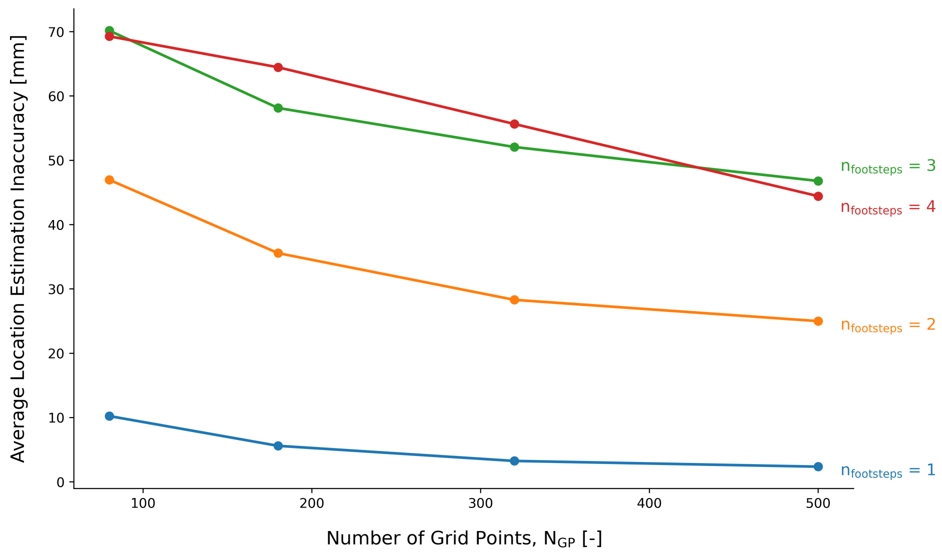


Figure 8.1: Average error in distance estimation for different grid number of grid points and number of footsteps.

number of footsteps increases, stabilizing after $n_{\text{footsteps}} = 3$. This stabilization is primarily influenced by the sensor configuration but is also affected by the bridge dimensions and the specific characteristics of the loads and their positions.

Despite the challenges posed by higher $n_{\text{footsteps}}$, the maximum error for GP3 is 56 mm, which is considered quite reasonable when compared to the overall length of the bridge, which is 3000 mm. This level of accuracy is deemed acceptable in the context of the system's performance, given the scale of the bridge. The results do underline the limitations of the fingerprinting algorithm when multiple loads are present. Increasing the number of sensors and improving image recognition techniques can significantly enhance the system's ability to handle multi-load situations. A greater number of sensors provides more detailed strain data, which improves the algorithm's capability to distinguish between multiple simultaneous loads. Meanwhile, advancements in image recognition can refine the initial load location and magnitude estimates, reducing the computational burden on subsequent steps in the pipeline and improving overall accuracy. Together, these enhancements can address the challenges posed by more complex load scenarios, leading to more precise and reliable estimates.

The hot spot stress error between the simulation results is evaluated using the median rather than the average. This approach is adopted because, at lower stress values, relative inaccuracies expressed as percentages can appear disproportionately large, even when the absolute differences are minor. Such occurrences are particularly common for loads situated far from the detail being analyzed. The relative error is determined according to

$$\%\Delta\sigma = \text{med}\left(\frac{|\sigma_{s,est} - \sigma_s|}{\sigma_s} \cdot 100\% \quad \forall s \in S\right) \tag{8.2}$$

for each of the load points in the subset, and then averaged to obtain the results. The isolated error caused by the fingerprinting algorithm are presented in Figure 8.2. The results reaffirm that the estimation error increases as $n_{\text{footsteps}}$ increases. For the selected grid point resolution GP3, the lowest median hot spot stress error is observed at $n_{\text{footsteps}} = 1$, with a value of 1.06%, while the highest error reaches 7.84% at greater $n_{\text{footsteps}}$ values. These errors are relatively small when compared to the uncertainties that must be taken into account according to the fatigue calculations in the standard. The results suggest that the methodology holds potential for practical applications, even under relatively complex loading conditions.

Interestingly, the location estimation (Figure 8.1) is influenced more by an increase in the number of loads than the hot-spot stress estimation (Figure 8.2). This is a sign that the model estimated a load configuration that does not exactly match the actual load situation, yet found an alternative that suits the geometry behavior reasonably accurately.

To show the behavior of the model when estimating individual load situations for $n_{\text{footsteps}} = 4$ are visualized in Figure 8.3. This shows that load locations often match well. The model seems to slightly struggle at appointing the right force magnitude to the correct locations, swapping around or halving combined forces of loads that are positioned close

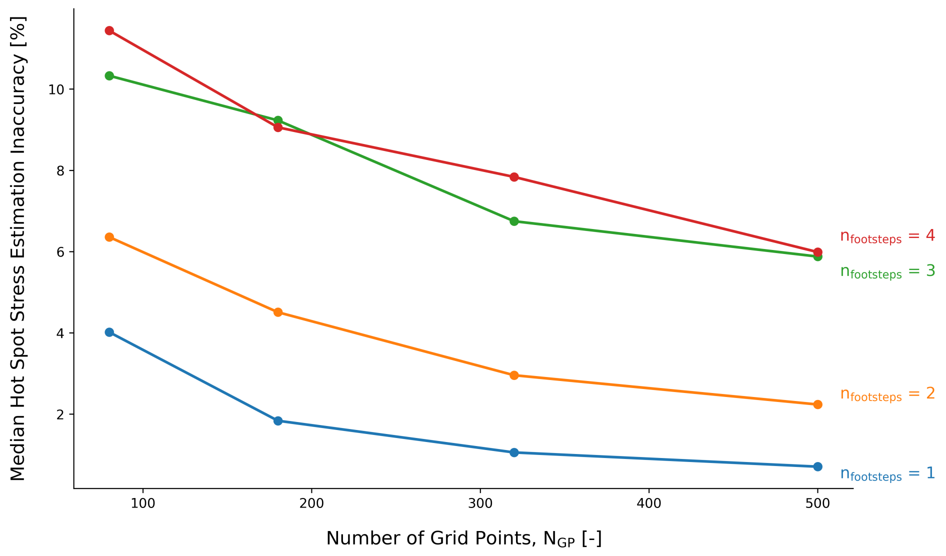


Figure 8.2: Median hot-spot stress estimation error for different grid number of grid points and number of footsteps.

together. This is a direct consequence of loads being positioned closer to each other than the sensors can distinguish properly. In addition, part of these inaccuracies fall into categories of estimation errors from minimization methods and local minima. Appointing the correct force magnitude is not a major issue, as the hot-spot stress estimates will still be comparable. However, using the load magnitudes for determining trends could pose a problem. This is because inaccurate load magnitude estimation can distort the trend analysis over time, potentially affecting fatigue life predictions and long-term assessments.

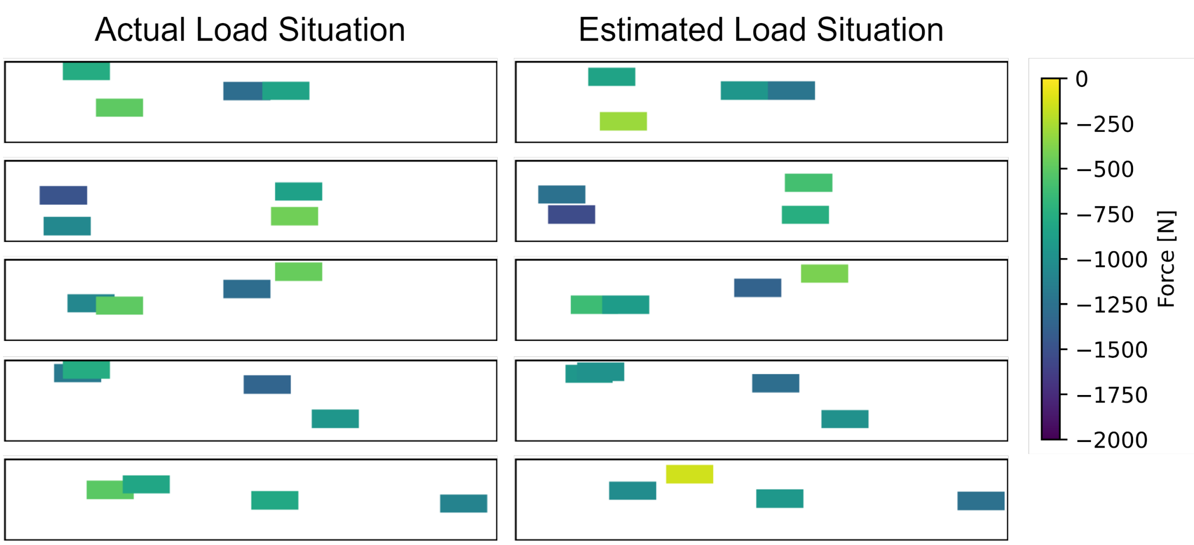


Figure 8.3: Individual estimation performance visualization. For different actual load situations (as computed using the digital model) and the estimated load situation as determined with the fingerprinting method.

8.2. Experimental Results of Single Load Testing

Single load experiments were conducted to assess the accuracy of the model in a real-world measurement setup and to identify the sources of the largest inaccuracies. For this analysis there was a large benefit to having more sensors than used. Sensor configuration 3 (only estimating loads using 12 out of 20 sensors) means that 8 sensor values can be used to test estimation performance. For these strain values to match properly we chose to load both the digital model and test setup with matching locations and force magnitudes. These 50 locations were marked on the bridge as seen in Figure 7.5.

The full set of data obtained from the different DAMs of this test is shown in Appendix J. In the simulation tests the fingerprinting model achieved an average load location error of 3.6 mm and a load magnitude error of 0.58 %. Then for the experimental tests, the fingerprinting model achieved an average load location error of 43.5 mm and a load magnitude inaccuracy of 14.79 %. The simulation model, which isolates only the fingerprinting inaccuracy, continues to demonstrate exceptional performance. However, the experimental setup shows noticeably poorer results, indicating that a significant portion of the overall pipeline inaccuracy arises from other factors. These factors may include loading inaccuracies, such as not stepping precisely on the designated locations or wobbling during movement, sensor inaccuracies, or discrepancies between the FEM model and the actual behavior of the physical asset.

To facilitate further comparison, the simulation estimation error (difference between DAM1 and DAM2) and the experimental estimation error (difference between DAM3 and DAM4) are calculated using the strain estimation errors at eight redundant sensor locations $i \in I^-$. These errors are expressed as the percentage of absolute strain difference relative to the maximum strain at each sensor location, as defined by:

$$\%\Delta\epsilon = \text{med}\left(\frac{|\epsilon_{i,est} - \epsilon_i|}{\epsilon_{\max_i}} \cdot 100\% \quad \forall i \in I^-\right) \tag{8.3}$$

Here we estimate $\epsilon_{\max_i}$ for each location i as the maximum value measured over the 50 load situations. This approach ensures the accuracy evaluation is scaled by the magnitude of the strain values, mitigating the impact of small absolute differences that could otherwise lead to disproportionately high relative percentage differences. This method provides a balanced and meaningful assessment of the strain estimation performance.

For each sensor in the set I^-, a box plot is created to visualize the variations in strain estimation accuracy and, by extension, how stress estimation varies across all tested loads. The box plot of simulation errors is shown in Figure 8.4, while the experimental errors are presented in Figure 8.5.

The simulation model demonstrates consistent strain estimation performance, with variations averaging less than 0.2%. In contrast, the experimental strain estimations show significantly larger discrepancies, favoring sensors positioned closer to the center. Notably, the outer sensors located on the right bottom flange (R2 and R9) exhibit greater inaccuracies, suggesting a dependence of measurement accuracy on sensor location.

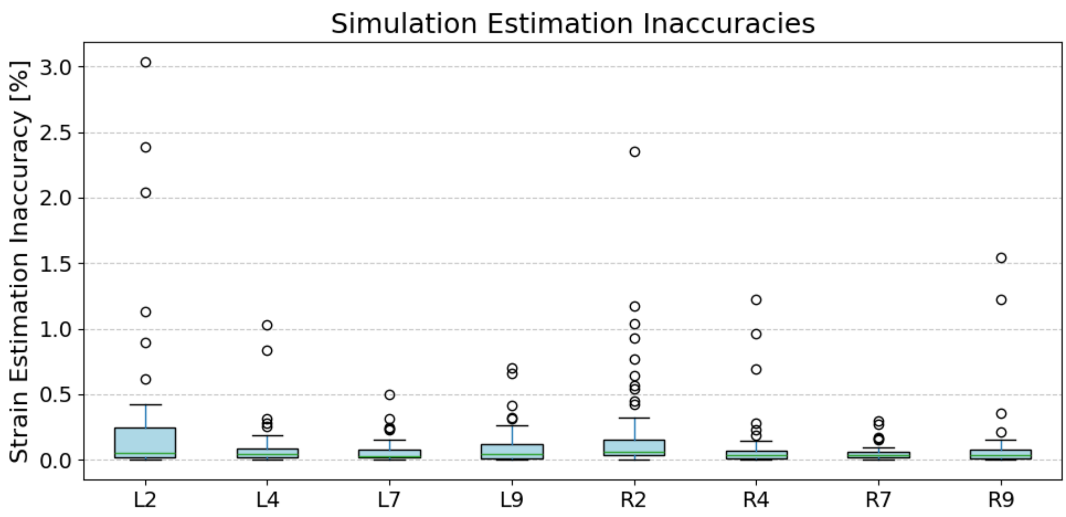


Figure 8.4: Box plot showing estimation inaccuracy of single load situation from a FEM load as determined by the fingerprinting model.

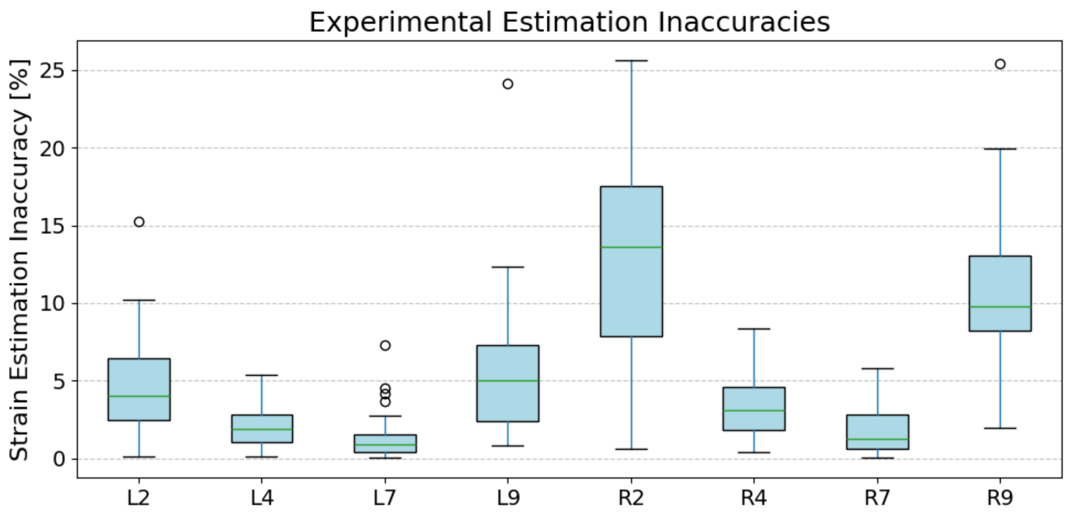


Figure 8.5: Box plot showing estimation inaccuracy of single load situation from the experimental setup as determined by the fingerprinting model.

The results suggest that the location of sensors significantly impacts estimation errors. Sensors closer to the center generally perform better, partly because they are closer to the average load location. However, further analysis reveals a skew in experimental location estimation errors across the bridge's geometry, as shown in Figure 8.6. This visualization highlights that the central region excels in estimating load locations, whereas areas toward the edges of the asset exhibit higher errors.

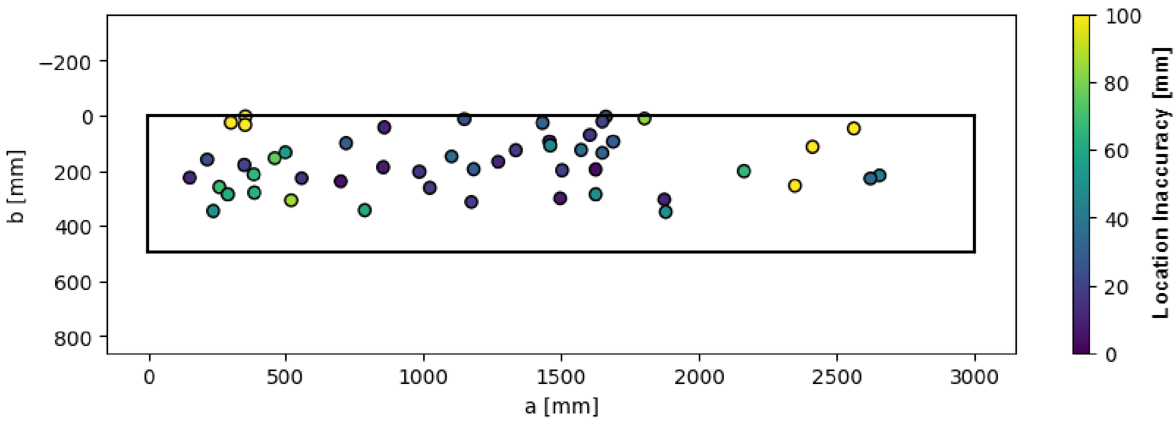


Figure 8.6: Errors in location estimation from the experimental setup.

