FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation: *Computational Engineering*	The *spring* semester, *2023* Open
Author: *René König, 268127*	
Supervisor at UiS: *Professor Aksel Hiorth* Co-supervisor: *Professor Jasna Bogunovic Jakobsen* External supervisor(s): *Bernardo Morais da Costa, Statens vegvesen*	
Thesis title: *Analysis of wind and response measurement data from a suspension bridge*	
Credits (ECTS): *30*	
Keywords: *Python, Data Science, Software Development, Suspension Bridge, Wind Engineering, Wind Response, Full-Scale Monitoring, Anemometers, Accelerometers*	Pages: *116* + appendix: *169* Stavanger, *14.06.2023* René König

Abstract

The Lysefjord Bridge is a suspension bridge at the entrance to the Lysefjord in south-western Norway at which full-scale measurements on wind conditions and bridge response are collected using anemometers and accelerometers. In this work Python is used to develop a toolset for analysing the wind and response measurement data from the Lysefjord Bridge. The functionality is provided through different methods compiled in a class. This includes methods for importing and combining data from multiple days, re-arranging and interpreting the data, feature engineering, data cleaning, filtering and various types of visualisations. The code is demonstrated in an analysis of 30 days of data. The analysis focuses on the wind conditions for south-westerly and north-easterly winds in terms of wind speeds, primary directions, turbulence intensity and angle of attack as well as the bridges lateral, vertical and torsional wind response. The analysis shows on average slightly higher wind speeds, lower turbulence intensities and higher angles of attack for south-westerly winds, compared to north-easterly winds. Towards the southern end of the bridge the wind direction has a south tendency for south-westerly winds and north tendency for north-easterly winds. Turbulence intensity is measured slightly higher on the downwind side of the bridge. The angle of attack is straightened towards 0° on the downwind side. Furthermore, the analysis shows that the assumption of a linear correlation between drag coefficient and angle of attack used in the so-called quasi-steady theory of wind loading and the corresponding numerical simulations underestimates most of the larger lateral bridge responses at angles of attack above 0°. The lift and moment coefficients estimated using similar linearity assumptions overestimate some of the larger vertical and torsional bridge responses at angles of attack above +5°.

Acknowledgements

I want to thank the *Department of Mechanical and Structural Engineering and Material Science* at *Univeritetet i Stavanger* for giving me the chance and freedom to examine the data of the Lysefjord Bridge from a different perspective. The well documented previous work on the full-scale monitoring system and data analysis at the Lysefjord Bridge have been very helpful in approaching this new domain.

A special thanks goes to my co-supervisor from this department, *Professor Jasna Bogunovic Jakobsen*, who helped me get up to speed with the complex topic, providing me with detailed information and insights. She enthusiastically supported my ideas, discussed different approaches with me and provided me guidance along the way during our regular supervision meetings.

I would also like to thank my supervisor *Professor Aksel Hiorth* who quickly got me and my fellow students up to speed with Python in our first semester in the *Computational Engineering* program at the *Department of Energy Resources.* He taught us the importance of proper code documentation in the various projects we did. In my third semester he gave me the opportunity to experience this importance from the perspective of a teaching assistant, helping the new students to get up to speed with Python and correcting their projects.

Finally, I want to thank *Statens vegvesen* and *Bernardo Morais da Costa* for supporting my work on this project. I hope future works on this topic will find my work helpful.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

The following abbreviations and acronyms are used in the data, code and documentation of this work:

acc	Accelerometer
anemo	Anemometer
CFD	Computational fluid dynamics
id	Identification
IDE	Integrated development environment
ind	Index
key	Key of a key-value pair in a dictionary
LFB	Lysefjord Bridge
MATLAB	Matrix laboratory
N/E/S/W	North, East, South or West. Combinations possible.
NaN	Not a number
NumPy	Numerical Python
pandas	Python Data Analysis Library, derived from panel data
PEP	Python Enhancement Proposal
SSV	Statens vegvesen
UiS	Universitetet i Stavanger
VSCode	Visual Studio Code

List of Symbols

The following symbols are used in the data, code and documentation of this work:

$C_{D/L/M}$ ..... Wind load coefficient for drag, lift or moment

$F_{D/L/M}$ ..... Drag, lift or moment

S_t ..... Strouhal number

U_r ..... Resonance wind velocity

f_S ..... Vortex shedding frequency

f_e ..... Eigenfrequency

AOA_ ..... Angle of attack, commonly incidence angle α

AOI_ ..... Horizontal angle of incidence, commonly yaw angle β

Aox_ ..... Lateral acceleration

Aoy_ ..... Longitudinal acceleration

Aoz_ ..... Vertical acceleration

Dir_ ..... Wind direction, direction from which the wind is blowing, commonly Θ

h ..... Height of the bridge girder

H/V/T|S/A|# ..... Horizontal, vertical or torsional | (a)symmetric | eigenmode number #

H_ ..... Horizontal wind component, commonly U

H|##|E/W|t/b .... Hanger | number ## | East or West | top or bottom

Hum_ ..... Humidity

P_ ..... Pressure

T_ ..... Temperature

theta_ ..... Torsional acceleration along the longitudinal axis (calculated), com. θ

Vx_ ..... Lateral across-bridge wind component in x-direction

Vy_ ..... Longitudinal along-bridge wind component in y-direction

W_ ..... Vertical wind component

U ..... Mean wind speed

b ..... Width of the bridge girder

ρ ..... Air density

List of Indices

The following indices are used in the data, code and documentation of this work in combination with some of the symbols from List of Symbols:

_C_ ..... Central acceleration (calculated)
_E_ ..... East side accelerometer
_max..... Maximum of 10-minute data package
_max_by_std ..... Quotient of _max and _std
_mean ..... Mean of 10-minute data package, commonly overlined, e.g. $\overline{U}$
_min..... Minimum of 10-minute data package
_std..... Standard deviation of 10-minute data package, commonly σ
_turb ..... Turbulence intensity of 10-minute data package, commonly I_i
_W_ ..... West side accelerometer

1 Introduction

The collapse of the Tacoma Narrows Bridge in 1940 is a famous example for the response of a mechanical structure to wind leading to its destruction. The still-frame from Barney Elliott´s film-recording of the Tacoma Narrows Bridge in Figure 1 shows the suspension bridge twisting in a torsional vibration mode in a dynamic response to wind with speeds of up to 65 km/h.

Figure 1: Tacoma Narrows Bridge as the wind induced destruction is imminent [1]

While the bridge has been designed and built for a static wind force of 160 km/h, this extreme dynamic response was unforeseen, as no dynamic analysis has been carried out. [2, p. 4]

This shows the importance of analysing the wind response of a structure to prevent over-stressing the structure.

"The Norwegian Public Road Administration is considering building long-span floating suspension bridges in Western Norway to cross deep fjords that are up to around 5 km wide. […] They are therefore extremely wind sensitive structures which require a dedicated investigation of wind-induced effects." [3]

Understanding the wind field characteristics in the complex terrain of the fjords and the wind response of a bridge is essential for those construction projects. Full-scale monitoring of wind and wind response at existing bridges is carried out to help understand these effects.

1.1 The Lysefjord Bridge

Figure 2: Lysefjord Bridge North view at H11. Adapted from [4]

Figure 3: Lysefjord Bridge viewed from the West. Adapted from [5]

The Lysefjord Bridge displayed in Figure 2 and Figure 3 is a suspension bridge located at the entrance of the Lysefjord in South-Western Norway, as shown in Figure 4 and Figure 5.

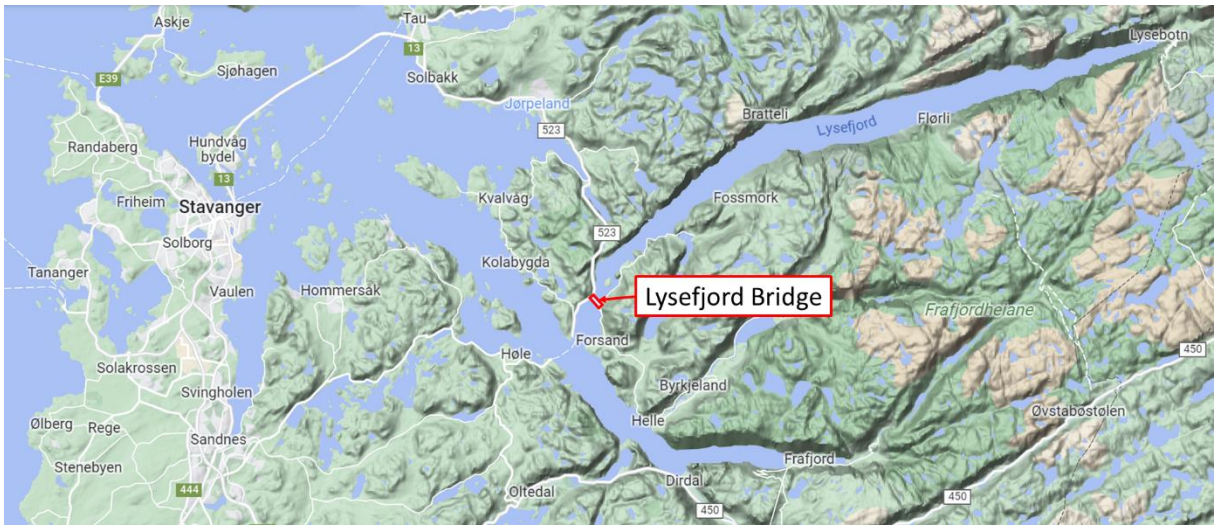


Figure 4: Map of the Stavanger and Lysefjord area. Adapted from [6]

Figure 5: Aerial footage of the Lysefjord Bridge.

Figure 5 shows an aerial photograph of the Lysefjord Bridge at the entrance to the Lysefjord taken by the author.

The bridge is a critical transportation link in the region. It facilitates the transport of goods and people from Stavanger to the municipality of Forsand and other communities south of the Lysefjord and back. This includes heavy industry such as the Norsk Spennbetong AS concrete plant and NCC Helle sandtak sand pit, as depicted in Figure 6. [7, p. 3], [8]

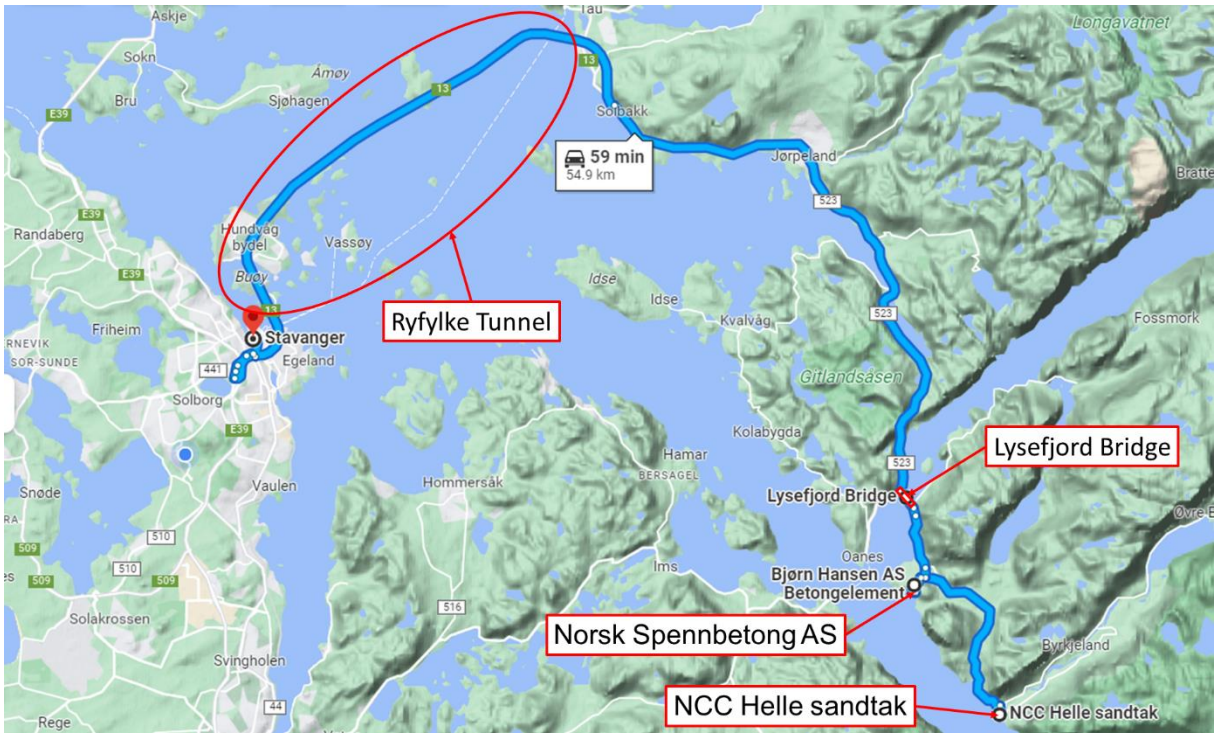


Figure 6: Heavy industry route to Stavanger via Lysefjord Bridge and Ryfylke Tunnel. Adapted from [8]

Note that the Ryfylke Tunnel connecting Stavanger to Ryfylke first opened at the end of 2019. Before the opening of the tunnel, the traffic across the bridge had to take the Lauvvik-Oanes Ferry, as depicted in Figure 7.

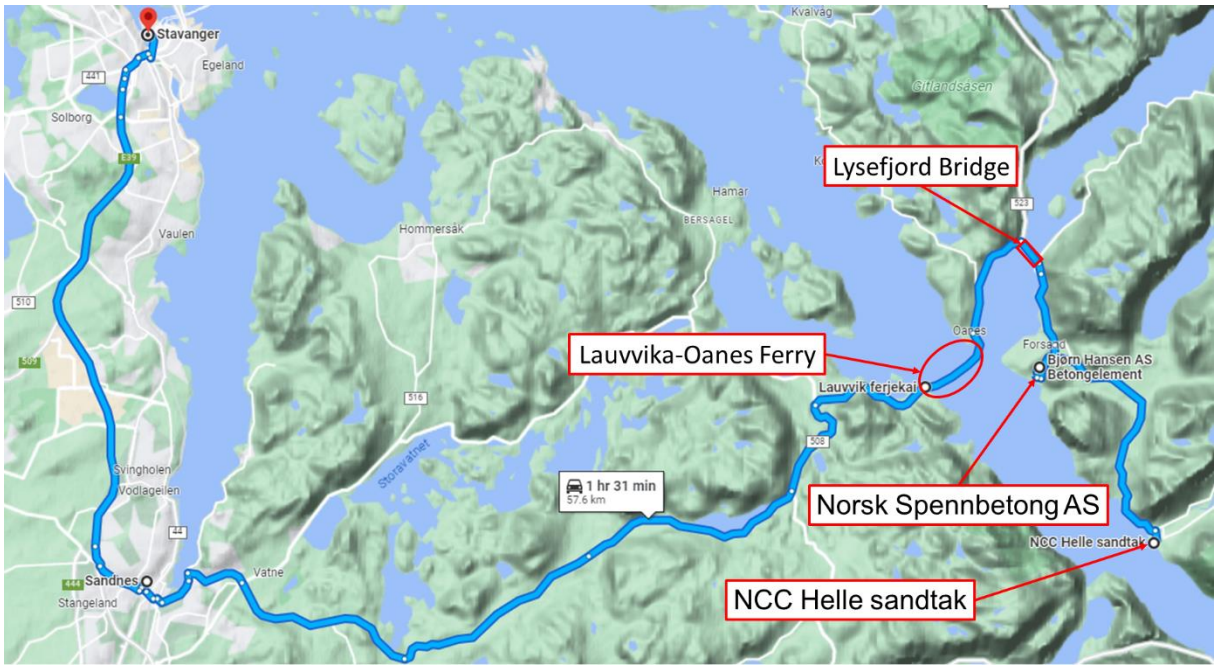


Figure 7: Heavy industry route to Stavanger via Lysefjord Bridge and Lauvvika-Oanes Ferry. Adapted from [9]

While there is also a ferry quay directly in Forsand, enabling traffic to take other ferry routes and by-passing the Lysefjord Bridge, it should be noted that ferry connections have a limited capacity, run at a schedule and are less reliable, compared to a road connection. The Lauvvika-Oanes ferry for example is currently discontinued. [10]

The bridge was built from 1995 to 1997 at a cost of 150 million Norwegian kroner. It has a total length of 639 m with a main span of 446 m. The towers stand at a height of 102.26 m. The main cables have a sag of 45 m, as depicted in Figure 8. [7, p. 3], [3, p. 2]

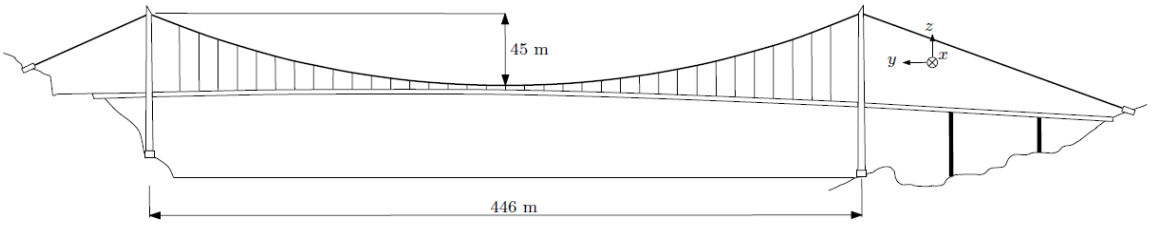


Figure 8: Schematic of the Lysefjord Bridge (sideview from the West) [3, p. 3]

The bridge is asymmetric with the bridge deck support at a height of 52.36 m at the north tower and 44.9 m at the south tower. The bridge girder is suspended from the two main cables by 35 hanger pairs with a longitudinal distance of 12 m between hanger pairs and 19 m between the outmost hangers and the towers. At mid-span, hanger 18 (H18), the deck is suspended 53.37 m above the sea surface with the main cables at their lowest point 3 m above the deck. The girder has a closed steel cross section with a height of 2.76 m and width of 12.3 m, as depicted in Figure 9. [7, pp. 3,5], [3, p. 2]

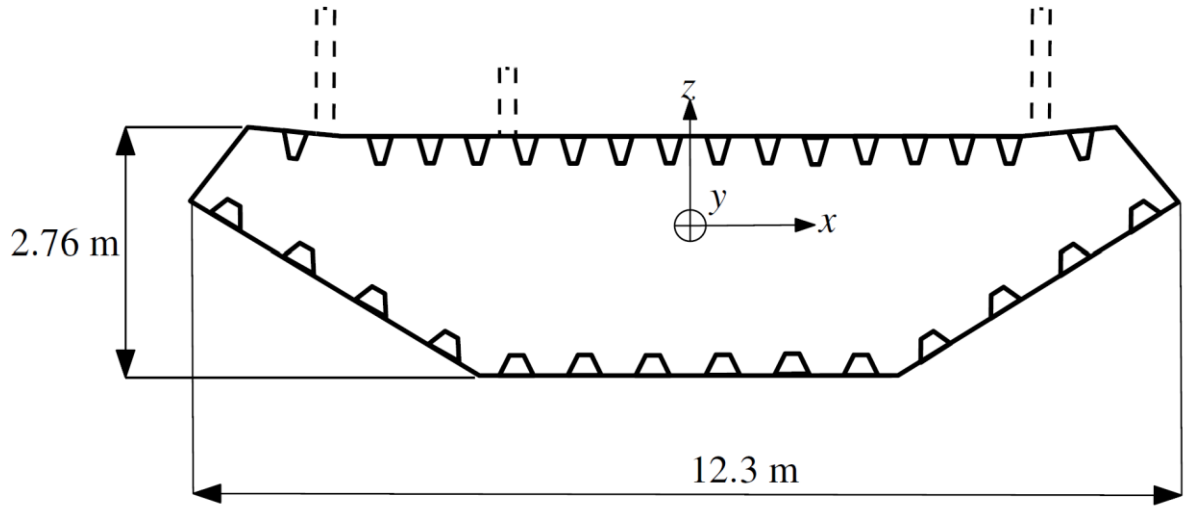


Figure 9: Girder cross section [7, p. 11]

1.1.1 Wind conditions in the surrounding area

The wind in the area around the Lysefjord Bridge is highly influenced by the rough terrain created by the mountains, valleys and fjords. Figure 11 - Figure 15 show the wind directions and monthly strength distributions at different locations around the Stavanger and Lysefjord area. Note that not all weatherstations have been actively reporting data in the same time periods and are therefore not directly compareable. Figure 10 shows a compilation of wind roses overlayed on an overview map of the area. Note that there are currently no weather stations further to the East of the bridge published on windfinder.com.

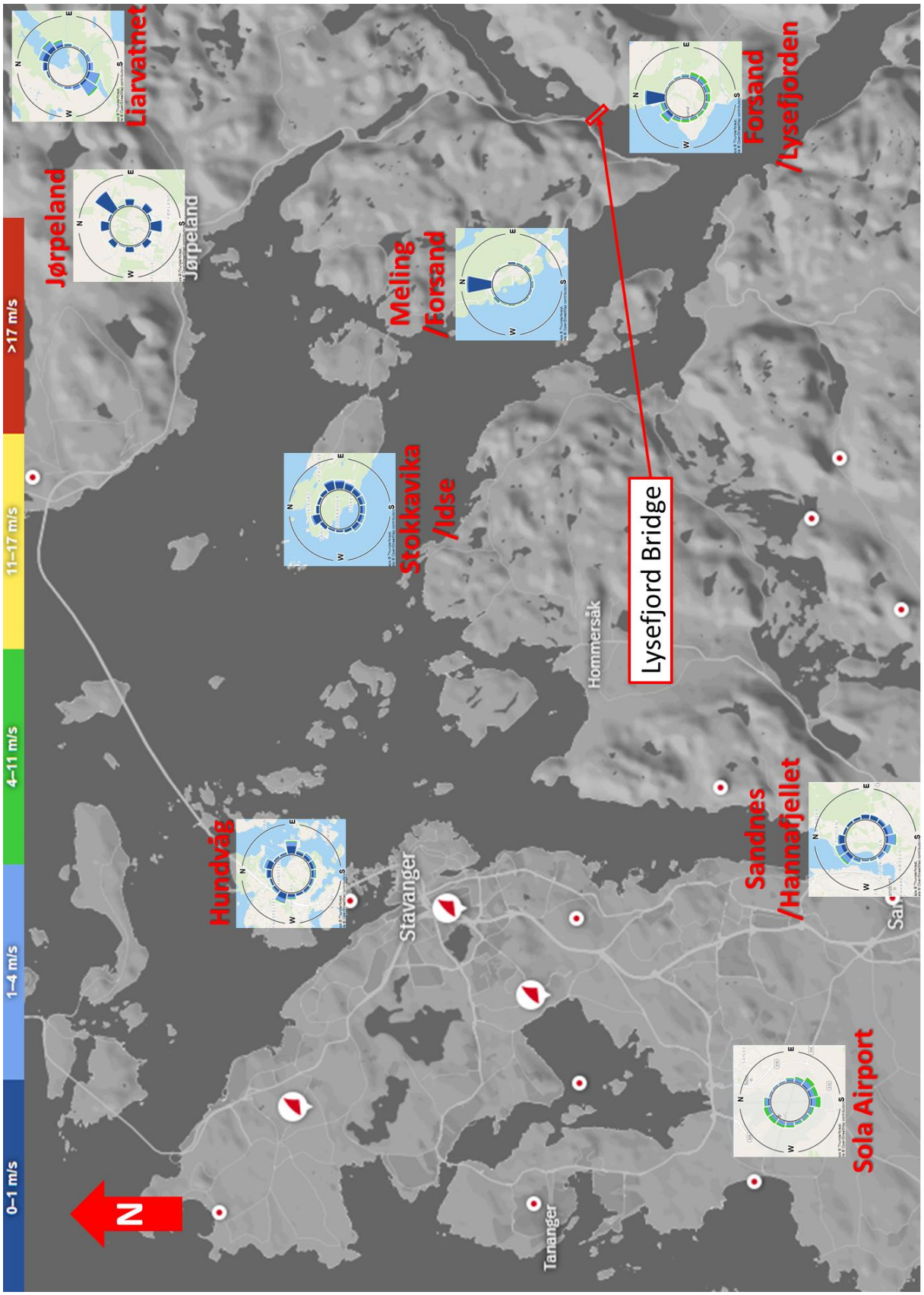


Figure 10: Wind map of the area around the Lysefjord Bridge. Adapted from [11], [12], [13], [14], [15], [16], [17]

The Forsand weather station depicted in Figure 11 has been the closest weather station to the Lysefjord Bridge, about 2 km to the South of the bridge, at the entrance to the Fjord.

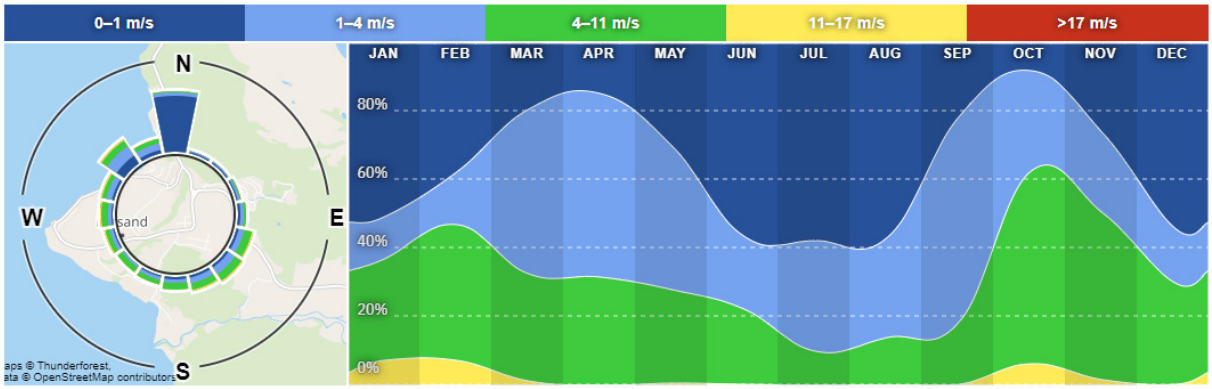


Figure 11: Wind directions and monthly strength distribution at Forsand/Lysefjorden (05/2012 - 02/2017) [11]

Note that the high number of data showing winds of up to 1 m/s from direct North visible in the wind rose might be outliers due to errors in the measurements or averaging method and can be disregarded, as they are for very low windspeeds. The primary wind directions at this weather station are therefore most likely NW and SE, following the orientation of the foot of the mountain to the east. Figure 11 shows relatively high windspeeds at Forsand, especially in the spring and autumn period where more than 40% or even 60% of the wind respectively is recorded at over 4 m/s. This is relatively high, compared to the other weather stations. An exception to this is the weather station at Sola Airport, which is depicted in Figure 12.

The weather station at Sola Airport depicted in Figure 12 is located about 20 km to the West of the Lysefjord Bridge, directly at the Atlantic coast.

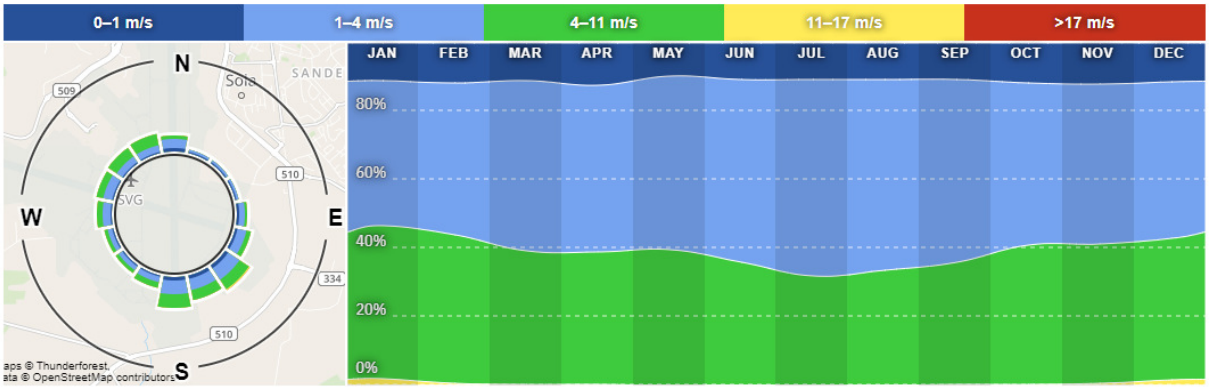


Figure 12: Wind directions and monthly strength distribution at Sola Airport (01/2002 - 04/2023) [14]

The primary wind directions at Sola Airport are NW and SSE. The wind strength distribution is relatively consistent throughout the year with about 40% of the wind recorded at over 4 m/s.

The Meling weather station depicted in Figure 13 has been the second closest weather station to the Lysefjord Bridge, located on the NE-shore of the Høgsfjord, about 5 km NW of the bridge.