A similar trend is observed for experimental force errors, as illustrated in Figure 8.7. In this case, loads positioned closer to the center tend to cause overestimated load magnitudes.

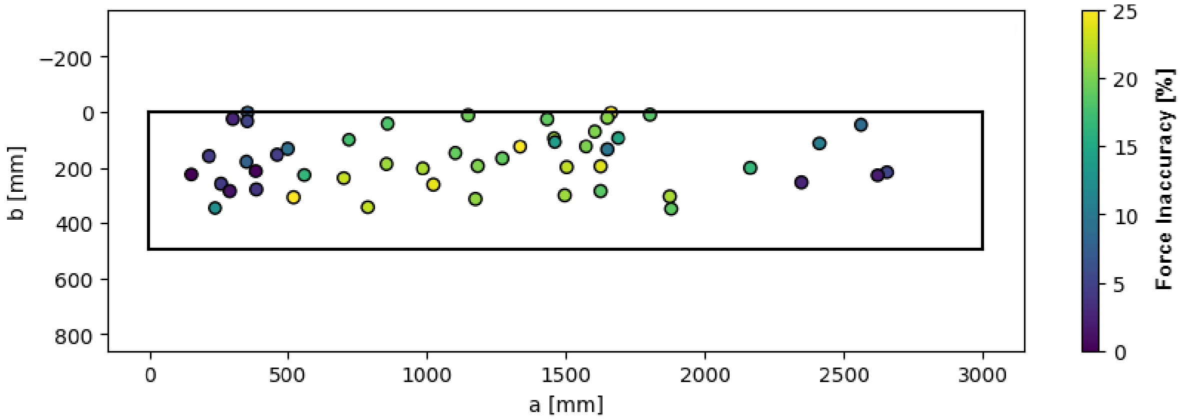


Figure 8.7: Errors in force estimation from the experimental setup.

The influencing factors between the experimental errors are the differences between the FEM model and the asset's behavior, the inaccuracy of applying the load onto the asset or the simplified footprint geometry and finally an error in the sensor reading of the asset. There is a clear systematic overestimation of the load magnitude for loads in the center, which cannot be attributed to the sensors or the way the load is applied. This means that the input in the FEM model slightly differs from the asset's properties. The FEM model is overall more rigid than the asset, causing larger strains in both flanges. A possible reason is that the connections between parts of the bridge are modeled as rigid where they are instead connected by welds. The FEM model is clearly a very important part of the fingerprinting performance, and thus needs to be validated thoroughly in future application.

9 Conclusion

In conclusion, this research aimed to achieve real-time insights into the stress state of bridge structures by addressing the core research question: How can real-time stress states of bridge structures be derived using a combination of sensor data and computational modeling?

The methodology developed integrated sensor data with a finite element model (FEM) to determine load locations and magnitudes by comparing measured strain values with simulated results. To enhance scalability, particularly for real-world traffic bridges, a camera with image recognition capabilities was introduced. This addition enabled the detection of the number and approximate locations of loads on the bridge, providing an initial estimate to complement the sensor-based approach. Linearizing the methodology for scalability was a critical step toward adapting the framework for larger, more complex structures.

The results demonstrate that the methodology is promising for estimating load scenarios and stress states, particularly when addressing the inaccuracies inherent to the fingerprinting approach. It provided reasonable estimations for single and multi-load scenarios using a limited amount of sensor data. However, during testing, the FEM model exhibited slight inaccuracies in representing the physical asset's geometric behavior under load. These inaccuracies propagated through the simulation database of grid points used for the fingerprinting algorithm, reducing estimation accuracy. Future applications will require precise FEM modeling and validation of both the global and detailed behavior of the asset to achieve more accurate and reliable estimations.

The image recognition for shoes proved to be a limiting factor in the current setup, frequently misidentifying multiple shoes at the same location or failing to detect them entirely. For successful application on traffic bridges, the system must function reliably under all weather and lighting conditions. Future work should focus on improving the consistency of image recognition to address this challenge, as well as expanding the functionality to the recognition of vehicles.

Additionally, while the use of optimization algorithms like COBYLA and SLSQP was effective, their performance in terms of computational speed and estimation accuracy was variable. Exploring alternative minimization methods or optimization techniques could further refine the model, though initial trials with other methods yielded inconsistent results. Future research should investigate new approaches to enhance performance.

Finally, achieving real-time analysis for large-scale traffic bridges will require a significant investment in computational resources. This can involve deploying the system on more powerful local hardware or leveraging cloud computing to allocate sufficient processing power. Addressing this will be essential for practical, real-time applications on full-scale traffic bridges.

References

[1] Chenyu "Zhou et al. “"An Example of Digital Twins for Bridge Monitoring and Maintenance: Preliminary Results"”. In: *"Proceedings of the 1st Conference of the European Association on Quality Control of Bridges and Structures"*. Ed. by Carlo "Pellegrino et al. "Cham": "Springer International Publishing", "2022", "1134–1143". ISBN: "978-3-030-91877-4".

[2] ASSEM. *Tot welke leeftijd groeien je voeten?* Accessed: 2024-09-10. 2024. URL: https://www.assem.nl/blogs/tot-welke-leeftijd-groeien-je-voeten--NL020.html#:~:text=Hoe%20groot%20je%20voeten%20worden,het%20gemiddelde%20rond%20maat%2043.

[3] M.S. Cheung and W.C. Li. “Probabilistic fatigue and fracture analyses of steel bridges”. In: *Structural Safety* 25.3 (2003), pp. 245–262. ISSN: 0167-4730. DOI: https://doi.org/10.1016/S0167-4730(02)00067-X. URL: https://www.sciencedirect.com/science/article/pii/S016747300200067X.

[4] Wikipedia contributors. *Median absolute deviation*. Accessed: 2024-09-10. 2024. URL: https://en.wikipedia.org/wiki/Median_absolute_deviation.

[5] Isaac Farreras-Alcover, Marios K. Chryssanthopoulos, and Jacob E. Andersen. “Data-based models for fatigue reliability of orthotropic steel bridge decks based on temperature, traffic and strain monitoring”. In: *International Journal of Fatigue* 95 (2017), pp. 104–119. ISSN: 0142-1123. DOI: https://doi.org/10.1016/j.ijfatigue.2016.09.019. URL: https://www.sciencedirect.com/science/article/pii/S0142112316303103.

[6] FBGS. *Strain Sensing*. 2024. URL: https://fbgs.com/solutions/strain-sensing/ (visited on 11/25/2024).

[7] FBGS. *Tailored Fiber Optic Sensing Components & Solutions*. 2024. URL: https://fbgs.com/ (visited on 11/25/2024).

[8] Ibai Gorordo Fernandez and Chikamune Wada. “Shoe Detection Using SSD-MobileNet Architecture”. In: *2020 IEEE 2nd Global Conference on Life Sciences and Technologies (LifeTech)*. 2020, pp. 171–172. DOI: 10.1109/LifeTech48969.2020.1570618965.

[9] Yan Gao et al. “AIoT-informed digital twin communication for bridge maintenance”. In: *Automation in Construction* 150 (2023), p. 104835. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2023.104835. URL: https://www.sciencedirect.com/science/article/pii/S092658052300095X.

[10] Eman Hussein et al. “Fatigue Measurements in an Existing Highway Concrete Bridge”. In: *Sensors* 22.9 (2022). Accessed: 2024-06-13, p. 2868. DOI: 10.3390/s22082868. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9030938/#B7-sensors-22-02868.

[11] Iv. *Engineering that excites*. 2024. URL: https://www.iv.nl/ (visited on 10/30/2024).

[12] Fei Jiang et al. “Digital Twin-driven framework for fatigue life prediction of steel bridges using a probabilistic multiscale model: Application to segmental orthotropic steel deck specimen”. In: *Engineering Structures* 241 (2021), p. 112461. ISSN: 0141-0296. DOI: https://doi.org/10.1016/j.engstruct.2021.112461. URL: https://www.sciencedirect.com/science/article/pii/S0141029621006118.

[13] Aljaž Jurca, Jana Žabkar, and Sašo Džeroski. “Analysis of 1.2 million foot scans from North America, Europe and Asia”. In: *Scientific Reports* 9.1 (2019), p. 19155. DOI: 10.1038/s41598-019-55432-z. URL: https://doi.org/10.1038/s41598-019-55432-z.

[14] Sakdirat Kaewunruen et al. “Digital Twin Aided Vulnerability Assessment and Risk-Based Maintenance Planning of Bridge Infrastructures Exposed to Extreme Conditions”. In: *Sustainability* 13.4 (2021). ISSN: 2071-1050. DOI: 10.3390/su13042051. URL: https://www.mdpi.com/2071-1050/13/4/2051.

[15] Zalando Lounge. *Schoenmaat meten*. Accessed: 2024-09-10. URL: https://www.zalando-lounge.nl/sizehelper/shoes/measure/#/.

[16] *Merwedebrug (Boven-Merwede)*. 2024. URL: https://nl.wikipedia.org/wiki/Merwedebrug_(Boven-Merwede) (visited on 10/30/2024).

[17] Masoud Mohammadi et al. "Quality Evaluation of Digital Twins Generated Based on UAV Photogrammetry and TLS: Bridge Case Study". In: *Remote Sensing* 13.17 (2021). ISSN: 2072-4292. DOI: 10.3390/rs13173499. URL: https://www.mdpi.com/2072-4292/13/17/3499.

[18] Vahid Mousavi et al. "Evolution of Digital Twin Frameworks in Bridge Management: Review and Future Directions". In: *Remote Sensing* 16.11 (2024). ISSN: 2072-4292. DOI: 10.3390/rs16111887. URL: https://www.mdpi.com/2072-4292/16/11/1887.

[19] NEN. *Eurocode 1: Belastingen op constructies - Deel 2: Verkeersbelasting op bruggen*. Standard NEN-EN 1991:2003 en. NEN, Oct. 2003.

[20] NEN. *Eurocode 1: Belastingen op constructies - Deel 2: Verkeersbelasting op bruggen*. Standard NEN-EN 1991-2+C2:2015. NEN, Nov. 2019.

[21] NEN. *Eurocode 3: Ontwerp en berekening van staalconstructies - Deel 1-14: Ontwerp ondersteund door eindige elementenanalyse*. Standard NEN-EN 1993-1-14. NEN, Sept. 2023.

[22] NEN. *Eurocode 3: Ontwerp en berekening van staalconstructies - Deel 1-9: Vermoeiing*. Standard NEN-EN 1993-1-9+C2:2012 nl. NEN, Sept. 2012.

[23] NEN. *Eurocode 9: Ontwerp en berekening van aluminiumconstructies - Deel 1-3: Vermoeiing*. Standard NEN-EN 1999-1-3:2007 en. NEN, May 2007.

[24] NEN. *Fatigue design of orthotropic bridge decks with the hot spot stress method*. Standard TS 1993-1-901:2021. NEN, 2021.

[25] Idilson A. Nhamage et al. "Performing Fatigue State Characterization in Railway Steel Bridges Using Digital Twin Models". In: *Applied Sciences* 13 (2023), p. 6741. ISSN: 2076-3417. URL: https://www.mdpi.com/2076-3417/13/11/6741.

[26] NOS. *Bedrijf krijgt schadevergoeding wegens sluiting Merwedebrug*. 2018. URL: https://nos.nl/artikel/2244379-bedrijf-krijgt-schadevergoeding-wegens-sluiting-merwedebrug (visited on 11/18/2024).

[27] Daniel Papán and Katarína Demeterová. *Trendprognose wegverkeer 2022-2027*. 2022. URL: https://www.kimnet.nl/publicaties/notities/2022/05/30/trendprognose-wegverkeer-2022-2027 (visited on 11/14/2024).

[28] Sadanandam Anupoju. *12 Types of Loads Considered for Design of Bridge Structures*. Accessed: 2024-06-13. URL: https://theconstructor.org/structures/bridge-design-loads/21478/#google_vignette.

[29] A. Sahrapeyma, A. Hosseini, and M.S. Marefat. "Life-cycle prediction of steel bridges using reliability-based fatigue deterioration profile: Case study of Neka bridge". In: *International Journal of Steel Structures* 13.2 (2013), pp. 229–242. DOI: 10.1007/s13296-013-2003-8.

[30] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[31] Kievit Schoenen. *Schoenmaat bepalen - Maattabel*. Accessed: 2024-09-10. URL: https://kievit-schoenen.nl/maattabel.

[32] Harald Schuler, Florian Meier, and Burkhart Trost. "Monitoring der Gerbergelenke im Erhaltungsprojekt Grenzbrücke Basel". In: *Beton- und Stahlbetonbau* 116.3 (2021), pp. 160–169. DOI: https://doi.org/10.1002/best.202000081. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/best.202000081. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/best.202000081.

[33] Sneakerjagers. *De gemiddelde schoenmaat van mannen is berekend*. Accessed: 2024-09-10. 2023. URL: https://www.sneakerjagers.com/n/de-gemiddelde-schoenmaat-van-mannen-is-berekend/148159?srsltid=AfmBOoqZWlKJQl-1G-w1zQqHOssJA3MzCeDr3F8TSWyvRqoOGNaFRAJs.

[34] *Trucks worden zwaarder*. 2024. URL: https://truckstar.nl/trucks-worden-zwaarder/ (visited on 11/14/2024).

[35] Hanwen Ju Yang Deng Taolei Liu and Aiqun Li. "Investigation on fatigue cracks of diaphragm cutout in bridge orthotropic steel deck". In: *Structure and Infrastructure Engineering* 20.5 (2024), pp. 682–698. DOI: 10.1080/15732479.2022.2120901. eprint: https://doi.org/10.1080/15732479.2022.2120901. URL: https://doi.org/10.1080/15732479.2022.2120901.

[36] Cong Ye et al. "A digital twin of bridges for structural health monitoring". In: *12th International Workshop on Structural Health Monitoring 2019*. Stanford University. 2019.

[37] X. W. Ye, Y. H. Su, and J. P. Han. "A State-of-the-Art Review on Fatigue Life Assessment of Steel Bridges". In: *Mathematical Problems in Engineering* 2014.1 (2014), p. 956473. DOI: https://doi.org/10.1155/2014/956473. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2014/956473. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2014/956473.

[38] Yang Yu et al. "Fatigue damage prognosis of steel bridges under traffic loading using a time-based crack growth method". In: *Engineering Structures* 237 (2021), p. 112162. ISSN: 0141-0296. DOI: https://doi.org/10.1016/j.engstruct.2021.112162. URL: https://www.sciencedirect.com/science/article/pii/S0141029621003126.

[39] Ziengs. *Schoenmaat bepalen*. Accessed: 2024-09-10. URL: https://www.ziengs.nl/maatwijzer/.

A

Scientific Research Paper

Real-Time Stress State Estimation for Steel Bridges

H. de Vries[a], W. van den Bos[a], G. Vandeberg[b], Y. Pang[a]

[a] *Delft University of Technology, Delft, The Netherlands*
[b] *Iv, Sliedrecht, The Netherlands*

Abstract

This research presents a novel approach for real-time stress state estimation in steel bridges using Fiber Bragg Grating (FBG) sensors and image recognition techniques. The methodology involves creating a digital model of the bridge, comprising a global finite element model (FEM) and detailed sub-models of critical areas. A database of precomputed load cases is generated, and real-time sensor data is matched to this database using the developed fingerprinting method. Image recognition is employed to detect multiple load scenarios, enhancing the accuracy of stress estimations and ensuring linear scalability for multi-load situations. The accuracy of the developed model was tested using a scaled setup using a 3 meter long aluminum bridge, proving its effectiveness in real-world conditions. The results demonstrate the feasibility of this approach, with reasonable accuracy achieved in both single and multi-load scenarios. Future work should focus on improving model accuracy, enhancing image recognition algorithms, and optimizing computational performance for large-scale applications.

1. Introduction

One of the most prominent failure mechanisms for steel bridges is fatigue [3], [1]. Loads applied by for example vehicles, trains, pedestrians, wind and temperature are typically below the yield strength of the material. However, the cyclic nature of the loads means that over time the structural integrity of the bridge decays, leading to fracture initiation, propagation and eventually structural failure. Most of the fatigue-related damage can be repaired and maintained doing routine maintenance, but this maintenance is costly and often unnecessary given the poor understanding of the fatigue status [2]. To safely extend the operational lifetime of existing structures, it is essential to perform continuous analysis of the stress state and, consequently, the fatigue status at every critical detail.

This analysis is conducted using a 3-meter-long aluminum pedestrian bridge equipped with sensors. A FEM model of the bridge is developed and analyzed under various load scenarios. Algorithms were developed to compare strain measurements from the physical bridge to FEM results to estimate hot-spot stresses across the structure. During the measuring phase, stress fluctuations are translated to load cases, which are used to determine stress intervals. The collected stress interval data can then be extrapolated to be able to perform fatigue accumulation and make predictions with regard to fatigue life during the measurement period. This can then be extrapolated for the full bridge life, so that predictions about the expected life of the structure can be made.

2. Methodology

As a proof-of-concept experiment, we consider a 3 meter long aluminum pedestrian bridge. Computationally, we build a database of strain profiles in the 3D FEM model of this bridge in response to a load at different locations on the surface. Experimentally, the bridge is equipped with a set of fiber Bragg grating (FBG) sensors that measure strain at up to 20 specific locations. In this research, we have developed a "fingerprinting" methodology to determine the locations and magnitudes of loads, from a comparison of the sensor values to the calculated database. Subsequently, the database is used to estimate the hot-spot stresses in the details of the structure. By running our algorithm in real time over prolonged time, we can keep track of fatigue build up.

2.1 Experimental Setup

The sensor setup consists of a FBG strain sensor strip and a camera. The instrumented FBG sensor is visualized in Figure A.1, which is a 20 meter long cable with a total of 20 measurement points at 1 meter intervals. Of these 20 sensors, we use 12 sensors for our algorithm while the other 8 are used to estimate the error in our experiments. In addition, a camera records events on the bridge to provide information on the number of loads and a rough estimate for their locations.

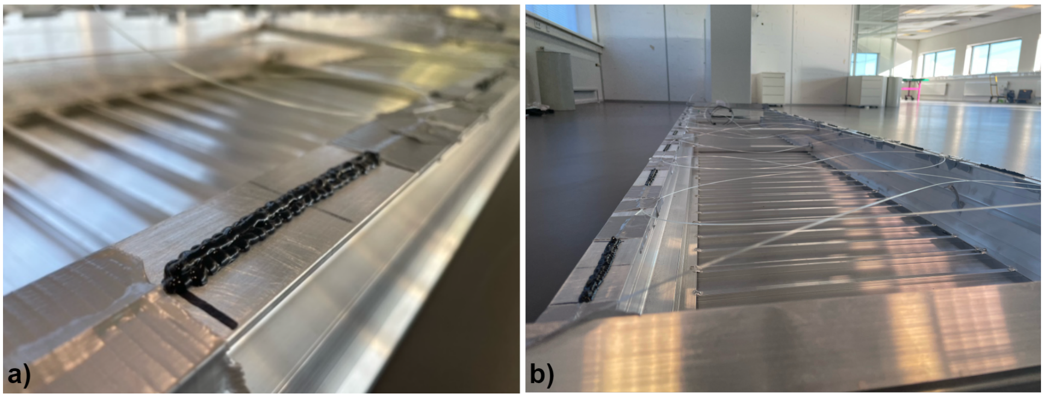


Figure A.1: a) Detail view of one of the FBG sensor locations connected to the asset. b) Visualization of the instrumented FBG sensor strip over the entire bridge

The sensors and the laptop running the computational script are all connected to the same router, enabling high-bandwidth data transfer with low latency. This setup is also adaptable for use in external locations. The required equipment, including a router, can be deployed on-site to transfer data either via Ethernet to a local computer or, ideally, over the internet to a cloud-based computer for processing.

2.2 Fingerprinting Algorithm

The fingerprinting algorithm is a methodology developed to determine the locations and magnitudes of loads, from a comparison of the sensor values to a database of FEM results. For each load situation, the fingerprinting algorithm estimates the location(s) and magnitude(s) of loads and estimates the resulting hot-spot stresses. To estimate the location(s) and magnitude(s), it goes through three main steps: **Coupling** to the closest grid points, **Interpolation** between grid points, **Scaling** of grid points. As a final operation in the scaling step, the algorithm uses the scaled grid points to estimate hot-spot stresses.

A major challenge was main- taining linear scaling in computational speed for multi-load situations, which was achieved by introducing a camera with image recognition to determine the number of loads on the bridge and to provide an initial guess of their locations. The second step to achieve linear scaling is to limit the tested combinations by locking all but one grid point and iteratively go through all of the loads, as seen in Figure A.2

(1) Coupling Step — In the multi-load algorithm we use an array $\gamma^{(n)} \quad \forall n \in N_{\text{GP}}$, where the value $\gamma^{(n)} = 0$ if grid point n has no load and $\gamma^{(n)} = 1$ if it has a load. In a scenario with $n_{\text{footsteps}}$ loads, we perform $n_{\text{footsteps}}$

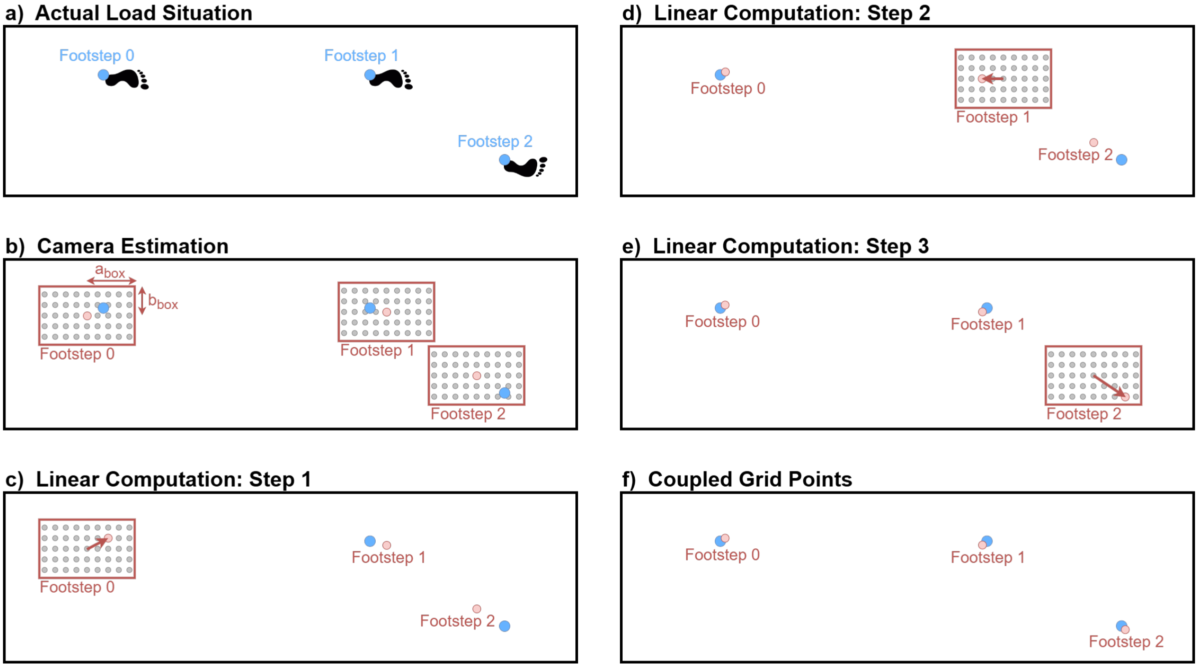


Figure A.2: Linear computation strategy for coupling grid point in a multi-load scenario. a) Depiction of an unknown load situation at a random moment in the asset's lifetime, where $n_{\text{footsteps}} = 3$. b) Camera prediction location (red dots), with the inaccuracy bounding boxes, in which grid points are considered. c), d), e) illustrate steps 1, 2 and 3 respectively, in determining the closest matching grid point within their respective bounding boxes. f) Presents the resulting coupled grid points obtained from the linear computation.

steps of the following type. The initial guess for the grid point index for each of the loads is based on a machine-learning interpretation of the camera image. All grid points indices containing a load are stored in array C. During each iteration, the grid point index of one of the loads (subset C^-) is optimized, while those of the other loads (subset C^+) are kept fixed.

In each iteration, we minimize

$$Z = \sum_{i \in I} \left| \sum_{n \in N_{\text{GP}}} \left(\alpha^{(n)} \, \gamma^{(n)} \, \epsilon_i^{(n)} - \epsilon_i \right) \right| \tag{A.1}$$

by finding a new grid point index n for the load we are optimizing, and by finding best estimates for $\alpha^{(n)} \quad \forall n \in C$. We use the following constraints:

$$\begin{cases} \sum_{n \in N_{\text{GP}}} \gamma^{(n)} = n_{\text{footsteps}} & \\ \gamma^{(c)} = 1 & \forall c \in C^+ \\ \gamma^{(n)} \in \{0, 1\} & \forall n \in N_{\text{box}} \\ \alpha^{(n)} \in \mathbb{R}_0^+ & \forall n \in C \end{cases} \tag{A.2}$$

(2) Interpolation Step — During each iteration there will be a set of locked grid points C^+, of size $n_{\text{footsteps}} - 1$ representing all but the currently optimized point, which itself falls in the set C^-. In each iteration, we first minimize

$$\text{MAD}^{(h_{c^-})} = \text{med} \left(\left| X_{ij}^{(h_{c^-})} - \tilde{X}^{(h_{c^-})} \right| \quad \forall i, j \in I \right), \quad \forall h_{c^-} \in H \tag{A.3}$$

with

$$\begin{cases} X_{ij}^{(h_{c^-})} = \left| \frac{\sum_{k\in K}\sum_{c\in C} \alpha^{(c)}\, \beta^{(k_c,h_c)}\, \epsilon_i^{(n_{k_c,h_c})}}{\sum_{k\in K}\sum_{c\in C} \alpha^{(c)}\, \beta^{(k_c,h_c)}\, \epsilon_j^{(n_{k_c,h_c})}} - \frac{\epsilon_i}{\epsilon_j} \right| & \forall i,j \in I, \forall h_{c^-} \in H \\ \tilde{X}^{(h_{c^-})} = \text{med}\left(X_{ij}^{(h_{c^-})} \quad \forall i,j \in I \right) & \\ h_c = h_{\min_c} & \forall c \in C^+ \\ \beta^{(k_c,h_c)} \in \mathbb{R}_0^+ & \forall k_c \in K, \forall h_c \in H \\ \sum_{k_c} \beta^{(k_c,h_c)} = 1 & \forall h_c \in H \end{cases} \tag{A.4}$$

to find the optimal values of $\beta^{(k_c,h_c)} \quad \forall c \in C$. In this step, the values of $\alpha^{(c)} \quad \forall c \in C$ are kept fixed at the values determined in the coupling phase.

Next, each iteration finds the optimal quadrant h_{c^-} of the load that is optimized as

$$h_{\min_{c^-}} = \underset{h_{c^-}}{\text{argmin}}\, \text{MAD}^{(h_{c^-})} \tag{A.5}$$

This value is then inserted back into the array of optimal area values $\{h_c\}$ for the next iteration. After all of the iterations have been executed, the areas list $\{h_{\min_c}\}$ and their respective interpolated scale factors $\beta_{k_c,h_{\min_c}} \quad \forall c \in C$ have been found.

(3) Scaling Step — The load factor $\alpha^{(n)}$, as determined during the coupling phase, provides an initial guess for the simplified locations. However, with the interpolation a more precise location has been determined. For this reason it is desirable to redetermine the scale factor $\alpha^{(c)}$ to better match the newly predicted load locations. We minimize

$$Z = \sum_{i\in I} \left| \sum_{c\in C} \sum_{k\in K} \alpha^{(c)} \beta_{k_c,h_{\min_c}}\, \epsilon_i^{\left(n_{k_c,h_{\min_c}}\right)} - \epsilon_i \right| \tag{A.6}$$

subject to

$$\alpha^{(c)} \in \mathbb{R}_0^+ \tag{A.7}$$

for global force scale factors $\alpha^{(c)}$. Here the previously found scale factors $\beta_{k_c,h_{\min_c}}$ are used.

The estimated loads for each of the footsteps follow

$$F_{\text{est}_c} = \alpha^{(c)} \sum_{k\in K} \beta_{k_c,h_{\min_c}}\, F_{k_c,h_{\min_c}} \quad \forall c \in C \tag{A.8}$$

where scale factors $\beta_{k_c,h_{\min_c}}$ and $\alpha^{(c)}$ are used to scale the force used for the database computation.

To determine the estimated hot-spot stresses in all of the researched details

$$\sigma_{s,\text{est}} = \sum_{c\in C} \alpha^{(c)} \sum_{k_c\in K} \beta_{k_c,h_{\min_c}}\, \sigma_s^{\left(n_{k_c,h_{\min_c}}\right)} \quad \forall s \in S \tag{A.9}$$

the same scale factors $\beta_{k_c,h_{\min_c}}$ and $\alpha^{(c)}$ are used. These values provide the live stress state of the desired details and the load description on the asset.

2.3 Fatigue Estimation

Gathering these results over time allows us to extrapolate realistic data to predict a load history for the full life of the bridge. The following extrapolated load spectrum can then be translated to fatigue damage in each investigated detail, using the Palmgren Miner Rule

$$D_s = \sum_x \frac{n_{x,s}}{N_x} \tag{A.10}$$

allows us to then estimate current fatigue damage and predict fatigue life.

2.4 Performance Testing

There are two ways performance is tested: Simulation and Experimental. Our simulation tests quantify the isolated performance of the fingerprinting algorithm by feeding it with FEM-generated strain values at the sensor locations. These tests compare the output of the algorithm with the input of the simulations in terms of hot-spot estimation error and they consider the computational speed.

For the experimental analysis, 50 experiments were performed where a person stepped on the physical bridge at a marked location. Meanwhile, the Real-Time Assessment pipeline was fully operational. For our performance tests, the strain values of 8 abundant sensors are measured experimentally, calculated with the FEM model with an input load at the marked location, and estimated using the fingerprinting algorithm. This constitutes the following Data Acquisition Modalities (DAMs):

- DAM1: Simulation calculation (FEM)
- DAM2: Fingerprinting, with input strain values from simulation
- DAM3: Experimental measurement (sensors)
- DAM4: Fingerprinting, with input strain values from experiment

3. Results

The performance of DAM2 when estimating individual load situations for $n_{\text{footsteps}} = 4$, is visualized in Figure A.3. This shows that load locations often match well. The model seems to slightly struggle at appointing the right force magnitude to the correct locations, swapping around or halving combined forces of loads that are positioned close together. This is a direct consequence of loads being positioned closer to each other than the sensors can distinguish properly. In addition, part of these inaccuracies are due to local minima to which the minimization methods converge.

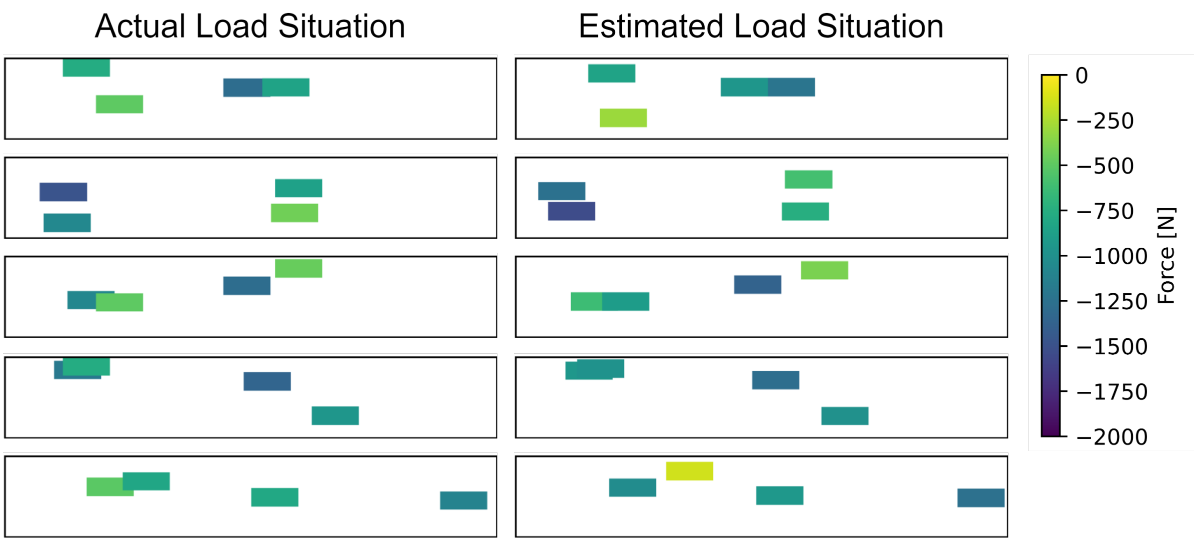


Figure A.3: Individual estimation performance visualization. For different actual load situations (as computed using the digital model) and the estimated load situation as determined with the fingerprinting method.

Figures A.4 and A.5 presents box plots for each of the 8 redundant sensors, showing variations in strain estimation accuracy. The box plot of simulation inaccuracies (%Δ between DAM1 and DAM2) is shown in Figure A.4, while the experimental inaccuracies are presented in Figure A.5 (%Δ between DAM3 and DAM4). The simulation model demonstrates consistent strain estimation performance, with variations averaging less than 0.2%. In contrast, the experimental strain estimations show significantly larger discrepancies. The locations positioned closer to the center (sensors L4, L7, R4, and R7) show smaller errors. The greatest errors are found for locations on the right bottom flange (R2 and R9). This shows a dependence of hot-stress estimation error on location. Further testing strongly suggests this is due to inaccuracies in the FEM model.