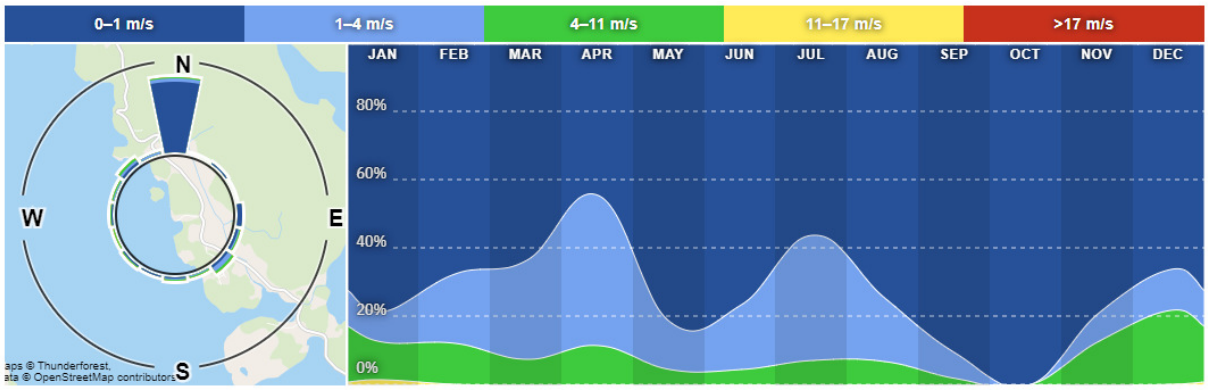


Figure 13: Wind directions and monthly strength distribution at Meling/Forsand (04/2014 - 02/2017) [16]

Note that similarly to the weather station in Forsand there are possibly outliers displaying a lot of wind from the North. This data can again be ignored. The primary wind directions are therefore most likely SE and NW, following the orientation of the shoreline. The wind speeds are below 4 m/s about 90% of the time from February to October. In December about 20% of the wind speeds recorded are above 4 m/s.

The weather station in Stokkavika depicted in Figure 14 is located on the island of Idse in the Høgsfjord, about 12.5 km NW of the Bridge.

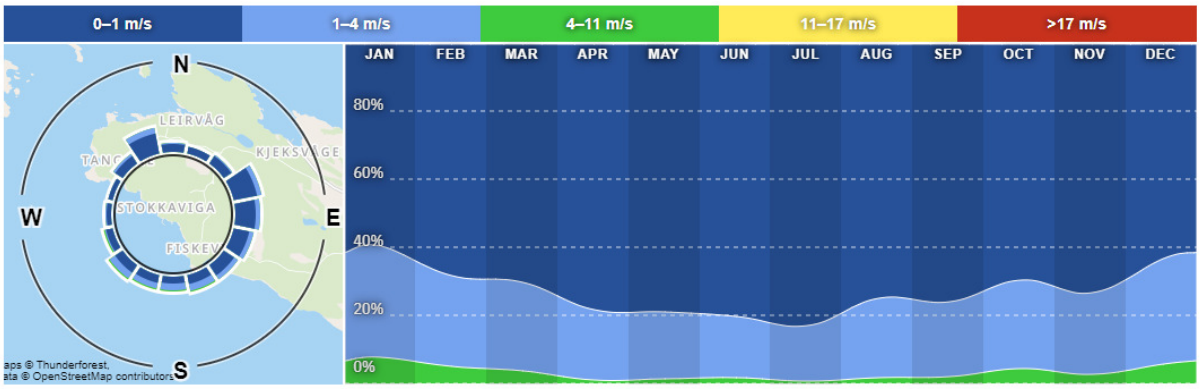


Figure 14: Wind directions and monthly strength distribution at Stokkavika/Idse (04/2014 - 05/2020) [15]

The windspeeds are below 4 m/s about 95% of the time throughout the year and the wind directions are variable with a slight tendency to a broad range of south-westerly winds and a narrow band of NNW winds.

The weather station at Liarvatnet depicted in Figure 15 is about 14 km to the North of the Lysefjord Bridge, in the valley of the Liarvatnet lake.

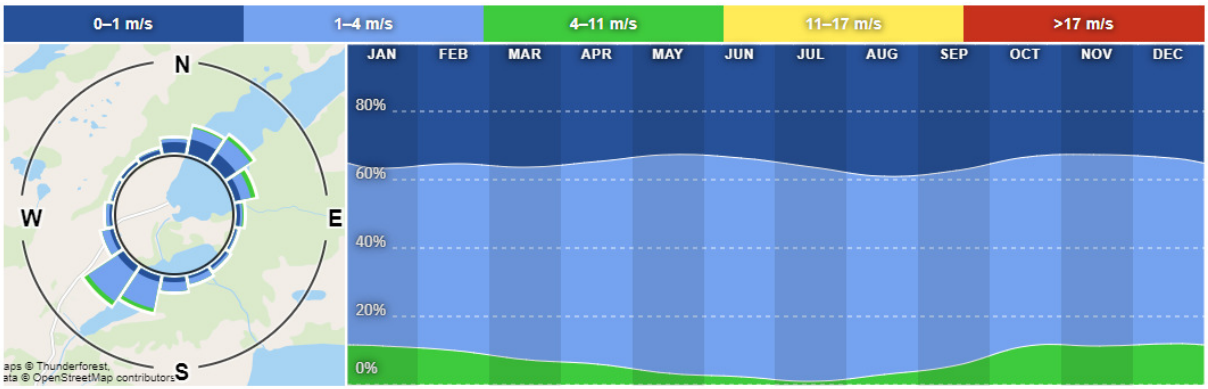


Figure 15: Wind directions and monthly strength distribution at Liarvatnet (04/2014 - 04/2023) [12]

The primary wind directions at Liarvatnet are SW and NE, which follows the orientation of the valley. About 90% of the wind is below 4 m/s throughout the whole year.

Computational fluid dynamics (CFD) simulations have been used in [18] to better understand the influence of the terrain on the wind flow characteristics at the entrance to the Lysefjord. Results from the CFD simulations with different wind directions are displayed in Figure 16 and Figure 17.

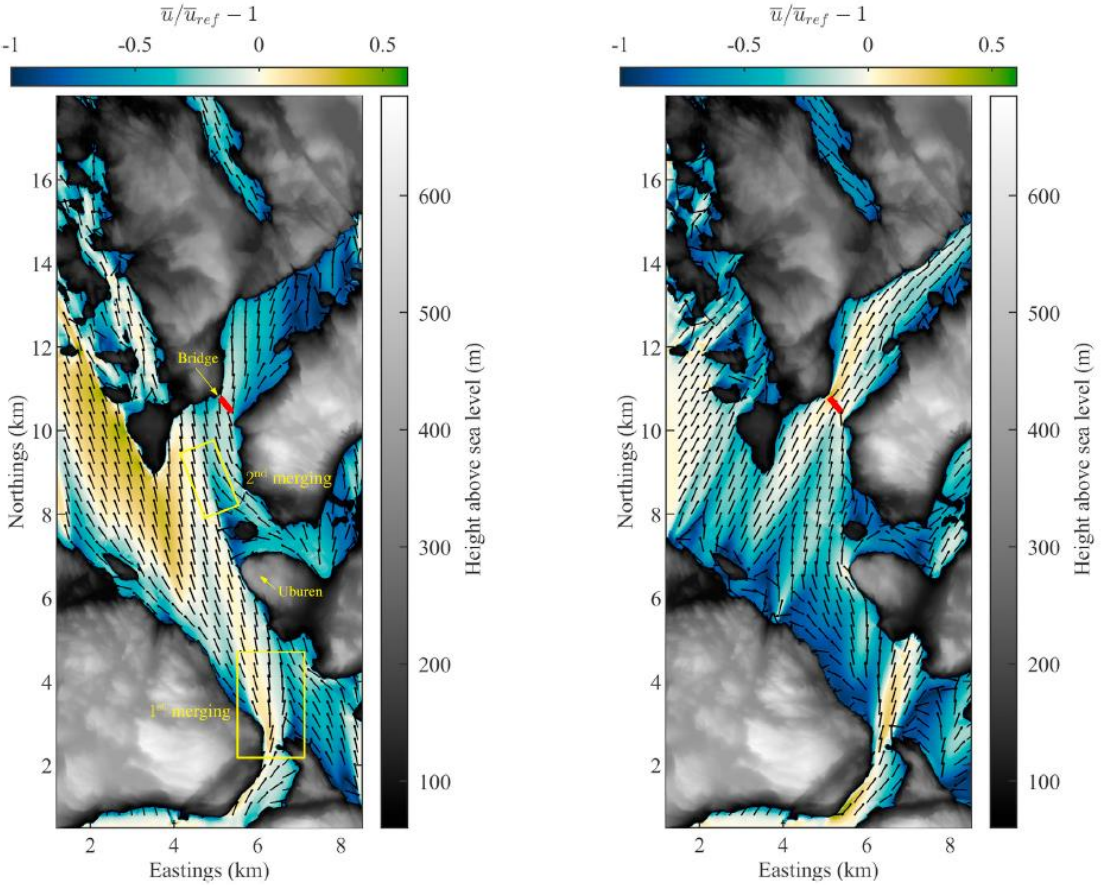


Figure 16: Normalized mean horizontal wind velocity 60m above the surface simulated with a direction of 168° (left) and 210° (right) at the inlet boundary [18, p. 10]

Figure 16 shows how a flow from south-easterly directions through the Høgsfjord can turn to a flow from SSW when entering the Lysefjord. It should be noted that a CFD simulation, such as any simulation or model, is only a model of reality that has its limitations and cannot replicate reality in all its complexity. However, they can still be useful.

"All models are wrong but some are useful" [19]

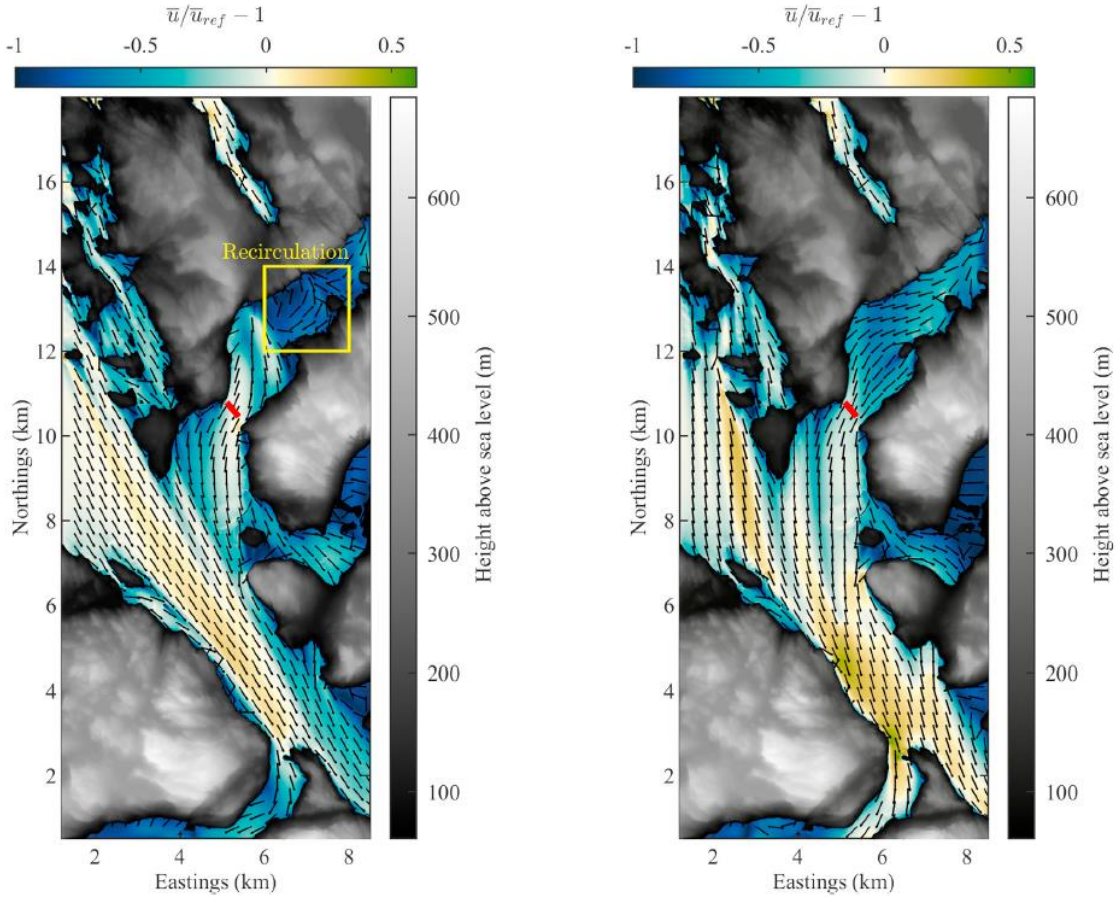


Figure 17: Normalized mean horizontal wind velocity 60m above the surface simulated with a direction of 335° (left) and 355° (right) at the inlet boundary [18, p. 12]

Figure 17 shows how small changes in the wind direction at the inlet boundary can create a recirculation zone in the Lysefjord north-east of the bridge. A detailed description of the simulation and the results can be found in [18].

1.1.2 Wind definitions

Figure 18 illustrates the wind definitions in the horizontal plane used in this work for SSW (left) and NNE wind conditions (right).

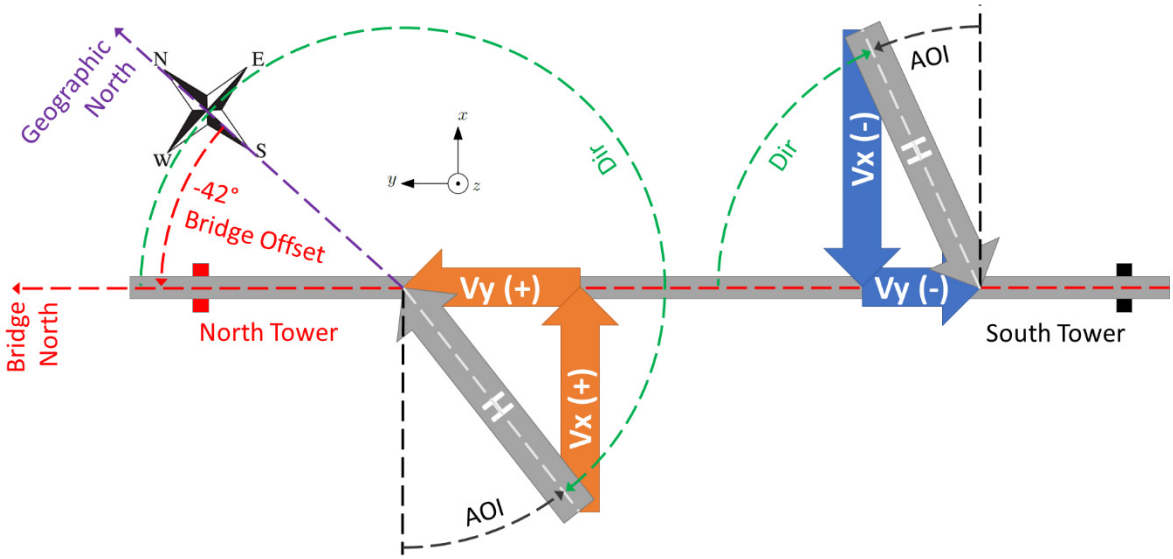


Figure 18: Horizontal wind definitions in this work for a SSW (left) and NNE wind condition (right)

The direction from the *south tower* (right) towards the *north tower* (left) is defined as *bridge north*. *Bridge north* is offset from *geographic north* by about -42°. The direction from which the horizontal wind component *H* is blowing from, measured from *bridge north*, is defined as *Dir* from 0° to 360° in clockwise direction. The lateral component of *H* along the x-axis of the bridge coordinate system is defined as *Vx*. The longitudinal component along the y-axis is defined as *Vy*. *Vx* and *Vy* are defined positive in the direction they are blowing to in reference to the bridge coordinate system. This means that a wind from SSW has positive *Vx* and *Vy* and wind from NNE has negative *Vx* and *Vy*.

Some wind engineering models assume a wind field that is coming in perpendicular to the bridge's primary axis. The deviation of the horizontal wind vector *H* from this perpendicular direction is defined as the angle of incidence (*AOI*), in wind engineering commonly referred to as the yaw angle. The *AOI* has no sign.

The vertical wind definitions used in this work for positive (bottom) and negative (top) angle of attack (*AOA*) are depicted in Figure 19.

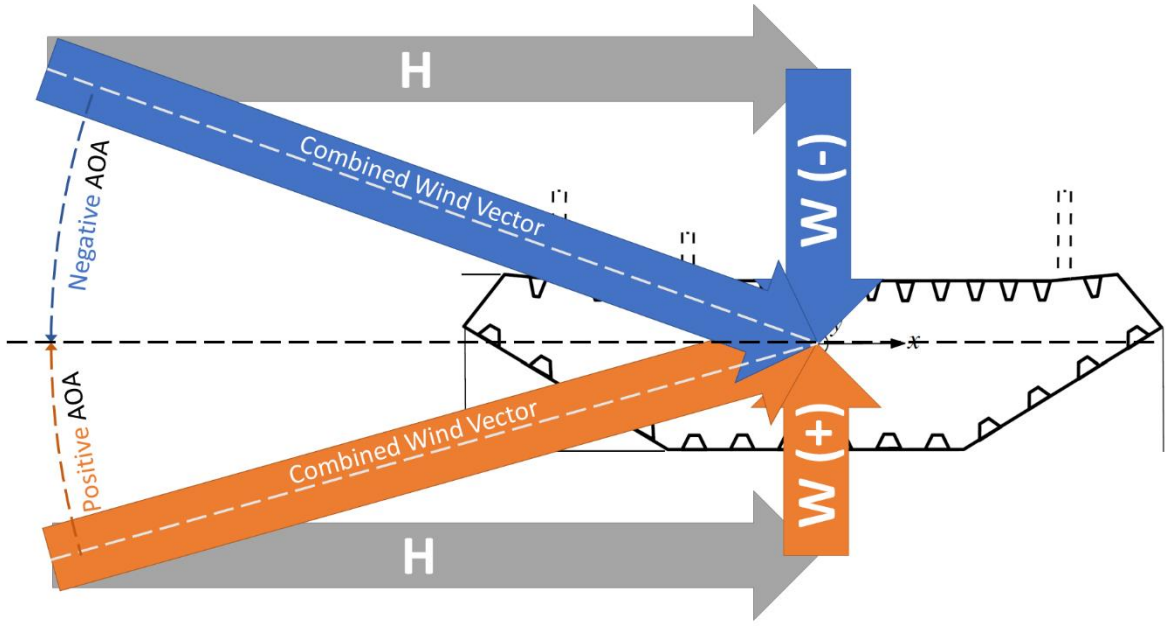


Figure 19: Vertical wind definitions in this work for a positive (bottom) and negative AOA (top). Adapted from [7, p. 11]

The wind vector is a combination of the horizontal component *H* and vertical component *W*. *W* is defined as positive in positive z-direction (bottom) and negative in negative z-direction (top). The angle between the horizontal x-y-plane of the bridge coordinate system and the combined wind vector is the angle of attack (*AOA*). The *AOA* is defined as positive for positive *W* and negative for negative *W*.

1.2 The data

A field measurement campaign on wind conditions and bridge response at the Lysefjord Bridge by the research group from University of Stavanger started in 2013. [3, p. 2]

The composition of the instrumentation on the bridge has developed over time. It primarily consists of sonic anemometers mounted above the bridge deck and pairs of accelerometers mounted inside the bridge girder.

1.2.1 Anemometers

The sonic anemometers used are 3D WindMaster Pro sonic anemometers from Gill Instrument Ltd. The anemometers can measure wind speeds of up to 65 m/s in all three directions, converted into horizontal and vertical wind components and wind direction. On hanger 10 the anemometer is part of the Vaisala weather transmitter WXT520 and measures horizontal wind speeds of up to 60 m/s and wind direction. [7, pp. 3-6], [20, pp. 35-38], [21, p. 1], [22, pp. 25-27,140]

Figure 20: Anemometer mounting positions on a hanger (left) and on a pole on top of the main cable (right) [7, p. 5]

The anemometers are usually mounted on the hangers, as depicted in Figure 20 (left). Where this method would not position them high enough above the deck, the anemometers are mounted on a pole above the main cable, as depicted in Figure 20 (right). Note that the anemometers are aligned with the bridge axis in a way that the reported wind direction from the North is wind blowing from the north tower parallel along the bridge towards the south tower. This direction is offset by -42° from geographic North due to the bridge's orientation.

1.2.2 Accelerometers

The accelerometers used are triaxial microelectromechanical silicon accelerometers with a range of ±5g by Canterbury Seismic Instruments Ltd. [23]

The accelerometers are mounted inside the bridge girder as depicted in Figure 21.

Figure 21: Accelerometer mounting position inside the bridge girder [7, p. 6]

Figure 22 shows the position of the accelerometers in the girder cross section.

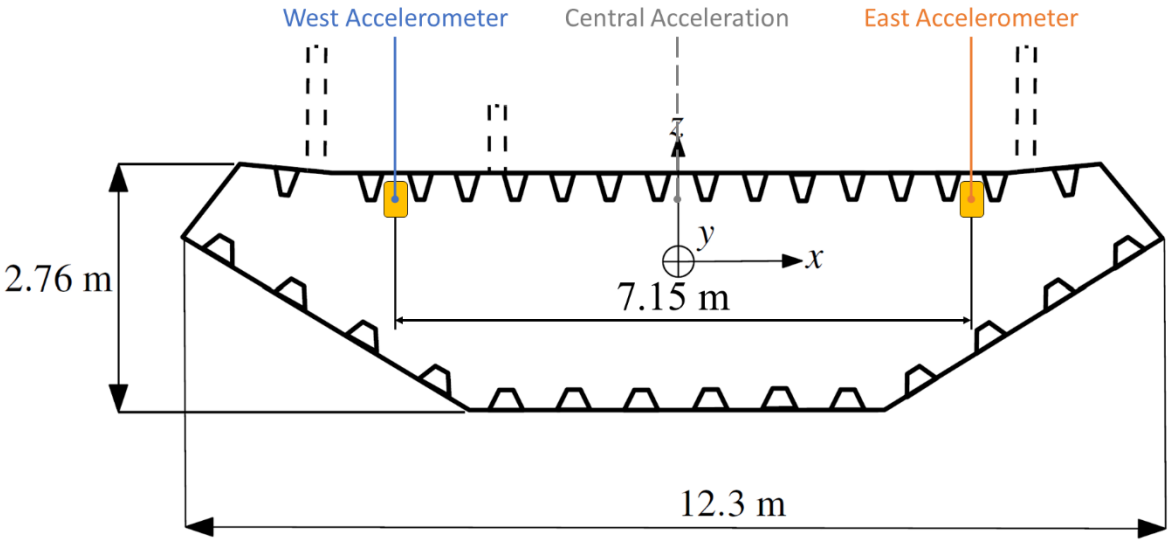


Figure 22: Girder cross section with accelerometers. Adapted from [7, p. 11]

The pairwise installation of the accelerometers allows the monitoring of torsional accelerations (theta) along the bridge's longitudinal axis (y-axis) in addition to the central translational accelerations in lateral, longitudinal and vertical direction (Aox, Aoy and Aoz). The lateral distance of the accelerometers is 7.15 m. [20, pp. 35-38]

1.2.3 Instrumentation Layout

The sensors' positions are denoted as shown in the naming convention depicted in Figure 23. Note that accelerometer pairs are denoted only by the hanger number.

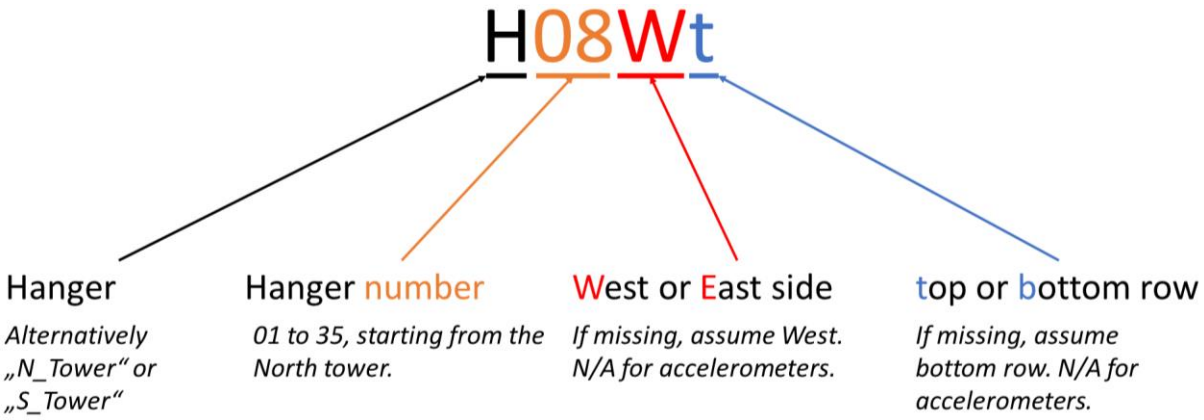


Figure 23: Sensor naming convention.

Bottom row anemometers are mounted 6 m above the bridge deck while top row anemometers are mounted 10 m above the bridge deck.

Figure 24 shows an overview of the instrumentation of the Lysefjord Bridge in July 2017.

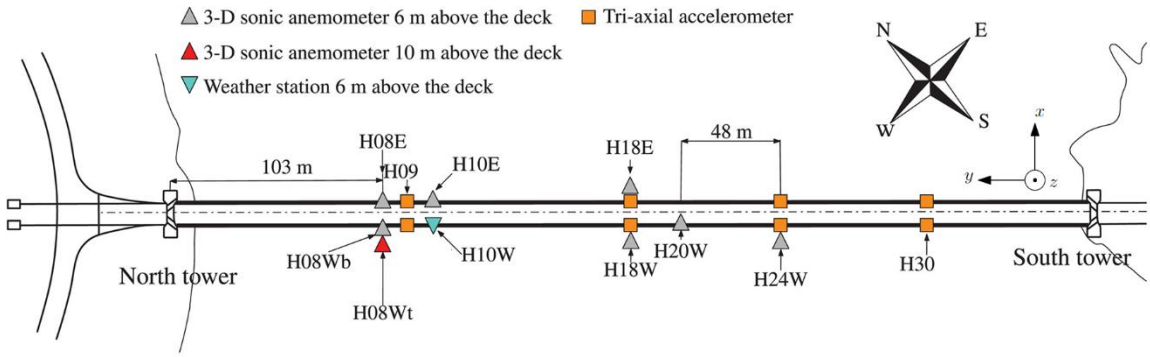


Figure 24: Overview of the instrumentation of the Lysefjord Bridge in July 2017. Adapted from [24, p. 3]

Initially six anemometers have been fitted to the west side of the bridge at hangers 08, 16, 18, 20 and 24, denoted H08Wb, H16W, H18W, H20W and H24W, with the weather station at H10W.

Four pairs of accelerometers have been installed at H09, H18, H24 and H30. Note that in the MATLAB files the position of the accelerometer pair at H24 is mistakenly denoted as H20. This mistake might have carried over in parts of this work during the import of the MATLAB files. Any mentioning of H20 in reference to an accelerometer is therefore to be interpreted as H24.

At hanger 08 an additional anemometer has been installed 10 m above the deck in June 2014, denoted H08Wt. [7, pp. 3-6], [20, pp. 35-38]

In 2014 an additional anemometer has been installed on top of the north tower, denoted N_Tower, which has been rendered un-operational in the same year.

In June 2017 additional anemometers have been installed on the east side of the bridge at H08E, H10E and H18E. [7, pp. 3-6]

Figure 25, Figure 26, Figure 27 and Figure 28 show the anemometers mounted to the bridge as of June 2022.

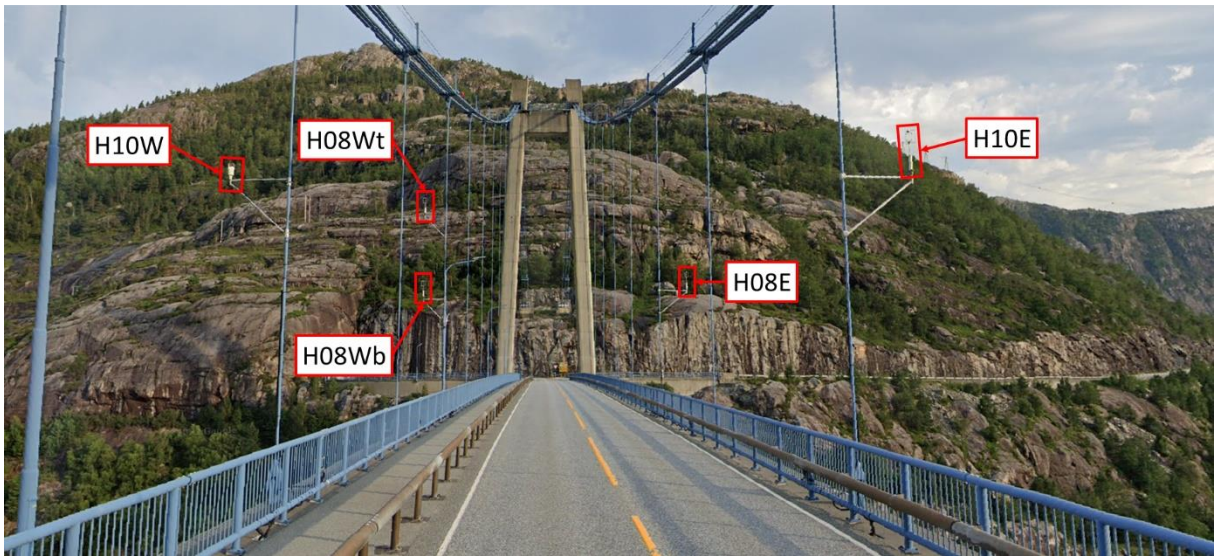


Figure 25: Location of anemometers H08Wb, H08Wt, H08E, H10W and H10E. Adapted from [4]

Figure 26: Location of anemometers H18W and H18W. Adapted from [25]

Figure 27: Location of anemometers H20W and H24W. Adapted from [26]

Figure 28: Location of anemometer H24W. Adapted from [27]

1.2.4 Data compilation

The sensors are connected to data acquisition units, which are connected to GPS receivers, providing an accurate timestamp to the data. The data, which is collected at a sample rate of 4 Hz, 32 Hz and 200 Hz by the weather station, anemometers and accelerometers respectively, is then resampled by a central data acquisition unit to 50 Hz. The central data acquisition unit outputs a single, synchronized, time-aligned dataset for every 10 minutes. This dataset is stored locally and simultaneously transferred to a server at University of Stavanger. [7, p. 7]

The 10-minute data packages of each day are compiled into a single MATLAB file. A MATLAB file for one day has an average file size of about 1.2 GB.