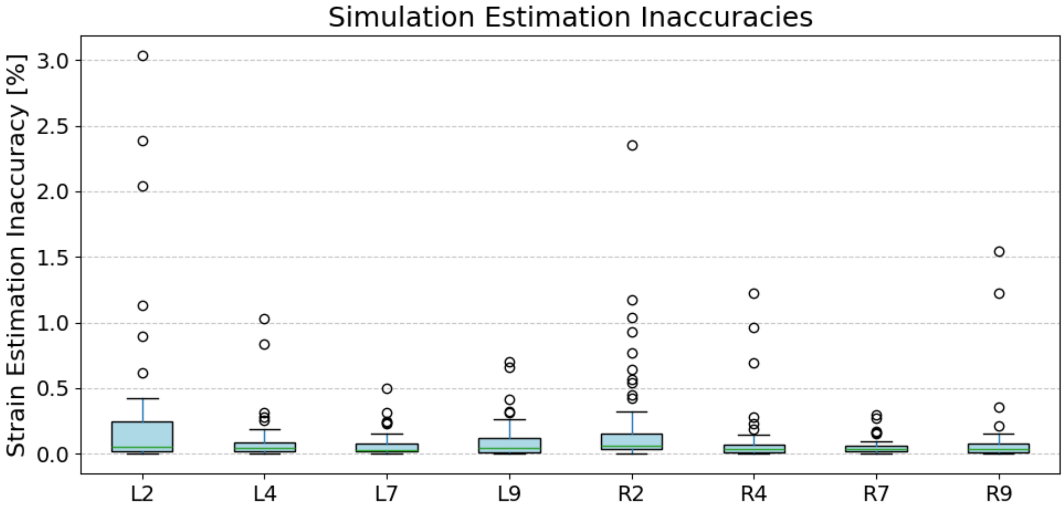


Figure A.4: Box plot showing estimation inaccuracy of single load situation from a FEM load as determined by the fingerprinting model.

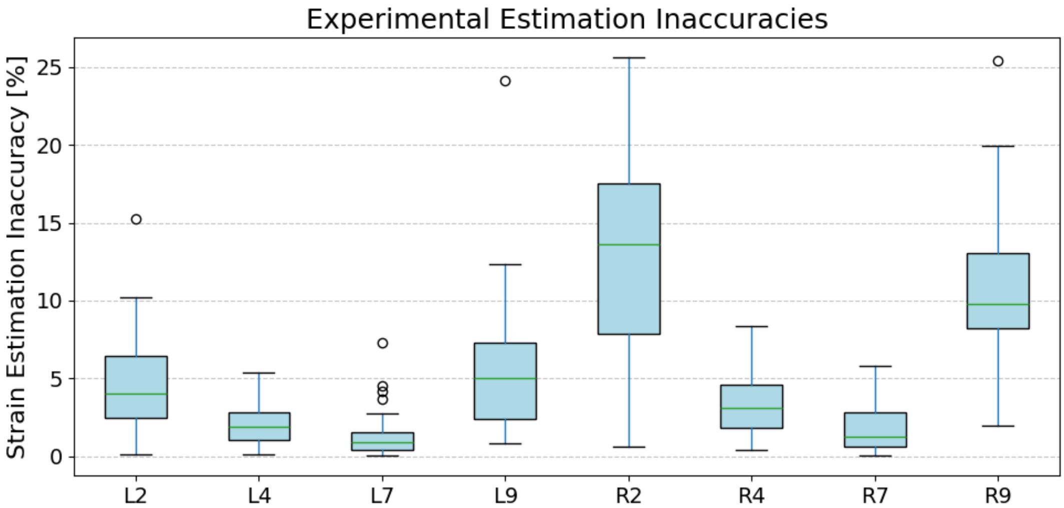


Figure A.5: Box plot showing estimation inaccuracy of single load situation from the experimental setup as determined by the fingerprinting model.

4. Conclusions

The results demonstrate that the methodology is promising for estimating load scenarios and stress states. Testing the fingerprinting algorithm in isolation, using simulated data, showed hot-spot stress estimation errors between 1.06% for a single-load scenario and 7.84% for a multi-load scenario of 4 footsteps. Experimental tests yielded larger errors. These could be attributed to inaccuracies in representing the physical asset's geometric behavior under load in the FEM model. These inaccuracies propagated through the simulation database of grid points used for the fingerprinting algorithm, increasing median strain estimation inaccuracy to between 1.28% and 9.76% already for single loads. Future applications will require precise FEM modeling and validation of both the global and detailed behavior of the asset to achieve more accurate and reliable estimations.

Finally, achieving real-time analysis for large-scale traffic bridges will require a substantial investment in computational resources. This could involve deploying the system on more powerful local hardware or utilizing cloud computing to ensure adequate processing capabilities. Additionally, improvements in image recognition technology will be critical. Addressing these challenges will be essential for implementing practical, real-time applications on full-scale traffic bridges.

References

[1] M.S. Cheung and W.C. Li. "Probabilistic fatigue and fracture analyses of steel bridges". In: *Structural Safety* 25.3 (2003), pp. 245–262. ISSN: 0167-4730. DOI: https://doi.org/10.1016/S0167-4730(02)00067-X. URL: https://www.sciencedirect.com/science/article/pii/S016747300200067X.

[2] Hanwen Ju Yang Deng Taolei Liu and Aiqun Li. "Investigation on fatigue cracks of diaphragm cutout in bridge orthotropic steel deck". In: *Structure and Infrastructure Engineering* 20.5 (2024), pp. 682–698. DOI: 10.1080/15732479.2022.2120901. eprint: https://doi.org/10.1080/15732479.2022.2120901. URL: https://doi.org/10.1080/15732479.2022.2120901.

[3] X. W. Ye, Y. H. Su, and J. P. Han. "A State-of-the-Art Review on Fatigue Life Assessment of Steel Bridges". In: *Mathematical Problems in Engineering* 2014.1 (2014), p. 956473. DOI: https://doi.org/10.1155/2014/956473. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2014/956473. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2014/956473.

B

Read FBG Sensor Data Source Code

```python
# Function to read, process and store the sensor data
def read_sensor_data(TCP_IP,
                    TCP_PORT,
                    sensor_data_separator):

    global sensor_data_storage_list

    sensor_data = []

    try:
        # Create a TCP socket
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:

            # Measure ping time
            start_ping = time.time()
            s.connect((TCP_IP, TCP_PORT))
            end_ping = time.time()
            ping_duration = (end_ping - start_ping) / 2  # Half round trip time
            logger.info(f"Connected to {TCP_IP}:{TCP_PORT}")

            # Keep receiving data
            while True:

                # First read the 4-byte integer for the length of the data string
                raw_len = s.recv(4)
                if not raw_len:
                    continue
                data_length = struct.unpack('!I', raw_len)[0]

                # Now read the full data string based on the received length
                data = s.recv(data_length)
                if not data:
                    continue
                decoded_data = data.decode('utf-8').strip()
                split_data = decoded_data.split(sensor_data_separator)

                # Store the parsed data
                sensor_data.append(split_data)

                # Retrieve current time, which forms the basis for the sensor iteration time
                    in all future lines
                sensor_measurement_time = retrieve_current_time(time_format)

                # Adjust the start time by subtracting the ping duration
                sensor_measurement_time -= timedelta(seconds=ping_duration)

                # Convert the time to the desired time format
                sensor_measurement_time = sensor_measurement_time.strftime(time_format)

```

```
                # Store sensor data if sensor data is read
                if len(sensor_data) > 0:
                    processed_sensor_data, sensor_data = process_sensor_data(sensor_data=sensor_data,
                    sensor_measurement_time=sensor_measurement_time,
                    sensor_strain_data_positions=sensor_strain_data_positions,
                    strain_scale_factor=strain_scale_factor)

                    # Continuously adjust for moving averages due to temperature changes within the material to prevent drift within the strain measurements
                    if continuous_strain_calibration_statement:
                        processed_sensor_data = remove_moving_average_effect(processed_sensor_data=processed_sensor_data,
                        window_size_calibration=window_size_calibration,
                        calibration_strains=calibration_strains,
                        processed_strain_data_positions=processed_strain_data_positions,
                        strain_cut_off_value=strain_cut_off_value,
                        reset_calibration_values=reset_calibration_values)

                    # Writing the found sensor values to memory storage, while preventing other threads from reading the list during this operation
                    with sensor_data_lock:
                        for i in range(len(processed_sensor_data)):
                            sensor_data_storage_list.append(processed_sensor_data[i])

    # Provide error information when no connection can be made to the sensor device
    except socket.error as e:
        logger.info(f"Error while reading sensor data: {e}")
```

C

Capture Camera Images Source Code

```python
# Function that captures images from camera stream and saves them on memory
def initiate_axis_stream_capture_to_images(
    time_format,
    camera_username,
    camera_ip_address,
    camera_password,
    camera_video_codec,
    image_storage_folder_path
):

    global temporary_frame_storage_dict

    # Formatting the stream URL used to give access to the stream
    stream_url = f'rtsp://{camera_username}:{camera_password}@{camera_ip_address}/axis-media/media.amp?videocodec={camera_video_codec}&camera=1'

    # Set up video capture from the camera's stream
    cap = cv2.VideoCapture(stream_url)
    cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)  # Set buffer size to 1

    # Check if the video capture is successful
    if not cap.isOpened():
        logger.info("Error: Could not open video stream.")

        if not stream_url.startswith((' http://', 'https://', 'rtsp://', 'ftp://')):
            logger.info("Possible cause: The URL provided is not a valid or supported protocol.")
        else:
            logger.info(f"Warning: If live camera images is desired, try rerunning the script with adjusted settings")
            logger.info(f"Warning: If the run is done to process existing data, then do not mind the warning")
            logger.info("Possible Causes:")
            logger.info("- The camera/server might be offline or unavailable")
            logger.info("- The laptop might be correctly set up in the same subnet as the camera")
            logger.info("- The credentials of the stream authentication are incorrect")
            logger.info("- Unsupported or incorrect video codec or format")

    # Start while loop to continue capturing frames
    while True:

        # Reading a frame from the captured camera video
        ret, frame = cap.read()

        # Exit the capture loop if the connection to the camera is lost
        if not ret:
            logger.info("Warning: failed to grab frame")
            break
```

```python

        # Determine recorded frame time by obtaining the current time and removing any delay
        current_time = (datetime.now() - timedelta(seconds=camera_image_delay)).strftime(time_format)
        image = Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))

        # Saving the camera images in memory
        with camera_data_lock:
            temporary_frame_storage_dict[current_time] = image

    # Stopping the entire Real-Time Assessment run when no images can be captured anymore
    cap.release()
    sys.exit()
```

D

Processing Sensor Data Source Code

```python
# Process the sensor data from raw bytes to list of desired column values
def process_sensor_data(sensor_data,
                        sensor_measurement_time,
                        sensor_strain_data_positions,
                        strain_scale_factor):

    processed_sensor_data = []

    # For each line of read sensor data, process the line
    for line in sensor_data:
        processed_sensor_data_line = []

        # Append the timestamp at which the sensor data was recorded to the processed data
            line
        processed_sensor_data_line.append(sensor_measurement_time)

        # Append each of the strain measurements to the processed
        for pos in sensor_strain_data_positions:

            # Scale the strain values from microstrain to strain
            strain_value = float(line[pos]) * strain_scale_factor
            processed_sensor_data_line.append(strain_value)

        processed_sensor_data.append(processed_sensor_data_line)

    # Empty the list used for temporarily storing read sensor data
    sensor_data = []

    return processed_sensor_data, sensor_data
```

E

Calibrate Strain Data for Temperature Variance Source Code

```python
# Function to remove temperature effect using the effect of a moving average over
    predetermined window size
def remove_moving_average_effect(processed_sensor_data,
                                 window_size_calibration,
                                 calibration_strains,
                                 processed_strain_data_positions,
                                 strain_cut_off_value,
                                 reset_calibration_values):

    # Initialize a list to store adjusted sensor data
    filtered_sensor_data = []

    # Function that determines if calibration using the moving average is required, the
        calibration is not done if the cut off value is met
    def check_strain_values_for_calibration_requirement(calibration_strains,
        processed_strain_data_positions, strain_cut_off_value):
        # Loop through each position in processed_strain_data_positions
        for pos in processed_strain_data_positions:
            # Use generator expression for early exit if a condition is not met
            if any(abs(value) >= strain_cut_off_value for value in calibration_strains[pos]):
                return False
        return True

    # Retrieve statement on if the calibration should be updated based on if the cut-off
        strain value is reached
    update_calibration_statement = check_strain_values_for_calibration_requirement(
        calibration_strains, processed_strain_data_positions, strain_cut_off_value)

    # Check if calibration data is insufficient or if any processed strain data exceeds the
        cut-off value
    if len(calibration_strains[processed_strain_data_positions[0]]) >=
        window_size_calibration and update_calibration_statement:
        # Reset calibration values to the mean of the calibration strains for each position
        for x in processed_strain_data_positions:
            reset_calibration_values[x] = np.median(calibration_strains[x])

    # Process each new incoming data
    for data in processed_sensor_data:
        # Extract timestamp and sensor values
        timestamp = data[0]
        sensor_values = data[1:]

        # Adjust sensor values by removing the moving average
        filtered_values = []
        for pos in processed_strain_data_positions:
            # Get the current sensor strain value for the given position
```

```
            current_value = sensor_values[pos - 1]  # adjust for 0-indexed list

            # Update the moving average list for the current position
            calibration_strains[pos].append(current_value)

            # Maintain a fixed window size
            if len(calibration_strains[pos]) > window_size_calibration:
                calibration_strains[pos].pop(0)

            # Subtract the moving average from the current value to remove slow changes
            filtered_value = current_value - reset_calibration_values[pos]

            filtered_values.append(filtered_value)

        # Append filtered sensor data (with timestamp) to the result list
        filtered_sensor_data.append([timestamp] + filtered_values)

    return filtered_sensor_data
```

F

Time Control Source Code

```python
### Time control ###
def run_time_control(current_iteration_time,
                     start_iteration_time,
                     non_zero_data_statement):

    global total_iterations
    global average_iteration_time
    global non_zero_iterations
    global average_non_zero_iteration_time

    # Save current iteration time
    if live_data_statement:
            save_current_iteration_time(current_iteration_time=current_iteration_time,
                            storage_folder_name=storage_folder_name,
                            current_iteration_time_file_name=current_iteration_time_file_name,
                            time_format=time_format)

    # Retrieve current time
    current_time = retrieve_current_time(time_format=time_format) - timedelta(seconds=iteration_delay)

    # Determine iteration time
    end_iteration_time = current_time
    iteration_time = (end_iteration_time - start_iteration_time).total_seconds()

    # Determine iteration speed and updating the information values
    average_iteration_time = (average_iteration_time*total_iterations + iteration_time) / (total_iterations + 1)
    total_iterations += 1

    if non_zero_data_statement:
        average_non_zero_iteration_time = (average_non_zero_iteration_time*non_zero_iterations + iteration_time) / (non_zero_iterations + 1)
        non_zero_iterations += 1

    # Store iteration speed information
    to_be_saved_string = f"{total_iterations},{non_zero_iterations},{average_iteration_time},{average_non_zero_iteration_time},{iteration_speed_info_string}"
    store_string_as_txt(storage_folder_name=storage_folder_name,
                        storage_file_name=iteration_speed_information_file_name,
                        to_be_saved_string=to_be_saved_string)

    # Pause analysis if the new iteration time would be later than the current time
    time_difference = (current_iteration_time + interval_dt) - current_time
    time_difference = time_difference.total_seconds()

    logger.info(f'time_difference: {time_difference}')

```

```
    if time_difference > 0:
        if time_difference > 1/desired_run_frequency:
            logger.info(f'Found a time difference larger than a single time step, make sure 
                the current time is not lower than the current iteration time.')
            raise ValueError('Time difference exceeds expected single time step.')
        else:
            sleep_time = time_difference
            time.sleep(sleep_time)

    # Update the current iteration time
    current_iteration_time = update_current_iteration_time(current_iteration_time=
        current_iteration_time,
                                                        interval_dt=interval_dt)

    return current_iteration_time
```

G

Image Recognition Source Code

Code presented here is a slightly altered version of the code developed by Fernandez et al. [8].

```
import cv2
import numpy as np
import tensorflow as tf

# Function to execute the shoe detection algorithm and obtain the footstep coordinates and
    their boundary boxes along the image dimensions
def detect_shoes(side_image,
                 side_left_clearance,
                 side_right_clearance,
                 bridge_length,
                 bridge_width,
                 footstep_width,
                 threshold,
                 PATH_TO_CKPT):

    # Load the TensorFlow model into memory
    detection_graph = tf.Graph()
    with detection_graph.as_default():
        od_graph_def = tf.compat.v1.GraphDef()
        with tf.io.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
            serialized_graph = fid.read()
            od_graph_def.ParseFromString(serialized_graph)
            tf.import_graph_def(od_graph_def, name='')

        sess = tf.compat.v1.Session(graph=detection_graph)

    img_array = np.array(side_image)
    side_image = cv2.cvtColor(img_array, cv2.COLOR_RGB2BGR)
    side_image = np.expand_dims(side_image, axis=0)

    # Define input and output tensors
    image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
    detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
    detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
    detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')
    num_detections = detection_graph.get_tensor_by_name('num_detections:0')

    def preform_detection(image):
        # Perform the actual detection
        (boxes, scores, classes, num) = sess.run(
            [detection_boxes, detection_scores, detection_classes, num_detections],
            feed_dict={image_tensor: image})

        # Squeeze arrays for easier handling
        boxes = np.squeeze(boxes)
        scores = np.squeeze(scores)
        classes = np.squeeze(classes).astype(np.int32)
```

```

        # Visualization parameters
        _, image_height, image_width, _ = image.shape

        # Store footsteps
        footsteps = []

        # Draw bounding boxes on the image
        for i in range(int(num[0])):
            if scores[i] > threshold:
                box = boxes[i]
                ymin, xmin, ymax, xmax = box
                (left, right, top, bottom) = (xmin * image_width, xmax * image_width,
                                              ymin * image_height, ymax * image_height)

                # Append drawn box to footsteps
                footsteps.append({'left': left, 'right': right, 'bottom': bottom, 'top': top, 'confidence': scores[i]})

        return footsteps, image_width

    side_footsteps, image_width = preform_detection(side_image)

    # Determine footstep location
    def determine_location_side(footstep, image_width):
        xmin = footstep['left'] / image_width
        a_coordinate = (bridge_length * (xmin - side_left_clearance))/(1 - side_left_clearance - side_right_clearance)
        footstep['a_coordinate'] = a_coordinate

        return footstep, a_coordinate

    # Determine the guessed location of the footstep from the shoe detection and obtaining the length coordinate along the bridge
    footsteps = []
    for index, footstep in enumerate(side_footsteps):
        side_footsteps[index], a_coordinate = determine_location_side(footstep, image_width)
        b_coordinate = bridge_width/2 - footstep_width/2
        dummy_force_value = 0
        footsteps.append((a_coordinate, b_coordinate, dummy_force_value))

    detection_boxes = side_footsteps
    return footsteps, detection_boxes
```

H

Fingerprinting Source Code

```python
import logging
from collections import Counter

# Use the already configured logger
logger = logging.getLogger('shared_logger')


# Defining the single function used to determine the fingerprint of a measurement point
def determine_fingerprint_of_row(measurement_row,
                                 fingerprints,
                                 strain_handle,
                                 disabled_strain_pattern,
                                 a_handle,
                                 b_handle,
                                 force_handle,
                                 fingerprint_handle,
                                 hot_spot_handle,
                                 n_footsteps_handle,
                                 a_box_boundary,
                                 b_box_boundary,
                                 d_a,
                                 d_b,
                                 grid_point_filter_offset):
    """
    Determines the fingerprint of a measurement point by processing measurement data,
    interpolating strain values, and scaling results.

    Args:
        measurement_row (pd.Series): Single row of measurement data.
        fingerprints (pd.DataFrame): DataFrame containing simulation results.
        strain_handle (str): Characteristic string for strain sensor measurements columns.
        disabled_strain_pattern (str): String pattern to filter out deactivated strain sensor
             handles.
        a_handle (str): String for the length-direction column.
        b_handle (str): String for the width-direction column.
        force_handle (str): String for the force column.
        fingerprint_handle (str): String for the simulation result unique identifier column.
        hot_spot_handle (str): String for hot spot value columns.
        n_footsteps_handle (str): String for footstep count.
        a_box_boundary (float): Boundary for the length-direction grid points.
        b_box_boundary (float): Boundary for the width-direction grid points.
        d_a (float): Distance between grid points in the length direction.
        d_b (float): Distance between grid points in the width direction.
        grid_point_filter_offset (float): Number used as offset range for finding
            neighbouring grid points.

    Returns:
        tuple: Contains the following:
            - coupled_fingerprint_data_scaled (pd.Series): Predicted values based on input
```

```
            measurement_row.
        - scale_factor (float): Scaling factor for force.
        - coupled_fingerprint_data (pd.Series): Coupled grid point values.
        - coupled_fingerprint (str): Unique identifier of the coupled grid point.
        - a_interpolated (pd.Series): Interpolated length coordinates.
        - b_interpolated (pd.Series): Interpolated width coordinates.
        - alpha (np.ndarray): Force scale factors for each of the footsteps.
"""


### START OF FUNCTION DEFINITIONS ###
def determine_scale_factors_iteratively(fingerprints_dict,
                                        fingerprints,
                                        fingerprint_index,
                                        measurement_array,
                                        a_values_measurement,
                                        b_values_measurement):
    """
    Iteratively determines the scaling factors to match measurement data with fingerprint data.

    Args:
        fingerprints_dict (dict): Dictionary of fingerprints for each footstep.
        fingerprints (pd.DataFrame): DataFrame of fingerprint data.
        fingerprint_index (int): Index of the fingerprint column.
        measurement_array (np.ndarray): Array of measurement values.
        a_values_measurement (np.ndarray): Length-direction values of the measurement.
        b_values_measurement (np.ndarray): Width-direction values of the measurement.

    Returns:
        tuple: Contains the following:
            - gamma_values (np.ndarray): Array of gamma values for scaling.
            - alpha_values (np.ndarray): Array of alpha values for scaling.
            - gamma_keys (list): List of keys for the optimal grid points.
            - a_values_measurement (list): List of length-direction measurement values.
            - b_values_measurement (list): List of width-direction measurement values.
    """

    ## Save the nearest keys ##
    current_optimal_keys = []

    for i in range(len(a_values_measurement)):
        key = f"footstep_{i}"

        # Compute the Euclidean distance for each row in the DataFrame
        distances = np.sqrt((fingerprints_dict[key][a_col_name] - a_values_measurement[i])**2 + (fingerprints_dict[key][b_col_name] - b_values_measurement[i])**2)

        if len(distances) > 0:
            # Find the minimum distance and its corresponding index
            min_distance_index = distances.idxmin()

            # Append the key corresponding to the minimum distance
            current_optimal_keys.append(fingerprints_dict[key].loc[min_distance_index, fingerprint_col_name])
        else:
            continue

    logger.info(f"current_optimal_keys: {current_optimal_keys}")

    # Filter the strain columns
    fingerprint_strain = np.array(fingerprints.filter(like=strain_handle).filter(regex=disabled_strain_pattern))

    # Convert the strain values for the fingerprints to a dictionary
    eps_FP_dict = {
        name: fingerprint_strain[i].tolist()
        for i, name in enumerate(fingerprints.iloc[:, fingerprint_index])
    }

    fingerprint_col = fingerprints.columns[fingerprint_index]
```

```
        # Extract all unique fingerprint_index values
        filtered_fingerprints = []
        for df in fingerprints_dict.values():
            fingerprints_temp = df[fingerprint_col].tolist()
            if fingerprints_temp:
                filtered_fingerprints.append(fingerprints_temp)

        # Set the relative matrix from the measurement to a more convenient name
        eps_array = measurement_array

        # Make a list out of all of the dictionary entries to access the grid point data
        keys = list(eps_FP_dict.keys())

        alpha_list = []

        # For loop to determine optimal gammas
        for i in range(n_footsteps):

            logger.info(f"current_optimal_keys:␣{current_optimal_keys}")
            locked_keys = []

            # Setting the initial dummy values of the optimization results
            min_value = float('inf')
            min_diff_entries = None
            optimized_alpha = None

            for k in range(n_footsteps):
                if i == k:
                    continue
                else:
                    locked_keys.append(current_optimal_keys[k])

            current_combinations = []
            for filt_fp in range(len(filtered_fingerprints[i])):
                current_combination = []
                for k in range(n_footsteps):
                    if i == k:
                        current_combination.append(filtered_fingerprints[i][filt_fp])
                    else:
                        current_combination.append(current_optimal_keys[k])
                current_combinations.append(current_combination)

            for combination in current_combinations:

                # Setting empty lists to store keys and epsilon values
                active_keys = combination
                eps_FP_list = []

                for index in combination:
                    eps_FP_list.append(np.array(eps_FP_dict[index]))

                # Define the objective function using NumPy for faster computation
                def objective(alpha):
                    alpha = np.expand_dims(alpha, axis=-1)
                    eps_FP_sum = np.array(np.sum(alpha * eps_FP_list, axis = 0))
                    return np.sum(np.abs(eps_FP_sum - eps_array))

                # Set initial guess for alpha value
                initial_alpha = np.full(len(eps_FP_list),1)

                # Set bounds to ensure alpha values are non-negative
                bounds = [(0, 10) for _ in range(len(initial_alpha))]

                # Use minimize with the vectorized objective function and bounds
                result = minimize(objective, initial_alpha, method='COBYLA', bounds=bounds,)


                # Save resulting scale factor
                alpha = result.x
                optimal_value = result.fun
```

```
            # Testing if the new minimum MAD is lower than previously found MAD value
            if optimal_value < min_value:
                min_value = optimal_value
                min_diff_entries = active_keys
                optimized_alpha = alpha


        # Export keys and alphas from minimal combination
        current_optimal_keys = min_diff_entries
        alpha_list.append(optimized_alpha[i])

    logger.info(f"current_optimal_keys: {current_optimal_keys}")

    # Converting gamma list to array
    current_optimal_key_numbers = []
    for index, value in enumerate(current_optimal_keys):
        current_optimal_key_numbers.append(int(value.rsplit(' ', 1)[-1]))

    # Zip the lists together
    paired_list = list(zip(current_optimal_key_numbers, alpha_list, a_values_measurement, b_values_measurement))

    # Sort the paired list based on the first element of each pair
    sorted_paired_list = sorted(paired_list, key=lambda x: x[0])

    # Unzip the sorted paired list
    current_optimal_key_numbers, alpha_list, a_values_measurement_temp, b_values_measurement_temp = zip(*sorted_paired_list)

    # Convert the results back to lists
    current_optimal_key_numbers = list(current_optimal_key_numbers)
    alpha_list = list(alpha_list)
    a_values_measurement = list(a_values_measurement_temp)
    b_values_measurement = list(b_values_measurement_temp)

    alpha_values = np.array(alpha_list)
    gamma_keys = current_optimal_key_numbers
    logger.info(f"alpha_values: {alpha_values}")
    logger.info(f"gamma_keys: {gamma_keys}")

    return current_optimal_keys, alpha_values, gamma_keys, a_values_measurement, b_values_measurement


def determine_interpolated_betas(relative_relationships_measurement,
                                 alpha,
                                 fingerprints,
                                 gamma,
                                 d_a,
                                 d_b,
                                 fingerprint_handle):
    """
    Interpolates betas for the surrounding area based on minimal difference in strain values.

    Args:
        relative_relationships_measurement (np.ndarray): Relative strain relationships from the measurement.
        alpha (np.ndarray): Alpha values for scaling.
        fingerprints (pd.DataFrame): DataFrame of fingerprint data.
        gamma (np.ndarray): Gamma values for scaling.
        d_a (float): Distance between each grid point in length direction.
        d_b (float): Distance between each grid point in width direction.

    Returns:
        tuple: Contains the following:
            - interpolated_betas (list[float]): List of interpolated betas.
            - square_points (pd.DataFrame): Data from the surrounding grid points.
            - a_interpolated (float): Interpolated length coordinate.
            - b_interpolated (float): Interpolated width coordinate.
```

```
            - a_coupled (np.ndarray): Length coordinates of the coupled grid points.
            - b_coupled (np.ndarray): Width coordinates of the coupled grid points.
    """

    # Find the column with the fingerprint handle
    fingerprint_column = next((col for col in fingerprints.columns if fingerprint_handle in col), None)
    if fingerprint_column:
        # Filter rows where the fingerprint column is in the keys of gamma
        active_fingerprints = fingerprints[fingerprints[fingerprint_column].isin(gamma)]

        # Count occurrences of each value in gamma
        gamma_counts = Counter(gamma)

        # Repeat rows based on gamma_counts
        active_fingerprints = active_fingerprints.loc[
            active_fingerprints.index.repeat(active_fingerprints[fingerprint_column].map(gamma_counts))
        ]
    else:
        logger.info("No matching column found for the specified fingerprint handle.")

    active_fingerprints = active_fingerprints.reset_index(drop=True)

    ## Data preperation ##
    # Obtain the column names for the a and b values
    a_column = next((col for col in fingerprints.columns if a_handle in col), None)
    b_column = next((col for col in fingerprints.columns if b_handle in col), None)

    # Obtaining the a and b values for the grid point that was coupled previously
    a_coupled = active_fingerprints[a_column].values
    b_coupled = active_fingerprints[b_column].values

    logger.info(f"a_coupled = {a_coupled}")
    logger.info(f"b_coupled = {b_coupled}")

    # Setting ranges for quadrants and grid points in quadrants
    G = range(len(active_fingerprints))
    H = range(4)
    K = range(4)
    area_names = ["top_left_area", "bot_left_area", "bot_right_area", "top_right_area"]

    ## Save the nearest keys ##
    current_optimal_areas = []
    current_optimal_betas = []

    for idx, value in enumerate(gamma_keys):
        if a_values_measurement[idx] <= active_fingerprints.loc[idx, a_col_name] and b_values_measurement[idx] <= active_fingerprints.loc[idx, b_col_name]:
            temp_area = area_names[0]
            temp_betas = np.array([0, 0, 1, 0])
        elif a_values_measurement[idx] <= active_fingerprints.loc[idx, a_col_name] and b_values_measurement[idx] >= active_fingerprints.loc[idx, b_col_name]:
            temp_area = area_names[1]
            temp_betas = np.array([0, 0, 0, 1])
        elif a_values_measurement[idx] >= active_fingerprints.loc[idx, a_col_name] and b_values_measurement[idx] >= active_fingerprints.loc[idx, b_col_name]:
            temp_area = area_names[2]
            temp_betas = np.array([1, 0, 0, 0])
        elif a_values_measurement[idx] >= active_fingerprints.loc[idx, a_col_name] and b_values_measurement[idx] <= active_fingerprints.loc[idx, b_col_name]:
            temp_area = area_names[3]
            temp_betas = np.array([0, 1, 0, 0])
        current_optimal_areas.append(temp_area)
        current_optimal_betas.append(temp_betas)

    # Calculating all the coordinates for each grid point in each area in each minimum coupled grid point
    area = []
    for g in G:
        area_g = {}
```

```python
            for h in H:
                area_hg = pd.DataFrame()
                for k in K:
                    a_khg = a_coupled[g] + d_a*(math.floor(k/2) + math.floor(h/2) - 1)
                    b_khg = b_coupled[g] + d_b*((1+(-1)**(math.floor((k-1)/2)))/2 + (1+(-1)**(math.floor((h-1)/2)))/2 - 1)
                    filtered_fingerprint = fingerprints[
    (fingerprints[a_column].between(a_khg - grid_point_filter_offset, a_khg + grid_point_filter_offset)) &
    (fingerprints[b_column].between(b_khg - grid_point_filter_offset, b_khg + grid_point_filter_offset))]

                    if not filtered_fingerprint.empty:
                        area_hg = pd.concat([area_hg, filtered_fingerprint], ignore_index=True)
                area_g[area_names[h]] = area_hg
            area.append(area_g)

        # Starting loop to find minimal interpolated locations
        for i in range(n_footsteps):

            logger.info(f"current_optimal_areas: {current_optimal_areas}")

            # Setting the initial dummy values of the optimization results
            min_diff = float('inf')
            min_combination = None
            a_interpolated = [float('inf') for g in G]
            b_interpolated = [float('inf') for g in G]

            # Obtaining the 4 area combination options of the footstep currently being optimized
            area_combinations_filtered = []
            for h in range(len(area_names)):
                combination = current_optimal_areas.copy()
                combination[i] = area_names[h]  # Change only the i-th position
                area_combinations_filtered.append(combination)

            # Running the interpolation for each of the 4 areas of footstep i
            for combination in area_combinations_filtered:

                # Obtain the active_areas_data for the current combination
                active_areas_keys = combination
                active_areas_data = []
                for index, key in enumerate(active_areas_keys):
                    active_areas_data.append(area[index][key])

                ## Minimization ##
                # Defining the objective function to find the minimal difference area and with that the betas
                def objective(betas, gp_points_strain, relative_relationships_measurement, alpha):

                    # Split betas in lengths of 4 for each of the coupled grid points
                    beta_part = [betas[b:b+4] for b in range(0, len(betas), 4)]
                    beta_part = np.array(beta_part)

                    # Restructure betas to be multipliable
                    beta_part = np.expand_dims(beta_part, axis=-1)
                    alpha = np.expand_dims(alpha, axis=-1)
                    alpha = np.expand_dims(alpha, axis=-1)

                    # Tranforming grid point strain to array
                    gp_points_strain = np.array(gp_points_strain)

                    # Determine relative square points strain after scaling and summation
                    scaled_points_strain = np.sum(beta_part * alpha * gp_points_strain, axis=(0,1))
                    relative_square_points_strain = scaled_points_strain[:, np.newaxis] / scaled_points_strain[np.newaxis, :]

                    # Determine the MAD values per square grid point
```

```
            diff_nested = np.abs(relative_square_points_strain - relative_relationships_measurement)
            median_diff = np.median(diff_nested)
            median_absolute_deviation = np.median(np.abs(diff_nested - median_diff))
            return median_absolute_deviation

        # Defining the constraint
        def constraint_sum(betas):
            constraints = []
            for g in G:
                beta_part = betas[g*4:(g+1)*4]
                constraints.append(np.sum(beta_part) - 1)
            return np.array(constraints)

        # Define the constraints dictionary
        constraints = {'type': 'eq', 'fun': constraint_sum}

        # Defining the bounds on the beta values to be between 0 and 1
        bounds = [(0, 1) for _ in range(4*len(active_fingerprints))]

        # Check if 4 points surround the grid points. Does not apply to grid points at the boundaries of the grid, such as (0, 0)
        if all(len(item) == 4 for item in active_areas_data):

            gp_points_strain = []

            for g in G:
                # Filter the strain values from the dataframe
                points_strain = active_areas_data[g].filter(like=strain_handle).filter(regex=disabled_strain_pattern).values
                points_strain[(points_strain >= 0) & (abs(points_strain) < infinitely_small_value)] = infinitely_small_value
                points_strain[(points_strain < 0) & (abs(points_strain) < infinitely_small_value)] = -infinitely_small_value
                gp_points_strain.append(points_strain)

            # Define intial guess for beta values
            initial_betas = np.full(4*len(active_fingerprints), 0.25)

            # Run the minimization function to obtain the optimal area and betas
            result = minimize(objective, initial_betas, args=(gp_points_strain, relative_relationships_measurement, alpha),
                            method='SLSQP', bounds=bounds, constraints=constraints,)

            # Check if the minimization was successful and save the result data
            if result.success:
                diff = result.fun
                if diff < min_diff:
                    min_diff = diff
                    betas = result.x
                    beta_part = [betas[b:b+4] for b in range(0, len(betas), 4)]
                    interpolated_betas = beta_part

                    min_combination = combination
                    current_optimal_areas[i] = combination[i]

                    min_areas_data = active_areas_data
                    a_interpolated = []
                    b_interpolated = []
                    for part in range(len(interpolated_betas)):
                        a_interpolated.append(np.sum(beta_part[part][b] * active_areas_data[part].iloc[b, :].filter(like=a_handle).values[0] for b in range(4)))
                        b_interpolated.append(np.sum(beta_part[part][b] * active_areas_data[part].iloc[b, :].filter(like=b_handle).values[0] for b in range(4)))

            else:
                print(f"Optimization failed for {combination}: {result.message}")
                logger.info(f"Optimization failed for {combination}: {result.message}")
```

```
            else:
                logger.info(f"Skipping combination {combination} because at least one area falls outside of asset boundaries")

    logger.info(f"min_combination: {min_combination}")
    logger.info(f"interpolated_betas: {interpolated_betas}")
    logger.info(f"a_interpolated: {a_interpolated}")
    logger.info(f"b_interpolated: {b_interpolated}")
    return interpolated_betas, min_areas_data, a_interpolated, b_interpolated, a_coupled, b_coupled


# Function to determine force and hot spot scale factor
def determine_alpha(measurement_row,
                    interpolated_betas,
                    min_areas_data):
    """
    Determines the overall force and hot spot scale factor (alpha) based on measurement data and interpolated betas.

    Args:
        measurement_row (Series): A single row of measurement data.
        interpolated_betas (list[float]): List of beta values interpolated in a surrounding area.
        min_areas_data (list[DataFrame]): List of DataFrames containing minimal area data for each grid point.

    Returns:
        np.ndarray: Array of scale factors (alpha) for overall force and hot spot.
    """

    min_areas_data_filtered = []
    # Filter to only give the used strain columns
    for i in range(len(min_areas_data)):
        min_areas_data_filtered.append(min_areas_data[i].filter(like=strain_handle).filter(regex=disabled_strain_pattern).values)
    measurement_row_filtered = measurement_row.filter(like=strain_handle).filter(regex=disabled_strain_pattern).values

    # Convert lists to arrays for multiplication
    interpolated_betas = np.array(interpolated_betas)
    min_areas_data_filtered = np.array(min_areas_data_filtered)

    # Restructure betas to be multipliable
    interpolated_betas = np.expand_dims(interpolated_betas, axis=-1)

    # Define the objective function using NumPy for faster computation
    def objective(alpha):
        alpha = np.expand_dims(alpha, axis=-1)
        alpha = np.expand_dims(alpha, axis=-1)
        return np.sum(np.abs(np.sum(alpha * interpolated_betas * min_areas_data_filtered, axis = (0,1)) - measurement_row_filtered))

    # Set initial guess for alpha value
    initial_alpha = np.full(len(interpolated_betas),1)

    # Set bounds to ensure alpha values are non-negative
    bounds = [(0, 10) for _ in range(len(interpolated_betas))]

    # Use minimize with the vectorized objective function and bounds
    result = minimize(objective, initial_alpha, method='TNC', bounds=bounds,)


    # Save resulting scale factor
    alpha = result.x

    logger.info(f"scale factors: {alpha}")
    return alpha


# Function to scale the measurement and coupled grid points data into an export of
```

```
        predicted values
def scale_data(alpha,
               measurement_row,
               interpolated_betas,
               min_areas_data,
               a_interpolated,
               b_interpolated):
    """
    Scales the measurement data and grid points data to predict values.

    Args:
        alpha (np.ndarray): Array of scale factors for overall force and hot spot.
        measurement_row (Series): A single row of measurement data.
        interpolated_betas (list[float]): List of beta values interpolated in a
            surrounding area.
        min_areas_data (list[DataFrame]): List of DataFrames containing minimal area data
             for each grid point.
        a_interpolated (list[float]): Interpolated values for 'a' parameter.
        b_interpolated (list[float]): Interpolated values for 'b' parameter.

    Returns:
        Series: Output row of all values as predicted by the model.
    """

    # Extract strain and hot spot data from the measurement_row
    strain_data = measurement_row.filter(like=strain_handle)

    # Hot spot data
    hot_spot_data = []
    for i in range(len(min_areas_data)):
        hot_spot_data.append(min_areas_data[i].filter(like=hot_spot_handle))
    hot_spot_data_filtered = np.array(hot_spot_data)

    # Create a copy of the measurement_row to avoid modifying the original data
    coupled_fingerprint_data_scaled = measurement_row.copy(deep=True)
    strain_data_indices = [col for col in measurement_row.index if strain_handle in col]
    hot_spot_data_indices = [col for col in fingerprints.columns if hot_spot_handle in
        col]

    # Defining column names
    a_column_name = measurement_row.filter(like=a_handle).index
    b_column_name = measurement_row.filter(like=b_handle).index
    force_column_name = measurement_row.filter(like=force_handle).index
    force_column_name_FP = min_areas_data[0].filter(like=force_handle).columns[0]

    # Adding all load description columns to the DataFrame
    for i in range(len(min_areas_data)):
        coupled_fingerprint_data_scaled[a_column_name[i]] = a_interpolated[i]
        coupled_fingerprint_data_scaled[b_column_name[i]] = b_interpolated[i]
        coupled_fingerprint_data_scaled[force_column_name[i]] = np.sum(alpha[i] *
            interpolated_betas[i] * min_areas_data[i].loc[:, force_column_name_FP])

    # Update the copied DataFrame with the scaled hot spot data and the strain data
    alpha = np.expand_dims(alpha, axis=-1)
    alpha = np.expand_dims(alpha, axis=-1)
    interpolated_betas = np.expand_dims(interpolated_betas, axis=-1)
    coupled_fingerprint_data_scaled[strain_data_indices] = strain_data.values

    # Update the hot spot data directly in the DataFrame without concatenation
    coupled_fingerprint_data_scaled.loc[hot_spot_data_indices] = np.sum(alpha *
        interpolated_betas * hot_spot_data_filtered, axis=(0, 1))

    return coupled_fingerprint_data_scaled

### END OF FUNCTION DEFINITIONS ###


### START OF CODE ###

# Import statements
import numpy as np
```

```
    import pandas as pd
    from scipy.optimize import minimize
    import math

    # Filter fingerprints based on boundary boxes as obtained by the cameras
    a_values_measurement = measurement_row.filter(like=a_handle).values
    b_values_measurement = measurement_row.filter(like=b_handle).values

    # Remove NaN values using numpy
    a_values_measurement =  [x for x in a_values_measurement if not math.isnan(x)]
    b_values_measurement =  [x for x in b_values_measurement if not math.isnan(x)]

    a_col_name = fingerprints.filter(like=a_handle).columns[0]
    b_col_name = fingerprints.filter(like=b_handle).columns[0]
    fingerprint_col_name = fingerprints.filter(like=fingerprint_handle).columns[0]

    # Opening a dictionary to store fingerprints for different footsteps
    fingerprints_dict = {}

    # Iterate over the range of a_values_measurement
    for i in range(len(a_values_measurement)):
        # Construct the condition for the current i
        condition = ((fingerprints[a_col_name] >= a_values_measurement[i] - a_box_boundary)
                     & (fingerprints[a_col_name] <= a_values_measurement[i] + a_box_boundary)
                     & (fingerprints[b_col_name] >= b_values_measurement[i] - b_box_boundary)
                     & (fingerprints[b_col_name] <= b_values_measurement[i] + b_box_boundary))

        # Dictionary entry name
        key_fp = f"footstep_{i}"

        # Combine the condition with the previous conditions using OR
        fingerprints_dict[key_fp] = fingerprints[condition]

    # Zero replacement value
    infinitely_small_value = 1*10**(-8)

    # Determine the column index of the column where the fingerprint name identifier appears
    fingerprint_index = fingerprints.columns.get_loc(fingerprint_handle)

    # Reuse the filtered array for the second calculation
    measurement_filtered = measurement_row.filter(like=strain_handle).filter(regex=
        disabled_strain_pattern)

    # Convert to numpy array for faster computation
    measurement_array = measurement_filtered.values

    # Replace zero values with infinitely small values
    measurement_array[(measurement_array >= 0) & (abs(measurement_array) <
        infinitely_small_value)] = infinitely_small_value
    measurement_array[(measurement_array < 0) & (abs(measurement_array) <
        infinitely_small_value)] = -infinitely_small_value

    # Compute the relative relationships using numpy operations directly
    relative_relationships_measurement = measurement_array[:, np.newaxis] / measurement_array

    # Obtain the number of footsteps
    n_footsteps = len(a_values_measurement)

    if n_footsteps > 0:

        # Determine gammas
        gamma, alpha, gamma_keys, a_values_measurement, b_values_measurement =
            determine_scale_factors_iteratively(fingerprints_dict=fingerprints_dict,
            fingerprints=fingerprints, fingerprint_index=fingerprint_index, measurement_array
            =measurement_array, a_values_measurement=a_values_measurement,
            b_values_measurement=b_values_measurement)

        # Determine interpolation
        interpolated_betas, min_areas_data, a_interpolated, b_interpolated, a_coupled,
            b_coupled = determine_interpolated_betas(relative_relationships_measurement=
            relative_relationships_measurement, alpha=alpha, fingerprints=fingerprints, gamma
```

```
            =gamma, d_a=d_a, d_b=d_b, fingerprint_handle=fingerprint_handle)

        # Determine force scale factor
        alpha = determine_alpha(measurement_row=measurement_row, interpolated_betas=
            interpolated_betas, min_areas_data=min_areas_data)

        # Scale new measurement point according to the scale factor together with the matched
             fingerprint
        coupled_fingerprint_data_scaled = scale_data(alpha=alpha, measurement_row=
            measurement_row, interpolated_betas=interpolated_betas, min_areas_data=
            min_areas_data, a_interpolated=a_interpolated, b_interpolated=b_interpolated)

    else:
        logger.info("Skipped␣0␣footstep␣row")
        coupled_fingerprint_data_scaled = measurement_row

    ### END OF CODE ###

    return coupled_fingerprint_data_scaled, a_coupled, b_coupled, a_interpolated,
        b_interpolated, alpha
```