1.2.5 Data structure

Figure 29 illustrates the data structure of the compiled MATLAB files.

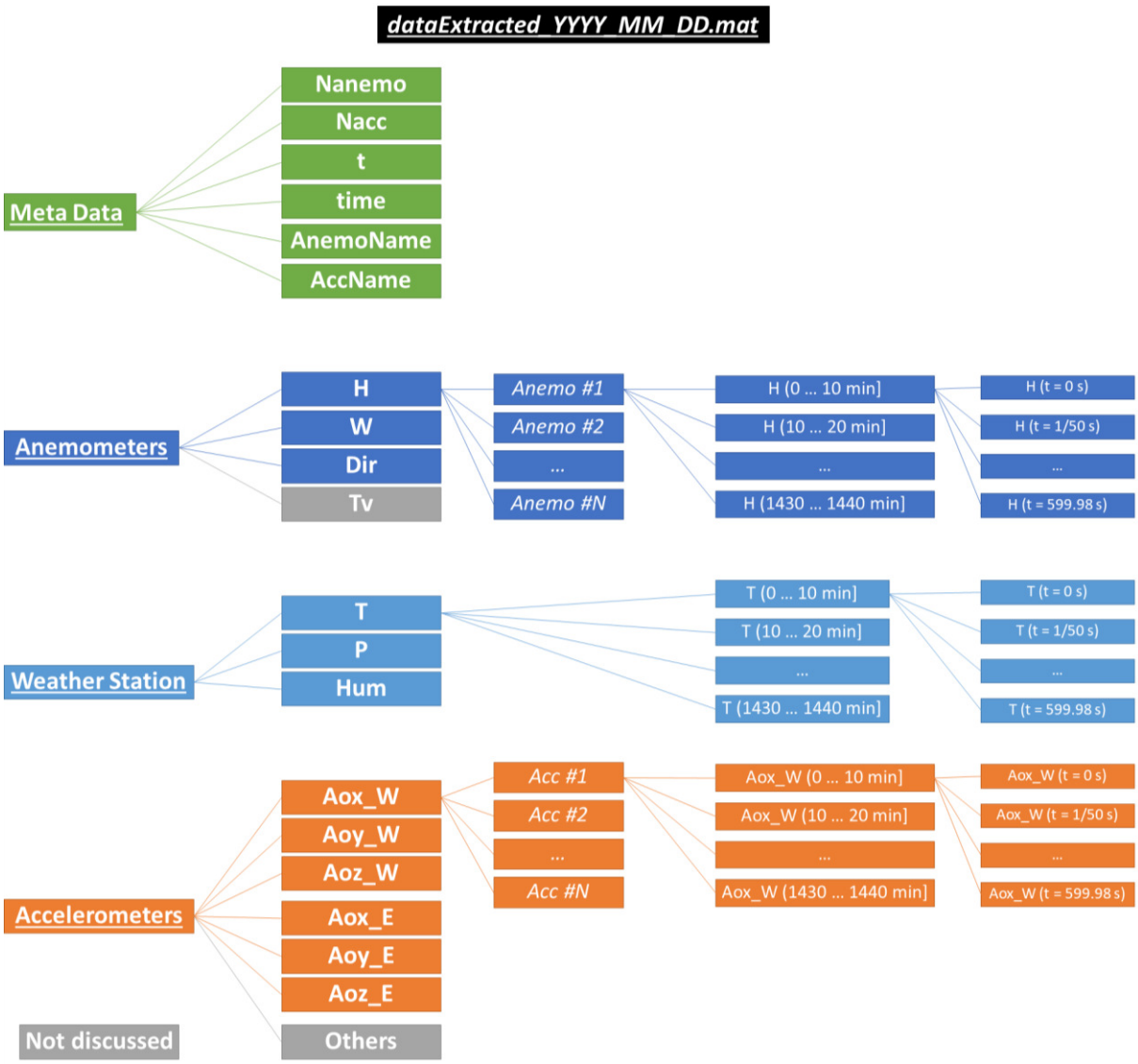


Figure 29: Data structure of MATLAB files.

The first level of the data structure can be grouped into meta data, data from anemometers, data from the weather station, data from accelerometers and data from other sources, which are not discussed in this thesis. The meta data includes information about the collected data such as timestamps as well as names of anemometers and accelerometers according to the naming convention illustrated in Figure 23. Measurements from anemometers include the horizontal wind component (H), the vertical wind component (W) and the direction from which the wind is blowing (Dir), as defined in *1.1.2 Wind definitions*. Note that all wind measurements are in

reference to the bridge coordinate system, as the sensors are mounted on the bridge. The bridge coordinate system can move translationally and rotationally relative to the world coordinate system. The measurements from the weather station include the temperature (T), pressure (P) and humidity (Hum). The measurements from each accelerometer pair are split into measurements from the west side accelerometer (_W) and east side accelerometer (_E) and into acceleration data in x-, y- and z-direction (Aox, Aoy, Aoz) respectively. For measurements from anemometers and accelerometers the measurements are stored in an array, or matrix, in which the first axis, or dimension, splits the data into the measurements from the different anemometers or accelrometers. The name of each anemometer or accelerometer can be obtained from *AnemoName* or *AnemoAcc* respectively using the index of this first axis. Note that this is not applicable to the weather station, as there is only one sensor of that type installed on the bridge. The second axis splits the data from that sensor into the 10-minute data packages. The final axis represents a 10-minute-long time-series of measurements from the respective sensor, sampled at 50 Hz.

1.3 Background information

Some background information from the wind engineering theory and previous works on the Lysefjord Bridge is presented below. This background information is referred to in *4 Analysis using this works code*.

1.3.1 Eigenfrequencies and eigenmodes

Eigenfrequencies are also known as the natural frequencies of a structure, at which it resonates with an external loading and, if undamped, starts to vibrate with steadily increasing amplitude until the deformation of the structure surpasses its stress limits leading to material failure. The shape of the deformation along a structure, at each eigenfrequency, is characterised in terms of the associated. A structure can have multiple symmetric and asymmetric eigenmode shapes. The true deformation might be a combination of multiple modes. [28], [29]

The eigenfrequencies and eigenmodes of the Lysefjord Bridge have been identified from the ambient vibration data and compared to those estimated using different computational models by E. Cheynet in [20, pp. 116-127]

The eigenfrequencies identified from the data are depicted in Table 1.

Table 1: Eigenfrequencies identified from the data by E. Cheynet. Adapted from [20, p. 118]

Mode	*HS1*	*HA1*	*HS2*	*HA2*	*HS3*	*HA3*	*VA1*	*VS1*	*VS2*	*VA2*	*VS3*	*VA3*	*TS1*	*TA1*
Hz	0.136	0.444	0.577	0.626	0.742	1.011	0.223	0.294	0.408	0.587	0.853	1.163	1.237	2.184

The first two symmetric and asymmetric eigenmode shapes for horizontal and vertical for horizontal and vertical vibrations, as well as the first symmetric and asymmetric eigenmode shape for torsional vibrations are depicted in Figure 30.

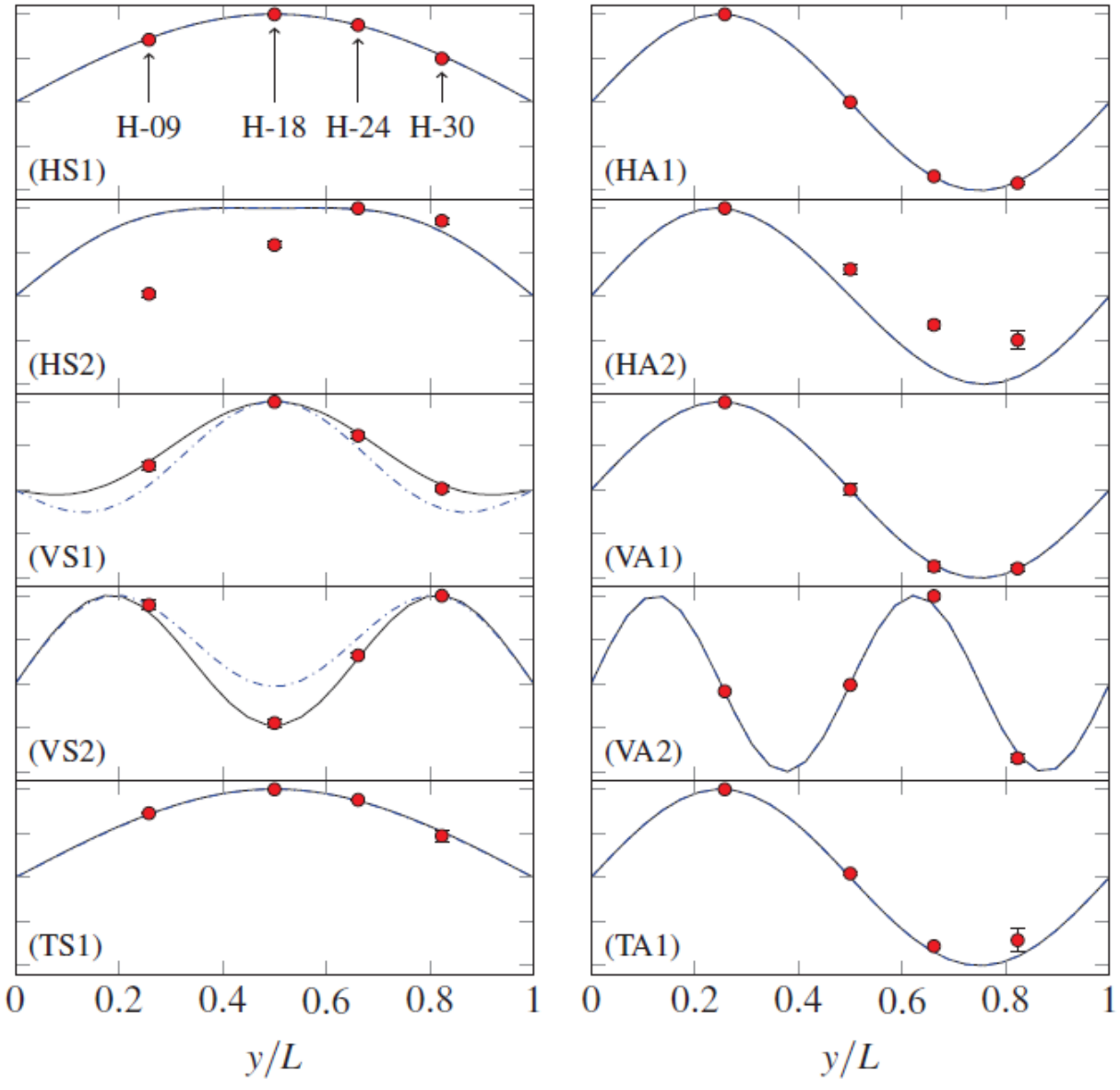


Figure 30: Identified eigenmode shapes from data (red dots) and models (lines). Adapted from [20, p. 126]

1.3.2 Wind load coefficients

The wind load coefficients for the drag (C_D), lift (C_L) and moment (C_M) of the girder cross section have been determined at different AOAs in wind tunnel testing.

The load coefficients are calculated from the respective measured forces F_D, F_L and F_M on the bridge girder cross section using the equations below:

$$C_D = \frac{F_D}{\frac{1}{2}\rho U^2 h}$$

$$C_L = \frac{F_L}{\frac{1}{2}\rho U^2 b}$$

$$C_M = \frac{F_M}{\frac{1}{2}\rho U^2 b^2}$$

in which ρ is the air density, U is the mean wind speed, h is the height of the bridge girder and b is the width of the bridge girder. [30, pp. 4-10]

Figure 31 displays the results from the wind tunnel testing as the measured datapoints for the drag coefficient (blue circles), lift coefficient (orange stars) and moment coefficient (yellow square).

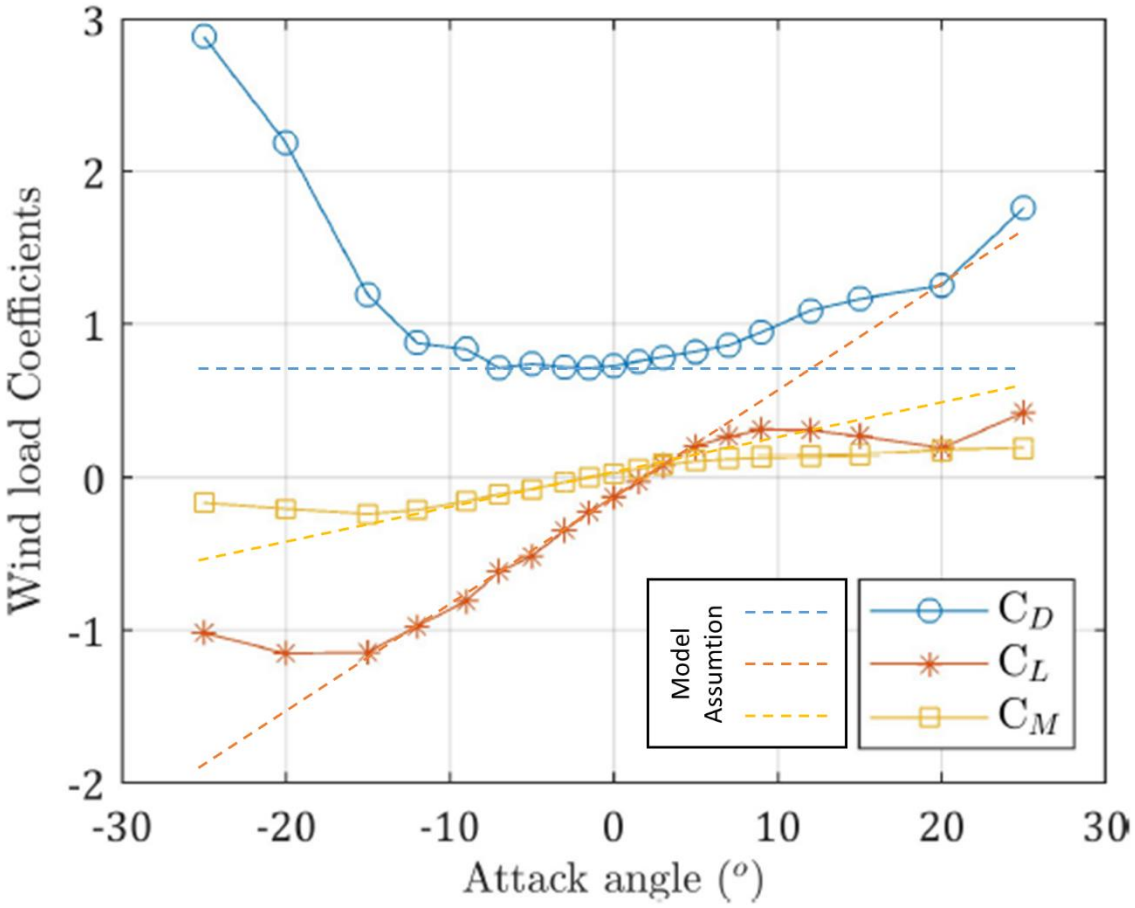


Figure 31: Wind load coefficients of the Lysefjord Bridge girder. Adapted from [30, p. 11]

In the so-called quasi-steady theory of wind loading and the corresponding numerical simulations a linear relation between AOA and the wind load coefficients, as illustrated in Figure 31, is assumed. The model assumption is illustrated by dashed lines, which are superimposed tangents at 0° AOA for the moment coefficient and lift coefficient and a horizontal line for the drag coefficient. Note that these are for illustrative purposes, the actual models might differ. While this linearity assumption is accurate enough at small AOAs, close to 0°, it leads to over- or underestimations in the wind response at higher positive or negative AOAs. From Figure 31 we see that the linearity assumption is accurate enough from about -15° to +5° for the lift coefficient, from about -12° to +5° for the moment coefficient and from about -8° to 0° for the drag coefficient.

1.3.3 Vortex shedding

Vortex shedding is a phenomenon in which the airflow detaches from both sides of a structure, with oppositely rotating vortices alternatingly shed from each side of the body, creating what is known as a Kármán vortex street, as depicted in Figure 32.

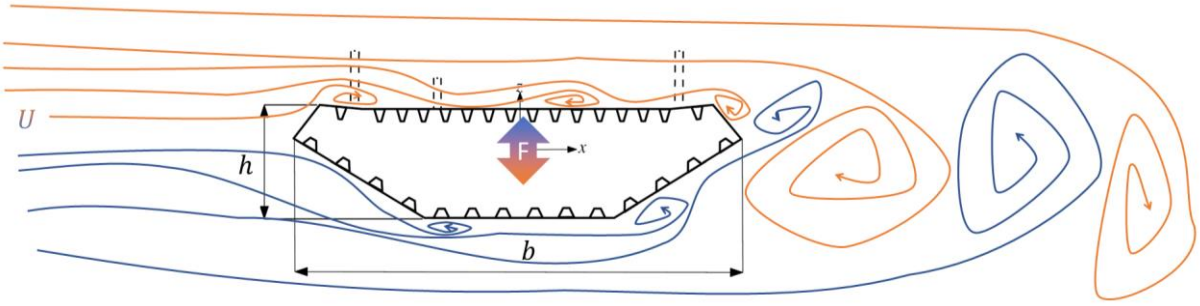


Figure 32: Illustration of vortex shedding at the bridge girder. Adapted from [7, p. 11]

These vortices create a harmonically varying force on the structure, primarily perpendicular to the wind direction with a distinct frequency of the vortex shedding. The shedding frequency f_S is

$$f_S = S_t \frac{U}{h}$$

Where U is the mean wind speed, h is the height of the bridge girder and S_t is the Strouhal number, which depends on the geometry of the structure and the Reynolds number. [2, pp. 59-60], [31, p. 5]

The Reynolds number, which captures the ratio between the inertia and the viscosity forces in the flow, has a smaller effect on the vortex shedding process for bodies with sharp edges, such as the girder of the Lysefjord Bridge. At a girder width to height ratio close to 4.5, which is the case at the Lysefjord Bridge, the Strouhal number can be assumed as 0.11. [2, p. 60], [32]

If the shedding frequency is close to one of the eigenfrequencies of the structure f_e this creates resonance, activating an eigenmode of the structure. The resonance wind velocity U_r is therefore

$$U_r = f_e \frac{h}{S_t}$$

[31, pp. 3, 5]

J. Tveiten has calculated critical wind velocities for vortex induced vibrations at 5.4 m/s, 7.6 m/s and 10.2 m/s, depending on the eigenmode, in [2, p. 61]

Using the eigenfrequencies for vertical modes identified by E. Cheynet from Table 1 and the equation from above we can calculate the respective resonance wind velocities depicted in Table 2.

Table 2: Critical resonance wind velocities for vortex shedding. Adapted from [20, p. 118]

Mode	*VA1*	*VS1*	*VS2*	*VA2*	*VS3*	*VA3*
Eigenfrequency [Hz]	0.223	0.294	0.408	0.587	0.853	1.163
Resonance wind velocity [m/s]	5.6	7.38	10.24	14.73	21.4	29.18

1.4 Motivation

The continues measurements since 2013 are now, almost 10 years later, equivalent to multiple terabytes of data. The data from the daily MATLAB files can be analysed directly in MATLAB, which has successfully been demonstrated in previous works in the department of Department of Mechanical and Structural Engineering and Materials Science at UiS in [20], [7] and [3]. However, the open-source programming language Python has become one of the most popular programming languages in recent years, as depicted in Figure 33.

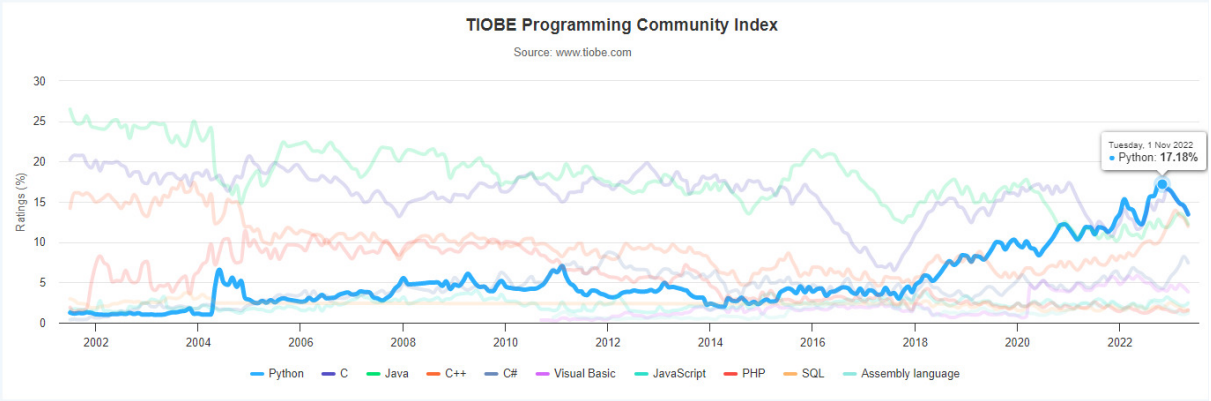


Figure 33: Popularity of different programming languages [33]

The open-source nature of Python and easy syntax provides opportunities to examine the data from a different perspective.

Research questions are related to the complex wind flow in the terrain surrounding the Lysefjord Bridge and the response of the bridge to the wind. Analysing datasets of multiple days in a consistent manner could help to answer these questions.

2 Method and design

The first goal of this work is to create a tool for detailed, consistent data analysis of multiple days of data from the Lysefjord Bridge using Python. The second goal is to demonstrate the code by using it to perform analysis on a dataset of a selected period, focusing on the local wind conditions at the Lysefjord Bridge and the bridge's response to different wind conditions.

2.1 Programming environment

Python has been chosen as the programming language for this work for its great readability and easy syntax allowing fast development of the code as well as open-source nature and extendibility, which are just a few reasons for its recent popularity. [34]

The code is written in a Jupyter Notebook, which allows to split the code into multiple code cells, have the output appear in an output cell right after each code cell and add markdown cells, as depicted in Figure 34.

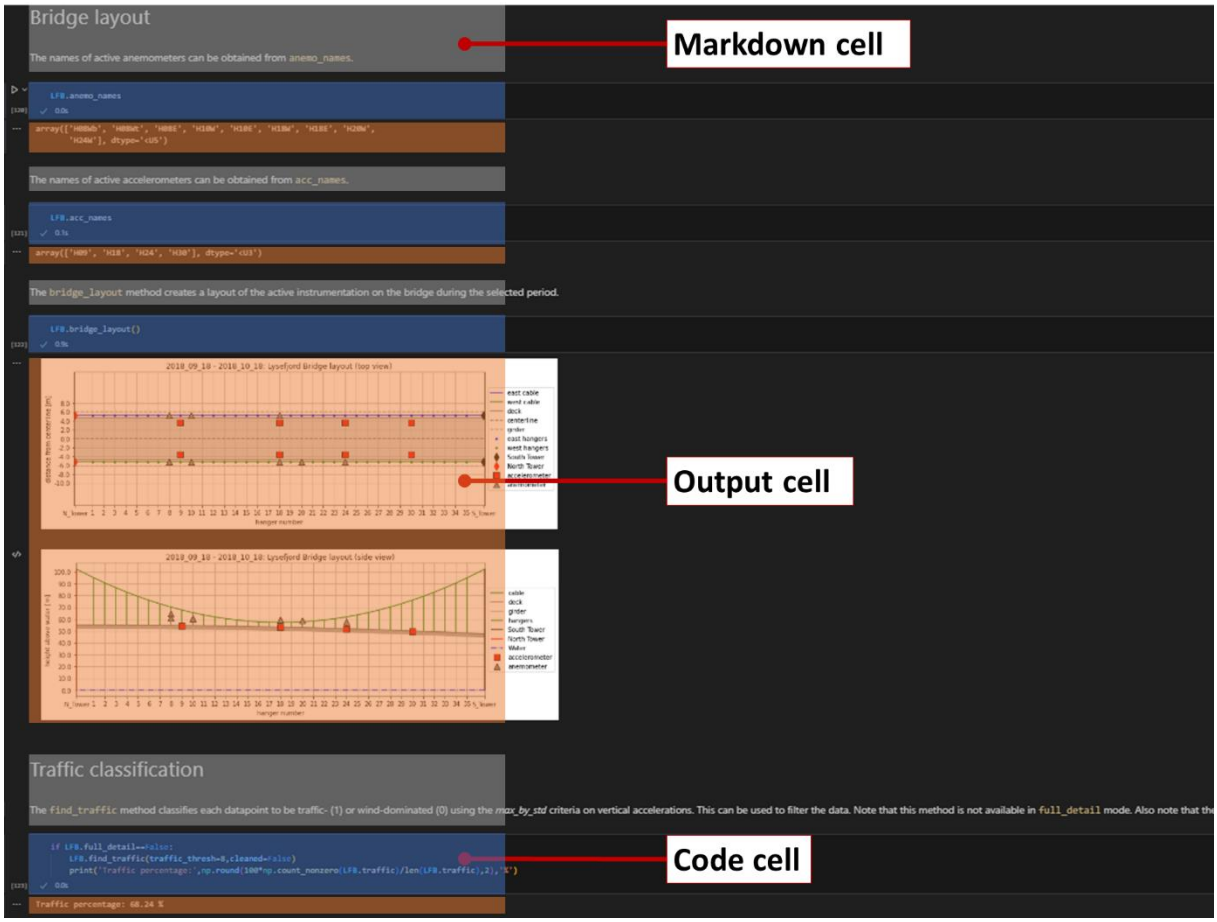


Figure 34: Structure of a Jupyter Notebook in VSCode with markdown-, code- and output-cells

The markdown cells can be used as explanatory text with formatting and as headings with different levels to group code cells together and give the notebook a structure. [35]

The integrated development environment (IDE) used is Visual Studio Code (VSCode).

2.2 Programming paradigms: Functional and object-oriented programming

A combination of functional and class-based object-oriented programming is used.

> "Functional programming decomposes a problem into a set of functions." *[36]*

Functions are defined to solve a type of problem, taking inputs, processing them and returning outputs. They can later be called to solve a specific problem, by giving them specific inputs so they return specific outputs.

> *"Object-oriented programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way." [36]*

An object can be an instance of a class, giving it the internal structure of that class.

> "Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state." *[37]*

A function inside a class is called a method of that class. Two different instances of a class have the same internal structure and methods of that class available but can have a different internal state.

In this work a class is defined to handle data from the Lysefjord Bridge of multiple dates. The functionality is contained in the methods of that class. An instance of that class is created with data from a given set of dates. The methods of the class are used to manipulate the internal state of that class instance, processing the data. This ensures that two different instances of the class, handling two different periods of data, have the same internal structure and the data can be processed in a consistent manner.

2.3 Data structures: NumPy array VS pandas dataframe

There is a powerful dedicated data analysis library for Python called pandas, which stores data in dataseries and dataframes. While pandas has a lot of inbuilt functionality for data analysis, it was deliberately decided to not use this library for this project. This is because this added functionality, such as explicit indexing, considerably slows down the data processing, compared to implicitly indexed NumPy arrays. It should be noted that this increased performance for NumPy arrays only holds up to datasets with about 50k columns, after which pandas performs better for some operations. On datasets of more than about 500k columns pandas outperforms Python on most operations. However, the bridge data in this work is by default down-sampled to 144 samples per day by taking statistical metrics of each 10-minute data package. Therefore, analysing up to 1 year of data, equating roughly 50k rows, still benefits from the higher performance of NumPy arrays. Additionally, NumPy arrays consume less than half the memory of an equally sized pandas dataframe. [38], [39], [40]

This however means that special care had to be taken to keep the data in the different arrays aligned.

2.4 Coding style and documentation

The coding style and the documentation of the code is based on the Python Enhancement Proposal (PEP) “PEP 8 – Style Guide for Python Code” [41]. The docstrings for the class and its methods are based on “PEP 257 – Docstring Conventions” [42], the NumPy style guide [43] and the pandas docstring guide [44]. Some of the potential invalid inputs are directly handled by raising exceptions, informing the user of the corrective measures necessary.