I

Rainflow Counting Source Code

```python
import pandas as pd
import numpy as np
import logging

# Use the already configured logger
logger = logging.getLogger('shared_logger')

def run_rainflow_counting(
    df, n_bins, hot_spot_handle, maximum_stress, minimum_stress,
    frequency_col_name, half_cycles_column_name, full_cycles_column_name, stress_cycles_column_name
):
    """
    Executes the rainflow counting process on a given stress dataset.

    Parameters:
    - df (DataFrame): The input DataFrame with stress values.
    - n_bins (int): Number of bins for discretizing stress values.
    - hot_spot_handle (str): Placeholder for specific handling (not implemented here).
    - maximum_stress (float): Maximum stress value for binning.
    - minimum_stress (float): Minimum stress value for binning.
    - frequency_col_name (str): Name of the column for cycle frequencies.
    - half_cycles_column_name (str): Name of the column for half-cycle stress values.
    - full_cycles_column_name (str): Name of the column for full-cycle stress values.
    - stress_cycles_column_name (str): Name of the column for combined stress values.

    Returns:
    - DataFrame: A DataFrame with rainflow counting results.
    """

    # Step 1: Apply peak-valley filtering to retain significant stress points.
    def apply_peakvalley_filter(df):
        """
        Identifies and retains only the peaks and valleys in the stress data.

        Parameters:
        - df (DataFrame): DataFrame containing the stress values.

        Returns:
        - DataFrame: Filtered DataFrame with only peaks and valleys.
        """
        logger.info("Applying peak-valley filtering.")
        stress = df.values
        peakvalley_drop = np.zeros(len(df), dtype=bool)

        # Identify peaks and valleys by checking neighboring values
        for i in range(1, len(df) - 1):
            if stress[i] > stress[i - 1] and stress[i] > stress[i + 1]:
                continue  # Peak
```

```
            elif stress[i] < stress[i - 1] and stress[i] < stress[i + 1]:
                continue  # Valley
            else:
                peakvalley_drop[i] = True  # Not a peak or valley

        # Filter out points that are neither peaks nor valleys
        df = df[~pd.Series(peakvalley_drop)].reset_index(drop=True)
        return df

    # Filter the stress data to retain only peaks and valleys
    df = apply_peakvalley_filter(df)

    # Step 2: Discretize stress values into bins.
    def apply_binning(df, n_bins, maximum_stress, minimum_stress):
        """
        Bins stress values into discrete intervals for analysis.

        Parameters:
        - df (DataFrame): DataFrame containing the stress values.
        - n_bins (int): Number of bins.
        - maximum_stress (float): Maximum stress value.
        - minimum_stress (float): Minimum stress value.

        Returns:
        - list: Binned stress values.
        - list: List of bin ranges and metadata.
        """
        logger.info("Applying stress value binning.")
        stress = df.values
        stress_range = abs(maximum_stress - minimum_stress)
        bin_size = stress_range / n_bins
        bins = []

        # Generate bin ranges and metadata (start, end, average value, bin index)
        start_value = minimum_stress
        for i in range(n_bins):
            end_value = start_value + bin_size
            avg_value = (end_value + start_value) / 2
            bins.append((start_value, end_value, avg_value, i))
            start_value = end_value

        # Map each stress value to the nearest bin's average value
        bin_stress_values = [min(bins, key=lambda b: abs(b[2] - s))[2] for s in stress]
        return bin_stress_values, bins

    # Discretize the filtered stress data
    bin_stress_values, bins = apply_binning(df, n_bins, maximum_stress, minimum_stress)

    # Step 3: Identify full cycles using four-point counting.
    def apply_fourpointcounting(bin_stress_values):
        """
        Applies four-point counting to detect full stress cycles.

        Parameters:
        - bin_stress_values (list): List of binned stress values.

        Returns:
        - list: List of identified full stress cycles.
        - list: Residual stress values.
        """
        logger.info("Performing four-point cycle counting.")
        stress = np.array(bin_stress_values)
        rainflow_cycles = []

        while True:
            # Look for a four-point cycle in the data
            for n in range(len(stress) - 3):
                S1, S2, S3, S4 = stress[n:n + 4]
                S_inner = abs(S2 - S3)
                S_outer = abs(S1 - S4)