2.5 Main class structure

In this work a class called *BridgeData*, containing most of the functionality in its methods, is defined. In the initialisation of an instance of that class parameters for the data, the design of the bridge and sensor ranges are defined. While the code is specifically tailored to the Lysefjord Bridge, this design allows it to be adapted to a different bridge where similar data might be collected.

The class is engineered in a modular design to allow for expendability. The methods for processing the data can be split into methods for pre-processing, processing and post-processing, as depicted in Figure 35.

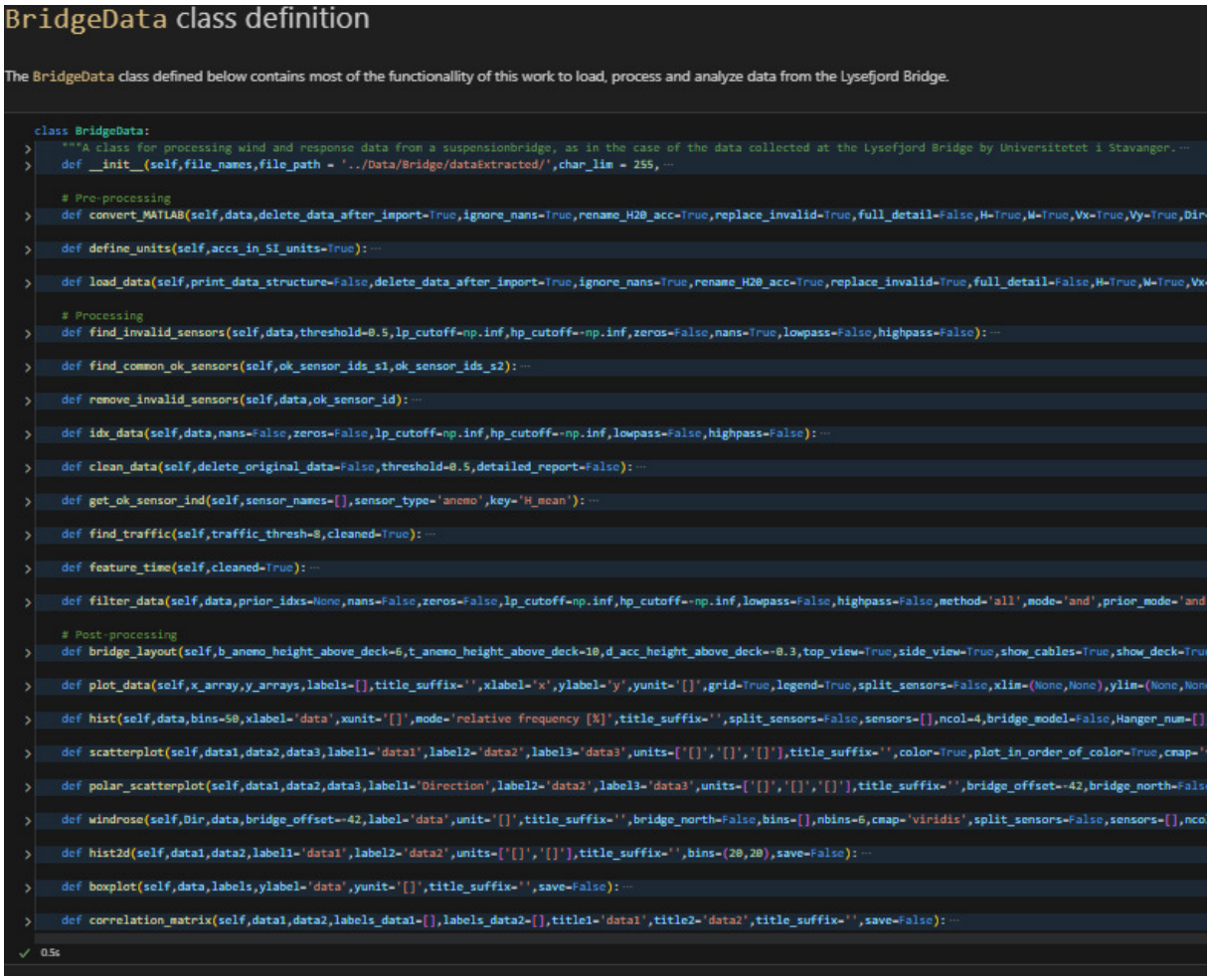


Figure 35: BridgeData class overview showing the different methods defined in the class.

The class-methods can be divided into secondary helper-methods and primary methods. Primary methods can either stand on their own or act as a wrapper-method that makes use of one or more helper-methods. Primary methods are designed to be called directly by the user. Secondary methods can technically also be called by the

user but might be less relevant. The methods are designed to be flexible, giving the user options in the analysis, while also providing a suitable default configuration.

In the following some of the methods for pre-processing, processing and post-processing are roughly described to illustrate the class structure and general workflow. Further details and explanations can be found in section *3 Implementation*, in the code documentation, the docstrings for each method and the code itself, which can be found in the appendix.

2.5.1 Pre-processing

The pre-processing consists of the helper-methods *convert_MATLAB* and *define_units* as well as the wrapper-method *load_data*.

convert_MATLAB converts MATLAB data of a single day into NumPy arrays and dictionaries of NumPy arrays. By default, the data is down-sampled to 144 samples per day by taking the mean (*_mean*), standard deviation (*_std*), maximum (*_max*) and minimum (*_min*) of each 10-minute data package. Readings that are outside the sensors' ranges are by default replaced by Not a Number (*NaN*) and ignored in the down-sampling. Additional features such as *AOA*, *AOI*, turbulence intensity (*_turb*) as well as central acceleration (*_C_*) and torsional acceleration (*theta_*) are engineered from the measurements.

define_units defines the units of each measurement and by default converts the accelerometric data from μG to $\frac{m}{s^2}$.

load_data loads and combines data from one or multiple days using the previous helper-methods. This is the primary function that gets called by the user to pre-process data from a specific set of dates.

2.5.2 Processing

The processing consists of the helper-methods *find_invalid_sensors*, *remove_invalid_sensors*, *idx_data* and the wrapper-method *clean_data*. These methods are used for data cleaning. The data cleaning process can be divided into two steps. Firstly, sensors that have by default more than 50% invalid data are marked as invalid and removed from the cleaned dataset using *find_invalid_sensors* and *remove_invalid_sensors*. This threshold can be adjusted by the user. Invalid data can be NaNs, zeros or values above or below a certain threshold, depending on

the measurement type. Secondly, invalid data from the remaining sensors is indexed using *idx_data* and removed from the cleaned dataset. Note that if there is invalid data for a single measurement type from a single sensor, all readings from all other sensors are also marked as invalid and removed to keep the data aligned. Additional methods that are useful for the preparation of visualisations are *get_ok_sensor_ind*, *find_common_ok_sensors, find_traffic*, *feature_time* and *filter_data*.

get_ok_sensor_ind and *find_common_ok_sensors* are useful methods to select the right sensors for visualisations. *feature_time* creates time arrays in days, hours, minutes and seconds, starting at 0, for plotting of time-series data. *filter_data* allows sophisticated filtering of the data returning indices of the filtered data. It also allows combining the filter with indices of previously filtered data to apply multiple filters to the data at once.

For some analysis it is important to be able to separate wind induced vibrations of the bridge from traffic induced vibrations. Sophisticated methods for the identification of traffic induced vibrations are discussed in [24].

For this work a simple approach has been chosen, where the maximum vertical acceleration of each 10-minute data package is divided by the standard deviation of the vertical acceleration, resulting in the new measurement statistic *_max_by_std*. The vertical acceleration and wind speed data have been studied with *full_detail* set to *True*, as depicted in Figure 36 - Figure 43.

Figure 36 shows the full detail horizontal wind speed data for one day of data.

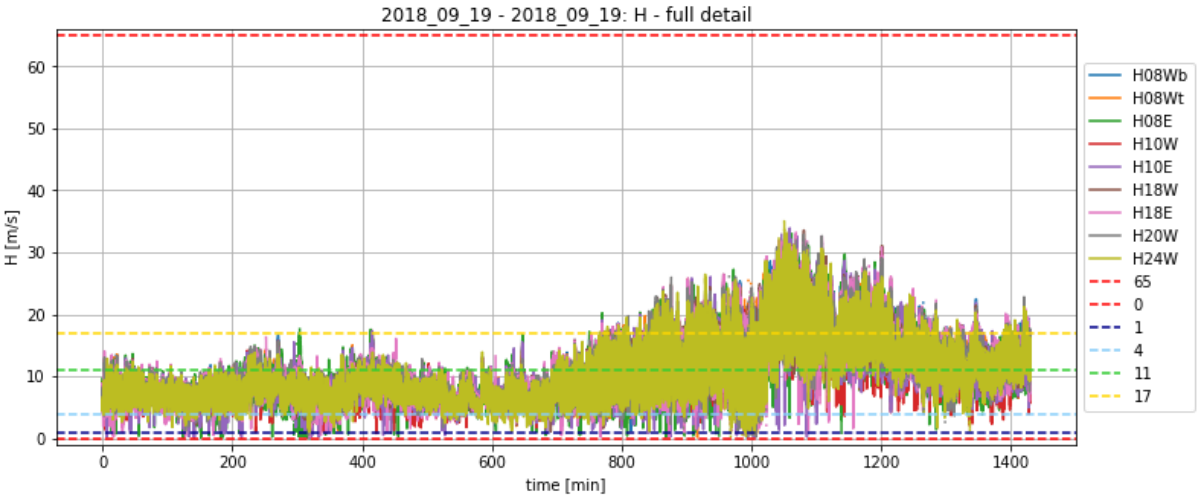


Figure 36: One day of full detail horizontal wind speed data

The data shows a horizontal wind speed mostly below 11 m/s in the first half of the day and reaching wind speeds above 17 m/s after about 13 hours or 780 minutes and mostly subsiding below 17 m/s at about 22 hours or 1320 minutes.

Figure 37 shows the central vertical acceleration data between minutes 600 and 610. Note that the naming error for accelerometer pair H24 mentioned in *1.2.3 Instrumentation Layout* might be carried over in some of the illustrations below. Any mentioning of H20 in reference to an accelerometer is to be interpreted as H24.

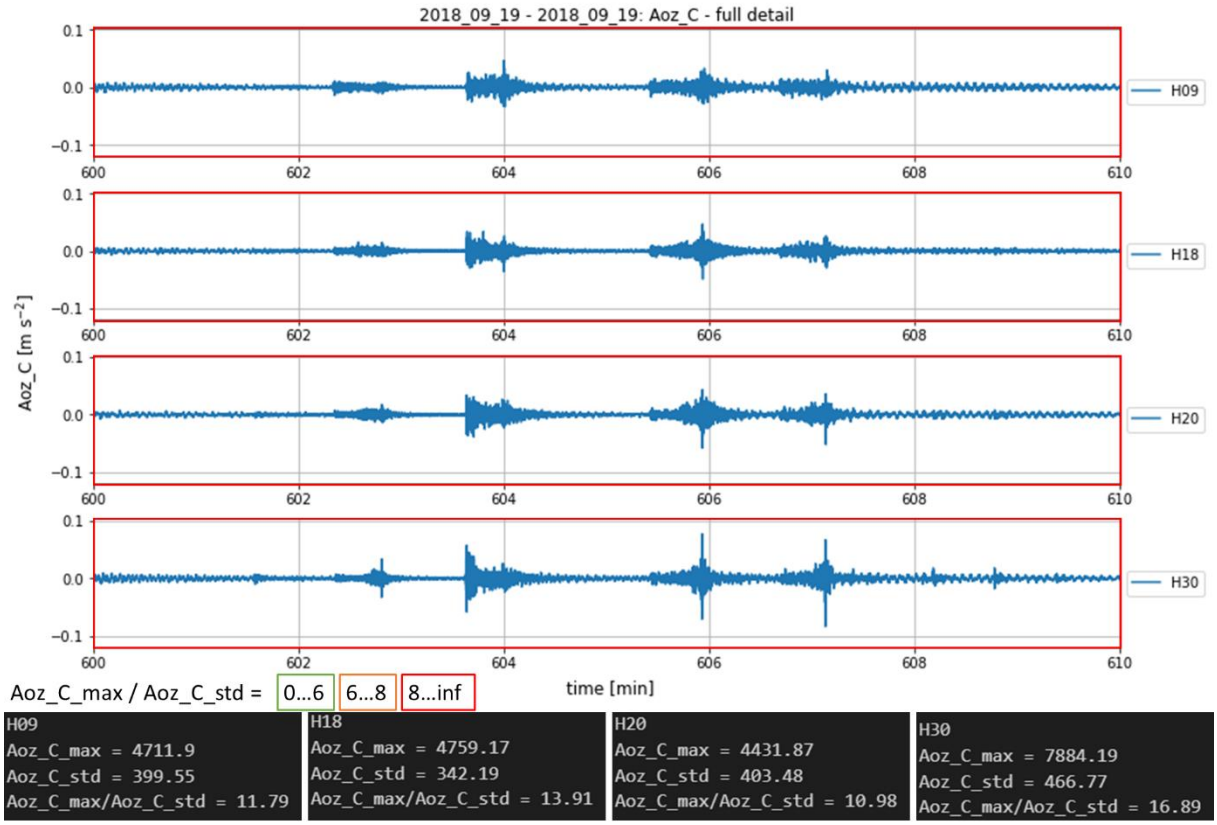


Figure 37: Full detail max_by_std criteria study (600min - 610min)

The respective calculation of maximum, standard deviation and the *max_by_std* criteria for each sensor in the given time period are below the plots and marked on each sensors plot according to the color code. The acceleration data shows four sharp patterns of clearly traffic induced vibrations at about 602.5 minutes, 603.5 minutes, 605.5 minutes and 607 minutes across all sensors. The *max_by_std* criteria is well above 8 across all sensors.

Figure 38 showcases that windspeeds were relatively low in the relevant period.

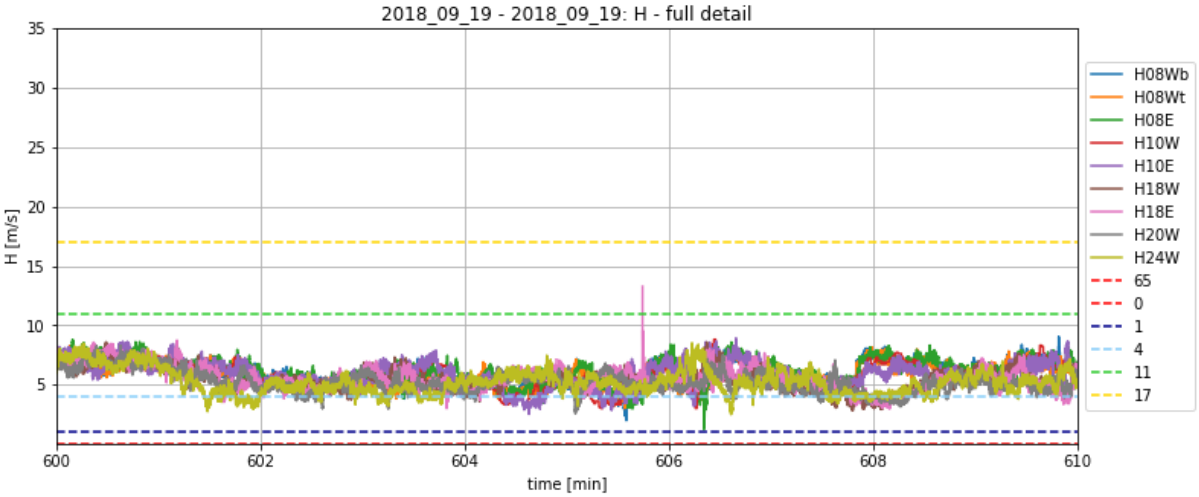


Figure 38: Full detail horizontal wind speed (600min - 610min)

Figure 39 shows the acceleration data between minutes 850 and 860.

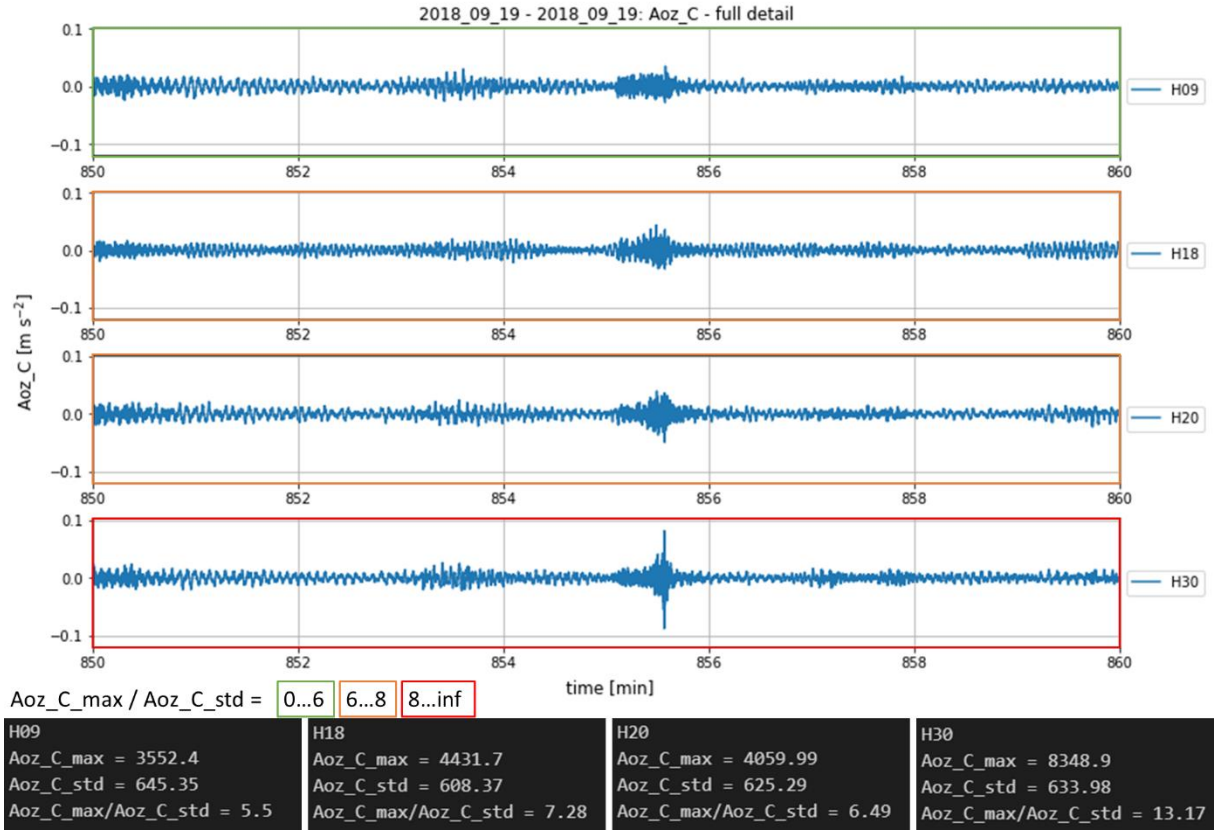


Figure 39: Full detail max_by_std criteria study (850min - 860min)

The acceleration data shows vibrations with higher amplitudes throughout the period, compared to the previous period, and a potential traffic induced vibration at about 855.5 minutes. The *max_by_std* criteria is below 6 at H09, between 6 and 8 at H18 and H24 and well above 8 at H30.

Figure 40 showcases that windspeeds are between 11 m/s and 17 m/s throughout most of the period and even peaking above 17 m/s around the time of the potential traffic induced vibration.

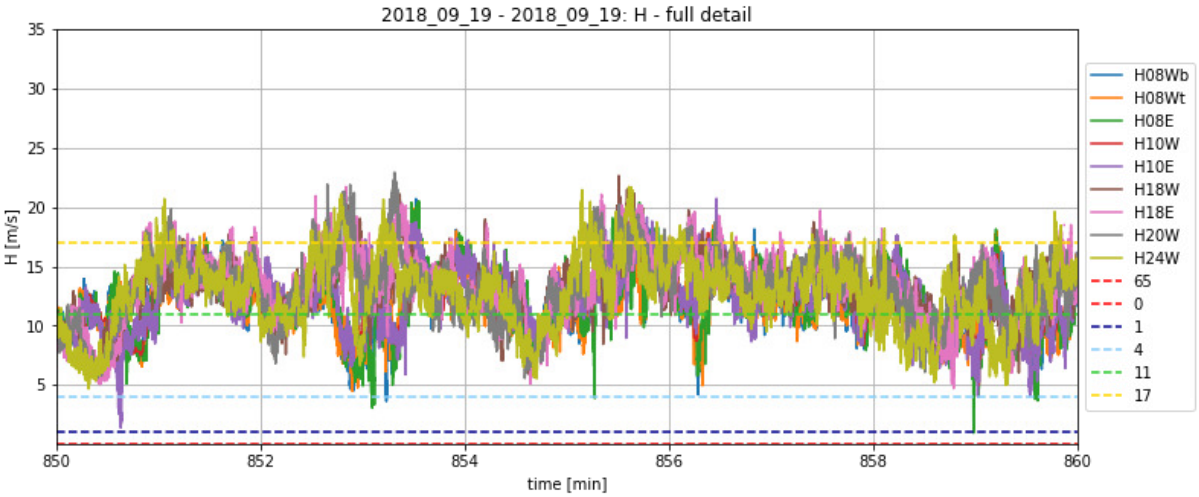


Figure 40: Full detail horizontal wind speed (850min - 860min)

This represents an edge case, where it is unclear if the vibrations in the period are traffic dominated or wind dominated.

Figure 41 shows the acceleration data between minutes 1050 and 1060.

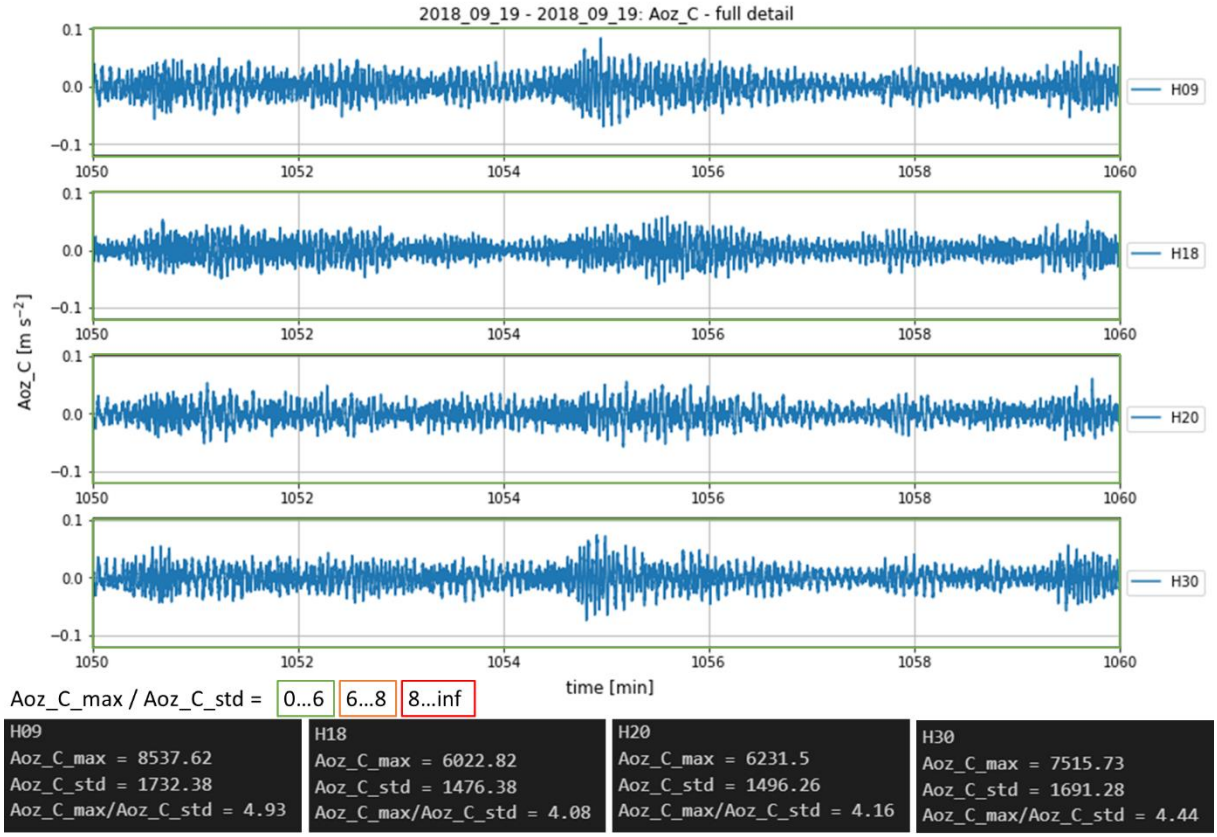


Figure 41: Full detail max_by_std criteria study (1050min - 1060min)

The acceleration data shows vibrations with even higher amplitudes throughout this period, compared to the previous periods. There is no clear sign of traffic induced vibrations. The max_by_std criteria is well below 6 across all sensors.

Figure 42 showcases that the wind speed is well above 17 m/s throughout most of the period.

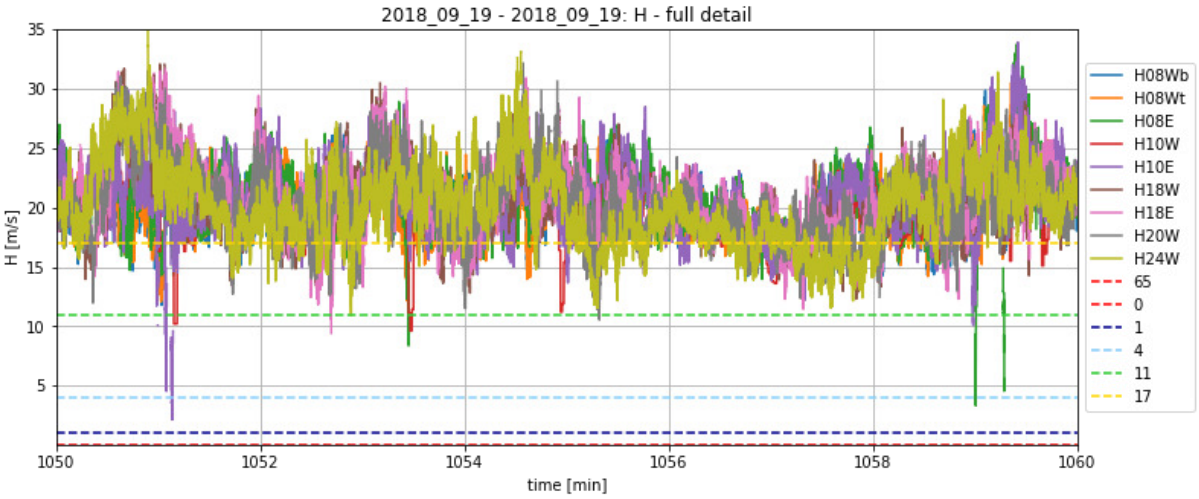


Figure 42: Full detail horizontal wind speed (1050min - 1060min)

This period is clearly a case of wind dominated vibrations.

Random 10-minute periods have been examined throughout the day in a similar way. A compilation of the results is showcased in Figure 43.

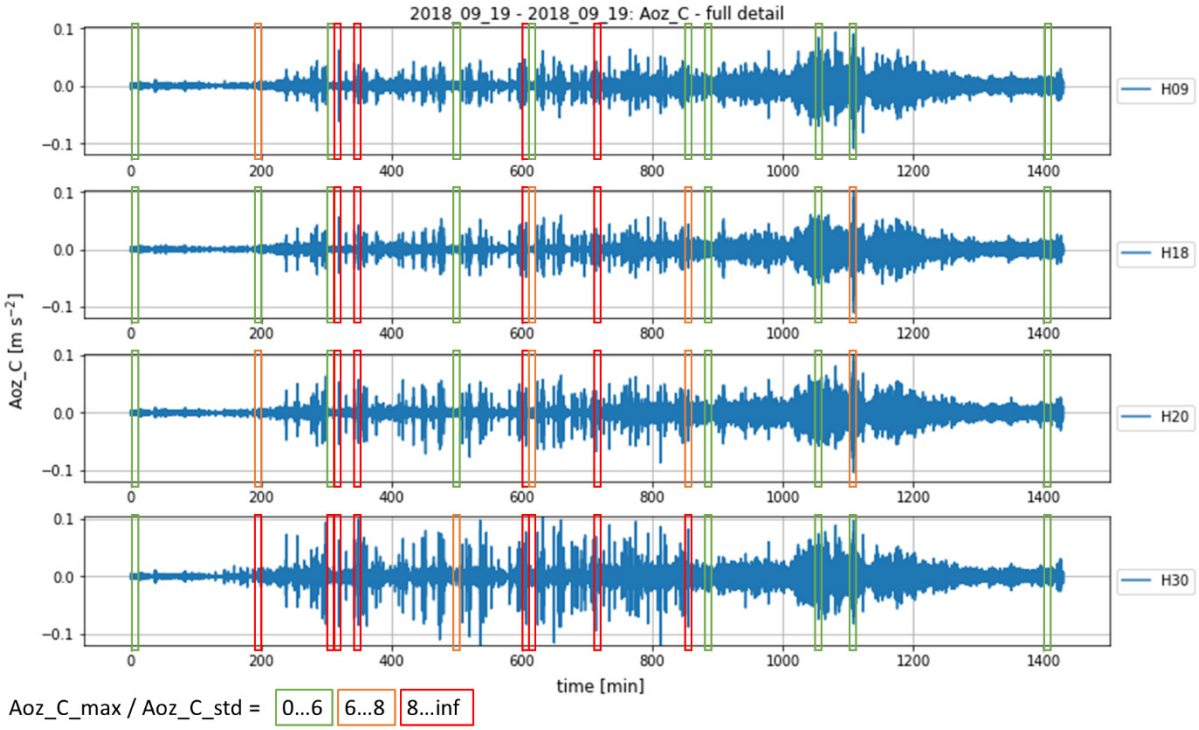


Figure 43: Full detail studies of central vertical acceleration with random samples of the max_by_std criteria

The sampled 10-minute periods are color-coded according to the respective result for the *max_by_std* criteria. These studies lead to the conclusion, that wind dominated

bridge responses have a vertical *max_by_std* value below 6 and traffic dominated bridge responses have a vertical *max_by_std* value above 8. *max_by_std* values between 6 and 8 are inconclusive edge cases, where the period can not be clearly classified as traffic dominated or wind dominated. The method *find_traffic* therefore uses a default lower threshold of 8 for the *max_by_std* criteria to find traffic dominated periods in the data, using the *idx_data* method. This threshold can be adjusted by the user.