```

```
                # Check the cycle conditions
                if (S1 > S4 and S_inner <= S_outer and S1 >= S3 and S4 <= S2) or \
                   (S1 < S4 and S_inner <= S_outer and S1 <= S3 and S4 >= S2):
                    rainflow_cycles.append((S2, S3))
                    # Remove the identified cycle from the data
                    stress = np.concatenate((stress[:n + 1], stress[n + 3:]))
                    break
            else:
                break

        return rainflow_cycles, stress.tolist()

    # Perform four-point rainflow counting
    rainflow_cycles, residue = apply_fourpointcounting(bin_stress_values)

    # Step 4: Export the results to a DataFrame.
    def export_rainflow(rainflow_cycles, residue):
        """
        Converts the rainflow counting results into a DataFrame.

        Parameters:
        - rainflow_cycles (list): Full cycles from the analysis.
        - residue (list): Remaining stress points not part of a full cycle.

        Returns:
        - DataFrame: Rainflow counting summary.
        """
        logger.info("Exporting rainflow counting results.")

        # Process full cycles (absolute stress differences)
        full_cycles = [abs(c[0] - c[1]) for c in rainflow_cycles]
        df_full = pd.DataFrame({full_cycles_column_name: full_cycles})
        df_full = df_full.groupby(full_cycles_column_name).size().reset_index(name=
            frequency_col_name)

        # Process half cycles (absolute stress differences in residue)
        half_cycles = [abs(residue[i + 1] - residue[i]) for i in range(len(residue) - 1)]
        df_half = pd.DataFrame({half_cycles_column_name: half_cycles})
        df_half = df_half.groupby(half_cycles_column_name).size().reset_index(name=
            frequency_col_name)
        df_half[frequency_col_name] *= 0.5  # Adjust frequency for half cycles

        # Combine full and half cycle results into one DataFrame
        df_combined = pd.concat([
            df_full.rename(columns={full_cycles_column_name: stress_cycles_column_name}),
            df_half.rename(columns={half_cycles_column_name: stress_cycles_column_name})
        ])
        df_combined = df_combined.groupby(stress_cycles_column_name)[frequency_col_name].sum
            ().reset_index()

        return df_combined

    # Export the rainflow counting results
    export_df = export_rainflow(rainflow_cycles, residue)
    return export_df
```

J

Controlled Location Loading Result Data

#	DAM	GP	SC	a [mm]	b [mm]	Force [N]	L2 [με]	L4 [με]	L7 [με]	L9 [με]	R2 [με]	R4 [με]	R7 [με]	R9 [με]
1	1	3	3	1495.0	300.0	-834.0	27.60	59.80	69.90	32.20	47.20	110.78	138.01	58.40
	2	3	3	1494.6	300.2	-834.0	27.60	59.82	69.86	32.17	47.26	110.87	137.87	58.40
	3	3	3	1495.0	300.0	-834.0	30.58	67.63	86.98	36.59	38.53	129.72	169.81	57.92
	4	3	3	1489.7	304.1	-1011.5	33.13	71.68	82.98	38.32	57.99	136.13	168.09	71.16
2	1	3	3	2654.0	217.0	-834.0	5.08	11.20	20.40	34.90	5.18	11.60	22.20	40.40
	2	3	3	2655.9	216.6	-841.0	5.06	11.18	20.41	34.87	5.17	11.58	22.16	40.34
	3	3	3	2654.0	217.0	-834.0	4.30	12.22	27.36	42.66	-1.36	8.60	24.04	45.40
	4	3	3	2617.0	196.5	-790.7	5.85	13.04	24.54	41.04	5.80	12.92	24.16	40.48
3	1	3	3	1334.0	125.0	-834.0	49.60	116.05	111.64	47.80	34.40	76.40	73.80	33.30
	2	3	3	1333.0	124.6	-834.3	49.70	116.26	111.74	47.81	34.44	76.35	73.65	33.23
	3	3	3	1334.0	125.0	-834.0	54.73	146.12	135.73	50.39	27.54	88.74	84.55	27.73
	4	3	3	1348.4	135.9	-1041.5	59.85	139.54	137.24	58.86	44.09	98.32	96.86	43.42
4	1	3	3	459.0	154.0	-834.0	79.40	78.40	39.30	17.40	60.00	61.60	33.30	15.00
	2	3	3	459.0	154.5	-834.5	79.01	78.34	39.34	17.40	59.97	61.74	33.38	15.04
	3	3	3	459.0	154.0	-834.0	85.24	85.29	48.49	16.67	79.51	76.36	35.16	9.66
	4	3	3	534.4	156.9	-871.5	80.63	92.63	46.43	20.50	62.43	73.61	39.46	17.77
5	1	3	3	498.0	133.0	-834.0	83.70	88.90	43.80	19.20	54.50	61.00	33.80	15.30
	2	3	3	487.6	133.6	-847.0	84.90	88.47	43.66	19.19	55.61	61.11	33.76	15.30
	3	3	3	498.0	133.0	-834.0	91.80	98.24	52.08	16.92	70.00	70.54	34.24	9.90
	4	3	3	551.8	145.8	-910.4	86.32	102.61	51.00	22.44	61.96	75.50	41.04	18.54
6	1	3	3	2347.0	254.0	-834.0	12.30	27.20	48.70	54.20	15.10	34.30	69.10	82.60
	2	3	3	2340.3	253.4	-824.0	12.31	27.24	48.84	53.64	15.18	34.41	69.24	81.28
	3	3	3	2347.0	254.0	-834.0	11.60	30.27	66.09	76.56	7.84	35.61	76.69	95.09
	4	3	3	2244.1	208.1	-812.6	15.73	35.14	66.79	64.17	16.60	37.31	72.80	70.93
7	1	3	3	1502.0	198.0	-834.0	36.90	83.80	103.38	45.10	37.60	85.80	105.97	46.10
	2	3	3	1502.4	198.0	-834.4	36.90	83.89	103.39	45.10	37.62	85.84	106.01	46.11
	3	3	3	1502.0	198.0	-834.0	40.12	93.35	120.10	49.72	27.50	101.18	136.40	44.28
	4	3	3	1503.4	216.8	-1023.1	43.08	97.27	119.25	52.47	48.20	110.68	137.69	59.50
8	1	3	3	557.0	227.0	-834.0	60.30	73.90	39.50	17.70	75.80	91.00	45.60	20.10
	2	3	3	557.7	227.0	-833.3	60.28	73.96	39.47	17.71	75.65	90.99	45.57	20.12
	3	3	3	557.0	227.0	-834.0	69.33	83.29	47.60	16.19	85.76	106.86	48.40	12.85
	4	3	3	569.6	216.8	-971.2	72.14	90.60	47.88	21.43	85.07	104.50	52.83	23.38
9	1	3	3	300.0	25.0	-834.0	97.80	73.40	34.70	15.00	26.20	26.90	17.70	8.27
	2	3	3	275.2	22.8	-886.9	101.01	73.30	34.65	14.99	26.74	26.91	17.63	8.27
	3	3	3	300.0	25.0	-834.0	104.40	80.19	40.87	12.73	39.67	35.07	18.41	4.60
	4	3	3	385.3	77.0	-855.8	96.90	83.31	40.01	17.43	41.36	41.30	24.81	11.46
10	1	3	3	1661.0	3.0	-834.0	47.40	111.91	178.13	77.70	17.80	36.80	45.00	22.90

#	DAM	GP	SC	a [mm]	b [mm]	Force [N]	L2 [με]	L4 [με]	L7 [με]	L9 [με]	R2 [με]	R4 [με]	R7 [με]	R9 [με]
	2	3	3	1659.1	2.7	-833.9	47.59	112.17	177.85	77.59	17.87	36.76	44.89	22.85
	3	3	3	1661.0	3.0	-834.0	48.78	134.20	222.48	88.57	9.97	37.82	58.20	22.77
	4	3	3	1647.1	24.3	-1040.9	57.93	136.28	212.36	92.54	24.67	51.65	64.80	32.08
11	1	3	3	1147.0	12.0	-834.0	71.40	168.16	120.27	50.80	23.60	47.80	40.30	19.50
	2	3	3	1144.8	11.9	-835.8	71.72	168.45	120.29	50.77	23.73	47.90	40.33	19.54
	3	3	3	1147.0	12.0	-834.0	74.32	204.17	137.08	48.45	21.27	55.32	42.55	14.93
	4	3	3	1123.6	19.2	-996.5	85.78	200.26	138.80	58.65	29.58	60.12	49.40	23.75
12	1	3	3	2162.0	201.0	-834.0	18.70	41.80	80.50	66.50	19.20	43.10	84.10	69.80
	2	3	3	2163.1	200.6	-835.1	18.70	41.86	80.54	66.66	19.18	43.10	84.02	69.71
	3	3	3	2162.0	201.0	-834.0	18.22	49.34	112.52	87.02	7.50	43.28	95.22	70.64
	4	3	3	2109.7	160.5	-972.6	25.34	57.42	114.44	86.14	22.12	49.36	92.30	68.80
13	1	3	3	1101.0	148.0	-834.0	55.60	128.03	87.10	37.80	42.70	95.30	67.60	30.30
	2	3	3	1114.7	147.5	-827.7	54.65	126.30	87.52	38.00	41.98	93.78	67.71	30.35
	3	3	3	1101.0	148.0	-834.0	65.78	168.80	107.56	40.08	34.42	100.40	66.54	20.62
	4	3	3	1073.8	126.3	-992.8	71.06	162.98	106.46	46.00	47.92	105.04	73.42	33.20
14	1	3	3	1431.0	26.0	-834.0	55.80	132.42	149.27	62.60	22.50	46.80	49.50	24.00
	2	3	3	1433.6	26.0	-836.2	55.80	132.44	149.83	62.87	22.53	46.87	49.77	24.07
	3	3	3	1431.0	26.0	-834.0	54.49	149.51	171.71	64.11	16.19	53.43	61.80	21.56
	4	3	3	1404.3	42.6	-991.8	65.81	156.01	168.38	70.77	29.13	61.33	63.76	30.37
15	1	3	3	385.0	279.0	-834.0	48.20	45.80	26.40	12.00	88.60	75.60	36.90	16.20
	2	3	3	379.1	279.2	-844.3	48.47	45.80	26.36	12.02	89.11	75.65	36.91	16.19
	3	3	3	385.0	279.0	-834.0	60.60	54.40	34.03	12.27	106.72	87.28	36.47	8.17
	4	3	3	421.9	226.4	-866.4	63.60	61.12	32.87	14.82	79.90	74.25	37.52	16.60
16	1	3	3	1571.0	124.0	-834.0	41.30	95.80	133.34	57.10	29.40	64.90	84.80	38.30
	2	3	3	1568.4	123.9	-834.1	41.43	96.12	132.91	56.95	29.42	64.91	84.71	38.28
	3	3	3	1571.0	124.0	-834.0	43.75	112.70	155.94	60.09	18.51	72.18	104.65	34.14
	4	3	3	1536.8	134.2	-1001.7	49.93	115.65	152.46	65.33	37.23	82.64	104.26	46.74
17	1	3	3	1624.0	195.0	-834.0	33.80	76.70	111.83	49.30	34.00	77.10	112.58	49.50
	2	3	3	1623.2	194.7	-834.0	33.86	76.82	111.67	49.26	33.96	77.08	112.30	49.50
	3	3	3	1624.0	195.0	-834.0	35.15	89.83	134.73	52.50	22.98	87.07	137.08	46.62
	4	3	3	1627.0	194.2	-1031.7	41.78	94.77	138.56	61.22	41.83	95.00	138.80	61.28
18	1	3	3	2411.0	113.0	-834.0	13.50	30.60	62.40	83.20	10.40	22.80	39.50	47.00
	2	3	3	2411.3	113.3	-835.4	13.48	30.71	62.51	82.98	10.39	22.83	39.57	46.87
	3	3	3	2411.0	113.0	-834.0	12.08	33.40	76.73	102.65	0.35	19.80	43.40	56.75
	4	3	3	2297.8	115.4	-744.6	15.38	35.08	71.75	77.85	11.55	25.35	44.50	44.80
19	1	3	3	1878.0	349.0	-834.0	17.50	37.20	55.00	30.00	35.90	83.70	164.77	85.30
	2	3	3	1878.1	349.1	-834.1	17.48	37.17	55.10	30.01	35.86	83.66	164.52	85.26
	3	3	3	1878.0	349.0	-834.0	16.77	36.28	62.47	33.07	23.85	99.12	213.08	90.33
	4	3	3	1831.5	365.5	-991.0	20.32	42.77	59.88	31.92	46.02	107.77	204.30	101.13
20	1	3	3	1456.0	94.0	-834.0	48.20	112.97	132.24	56.00	28.80	62.80	70.30	32.20
	2	3	3	1456.6	94.0	-834.5	48.15	112.94	132.34	56.03	28.89	62.88	70.39	32.23
	3	3	3	1456.0	94.0	-834.0	50.40	139.35	163.50	59.33	19.83	66.55	79.93	28.35
	4	3	3	1453.5	82.1	-1012.6	60.00	141.11	164.40	69.50	33.65	72.75	80.95	37.40
21	1	3	3	718.0	100.0	-834.0	81.30	131.71	64.50	28.00	43.10	70.50	40.90	18.70
	2	3	3	716.3	99.9	-835.8	81.55	131.44	64.53	28.04	43.17	70.52	40.95	18.78
	3	3	3	718.0	100.0	-834.0	92.78	166.66	79.46	26.10	40.76	73.02	39.60	11.20
	4	3	3	743.1	85.4	-980.2	97.20	163.52	80.48	34.86	46.70	79.00	47.08	21.70
22	1	3	3	353.0	2.0	-834.0	105.80	87.50	41.10	17.70	22.30	25.90	17.80	8.40
	2	3	3	348.3	1.6	-841.4	106.46	87.36	41.02	17.73	22.35	25.92	17.78	8.40
	3	3	3	353.0	2.0	-834.0	113.06	96.57	46.77	14.26	34.27	33.77	19.04	5.60
	4	3	3	484.6	34.0	-768.7	95.59	99.48	46.96	20.27	28.54	34.93	22.64	10.59
23	1	3	3	519.0	307.0	-834.0	41.80	50.70	30.20	13.90	94.40	104.38	49.90	21.70
	2	3	3	519.4	306.7	-833.5	41.83	50.85	30.23	13.91	93.97	104.18	49.88	21.70
	3	3	3	519.0	307.0	-834.0	62.05	69.00	41.75	15.60	92.25	108.80	48.65	13.50
	4	3	3	450.2	255.4	-1102.5	72.10	74.35	41.30	18.75	110.93	107.84	53.20	23.40
24	1	3	3	984.0	203.0	-834.0	51.50	112.21	68.10	30.20	54.30	119.07	71.40	31.50
	2	3	3	984.2	203.0	-834.1	51.52	112.07	68.14	30.17	54.32	118.90	71.47	31.49
	3	3	3	984.0	203.0	-834.0	62.26	148.62	83.76	29.84	52.00	143.10	78.88	25.48