2.5.3 Post-processing

bridge_layout creates an overview of the bridge illustrating the instrumentation layout for the chosen dates, as depicted in Figure 44 and Figure 45.

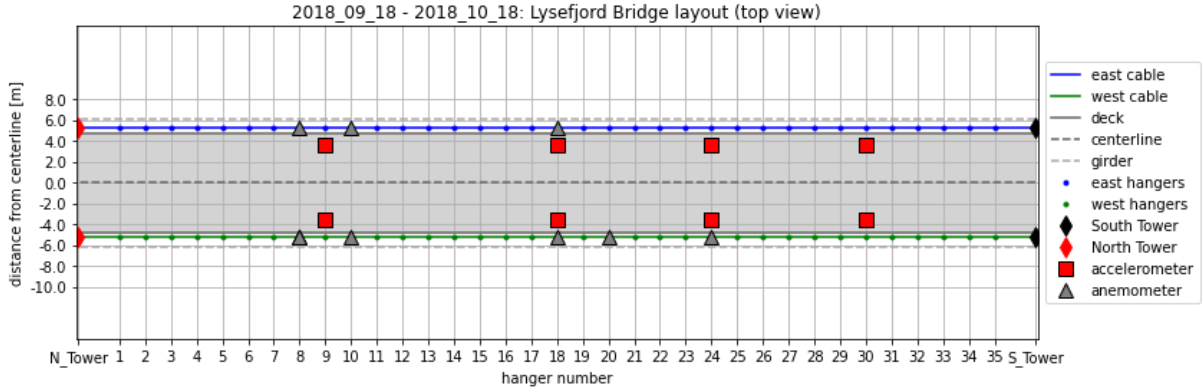


Figure 44: Lysefjord Bridge instrumentation layout from bridge_layout (top view)

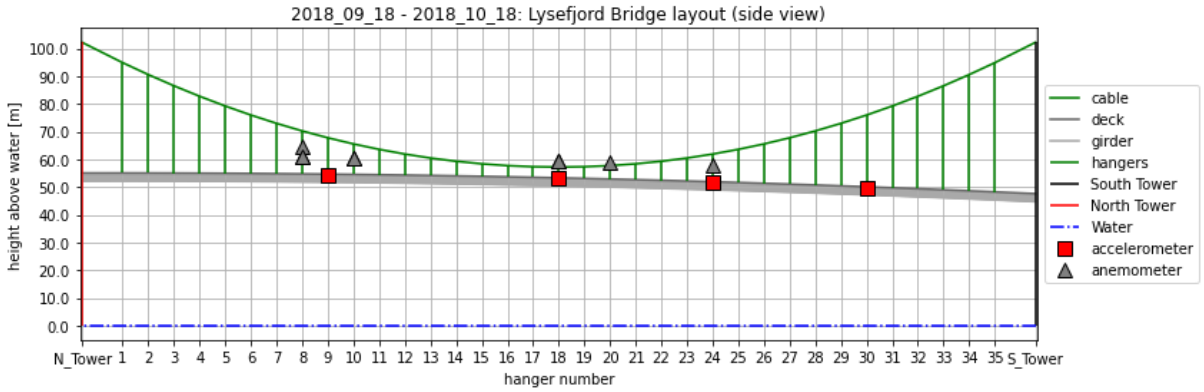


Figure 45: Lysefjord Bridge instrumentation layout from bridge_layout (side view)

Note that these layouts are for illustrative purposes only and not to be used as an engineering model.

The methods *plot_data*, *hist*, *scatterplot*, *polar_scatterplot*, *windrose*, *hist2d*, *boxplot* and *correlation_matrix* are designed similarly for a consistent visualisation of data from the Lysefjord Bridge. They take raw or processed data and create time-series

plots, histograms, scatterplots, polar scatterplots, wind roses, 2-dimensional histograms, boxplots or a heatmap of a correlation matrix respectively. In most of the methods it is possible to either display data for a measurement type combined from all sensors, or split it up, creating a visualisation per sensor. This allows comparisons of behaviours along the bridge span and on the upwind versus the downwind side of the bridge.

3 Implementation

The following Python libraries are used to implement the functionality of the class methods: *numpy* for mathematical operations and handling, transforming and shaping the data arrays, *scipy* to read MATLAB files and perform curve-fitting, *matplotlib* for most visualisations, *seaborn* for heatmaps of correlation matrices, *windrose* for wind roses and *gc* for garbage collection to free up memory.

3.1 Class methods

In this section the implementation of some of the class methods is explored and explained. To examine the full implementation of all methods, refer to the full documented code in the appendix.

3.1.1 convert_MATLAB

Data outside the sensor range for anemometers and accelerometers, specified by *anemo_range* and *acc_range*, is by default considered invalid and replaced with NaN in the method *convert_MATLAB*, as depicted in Figure 46.

```
if replace_invalid:
    # Replace data out of technical range of sensors with nan.
    data['H'][(data['H']>self.anemo_range)|(data['H']<0)] = np.nan
    data['W'][(data['W']>self.anemo_range)|(data['W']<-self.anemo_range)] = np.nan
    data['Dir'] = np.where((data['Dir']>360)|(data['Dir']<0),np.nan,data['Dir'])

    data['Aox_W'][(data['Aox_W']>self.acc_range*1e6)|(data['Aox_W']<-self.acc_range*1e6)] = np.nan
    data['Aoy_W'][(data['Aoy_W']>self.acc_range*1e6)|(data['Aoy_W']<-self.acc_range*1e6)] = np.nan
    data['Aoz_W'][(data['Aoz_W']>(self.acc_range+1)*1e6)|(data['Aoz_W']<-(self.acc_range-1)*1e6)] = np.nan

    data['Aox_E'][(data['Aox_E']>self.acc_range*1e6)|(data['Aox_E']<-self.acc_range*1e6)] = np.nan
    data['Aoy_E'][(data['Aoy_E']>self.acc_range*1e6)|(data['Aoy_E']<-self.acc_range*1e6)] = np.nan
    data['Aoz_E'][(data['Aoz_E']>(self.acc_range+1)*1e6)|(data['Aoz_E']<-(self.acc_range-1)*1e6)] = np.nan
```

Figure 46: Code snippet of method convert_MATLAB, replace_invalid

Note that the *acc_range* is given in G and therefore needs to be converted to µG by multiplying with 1e6.

The data is by default then down sampled to 144 samples per day, unless *full_detail* is set to *True*. This is achieved by calculating the statistics mean, std, min and max for each 10-minute data package of each measurement. There are two methods implemented to calculate the statistics. By default *np.nanmean* is used to calculate the mean of a 10-minute data package, as depicted in Figure 47, which ignores NaNs.

```
if ignore_nans:
    # Use np.nanmean etc. instead of np.mean to ignore nans.
    if H:
        anemo['H_mean'] = np.nanmean(data['H'],axis=2)
        anemo['H_std'] = np.nanstd(data['H'],axis=2)
        anemo['H_min'] = np.nanmin(data['H'],axis=2)
        anemo['H_max'] = np.nanmax(data['H'],axis=2)

    if W:
        anemo['W_mean'] = np.nanmean(data['W'],axis=2)
        anemo['W_std'] = np.nanstd(data['W'],axis=2)
        anemo['W_min'] = np.nanmin(data['W'],axis=2)
        anemo['W_max'] = np.nanmax(data['W'],axis=2)
```

Figure 47: Code snippet of method convert_MATLAB, ignore_nans

Note that the calculations are performed along *axis 2*, meaning that they are performed for all sensors and all 10-minute data packages simultaneously, showcasing the strength of array operations using NumPy, compared to running a double for-loop over all sensors and all data packages for example.

Also note how the calculations for a given measurement, *H* or *W* for example, are only performed if the respective variable is set to *True.* While this is the default for all measurements, it allows the user to de-select measurements that are not relevant for their analysis to save memory and speed up the processing.

If *ignore_nans* is set to *False, np.mean* is used instead to calculate the mean of a 10-minute data package, which sets the value for that mean to NaN if there is at least one NaN value in the 10-minute data package. *np.nanstd, np.nanmin* and *np.nanmax* or *np.std, np.min* and *np.max* are used similarly.

The statistics for the direction cannot be calculated directly in the same manner, as for example taking the mean of 30° and 350° in that way would give a result of 190°, although the correct mean is 10°. This is due to the 360° - 0° discontinuity. To avoid this issue, the horizontal wind vector, defined by the horizontal wind component and direction, is de-composed into wind in x-direction and y-direction according to bridge coordinates, denoted *vx* and *vy*, using $\mathrm{v_x} = -\sin(\mathrm{Dir} \cdot \mathrm{H})$ and $\mathrm{v_y} = -\cos(\mathrm{Dir} \cdot \mathrm{H})$, as depicted in Figure 48.

```
# Decompose horizontal wind vector into vx and vy component
# according to the bridge coordinate system.
vx = -np.sin(np.deg2rad(data['Dir']))*data['H']
vy = -np.cos(np.deg2rad(data['Dir']))*data['H']
```

Figure 48: Code snippet of method convert_MATLAB, vector-decomposition

Note the minus sign at the front of the equation. This is to keep the orientation of *vx* and *vy* consistent with the definition in Figure 18. Also note that the direction in degrees has to be converted to radians for the trigonometric functions in NumPy using *np.deg2rad*.

The mean of *Vx* and *Vy* is calculated separately, as depicted in Figure 49.

```
if Vx or Dir or Upwind:
    anemo['Vx_mean'] = np.nanmean(vx,axis=2)
    anemo['Vx_std'] = np.nanstd(vx,axis=2)
    anemo['Vx_min'] = np.nanmin(vx,axis=2)
    anemo['Vx_max'] = np.nanmax(vx,axis=2)

if Vy or Dir or Upwind:
    anemo['Vy_mean'] = np.nanmean(vy,axis=2)
    anemo['Vy_std'] = np.nanstd(vy,axis=2)
    anemo['Vy_min'] = np.nanmin(vy,axis=2)
    anemo['Vy_max'] = np.nanmax(vy,axis=2)
```

Figure 49: Code snippet of method convert_MATLAB, Vx and Vy statistics

Note that these calculations are also performed if *Dir* or *Upwind* are selected, even if *Vx* or *Vy* are de-selected, as these measurements are dependent on the statistics of *Vx* and *Vy*.

From these mean wind vectors in x- and y- direction the actual mean-direction is calculated using the *np.arctan2* function and modulo operator (%), as depicted in Figure 50.

```
if Dir or Upwind:
    # Calculate mean of direction from Vx_mean and Vy_mean
    # instead of original data to deal with 360-0 discontinuity.
    anemo['Dir_mean'] = (np.rad2deg(np.arctan2(anemo['Vx_mean'],anemo['Vy_mean']))+180+360)%360
```

Figure 50: Code snippet of method convert_MATLAB, Dir_mean

Note that *np.arctan2* returns the arctangent, " […] choosing the quadrant correctly […] so that arctan2(x1, x2) is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (x2, x1)." [45] Note that this requires the first argument to be the y-coordinate and the second argument to be the x-coordinate. [45] By reversing the arguments, giving the x-coordinate to the first argument (x1) and the y-coordinate to the second argument (x2), we get the angle between the ray passing through the origin and the point (0,1) and the ray passing through the origin and the

point (y,x). Therefore 0° will be at the top of the wind rose, instead of at the East position.

np.rad2deg converts the radians back to degrees, giving a result of -180° to +180°. Adding 360° and applying the modulo operator, which returns the remainder of the division by 360°, converts the result to a range of 0° to 360°. Adding another 180° reverses the minus sign applied in the vector-decomposition and ensures the correct direction, from which the wind is blowing, is preserved.

The *AOA* is calculated using $\mathrm{AOA} = \tan^{-1}(\frac{W}{H})$, as depicted in Figure 51.

```
if AOA:
    anemo['AOA_mean'] = np.nanmean(np.rad2deg(np.arctan(data['W']/data['H'])),axis=2)
    anemo['AOA_std'] = np.nanstd(np.rad2deg(np.arctan(data['W']/data['H'])),axis=2)
    anemo['AOA_min'] = np.nanmin(np.rad2deg(np.arctan(data['W']/data['H'])),axis=2)
    anemo['AOA_max'] = np.nanmax(np.rad2deg(np.arctan(data['W']/data['H'])),axis=2)
```

Figure 51: Code snippet of method convert_MATLAB, AOA

The turbulence intensity (*_turb*) describing the relative wind variability for each wind component is calculated in this work by dividing the standard deviation of each component by the combined mean wind vector magnitude, as depicted in Figure 52.

```
if Turb:
    if H:
        anemo['H_turb'] = anemo['H_std']/np.sqrt(np.square(anemo['H_mean'])+np.square(anemo['W_mean']))
    if W:
        anemo['W_turb'] = np.abs(anemo['W_std']/np.sqrt(np.square(anemo['H_mean'])+np.square(anemo['W_mean'])))
    if Vx:
        anemo['Vx_turb'] = np.abs(anemo['Vx_std']/np.sqrt(np.square(anemo['H_mean'])+np.square(anemo['W_mean'])))
    if Vy:
        anemo['Vy_turb'] = np.abs(anemo['Vy_std']/np.sqrt(np.square(anemo['H_mean'])+np.square(anemo['W_mean'])))
```

Figure 52: Code snippet of method convert_MATLAB, turbulence intensity

The code for *H_turb* for example can be translated into the equation

$$H_{turb} = \frac{H_{std}}{\sqrt{{H_{mean}}^2 + {W_{mean}}^2}}$$

Note that the commonly used definition of turbulence intensity slightly differs from the way it is implemented in this work, considering the standard deviation along and across a mean wind vector, as described in [20, pp. 12-14].

The central accelerations *Aox_C*, *Aoy_C* and *Aoz_C* are calculated from the east- and west- accelerometer of each accelerometer pair using $C = \frac{W+E}{2}$, as depicted in Figure 53.

```
if centr:
    acc['Aox_C_mean'] = np.nanmean((data['Aox_W']+data['Aoy_E'])/2,axis=2)
    acc['Aox_C_std'] = np.nanstd((data['Aox_W']+data['Aox_E'])/2,axis=2)
    acc['Aox_C_min'] = np.nanmin((data['Aox_W']+data['Aox_E'])/2,axis=2)
    acc['Aox_C_max'] = np.nanmax((data['Aox_W']+data['Aox_E'])/2,axis=2)
    acc['Aox_C_max_by_std'] = acc['Aox_C_max']/acc['Aox_C_std']

    acc['Aoy_C_mean'] = np.nanmean((data['Aoy_W']+data['Aoy_E'])/2,axis=2)
    acc['Aoy_C_std'] = np.nanstd((data['Aoy_W']+data['Aoy_E'])/2,axis=2)
    acc['Aoy_C_min'] = np.nanmin((data['Aoy_W']+data['Aoy_E'])/2,axis=2)
    acc['Aoy_C_max'] = np.nanmax((data['Aoy_W']+data['Aoy_E'])/2,axis=2)
    acc['Aoy_C_max_by_std'] = acc['Aoy_C_max']/acc['Aoy_C_std']
```

Figure 53: Code snippet of method convert_MATLAB, central acceleration

Note that for accelerometer data an additional statistic *_max_by_std* is calculated. This quotient of *_max* and *_std* is later used to find bridge responses to traffic-dominated periods in the accelerometer data and separate it from responses to wind-dominated periods, as described in *2.5.2 Processing*.

The torsional accelerations are calculated using $theta = tan^{-1}(\frac{W-E}{x\_dist\_from\_centerline})$, as depicted in Figure 54.

```
if theta:
    acc['theta_mean'] = np.nanmean(np.rad2deg(np.arctan(((data['Aoz_W'] - data['Aoz_E'])*self.g*1e-6/self.acc_x_dist_from_centerline))),axis=2)
    acc['theta_std'] = np.nanstd(np.rad2deg(np.arctan(((data['Aoz_W'] - data['Aoz_E'])*self.g*1e-6/self.acc_x_dist_from_centerline))),axis=2)
    acc['theta_min'] = np.nanmin(np.rad2deg(np.arctan(((data['Aoz_W'] - data['Aoz_E'])*self.g*1e-6/self.acc_x_dist_from_centerline))),axis=2)
    acc['theta_max'] = np.nanmax(np.rad2deg(np.arctan(((data['Aoz_W'] - data['Aoz_E'])*self.g*1e-6/self.acc_x_dist_from_centerline))),axis=2)
    acc['theta_max_by_std'] = acc['theta_max']/acc['theta_std']
```

Figure 54: Code snippet of method convert_MATLAB, theta

Note that the original data is in μG and needs to be converted to $\frac{m}{s^2}$ by multiplying with *self.g* and *1e-6* to obtain $\frac{rad}{s^2}$, which is then converted to $\frac{^\circ}{s^2}$ using *np.rad2deg*.

The names of the anemometers, saved in the *anemo_names* meta data, is used to determine if a sensor is up- or downwind for a given measurement, depending on the wind direction, as depicted in Figure 55.

```
if Upwind:
    # Create boolean arrays determening if a sensor is up- or downwind
    # at each datapoint, depending on direction measurement.
    anemo['Upwind'] = np.zeros(anemo_shape)
    for ind,anemo_name in enumerate(anemo_names):
        if anemo_name.endswith('W') or anemo_name.endswith('Wb') or anemo_name.endswith('Wt'):
            anemo['Upwind'][ind] = np.where(anemo['Dir_mean'][ind]>=180,1,0)
        elif anemo_name.endswith('E') or anemo_name.endswith('Eb') or anemo_name.endswith('Et'):
            anemo['Upwind'][ind] = np.where(anemo['Dir_mean'][ind]<180,1,0)
    anemo['Downwind'] = (~anemo['Upwind'].astype(bool)).astype(int)
```

Figure 55: Code snippet of method convert_MATLAB, Upwind

Similarly, the hanger number, y-position and information if an anemometer is on the West or East side of the bridge and if it is on the top or bottom row is determined from the anemometer name and encoded for each datapoint.

3.1.2 load_data

Data for one or multiple days is loaded using the *load_data* method. This method first uses the *scipy.io.loadmat* function to load the MATLAB file for one day and then the convert_matlab method to pre-process it. This is repeated for further days in a for-loop as a form of batch-processing, as depicted in Figure 56.

```
# Batch-processing of further days of data, if any,
# with temporary variables being created for each day
# and deleted afterwards to free up memory for processing of the next day.
for day in range(1,len(self.file_names)):
    print('Day',day+1,':',self.file_names[day][:-4][14:])
    temp_data = scipy.io.loadmat(self.file_path+self.file_names[day])
    temp_anemo_names, temp_acc_names, temp_time_array, temp_t_array, temp_anemo, temp_acc, temp_weather = self.convert_MATLAB(
```

Figure 56: Code snippet of method load_data, for-loop

This is necessary because the MATLAB files loaded via *scipy.io.loadmat* take up a lot of memory.

Through the use of batch-processing, temporary data can be cleaned up using *del* and the garbage collector function *gc.collect()*, as depicted in Figure 57.

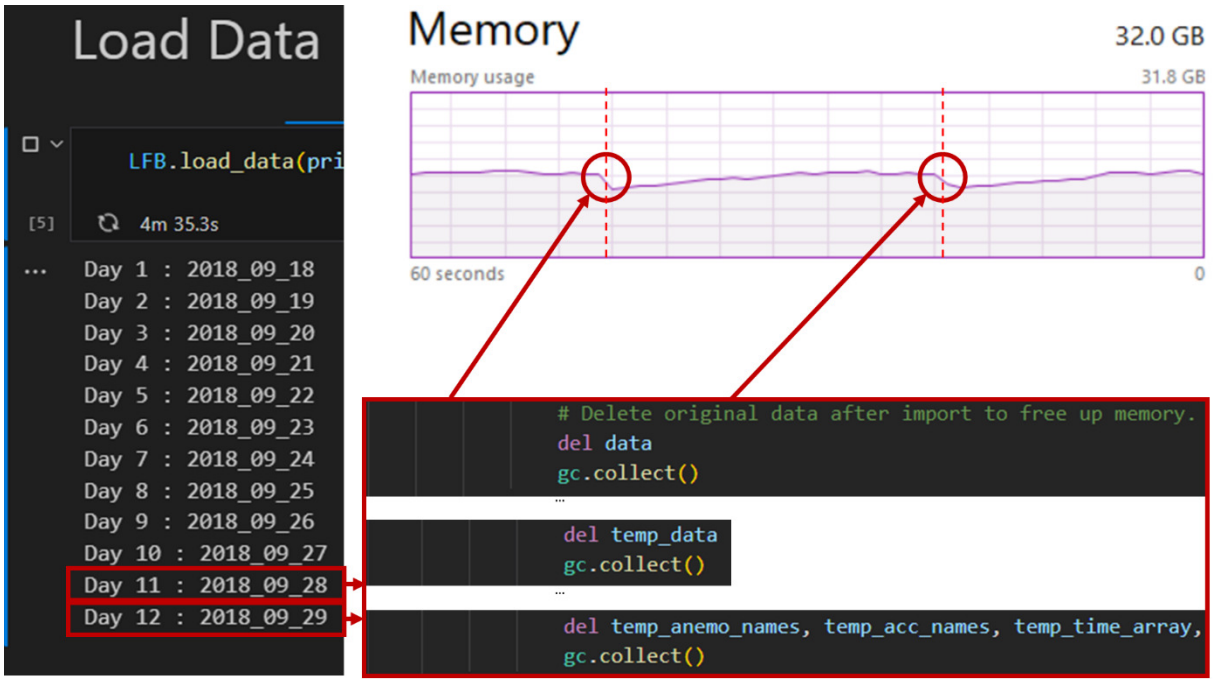


Figure 57: Memory usage during batch import

Figure 57 shows how the memory usage is reduced by using *batch-processing* to import the data. Each day of MATLAB data is processed sequentially in a *for-loop*. Temporary data variables are created for each day. These are deleted before moving on to the next step in the import pipeline, to free up memory for processing of the next day. This keeps the memory usage at a relatively constant level.

The temporary data arrays are stacked to the existing data arrays horizontally, as depicted in Figure 58.

```
for key in temp_anemo.keys():
    if key in self.anemo.keys():
        self.anemo[key] = np.hstack((self.anemo[key],temp_anemo[key]))
    else:
        self.anemo[key] = np.hstack((np.zeros_like(self.time_array),temp_anemo[key]))
```

Figure 58: Code snippet of method load_data, horizontal stacking

If a measurement type was not available in previous days, the data from those days is engineered as zeros to keep all data synchronized.

After the batch-processing the method *define_units* is called to define the units of the loaded measurements and by default convert accelerations to $\frac{m}{s^2}$.

3.2 Quick start guide

In the following a quick start guide is provided on how to use the *BridgeData* class and its methods to import MATLAB bridge data, process it and visualize it. This only covers the basics and aims to encourage the user to explore how to perform more detailed analysis. It is encouraged to refer to the documentation for the class methods by using the *help(class.method)* function and explore the documented example code in the appendix as well as the example analysis performed in *4 Analysis using this works code*.

3.2.1 Data import

Firstly, an instance of the *BridgeData* class needs to be initialized. In the example below an instance called *LFB* is created:

```
file_path = '../Data/Bridge/dataExtracted/'
file_names = ['dataExtracted_2018_09_18.mat','dataExtracted_2018_09_19.mat']
LFB = BridgeData(file_names=file_names,file_path=file_path)
```

The *BridgeData* class requires a file path relative to the location of the notebook to where the MATLAB data is stored and a list of names for the files to import. Once the class instance is initialized as *LFB*, the MATLAB data can simply be loaded into the class instance with the code below:

```
LFB.load_data()
```

Use the following line of code to access the documentation of *load_data* and find out more about the import options:

```
help(BridgeData.load_data)
```

Once the data is loaded, it can be processed and analysed.

3.2.2 Data processing

Use the line of code below to clean the data:

```
LFB.clean_data()
```

Note that the cleaned data is stored in class variables with the suffix *_cleaned,* such as *anemo_cleaned* and *acc_cleaned*.

The following line of code can be used to create time arrays for time series analysis:

```
LFB.feature_time(cleaned=False)
```

This creates the class variables *LFB.days*, *LFB.hours*, *LFB.minutes*, and *LFB.seconds*. Note that the keyword *cleaned* needs to be set to *True* or *False* accordingly, depending on which type of data is to be analysed with it.

The following line of code can be used to classify the data into data that is traffic- or wind-dominated:

```
LFB.find_traffic(traffic_thresh=8)
```

The *find_traffic* method uses a threshold *traffic_thresh* for the *max_by_std* criteria discussed in *2.5.2 Processing*. The classification is stored in the class variable *LFB.traffic*.

The *filter_data* method can be used to filter the data, as in the code below:

```
filter_idxs = LFB.filter_data(LFB.anemo['H_mean'], hp_cutoff=8, highpass=True)
```

Note that this only creates the indices *filter_idxs* of data applying to that filter, in this case mean horizontal wind speed measurements above 8 m/s. These filter indices need to be applied to the data in the visualisation to see the filter in effect. This can be done as showcased in the code below:

```
filtered_data = LFB.anemo['H_mean'].T[filter_idxs].T
```

Note how the array of anemometer data needs to be transposed using the *.T* operation before applying the filter indices and transposed back to its original shape afterwards using another *.T* operation. This is also the case for accelerometer data, but not for weather data or time arrays created by *LFB.feature_time*, as they are of lower dimension.

3.2.3 Saving and loading the state of a class instance

A class instance with the name *LFB* can be saved in its current state, including imported and processed data, using the *dump* function of the *dill* module, as in the example below:

```
dill.dump(LFB,file=open('Your/File/Path/YourFileName','wb'))
```

Reversely, a previously saved class instance state can be loaded using *load* function as in the code below:

```
LFB = dill.load(open('Your/File/Path/YourFileName','rb'))
```

This is especially useful when you want to come back to the analysis of a larger dataset, where re-importing the data would take up considerable amounts of time.

3.2.4 Data analysis

As described in *2.5.3 Post-processing*, there are multiple methods implemented for the analysis of the data. The code below shows a simple example to plot the time-series data on *H_mean* with days on the x-axis:

```
LFB.plot_data(LFB.days,LFB.anemo['H_mean'],ylabel='H_mean',
      yunit=LFB.units['H_mean'],xlabel='days',labels=LFB.anemo_names)
```

The visualisation methods are designed with a similar structure, sharing some of their functionality and the respective *keywords* to interact with them. To examine some more complex implementations of visualisations study the documented example code in the appendix. The visualisation methods of the *BridgeData* class are based on the default visualisation methods from the *matplotlib* library. However, they are optimized for a consistent visualisation of the bridge data. Nonetheless any of the standard methods from the *matplotlib* library or any other library can be used to visualise and analyse the data. This includes statistical analysis using the *numpy* and *scipy* libraries.

Tip: Should your Jupyter Notebook display graphical bugs upon opening, click *Clear All Outputs*, save it, close it and re-open it. Then run the code again from the top.

4 Analysis using this works code

In the following section 30 days of data are analysed using the developed code. The data is from 18th September 2018 until 18th October 2018. Further visualisations can be found in the code in the appendix.

4.1 Local wind conditions

This specific period has been chosen for its high maximum wind speeds at Sola observation station from data published for 2018 on Yr.no, as depcited in Figure 59.

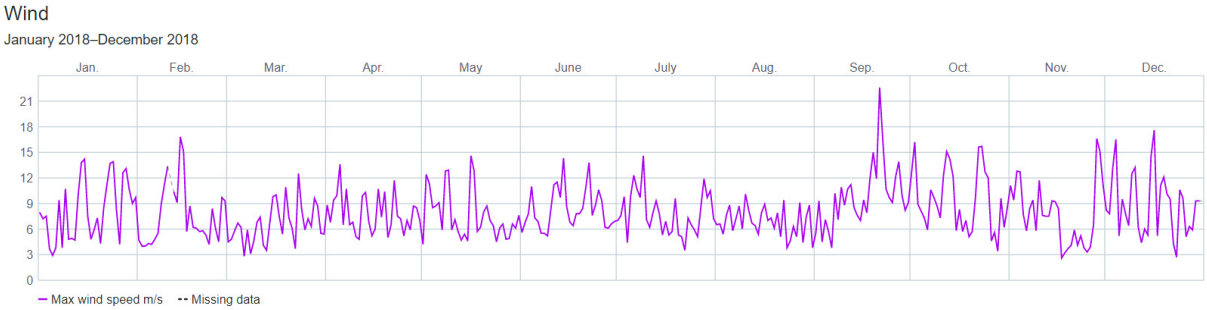


Figure 59: Maximum wind speeds at Sola observation station in 2018 [46]

The strongest gust of that year has been recorded at about 29.4 m/s on 21st September. [46]

4.1.1 Wind speeds and primary direction

Maximum wind speeds *H_max* have been gusting up to 35 m/s at the Lysefjord Bridge in this period, as depicted in Figure 60.

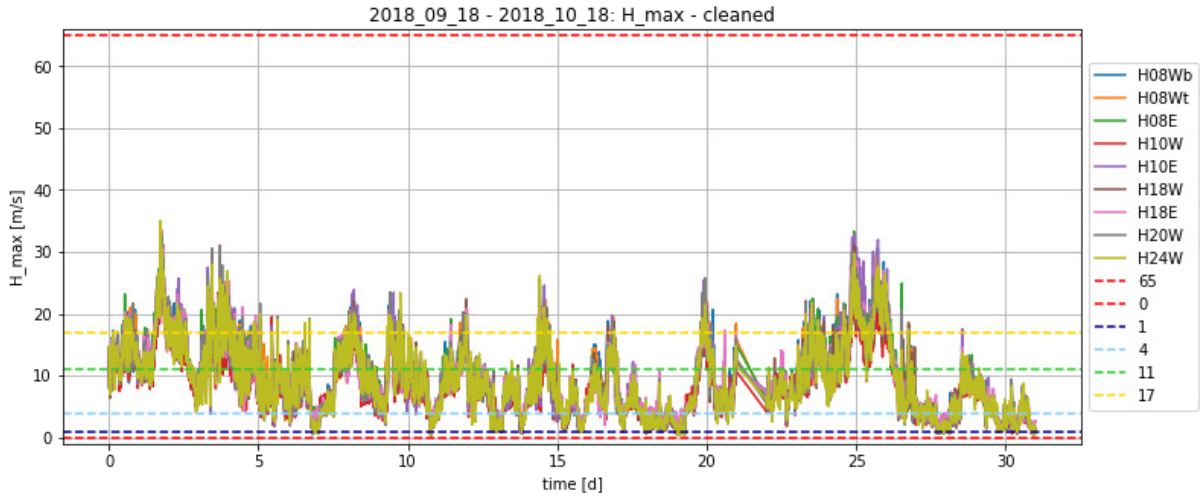


Figure 60: Maximum wind speeds per anemometer at Lysefjord Bridge during 30 days in autumn 2018

In the chosen period maximum wind speeds above 11 m/s have been recorded for multiple days at a time. Note that there is missing data on the 21st day of this period.

The primary wind directions at the Lysefjord Bridge during this period are NNE and SSW, as can be seen in the wind roses in Figure 61 and Figure 62.

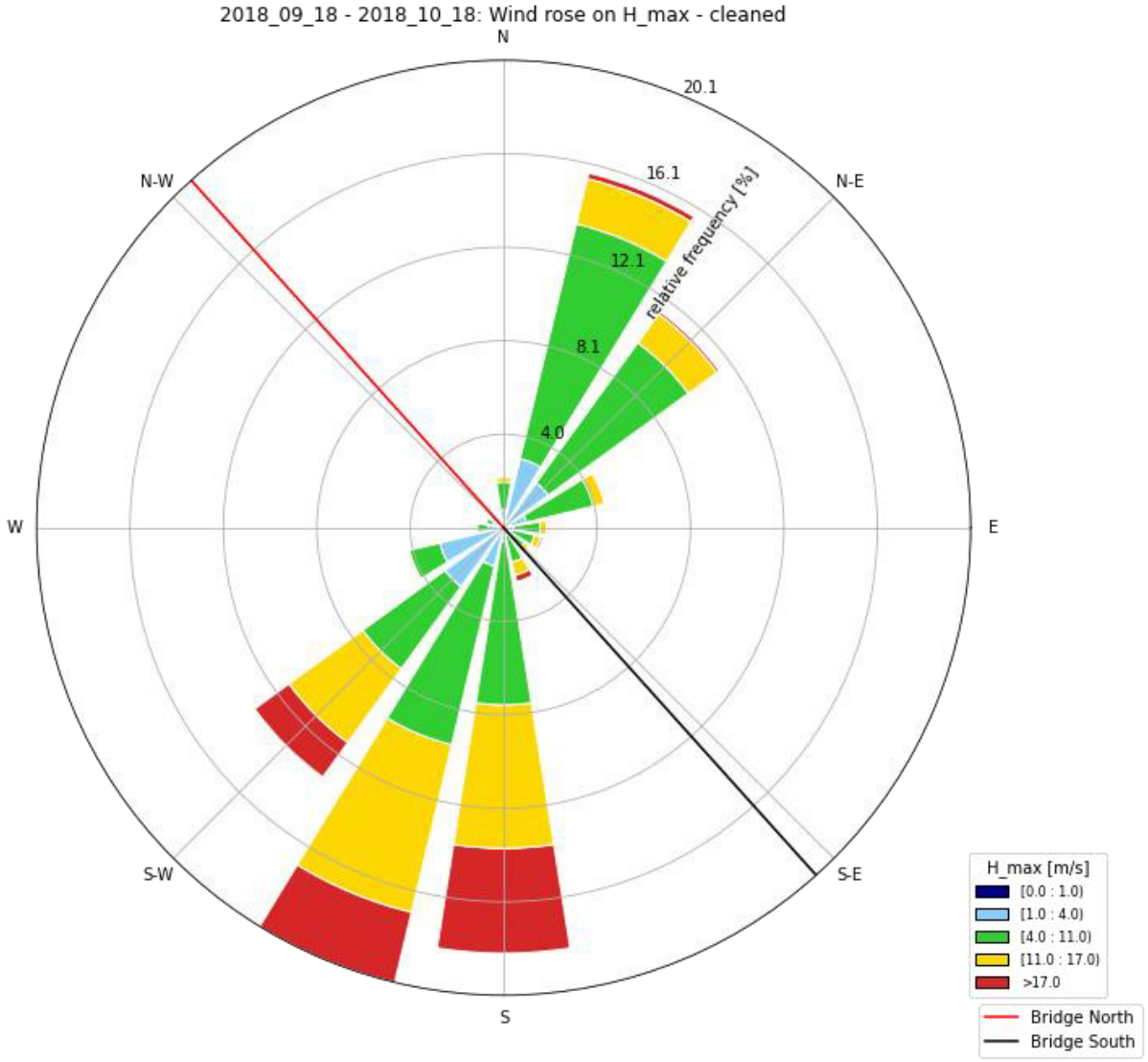


Figure 61: Combined wind rose on maximum wind speeds from all anemometers at Lysefjord Bridge during 30 days in autumn 2018

The two primary wind directions are from here on referred to as north-easterly (NE) wind for winds coming from anywhere NE of the bridge, meaning from -42° to 138° in geographic coordinates or 0° to 180° in bridge coordinates, and south-westerly (SW) wind for winds coming from anywhere SW of the bridge, the remaining half of the wind rose.

Note that the wind rose in Figure 62 displays the mean wind speeds *H_mean* of each 10-minute data package instead of the maximum wind speeds *H_max* displayed in Figure 60 and Figure 61. The mean wind speeds described the average wind condition while the maximum wind speeds describe the gusts.

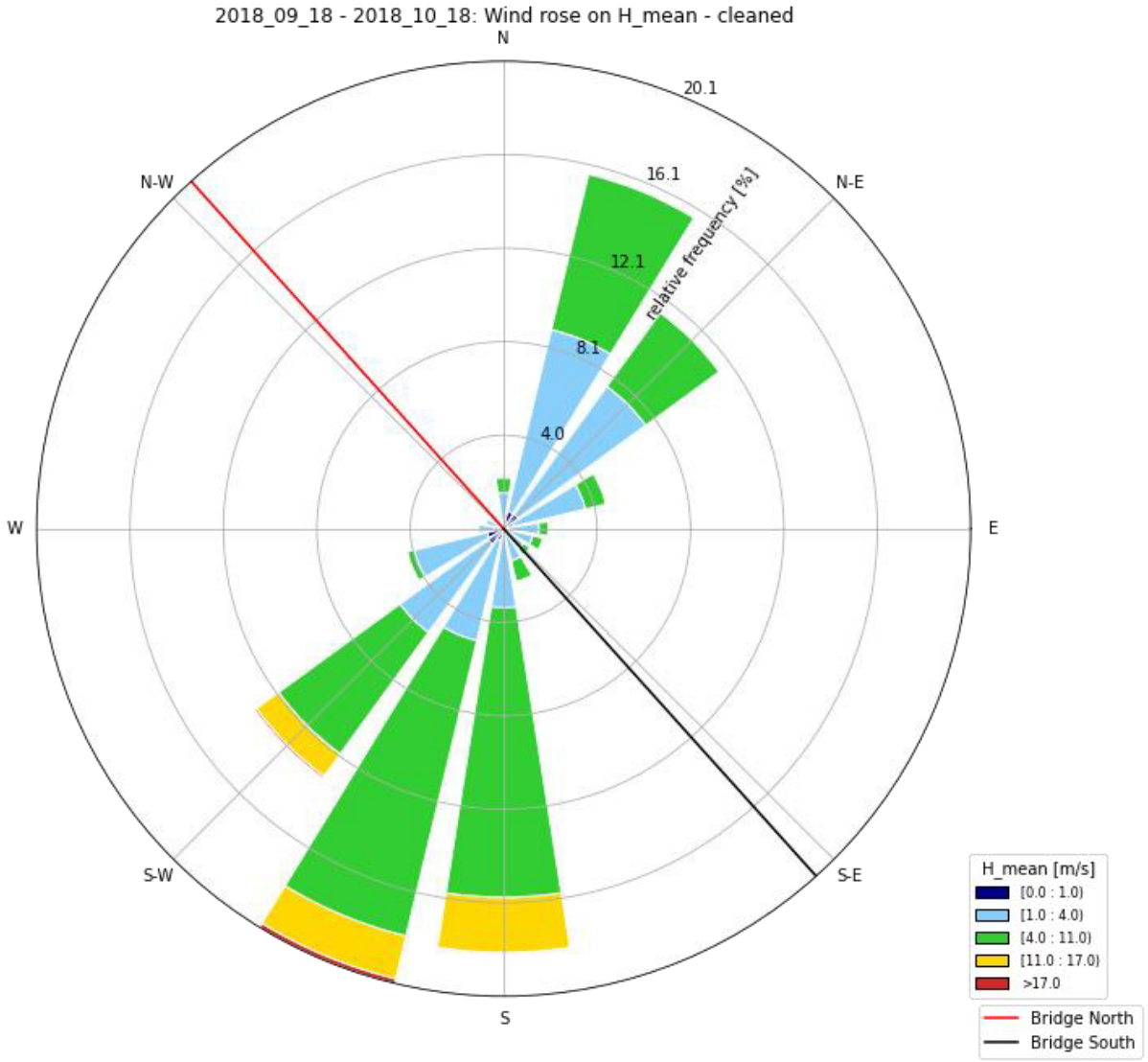


Figure 62: Combined wind rose on mean wind speeds from all anemometers at the Lysefjord Bridge during 30 days in autumn 2018

The wind roses in Figure 61 and Figure 62 show, that there are slightly stronger average winds and gusts from SSW compared to average winds and gusts from NNE. We see that there is a significant angle of incidence, or yaw angle, of up to about 48° for most winds from south-westerly direction and up to about 25° for most winds from north-easterly direction.

The histogram in Figure 63 is displaying the wind direction referring to bridge North, which is offset by -42° from geographic North.

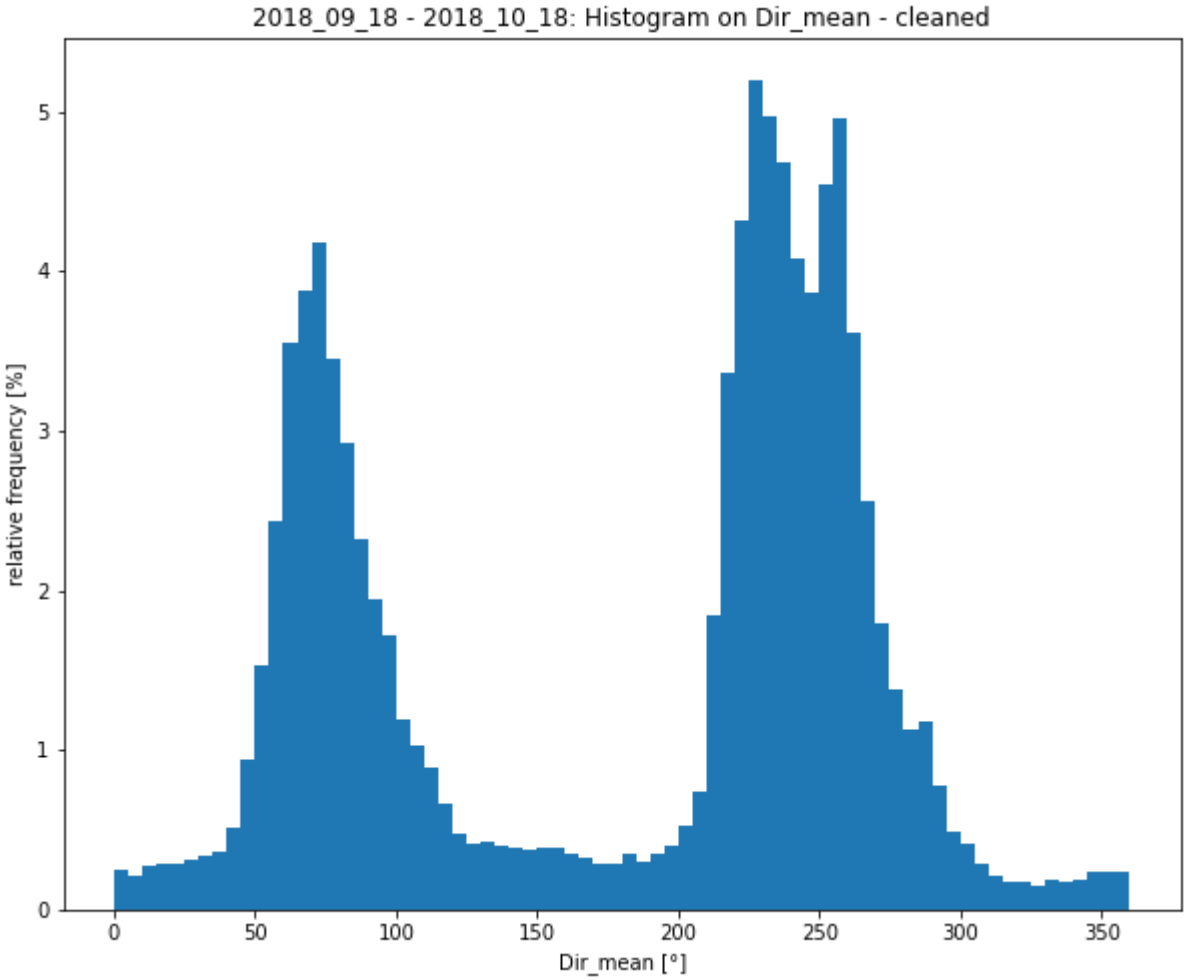


Figure 63: Histogram on mean wind direction

As expected, the histogram displays a bi-polar distribution. There is slightly more data with wind from SSW than wind from NNE in the 30 days period. The bins in Figure 63 have a width of 5°. When accounting for bins that make up more than 0.6% of the data, the wind from SSW is in the range of about 205° to 295° while the wind from NNE is in the range of about 45° to 120°. This gives the winds from SSW a broader bandwidth of about 90°, compared to the bandwidth of winds from NNE, which have a slightly narrower bandwidth of about 75°. This makes sense, as the wind is more confined by the walls of the fjord when coming from NNE, compared to the more open terrain when coming from the entrance to the fjord from SSW. Note that the distribution of wind from SSW is prominently bi-polar itself with one peak at about 225° to 230° and another at about 255° to 260°. There is also another small

peak at about 285° to 295°. This can be explained by the wind direction slightly changing along the span of the bridge, as depicted in Figure 64, which shows the wind rose for each anemometer, arranged by the anemometers position on the bridge. Note that the illustration is rotated 90 degrees counterclockwise with geographic North at the left of the page to increase readability. The bottom (right) row of wind roses is from anemometers on the west side of the bridge, ordered by hanger number with the windrose from H08Wt above (left to) the windrose from H08Wb and the top (left) row of wind roses is from anemometers on the east side of the bridge.

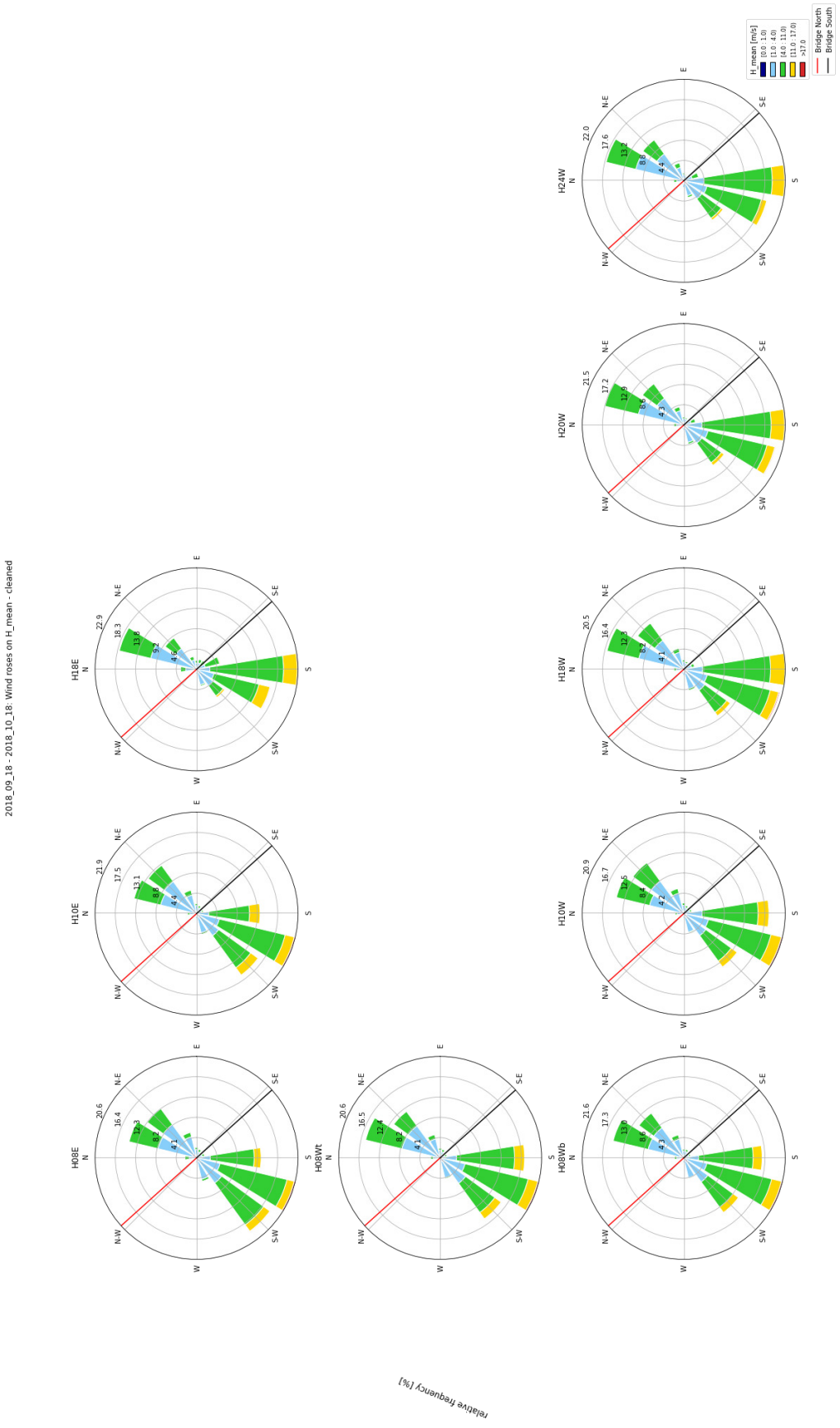


Figure 64: Wind roses on average wind speeds per anemometer, arranged by position on the bridge

For winds from SSW the wind tends to come more from the South for anemometers that are further down south on the bridge. Similarly, for winds from NNW the wind tends to come more from the North for anemometers that are further down south on the bridge.

The data can easily be split into south-westerly and north-easterly winds by applying a filter on the wind direction, as discribed in *3.2.4 Data analysis*. This allows the comparison of wind conditions with south-westerly wind and north-easterly wind.

Figure 65 and Figure 66 show the respective histograms on average windspeeds for south-westerly and north-easterly wind.

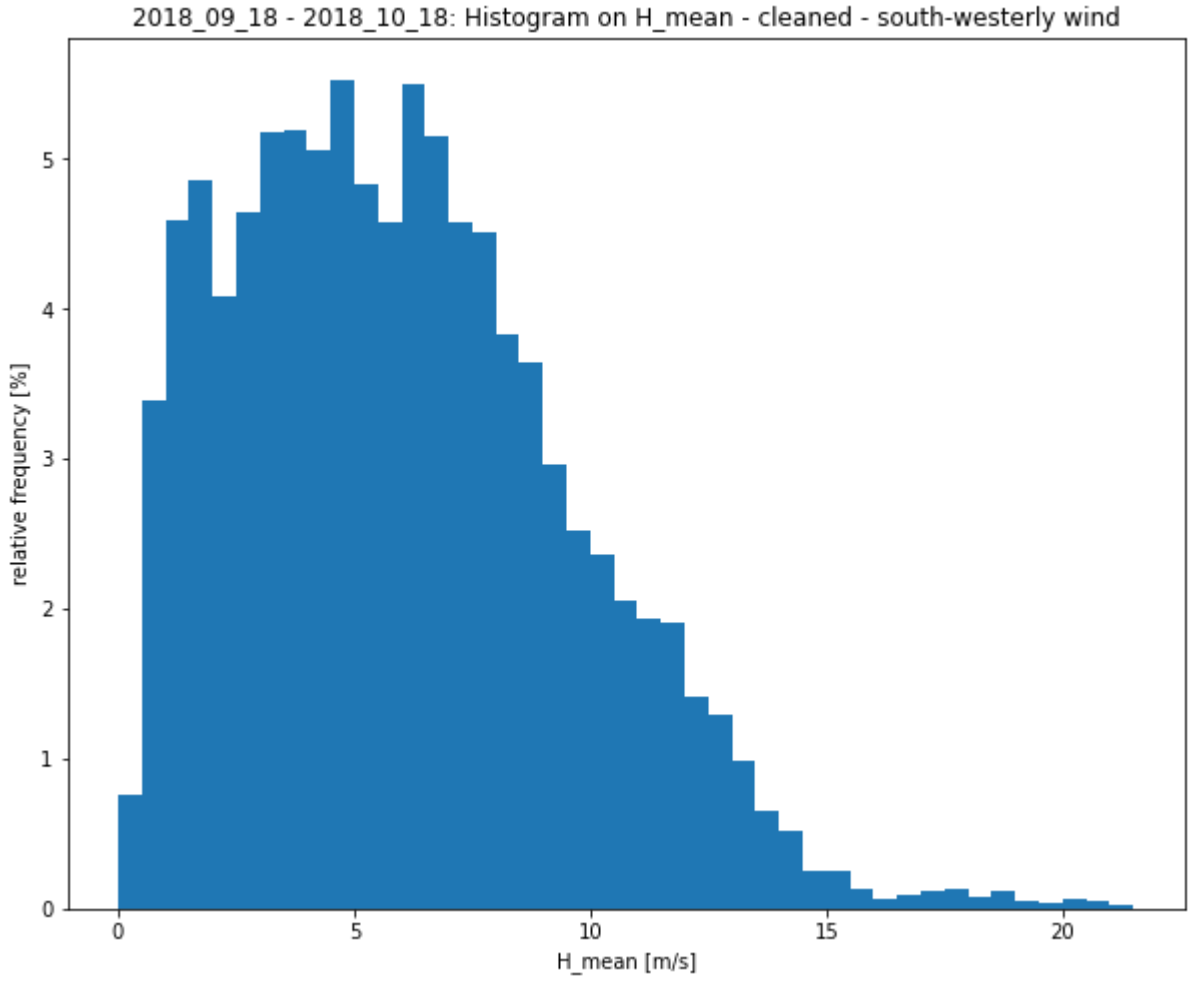


Figure 65: Histogram on average wind speeds for south-westerly wind

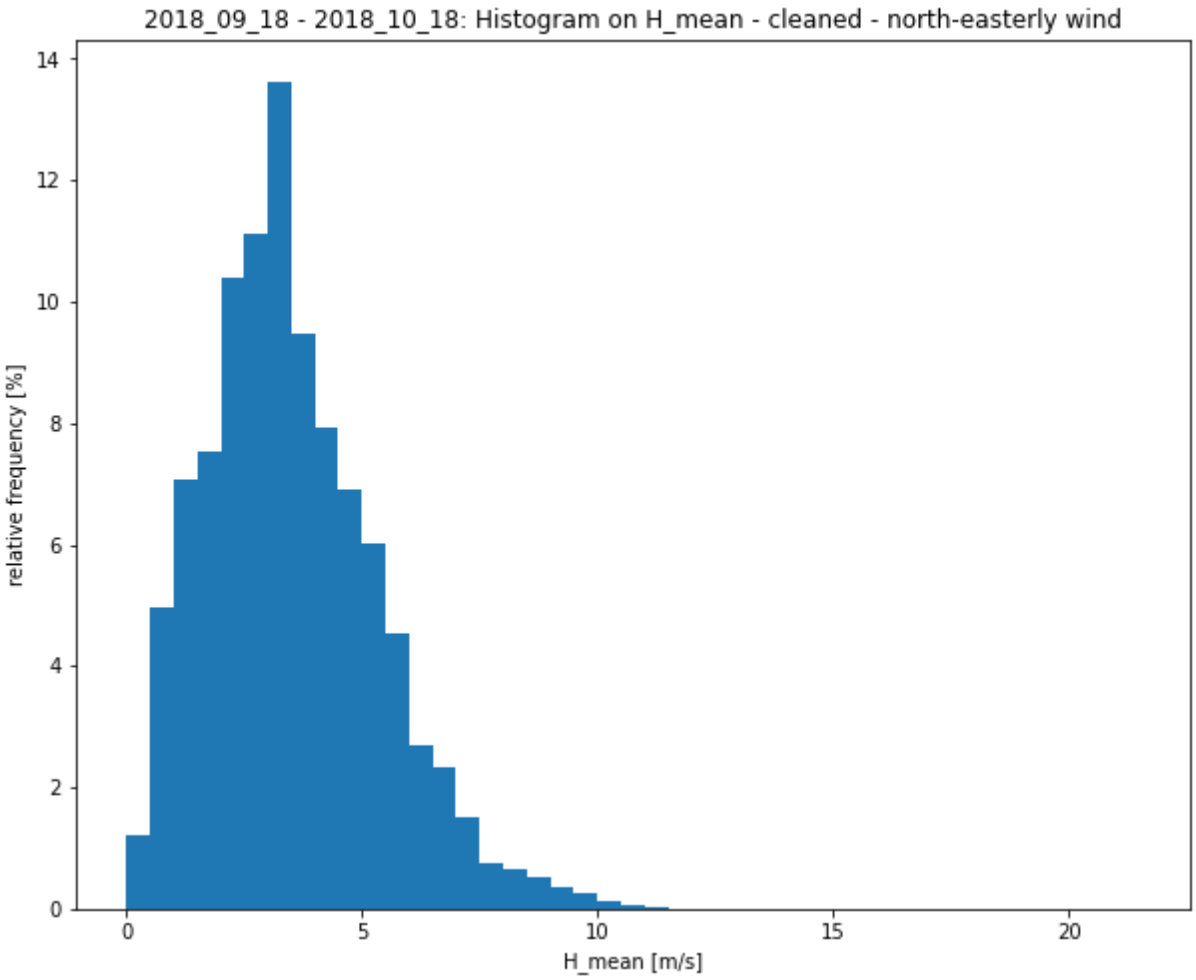


Figure 66: Histogram on average wind speeds for north-easterly wind

The histograms illustrate the different distributions of average wind speeds for south-westerly and north-easterly winds, with generally lower windspeeds from north-easterly directions. The distributions can be further examined using statistics on the filtered data.