#	DAM	GP	SC	a [mm]	b [mm]	Force [N]	L2 [με]	L4 [με]	L7 [με]	L9 [με]	R2 [με]	R4 [με]	R7 [με]	R9 [με]
	4	3	3	998.0	198.9	-1015.8	62.94	137.69	85.04	37.62	64.90	142.31	87.04	38.40
25	1	3	3	853.0	187.0	-834.0	58.60	115.23	62.50	27.60	56.10	110.14	60.30	26.80
	2	3	3	829.5	186.7	-855.0	61.13	116.63	62.50	27.63	58.32	111.24	60.27	26.82
	3	3	3	853.0	187.0	-834.0	65.91	144.31	75.03	26.86	56.04	135.19	70.02	22.55
	4	3	3	857.1	180.0	-1012.0	72.45	142.25	77.36	34.10	66.64	130.05	72.03	32.11
26	1	3	3	151.0	225.0	-834.0	45.00	29.80	16.30	7.38	55.50	34.20	17.50	7.81
	2	3	3	150.0	225.2	-837.5	44.97	29.79	16.25	7.38	55.44	34.16	17.55	7.82
	3	3	3	151.0	225.0	-834.0	47.86	28.39	15.20	4.50	56.46	36.28	16.43	2.31
	4	3	3	153.7	212.8	-835.1	47.06	30.79	16.60	7.51	53.63	33.68	17.50	7.81
27	1	3	3	786.0	343.0	-834.0	31.60	55.60	36.20	17.00	87.60	157.22	77.70	33.40
	2	3	3	784.8	343.4	-834.9	31.55	55.57	36.17	16.94	87.72	156.91	77.71	33.38
	3	3	3	786.0	343.0	-834.0	36.91	59.45	36.92	13.38	92.76	199.61	89.64	23.58
	4	3	3	728.7	360.7	-1024.5	35.36	58.66	39.15	18.41	115.55	189.16	91.68	39.34
28	1	3	3	2621.0	228.0	-834.0	5.92	13.10	23.60	38.00	6.20	13.90	27.00	47.40
	2	3	3	2625.9	225.5	-850.7	5.89	13.01	23.60	38.21	6.18	13.90	26.91	47.09
	3	3	3	2621.0	228.0	-834.0	6.30	13.90	29.00	44.30	-1.10	10.10	27.10	49.20
	4	3	3	2613.1	195.2	-817.9	6.16	13.70	25.80	42.80	6.16	13.80	25.80	43.00
29	1	3	3	1022.0	262.0	-834.0	41.40	89.40	59.20	26.80	61.90	139.78	85.20	36.90
	2	3	3	1022.3	262.0	-834.1	41.40	89.26	59.20	26.78	61.93	139.42	85.23	36.94
	3	3	3	1022.0	262.0	-834.0	48.79	116.38	72.31	25.68	57.46	167.00	97.25	30.78
	4	3	3	1006.8	252.3	-1034.9	53.65	115.53	74.83	33.71	75.81	169.17	101.96	44.33
30	1	3	3	699.0	238.0	-834.0	53.90	85.20	46.20	20.80	72.20	113.66	56.90	25.00
	2	3	3	697.9	238.1	-834.5	53.88	85.17	46.18	20.79	72.26	113.52	56.80	24.94
	3	3	3	699.0	238.0	-834.0	64.50	101.02	54.23	18.97	77.27	139.50	65.30	18.95
	4	3	3	701.0	242.2	-1024.9	65.10	103.05	56.25	25.35	89.62	141.14	70.63	30.95
31	1	3	3	1648.0	21.0	-834.0	46.60	109.88	172.15	74.60	19.50	40.60	50.90	25.30
	2	3	3	1647.2	20.4	-834.7	46.80	110.16	171.92	74.62	19.44	40.54	50.86	25.29
	3	3	3	1648.0	21.0	-834.0	47.60	133.50	210.76	80.04	9.16	39.38	60.88	22.50
	4	3	3	1627.8	19.7	-1006.2	57.48	135.39	205.91	88.80	23.68	49.34	60.82	30.18
32	1	3	3	1459.0	108.0	-834.0	46.70	109.21	128.29	54.50	30.10	66.20	74.70	34.00
	2	3	3	1459.2	107.9	-834.2	46.72	109.30	128.33	54.47	30.14	66.09	74.77	33.93
	3	3	3	1459.0	108.0	-834.0	49.00	125.25	144.80	56.10	20.80	69.00	81.65	27.80
	4	3	3	1413.0	103.7	-949.9	55.65	130.54	142.34	60.40	34.85	76.30	81.40	37.05
33	1	3	3	1624.0	285.0	-834.0	26.50	58.00	79.10	36.50	41.20	95.80	144.34	62.20
	2	3	3	1626.1	284.9	-835.5	26.51	57.94	79.36	36.68	41.19	95.84	144.40	62.33
	3	3	3	1624.0	285.0	-834.0	26.77	63.87	95.30	38.50	29.83	110.97	174.83	59.37
	4	3	3	1576.8	297.3	-988.2	31.27	67.97	87.00	40.23	51.97	121.49	171.37	73.20
34	1	3	3	383.0	213.0	-834.0	64.50	57.10	30.40	13.60	73.70	63.80	32.70	14.50
	2	3	3	383.2	213.1	-834.1	64.23	57.14	30.37	13.64	73.39	63.86	32.72	14.51
	3	3	3	383.0	213.0	-834.0	71.62	61.34	37.62	13.44	95.96	76.72	32.20	6.94
	4	3	3	444.9	190.7	-831.9	70.22	68.72	35.62	15.88	68.64	67.38	35.12	15.66
35	1	3	3	1181.0	194.0	-834.0	46.60	107.07	82.50	36.30	46.60	107.09	82.50	36.30
	2	3	3	1184.3	193.8	-833.1	46.51	106.76	82.67	36.37	46.49	106.66	82.55	36.32
	3	3	3	1181.0	194.0	-834.0	50.93	126.63	94.88	35.95	41.53	129.08	94.80	29.08
	4	3	3	1151.2	198.6	-1001.8	56.40	128.79	95.58	42.15	57.90	132.65	97.93	43.05
36	1	3	3	1648.0	135.0	-834.0	37.80	87.30	134.69	58.70	28.60	63.40	91.50	41.60
	2	3	3	1649.1	135.0	-834.9	37.84	87.29	134.56	58.80	28.61	63.41	91.58	41.71
	3	3	3	1648.0	135.0	-834.0	37.07	89.40	140.13	58.10	21.20	68.40	109.97	39.93
	4	3	3	1638.4	164.2	-915.3	39.37	90.10	135.57	59.47	34.23	76.73	111.69	50.03
37	1	3	3	857.0	42.0	-834.0	83.70	164.86	84.90	36.40	30.40	56.90	38.10	17.90
	2	3	3	856.4	42.0	-834.3	83.78	164.34	84.93	36.42	30.37	56.90	38.08	17.85
	3	3	3	857.0	42.0	-834.0	86.53	196.53	99.03	34.60	30.98	69.63	40.58	12.70
	4	3	3	854.8	52.3	-984.4	97.00	189.85	98.20	42.15	37.93	71.08	46.73	21.78
38	1	3	3	1687.0	94.0	-834.0	39.60	92.10	150.87	66.40	24.60	53.50	77.10	36.40
	2	3	3	1686.7	94.2	-834.3	39.58	92.04	150.45	66.35	24.64	53.61	77.25	36.42
	3	3	3	1687.0	94.0	-834.0	38.50	99.15	172.05	69.30	15.95	58.15	99.85	37.55
	4	3	3	1668.0	116.1	-956.8	44.30	102.55	162.65	71.40	30.65	67.35	97.31	45.05
39	1	3	3	236.0	346.0	-834.0	27.70	25.60	16.30	7.60	87.20	59.00	28.00	12.10
	2	3	3	235.5	346.2	-834.7	27.82	25.67	16.31	7.59	86.60	58.93	27.97	12.16

#	DAM	GP	SC	a [mm]	b [mm]	Force [N]	L2 [με]	L4 [με]	L7 [με]	L9 [με]	R2 [με]	R4 [με]	R7 [με]	R9 [με]
	3	3	3	236.0	346.0	-834.0	30.95	24.90	16.25	5.30	94.15	62.75	25.60	3.50
	4	3	3	257.5	300.1	-730.2	33.70	28.30	16.80	7.75	72.20	50.40	24.45	10.65
40	1	3	3	258.0	259.0	-834.0	47.80	37.40	21.10	9.61	75.50	52.70	26.20	11.50
	2	3	3	257.0	259.3	-836.2	47.70	37.45	21.08	9.61	75.10	52.63	26.13	11.53
	3	3	3	258.0	259.0	-834.0	54.98	40.58	24.36	7.56	85.36	57.60	23.68	3.52
	4	3	3	304.9	207.4	-798.1	59.54	46.66	24.80	11.12	65.44	50.54	26.12	11.66
41	1	3	3	214.0	159.0	-834.0	64.60	43.00	22.00	9.76	50.80	36.20	19.80	8.96
	2	3	3	209.0	157.8	-845.1	65.04	43.04	21.92	9.75	50.73	35.97	19.70	8.92
	3	3	3	214.0	159.0	-834.0	68.90	43.40	22.70	5.80	62.00	42.60	20.00	4.40
	4	3	3	229.7	175.9	-870.4	65.70	45.40	23.50	10.50	57.80	41.30	22.10	9.98
42	1	3	3	349.0	179.0	-834.0	71.60	58.60	30.20	13.50	64.50	53.90	28.60	12.80
	2	3	3	349.5	179.0	-833.2	71.40	58.59	30.19	13.41	64.36	53.87	28.59	12.80
	3	3	3	349.0	179.0	-834.0	78.24	61.02	35.70	12.24	82.60	61.48	27.08	6.70
	4	3	3	343.2	159.6	-898.8	81.04	65.52	33.24	14.70	63.80	53.92	29.24	13.18
43	1	3	3	352.0	33.0	-834.0	101.10	82.40	38.90	16.80	29.20	30.80	20.00	9.33
	2	3	3	335.9	30.6	-863.3	103.26	82.30	38.93	16.86	29.48	30.81	19.90	9.31
	3	3	3	352.0	33.0	-834.0	114.32	99.06	48.28	14.32	37.74	36.40	19.18	4.70
	4	3	3	467.3	65.0	-876.2	102.69	102.81	49.02	21.24	40.24	45.50	27.84	12.90
44	1	3	3	1873.0	304.0	-834.0	20.40	44.20	71.20	37.90	33.40	77.40	151.15	77.40
	2	3	3	1881.0	304.4	-840.7	20.37	44.20	71.46	38.25	33.43	77.38	151.59	78.44
	3	3	3	1873.0	304.0	-834.0	20.77	45.50	86.05	44.52	22.37	86.93	187.53	85.45
	4	3	3	1883.2	306.2	-1015.1	24.42	52.98	85.27	45.77	40.37	93.50	183.05	95.32
45	1	3	3	1173.0	313.0	-834.0	31.50	67.50	54.80	25.50	62.20	146.79	109.13	46.50
	2	3	3	1184.3	312.8	-830.0	31.15	67.03	55.02	25.56	61.32	144.86	109.62	46.68
	3	3	3	1173.0	313.0	-834.0	37.82	82.88	65.00	25.04	52.76	174.82	122.14	35.90
	4	3	3	1155.6	308.4	-1008.0	39.08	84.14	66.78	30.94	75.42	177.20	128.62	54.88
46	1	3	3	2561.0	46.0	-834.0	9.12	21.00	44.20	73.30	6.23	13.40	21.00	24.90
	2	3	3	2562.6	45.6	-838.5	9.12	21.00	44.18	73.03	6.21	13.37	20.94	24.90
	3	3	3	2561.0	46.0	-834.0	9.70	25.25	59.83	91.45	-0.45	11.65	25.50	32.68
	4	3	3	2457.5	63.6	-761.0	11.58	26.65	55.73	78.80	7.65	16.53	26.68	30.30
47	1	3	3	1603.0	70.0	-834.0	44.60	104.58	153.68	65.70	24.20	51.90	67.00	31.60
	2	3	3	1602.9	69.7	-833.9	44.67	104.62	153.62	65.65	24.15	51.89	67.04	31.57
	3	3	3	1603.0	70.0	-834.0	44.10	119.30	184.20	69.10	14.05	55.85	86.90	29.60
	4	3	3	1598.8	84.3	-1002.2	52.50	122.62	178.12	76.40	30.50	66.00	86.45	40.30
48	1	3	3	1270.0	167.0	-834.0	47.10	109.11	95.30	41.40	40.70	92.40	81.40	36.10
	2	3	3	1270.0	166.9	-834.5	47.13	109.22	95.34	41.39	40.71	92.46	81.46	36.09
	3	3	3	1270.0	167.0	-834.0	49.77	131.03	109.90	40.43	36.17	111.93	95.20	30.20
	4	3	3	1265.5	157.1	-992.7	57.60	133.83	116.03	50.23	47.23	106.83	93.57	41.63
49	1	3	3	289.0	285.0	-834.0	43.60	36.80	21.50	9.84	83.60	60.80	29.70	13.00
	2	3	3	294.8	285.0	-822.6	43.17	36.86	21.43	9.81	82.62	60.91	29.71	13.05
	3	3	3	289.0	285.0	-834.0	53.68	41.95	25.40	7.60	92.33	64.93	27.13	4.60
	4	3	3	321.0	232.4	-823.5	56.53	46.38	25.28	11.40	73.93	57.73	29.20	12.93
50	1	3	3	1801.0	10.0	-834.0	40.90	95.80	177.82	85.10	16.70	34.80	46.60	24.70
	2	3	3	1796.9	10.1	-831.2	40.91	95.91	176.93	84.50	16.65	34.82	46.59	24.64
	3	3	3	1801.0	10.0	-834.0	43.64	118.56	224.34	95.28	6.32	28.20	45.88	22.04
	4	3	3	1718.0	15.4	-987.1	52.14	122.47	207.99	93.82	21.52	44.94	58.30	29.64

K

Frequency per Stress Range Table

See next page

$\Delta\sigma$ [MPa]	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
1	129	266.5	154.5	281	163.5	302	118	242.5
2	44.5	26.5	53.5	14	42.5	7.5	48.5	35.5
3	24.5	7	23.5	0	30.5	0.5	28.5	10
4	21	1	13	0	10	0	24	3
5	11.5	2	6.5	0	5	0	15	2.5
6	10.5	1.5	3	0	8	0	8.5	2
7	9	0	5.5	0	6	0	5.5	3
8	8.5	0	4	0	2	0	16	0.5
9	5.5	0	3	0	2.5	0	4.5	0
10	7	0	5.5	0	2	0	7.5	0
11	4.5	0	3	0	2	0	2.5	0
12	2.5	0	3	0	4	0	3.5	0
13	2	0	2.5	0	1.5	0	2.5	0
14	4.5	0	1	0	0	0	3.5	0
15	1.5	0	1.5	0	0	0	2.5	0
16	1	0	0.5	0	0	0	2	0
17	2	0	0	0	0	0	1	0
18	2	0	0	0	0	0	0.5	0
19	1.5	0	0	0	0	0	0.5	0
20	0.5	0	0	0	0	0	0	0
21	0.5	0	0	0	0	0	4	0
22	2.5	0	0	0	0	0	1.5	0
23	3	0	0	0	0	0	2.5	0
24	3	0	0	0	0	0	2.5	0
25	0.5	0	0	0	0	0	2.5	0
26	0	0	0	0	0	0	1	0
27	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	0.5	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0

$\Delta\sigma$ [MPa]	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
1	123	246	125	269	126	270	166	289.5
2	51	27.5	58	21	47	22.5	65	15
3	25	9	23	5.5	28.5	5	27	0
4	23	6	17.5	3	19	3.5	14	0
5	12.5	2.5	17	2	8.5	3	3	0
6	12	2	9.5	1	14	0	5	0
7	7	3	7.5	0.5	9.5	0	5.5	0
8	12.5	0	7.5	0	6	0	6.5	0
9	4.5	0	6.5	0	3.5	0	2	0
10	4.5	0	4.5	0	4.5	0	4	0
11	4.5	0	2.5	0	5	0	1.5	0
12	6	0	3	0	5.5	0	0.5	0
13	4	0	1	0	2.5	0	1.5	0
14	2	0	2.5	0	1.5	0	2.5	0
15	2.5	0	1.5	0	3	0	1	0
16	1	0	0	0	2	0	0	0
17	0.5	0	0.5	0	0.5	0	0	0
18	1.5	0	0.5	0	0.5	0	0	0
19	1.5	0	1	0	1	0	0	0
20	1.5	0	0	0	0	0	0	0
21	1	0	2	0	1	0	0	0
22	0.5	0	1.5	0	1.5	0	0	0
23	1	0	2	0	3	0	0	0
24	3.5	0	4	0	4	0	0	0
25	1.5	0	0.5	0	1.5	0	0	0
26	1	0	0	0	0	0	0	0
27	0	0	0.5	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
33	0	0	0.5	0	0	0	0	0
34	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0

$\Delta\sigma$ [MPa]	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}	f_{21}	f_{22}	f_{23}
1	167.5	298.5	122	248	132.5	251.5	125.5	261
2	64	9	58.5	29	53.5	36.5	49.5	20
3	28	0.5	25.5	12	25.5	7	28.5	6.5
4	13	0	20	5.5	16	4	20	3
5	6	0	16.5	4.5	13	3.5	13	2.5
6	5.5	0	5.5	0.5	9.5	1	11	0
7	2.5	0	8	0.5	6.5	0.5	8.5	0
8	2	0	5	0	6	0.5	6	0
9	2	0	7	0	4	0	3.5	0
10	3	0	5	0	5	0	5	0
11	1	0	3	0	5	0	6	0
12	1	0	2.5	0	2	0	1.5	0
13	0.5	0	3	0	3	0	1	0
14	0.5	0	3	0	3.5	0	2.5	0
15	0	0	1.5	0	0	0	0.5	0
16	0	0	1.5	0	1.5	0	0.5	0
17	0	0	0.5	0	1.5	0	0.5	0
18	0	0	1.5	0	0	0	1	0
19	0	0	1	0	2	0	1	0
20	0	0	2.5	0	0	0	0	0
21	0	0	1	0	0.5	0	2.5	0
22	0	0	0.5	0	2	0	2.5	0
23	0	0	3	0	2	0	2	0
24	0	0	3	0	3.5	0	3	0
25	0	0	0	0	0	0	0.5	0
26	0	0	0.5	0	0	0	0	0
27	0	0	0	0	0.5	0	0.5	0
28	0	0	0.5	0	0.5	0	0	0
29	0	0	0.5	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0
43	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0

L

Endurance Limit Aluminium Alloy Details

$\Delta\sigma$ [MPa]	N_x
1	∞
2	∞
3	∞
4	∞
5	∞
6	51.0E+06
7	22.1E+06
8	10.7E+06
9	5.6E+06
10	3.8E+06
11	2.7E+06
12	2.0E+06
13	1.5E+06
14	1.2E+06
15	945.3E+03
16	758.7E+03
17	617.1E+03
18	507.9E+03
19	422.4E+03
20	354.7E+03
21	300.4E+03
22	256.3E+03
23	220.3E+03
24	190.6E+03
25	165.8E+03

$\Delta\sigma$ [MPa]	N_x
26	145.1E+03
27	127.6E+03
28	112.7E+03
29	100.0E+03
30	89.1E+03
31	79.7E+03
32	71.5E+03
33	64.4E+03
34	58.2E+03
35	52.7E+03
36	47.9E+03
37	43.6E+03
38	39.8E+03
39	36.4E+03
40	33.4E+03
41	30.7E+03
42	28.3E+03
43	26.1E+03
44	24.2E+03
45	22.4E+03
46	20.8E+03
47	19.3E+03
48	18.0E+03
49	16.7E+03
50	15.6E+03

Table L.1: Endurance limit of aluminium alloy welds with detail category 12-3,4. [23]