Table 3 Compares the statistics on H_mean for the primary wind directions.

Table 3: Comparison of statistics on mean wind speeds for south-westerly wind and north-easterly wind

Statistic on H_mean [m/s]	**south-westerly wind**	**north-easterly wind**
Mean	6.06	3.51
Median	5.72	3.28
Percentile [10, 25, 75, 90]	[1.62 3.25 8.33 11.04]	[1.26 2.22 4.61 5.9]
Minimum	0.18	0.22
Maximum	21.48	11.65
Variance	12.68	3.29
Standard deviation	3.56	1.81
Skewness	0.64	0.71
Kurtosis	0.21	0.54

The mean and median of mean wind speeds for south-westerly winds are about 2.5 m/s higher than for north-easterly winds. The highest mean windspeed for south-westerly winds are almost double those for north-easterly winds. The distribution for north-easterly mean wind is slightly more skewed towards lower wind speeds. While it has a lower standard deviation, it has slightly heavier tails.

Note that the applied kurtosis uses Fisher´s definition, in which the normal distribution has a kurtosis of 0, a distribution with lower kurtosis has thinner tails and a distribution with higher kurtosis has heavier tails. [47]

4.1.2 Turbulence intensity

Similarly, the distribution of turbulence intensities can be examined and compared from the histograms in Figure 67 and Figure 68 as well as Table 4.

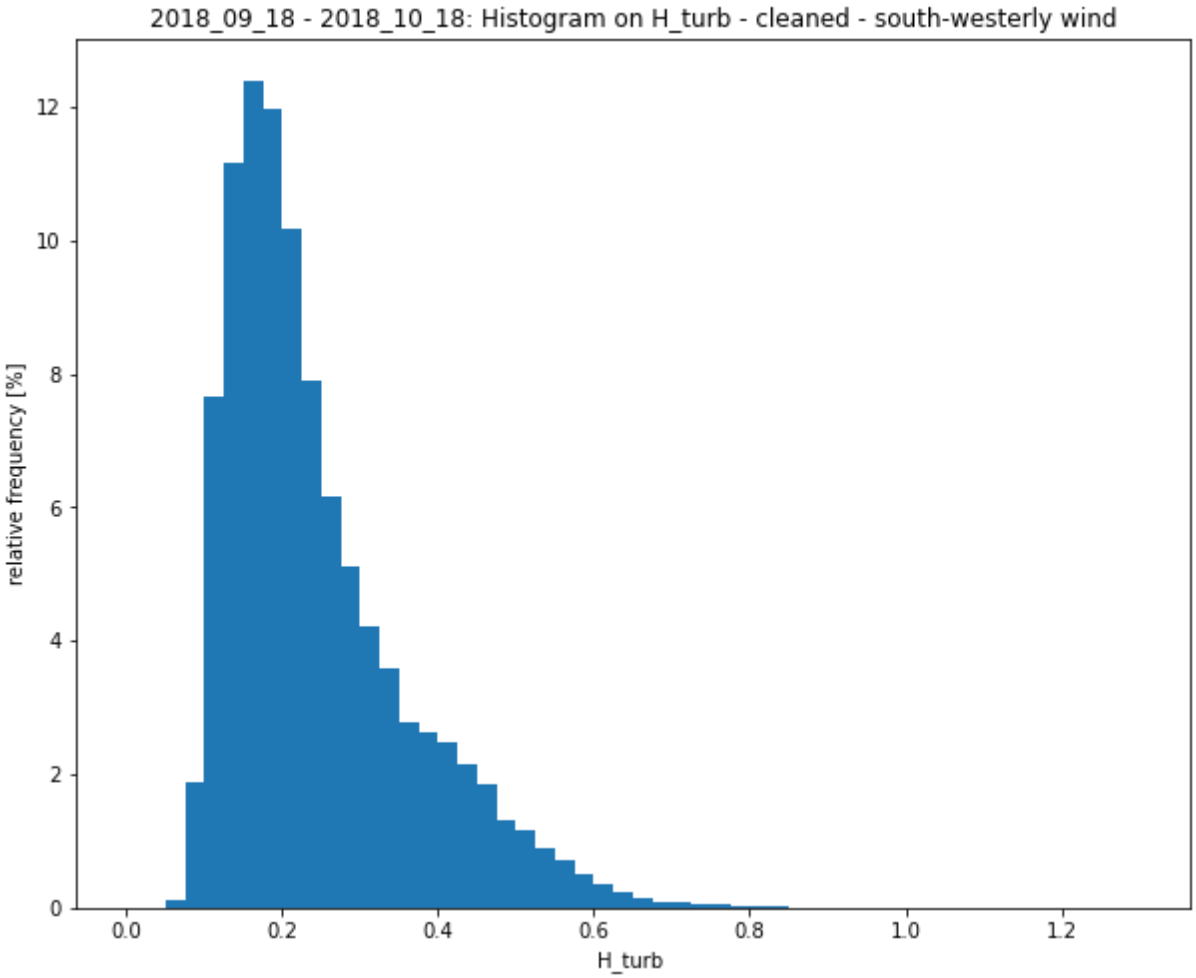


Figure 67: Histogram on horizontal turbulence intensity for south-westerly wind

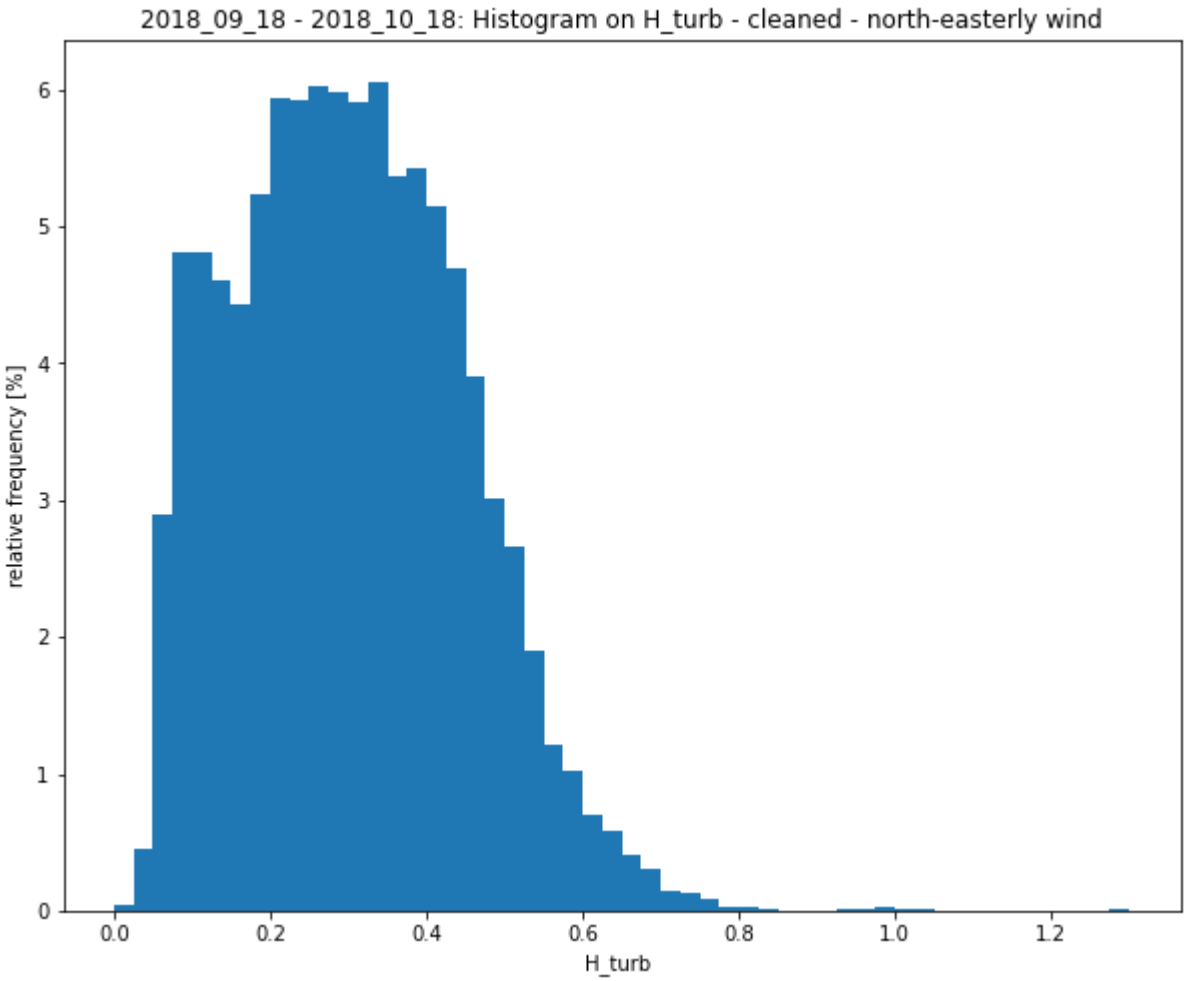


Figure 68: Histogram on horizontal turbulence intensity for north-westerly wind

Table 4: Comparison of statistics on horizontal turbulence intensity for south-westerly wind and north-easterly wind

Statistic on H_turb	**south-westerly wind**	**north-easterly wind**
Mean	0.25	0.3
Median	0.21	0.29
Percentile [10, 25, 75, 90]	[0.13 0.16 0.3 0.42]	[0.11 0.19 0.41 0.49]
Minimum	0.06	0.0
Maximum	0.98	1.29
Variance	0.01	0.02
Standard deviation	0.12	0.15
Skewness	1.29	0.42
Kurtosis	1.68	0.07

From the histograms in Figure 67 and Figure 68 as well as Table 4 we see that winds from north-easterly direction are slightly more turbulent. The distribution is less skewed towards lower turbulence intensities. While it has a higher standard deviation it has noticeably thinner tails.

When comparing the turbulence intensity measured by anemometers on the downwind side of the bridge to those on the upwind side, as in Table 5, we see that the turbulence intensity on the upwind side is on average slightly lower.

Table 5: Statistical comparison of turbulence intensity between upwind and downwind anemometers for south-westerly and north-easterly winds

Statistic on H_turb	**south-westerly wind**				**north-easterly wind**			
Anemometer	*H08Wb*	*H08E*	*H18W*	*H18E*	*H08Wb*	*H08E*	*H18W*	*H18E*
Up-/Downwind	*U*	*D*	*U*	*D*	*D*	*U*	*D*	*U*
Mean	0.24	0.25	0.25	0.25	0.31	0.3	0.31	0.3
Median	0.2	0.22	0.21	0.22	0.31	0.29	0.3	0.29
Percentile [10, 25, 75, 90]	[0.12 0.15 0.3 0.42]	[0.13 0.16 0.31 0.43]	[0.12 0.16 0.31 0.42]	[0.13 0.16 0.3 0.41]	[0.11 0.2 0.42 0.51]	[0.11 0.18 0.4 0.48]	[0.11 0.2 0.41 0.5]	[0.12 0.19 0.4 0.49]
Minimum	0.06	0.07	0.07	0.06	0.04	0.04	0.03	0.0
Maximum	0.89	0.95	0.84	0.98	1.03	0.98	1.0	1.02
Variance	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02
Standard deviation	0.12	0.12	0.12	0.11	0.15	0.14	0.15	0.15

For south-westerly winds all statistics on the turbulence intensity at the upwind anemometer H08Wb, except variance and standard deviation, are slightly lower than at the downwind anemometer H08E. At H18 the median, 10th percentile and maximum are slightly lower on the upwind anemometer H18W. The minimum and percentiles 75 and 90 are slightly higher.

For north-easterly winds all statistics on the turbulence intensity at the upwind anemometer H08E, except minimum, 10th percentile and variance, are slightly lower than at the downwind anemometer H08W. At H18 the mean, median, minimum, 25th, 75th and 90th percentiles are slightly lower on the upwind anemometer H18E. The 10th percentile and maximum are slightly higher.

It is also possible to plot the horizontal turbulence intensity on a wind rose, as depicted in Figure 69.

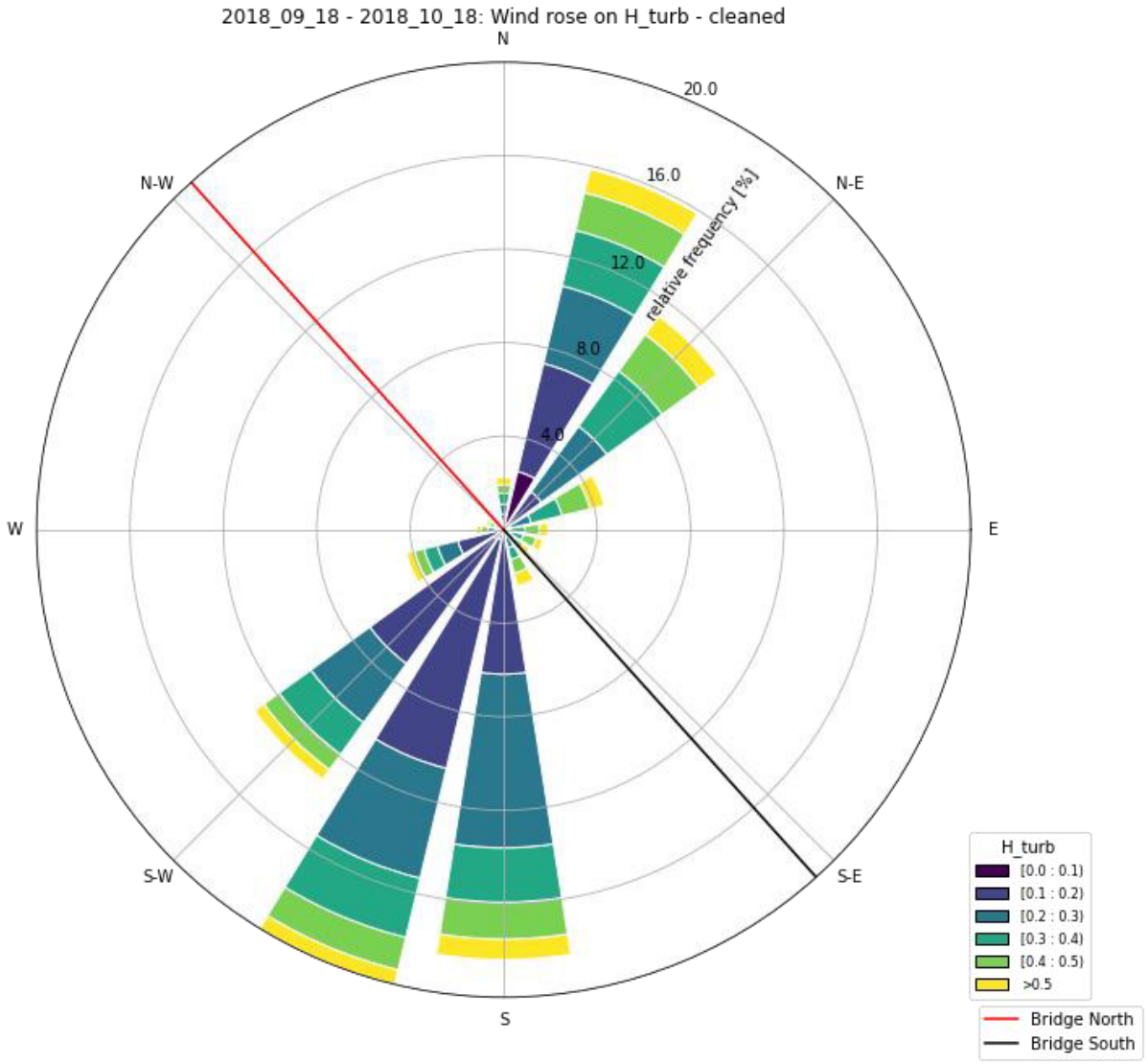


Figure 69: Wind rose on horizontal turbulence intensity, combined data from all anemometers

It is again also possible to split the wind rose into the different anemometers, as depicted in Figure 70, which allows a more detailed analysis of the turbulence distribution across the bridge.

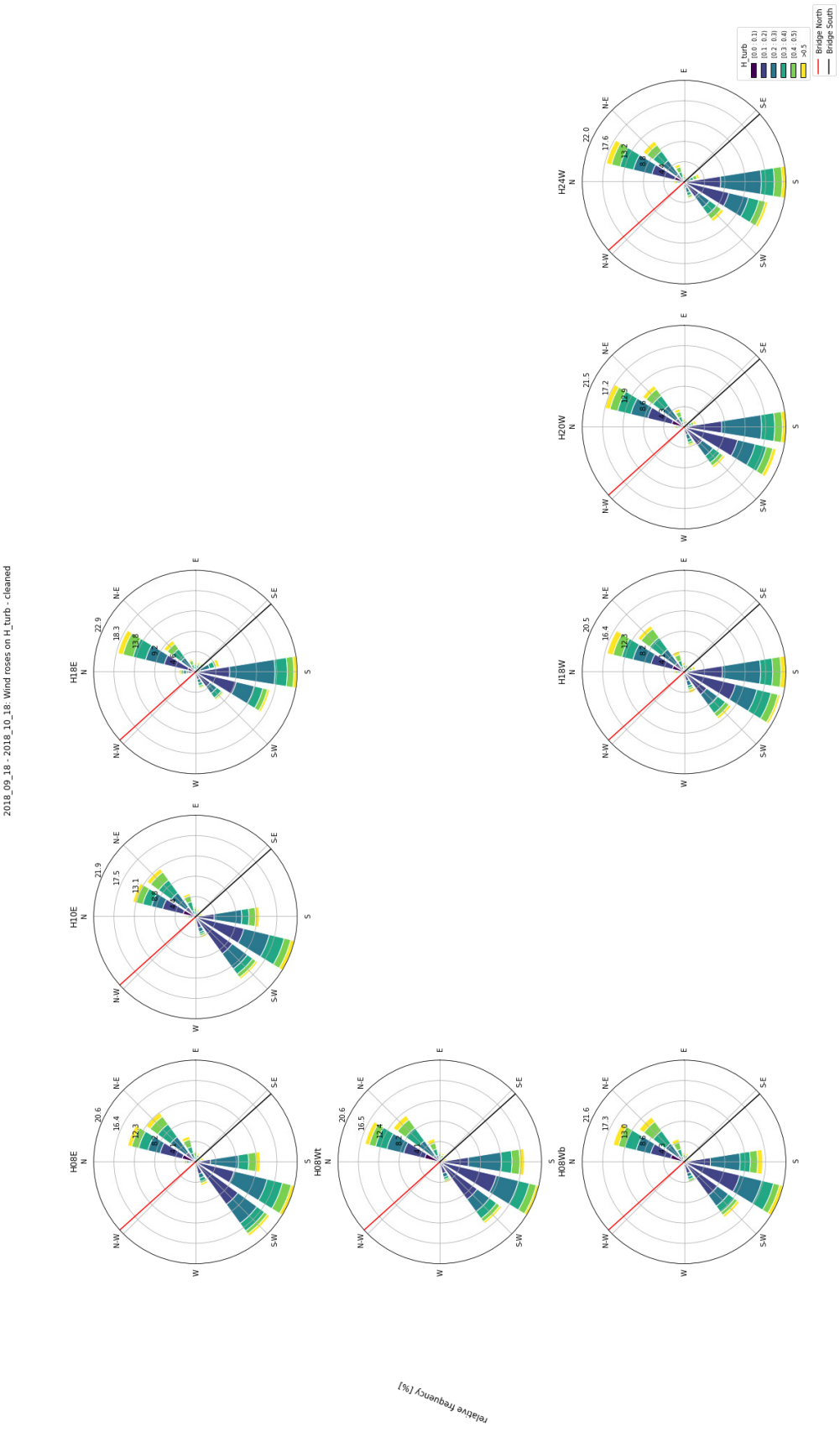


Figure 70: Wind roses on horizontal turbulence intensity per anemometer, arranged by position on the bridge

Figure 70 illustrates, how the turbulence intensity measured on the downwind anemometers is slightly higher for both winds from north-easterly and south-westerly direction. From their perspective the wind flow is disrupted by the upwind structures of the bridge, such as the cables, railings and bridge deck, as well as any traffic that might be on the bridge. We can also see that the top anemometer on H08W experiences slightly less turbulent winds compared to the bottom anemometer, as there is less influence on the wind flow by the bridge deck.

Figure 71 illustrates the horizontal and vertical turbulence intensity on the y-axis and color-axis respectively at different mean horizontal wind speeds on the x-axis.

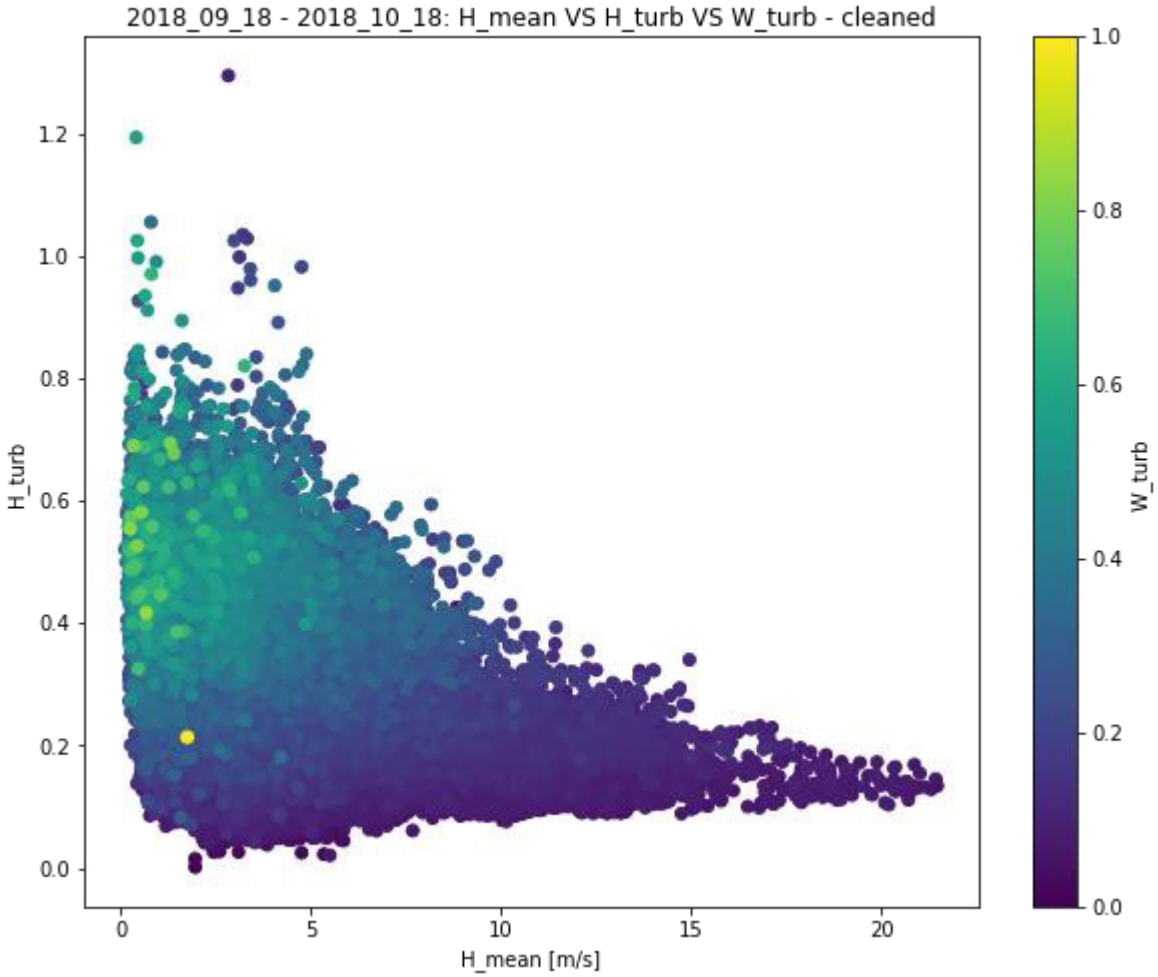


Figure 71: Horizontal and vertical turbulence intensity at different horizontal wind speeds

As expected, turbulence intensity decreases with higher wind speeds and is below 0.3 for wind speeds above 15 m/s. Vertical turbulence intensity increases with horizontal turbulence intensity up to about 0.5 after which the vertical component decreases as the horizontal component increases.

Figure 72 shows the correlations between *H_mean* and *H_turb* across all sensors.

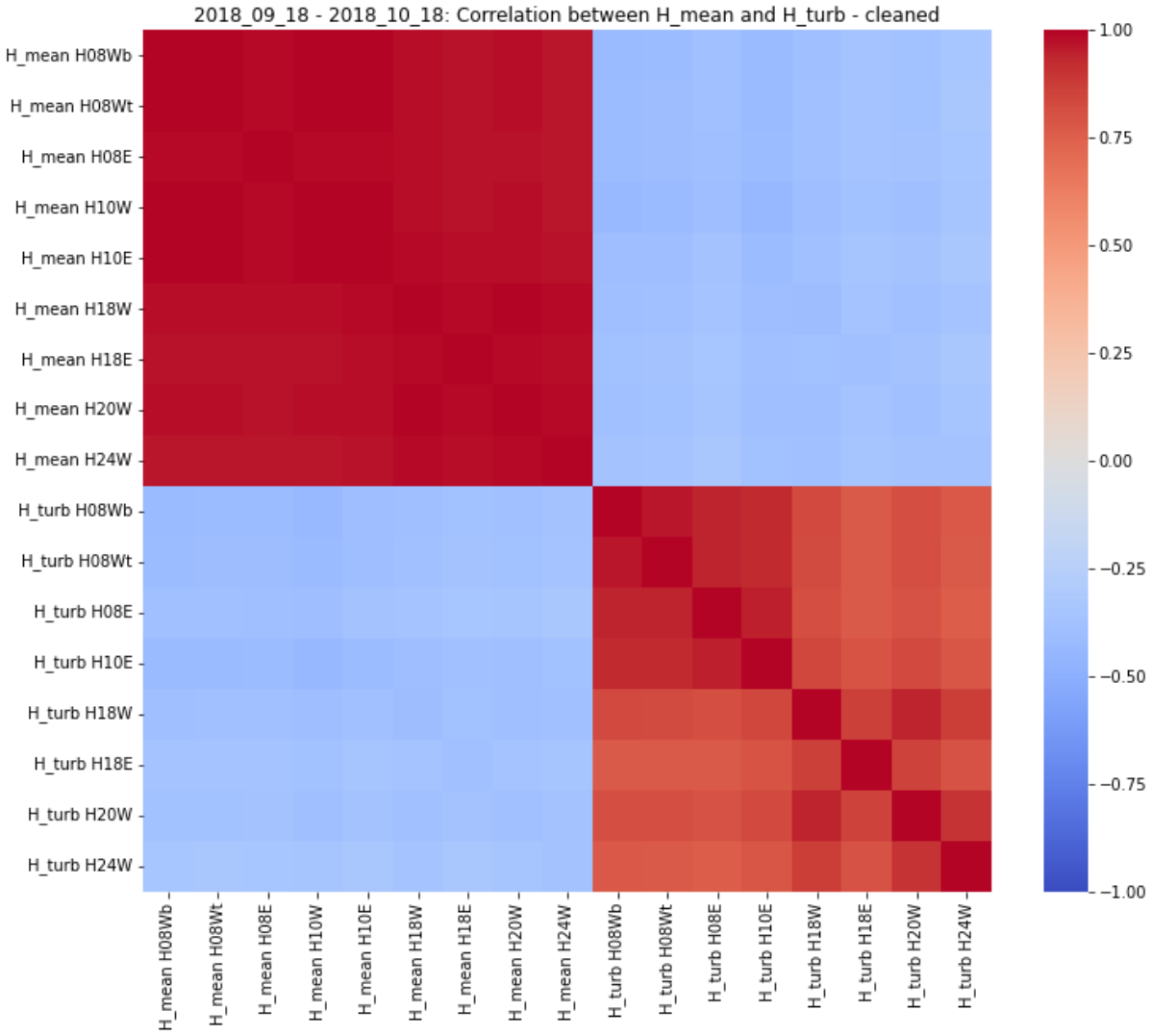


Figure 72: Correlation matrix between horizontal mean wind speed and turbulence intensity per anemometer

From Figure 72 we see that the measurements of mean horizontal wind speed are very well correlated between the anemometers, displayed as a homogeneous dark red coloring in the upper-left quadrant of the correlation matrix heatmap, corresponding to a high positive correlation coefficient close to 1. In the lower right quadrant, we see that the turbulence intensity is slightly less correlated across the anemometers. However, note that there is a higher correlation between the anemometers at hangers H08 to H10. There is also a slightly lower correlation between the anemometers at hangers H18 to H24, however it is still slightly higher compared to their correlation to the previous group of anemometers. The correlation coefficient between *H_mean* and *H_turb* is negative across all anemometers at

about -0.4, colored in light blue in the top right and bottom left quadrants. This means that with higher wind speeds turbulence intensity decreases, as expected.

4.1.3 Angle of attack

Figure 73 shows a wind rose on the mean angle of attack, created from the combined data of all anemometers during the 30-day period. Warmer colors represent positive angles of attack while colder colors represent negative angles of attack, as previously illustrated in Figure 19.

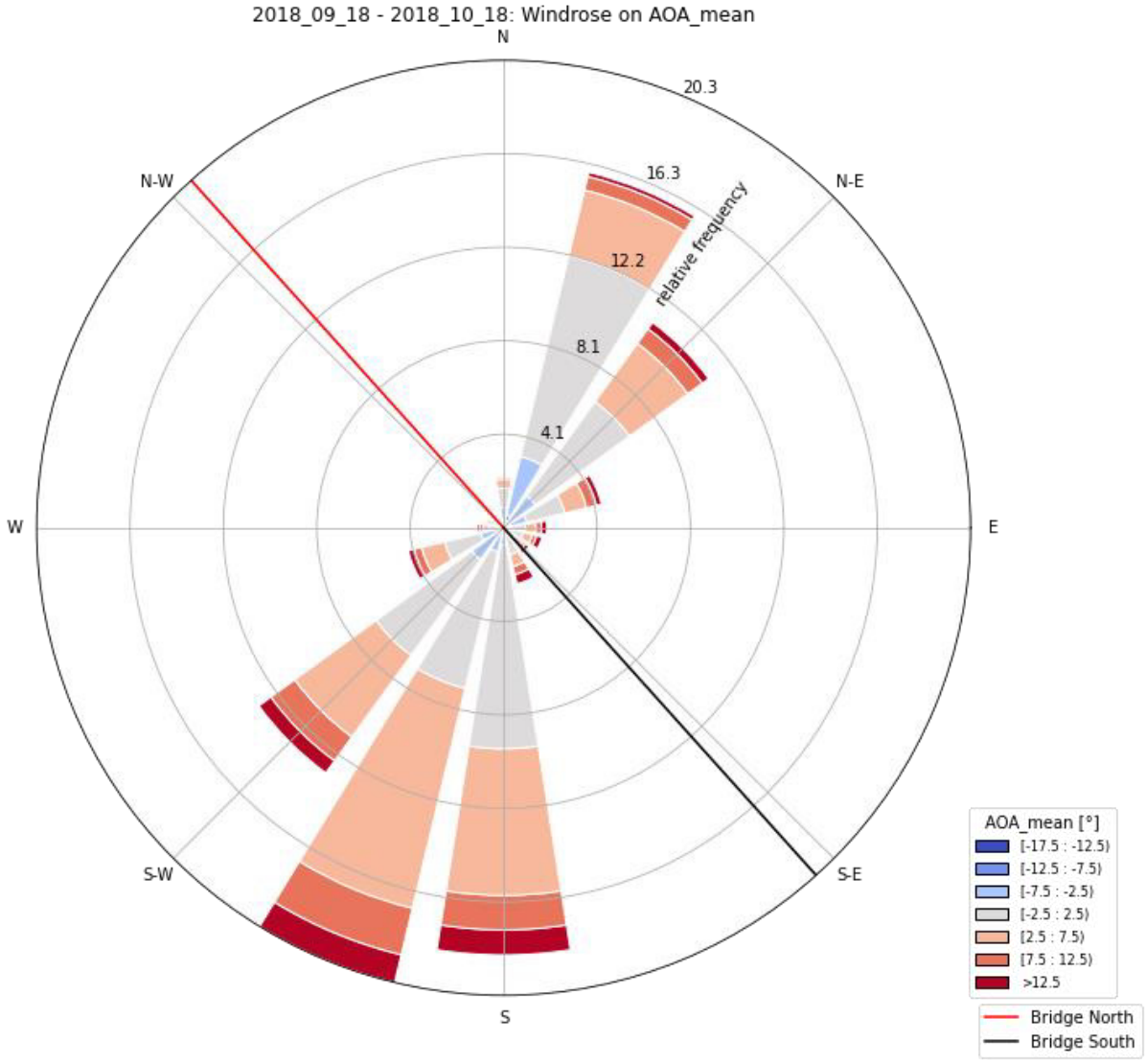


Figure 73: Mean angle of attack wind rose combined from all anemometers

Note how while a good portion of the data displays relatively neutral AOAs there is a tendency to more positive AOAs for winds from SSW and more negative AOAs from NNE. This can also be seen when comparing the histograms on AOA from

north-easterly and south-westerly winds in Figure 74 and Figure 75 as well as Table 6.

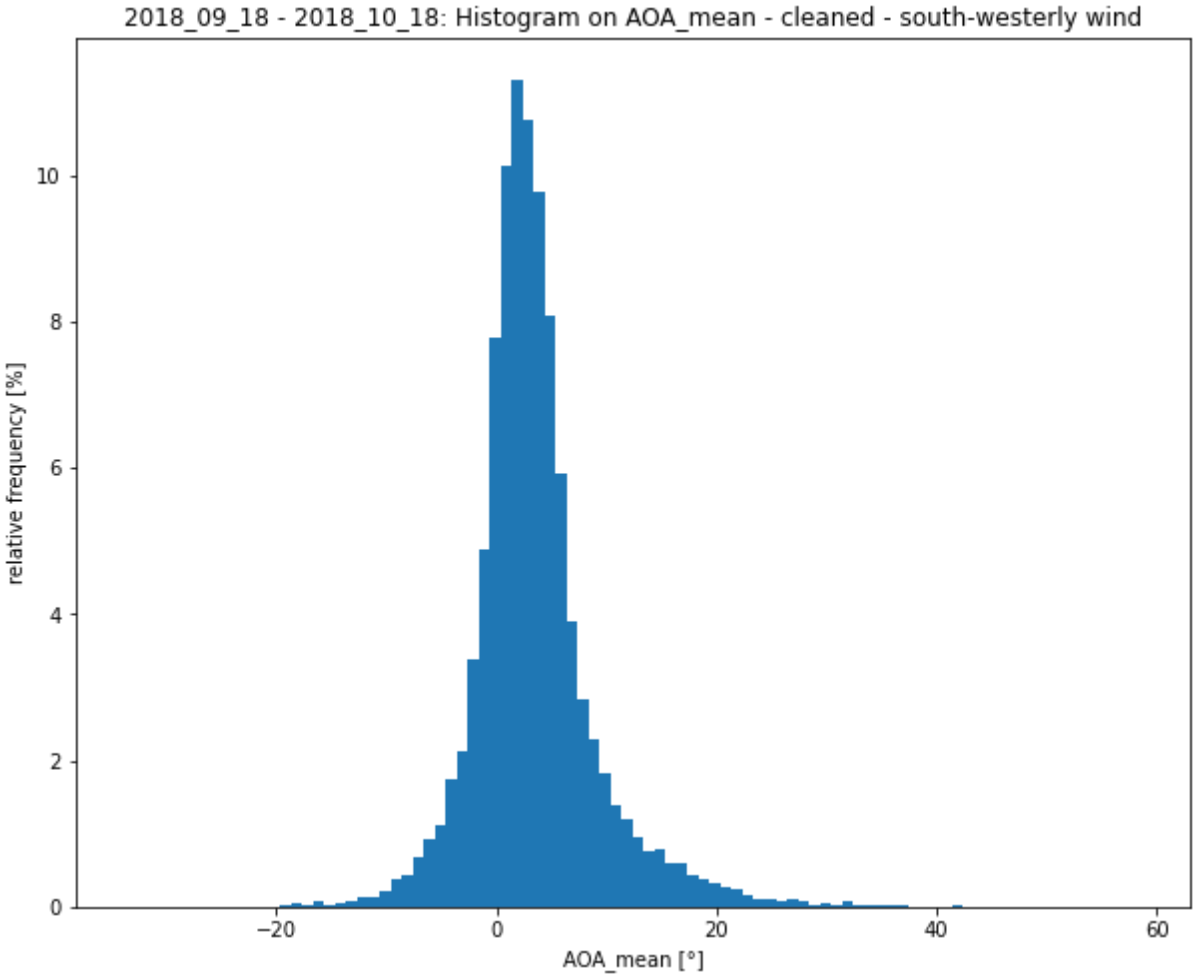


Figure 74: Histogram on mean angle of attack for south-westerly wind

Figure 74 clearly shows the tendency to positive AOAs in the distribution for south-westerly winds.

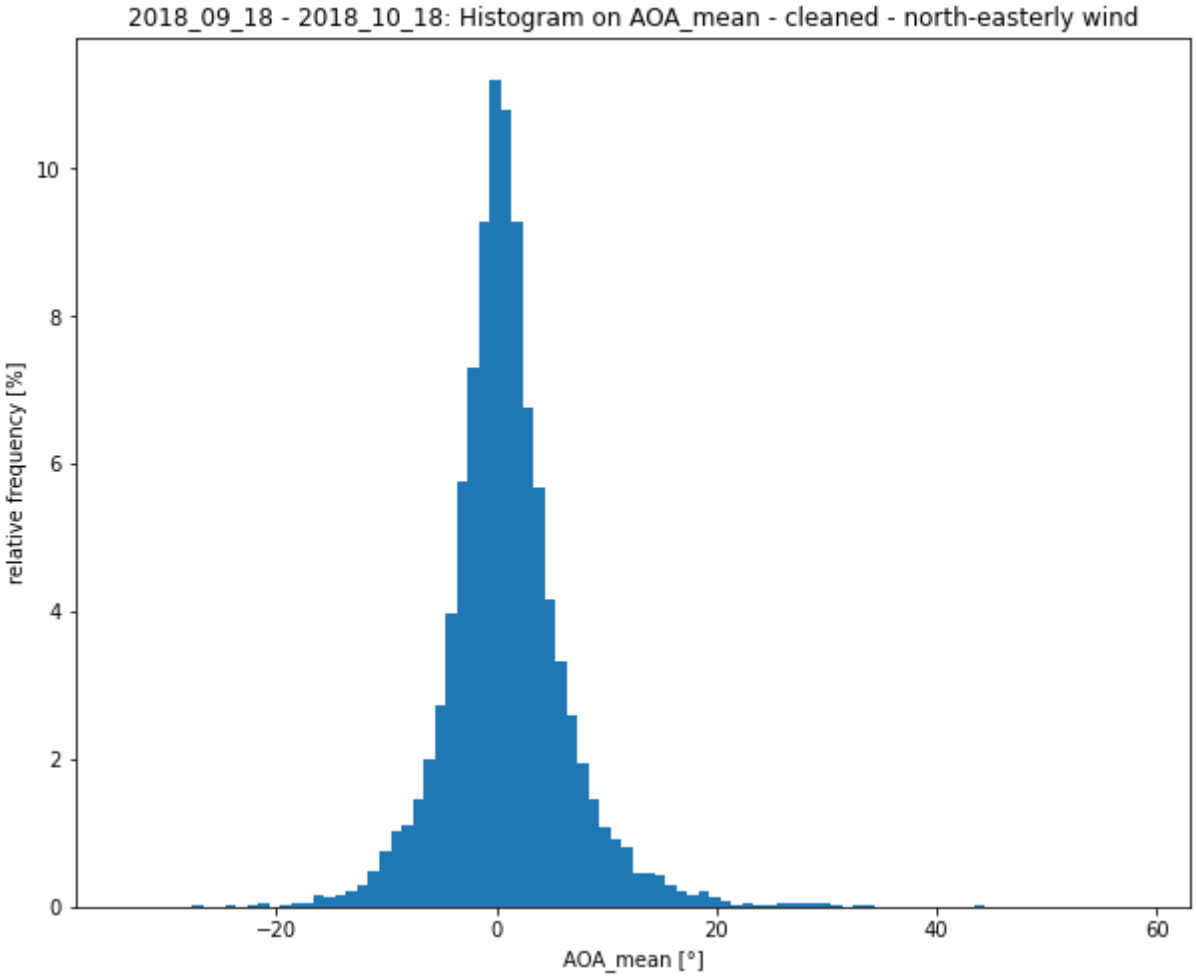


Figure 75: Histogram on mean angle of attack for north-easterly wind

Figure 75 shows a distribution centred around 0° for north-easterly winds.

Table 6: Comparison of statistics on mean angle of attack for south-westerly wind and north-easterly wind

Statistic on AOA_mean [°]	**south-westerly wind**	**north-easterly wind**
Mean	3.41	0.86
Median	2.76	0.53
Percentile [10, 25, 75, 90]	[-2.05 0.44 5.46 9.73]	[-4.85 -1.99 3.34 7.03]
Minimum	-21.9	-33.11
Maximum	53.22	45.93
Variance	31.51	29.7
Standard deviation	5.61	5.45
Skewness	1.23	0.73
Kurtosis	5.24	4.86

As can be seen in Figure 74 and Figure 75 as well as Table 6, both distributions have a similar standard deviation of about 5.5°. South-westerly winds have a slightly positive mean and median angle of attack at about 3.4° and 2.8°, while AOAs of north-easterly winds have a mean and median at about 0.9° and 0.5°. The 10^{th} percentile is about -2° and -5° respectively. The 90^{th} percentile is about 10° and 7° respectively. Therefore 80% of the data is inside a 12° bandwidth for both south-westerly and north-easterly winds. We see that more than 80% of the data is inside the range of -15° to +15°, the critical AOAs at which an aerofoil, such as the wing of an aircraft, would typically experience stall conditions. [48]

However, the wind flow around a bridge girder, as illustrated in Figure 32, is not as laminar and attached as the wind flow around a wing and subjected to vortex shedding, as discussed in *1.3.3 Vortex shedding*.

As we have seen in Figure 31, the lift coefficient has a local maximum at about +10° AOA and minimum at about -18°. From Figure 31 we also see that the linearity assumption is accurate enough from about -15° to +5° for the lift coefficient, from about -12° to +5° for the moment coefficient and from about -8° to 0° for the drag coefficient, as discussed in *1.3.2 Wind load coefficients*.

From the histograms in Figure 74 and Figure 75 and statistics in Table 6 we see that a considerable amount of data is outside these ranges, especially for positive AOAs and wind from south-westerly direction, where about 25% of the data showcases AOAs over +5° and about 75% of the data showcase AOAs over 0°. This suggests that while the linearity assumption is accurate enough for negative AOAs most of the time, it creates an overestimation of lift and moment coefficients 25% of the time and underestimation of drag coefficients 75% of the time for winds from south-westerly direction.

Figure 76 shows that the mean AOA gets closer to 0° at higher wind speeds, with most of the data between 0° and +8° for wind speeds above 15 m/s.

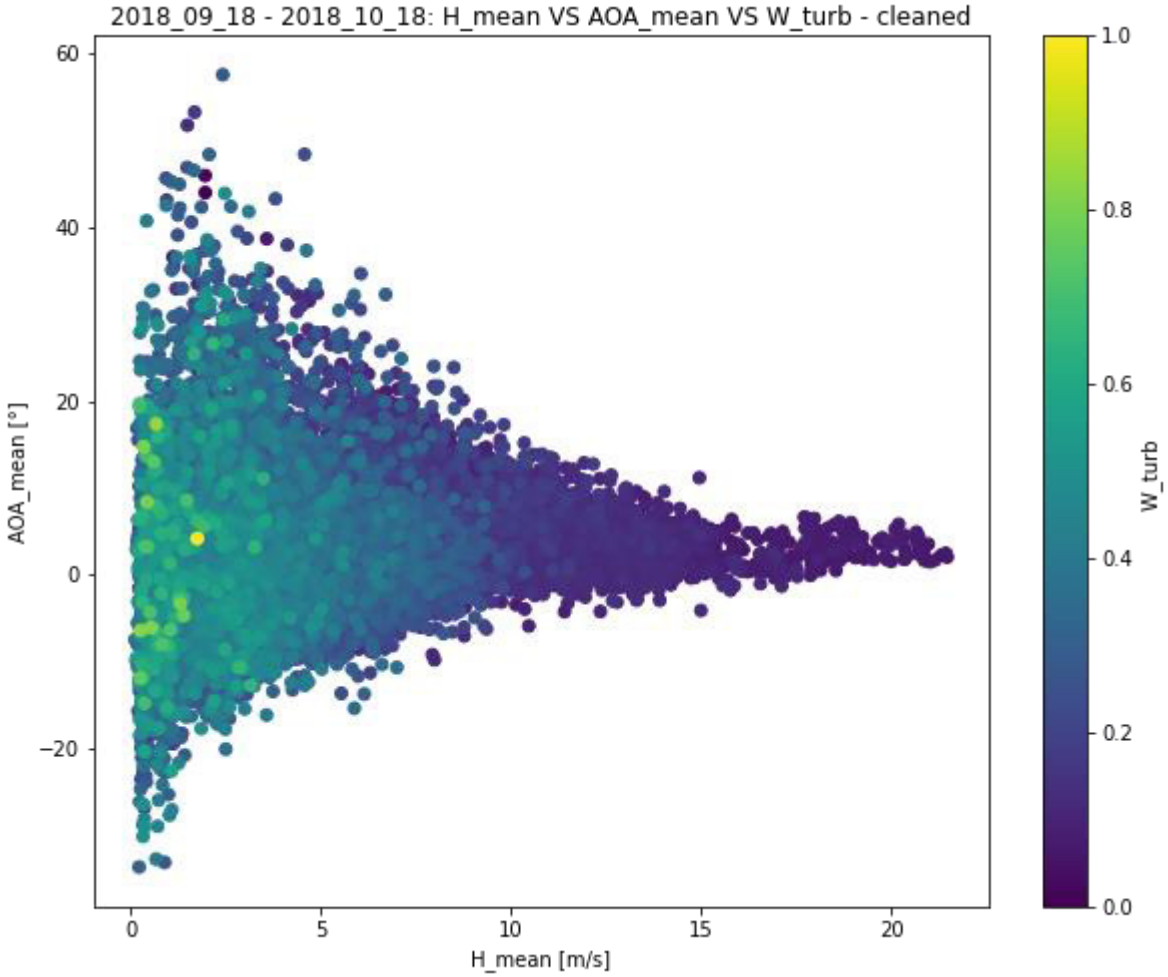


Figure 76: Mean angle of attack and vertical turbulence intensity at different wind speeds

The vertical turbulence intensity, which also decreases with higher wind speeds, is below 0.5 for wind speeds above 10 m/s.

When comparing the mean angle of attack measured by anemometers on the downwind side of the bridge to those on the upwind side, as in Table 5, we see that the mean AOA gets straightened towards 0° as the wind flow passes over the bridge.

Table 7: Statistical comparison of mean angle of attack between upwind and downwind anemometers for south-westerly and north-easterly winds

Statistic on AOA_mean	**south-westerly wind**				**north-easterly wind**			
Anemometer	*H08Wb*	*H08E*	*H18W*	*H18E*	*H08Wb*	*H08E*	*H18W*	*H18E*
Up-/Downwind	*U*	*D*	*U*	*D*	*D*	*U*	*D*	*U*
Mean	4.2	2.18	5.47	3.3	-0.42	2.41	0.39	2.41
Median	4.13	1.75	4.5	1.18	-0.09	2.36	0.22	2.3
Percentile [10, 25, 75, 90]	[-1.29 2.15 6.08 8.93]	[-3.35 -0.39 4.02 8.75]	[0.51 2.27 7.21 12.02]	[-3.54 -0.73 5.37 15.46]	[-4.95 -2.32 1.73 3.62]	[-5.48 -2.15 6.54 10.84]	[-4.11 -1.63 2.04 4.6]	[-5.12 -0.84 5.62 9.67]
Minimum	-17.55	-21.04	-17.74	-21.9	-17.86	-27.67	-25.3	-33.11
Maximum	35.22	30.51	39.49	53.22	26.1	40.72	36.82	45.93
Variance	23.56	26.74	30.28	64.0	13.94	47.37	22.95	45.89
Standard deviation	4.85	5.17	5.5	8.0	3.73	6.88	4.79	6.77

For south-westerly winds all statistics on the mean AOA at the downwind anemometer H08E, except 10th percentile, variance and standard deviation, are lower than at the upwind anemometer H08Wb. At H18 all statistics except 90th percentile, maximum, variance and standard deviation are lower on the downwind anemometer H18E.

For north-easterly winds all statistics on the mean AOA at the downwind anemometer H08Wb, except minimum and 25th percentile, are lower than at the upwind anemometer H08E. At H18 all statistics are lower on the downwind anemometer H18W. Note how the variance increases on the downwind side for south-westerly winds but decreases for north-easterly winds.

Comparing the AOA wind roses for H08E and H08Wb in Figure 77 illustrates how more extreme positive AOAs (orange-red) on the upwind side become more neutral (grey-orange) on the downwind side, as the wind follows the shape of the bridge girder. At H18E an anomaly to this occurs, showing a large population of high positive AOAs (red) from direct south to east.

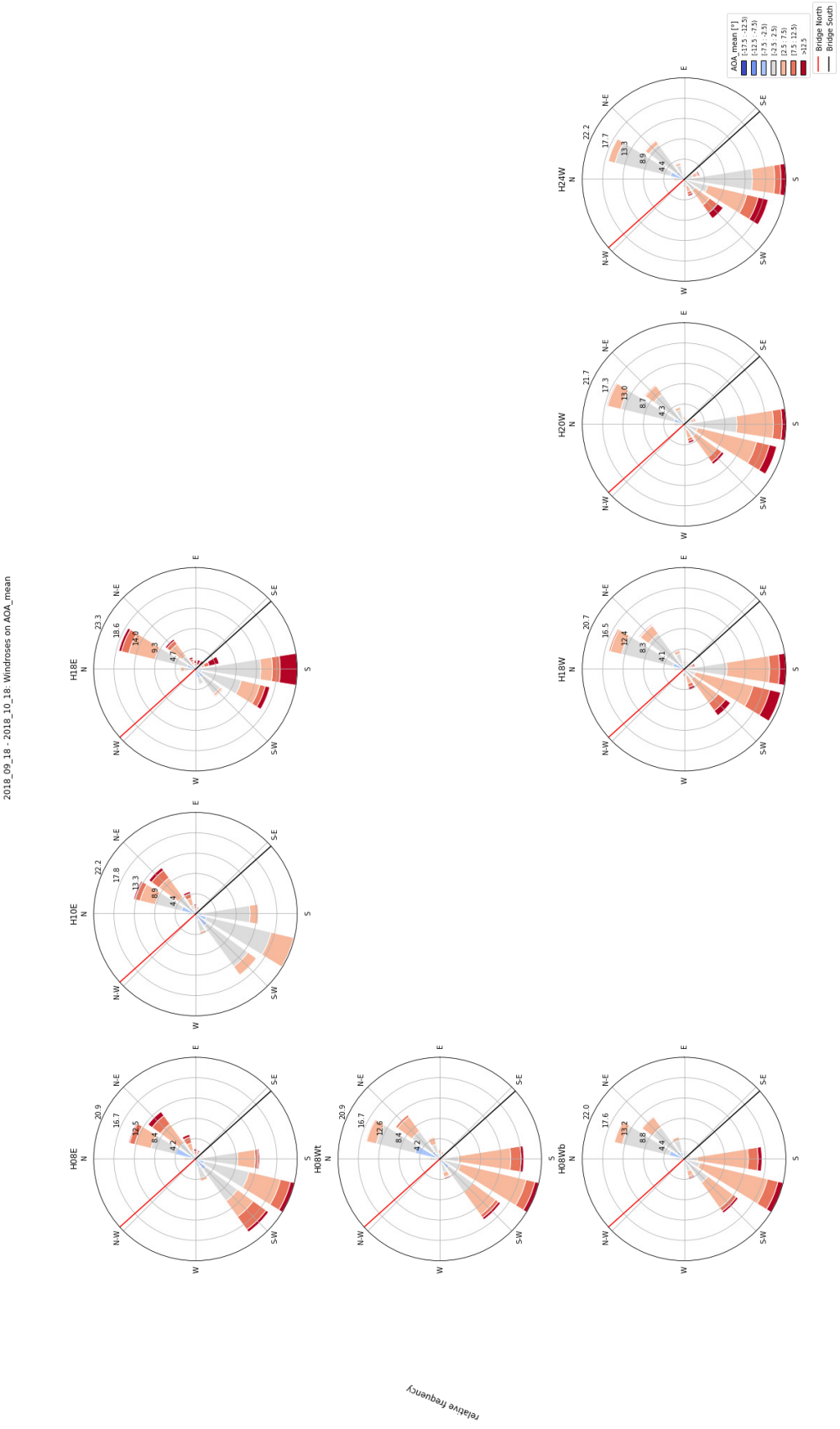


Figure 77: Mean angle of attack wind roses per anemometer, arranged by position on the bridge

This anomaly can be further examined using the polar scatterplots depicted in Figure 78. Note that the datapoints with the highest values on the color-axis, AOA_mean, are plotted on top of datapoints with lower mean angles of attack. Therefore, datapoints with negative AOAs are not very visible in this kind of plot, as they are hidden below datapoints with high AOAs. However, for a closer examination of the before mentioned anomaly at H18E the datapoints with extreme high mean AOAs of +15° or higher (dark red) are of interest. There is a large population of these datapoints in the sector between 0° and 225° with wind speeds mostly below 5 m/s in the sector between 0° and 90° and reaching 10 m/s in the sector between 90° and 225°. It should be noted that we can also observe a population of high AOA datapoints in the sector between 180° and 225° with wind speeds reaching 10 m/s at H08E, but not at H10E. There is also a smaller population of high AOA datapoints in the sector between 0° and 90° with wind speeds below 5 m/s at both H08E and H10E. However, there is no population of high AOA datapoints in the sector between 135° and 180° at both H08E and H10E.

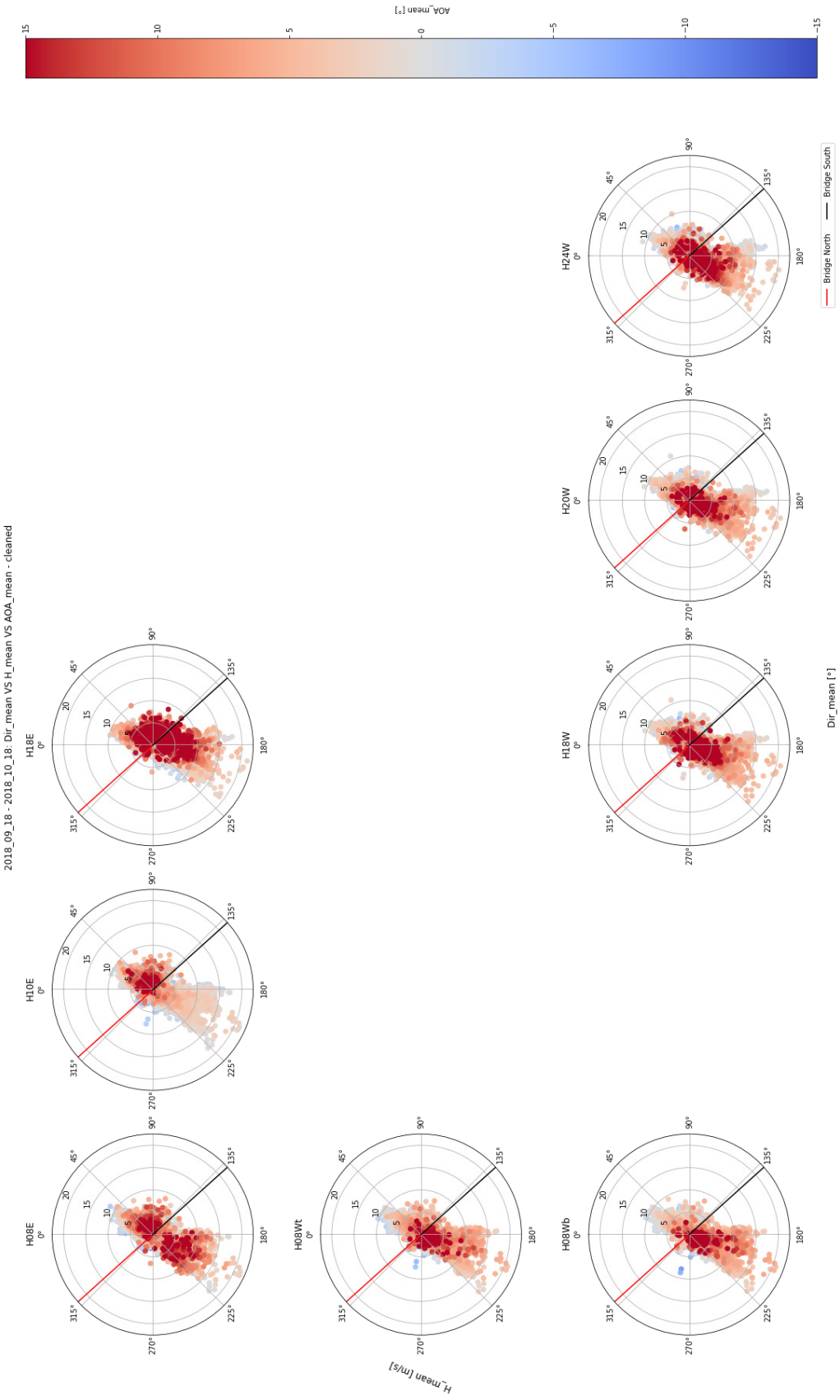


Figure 78: Polar scatterplots on mean angle of attack and horizontal wind speed per anemometer, arranged by position on the bridge

The reasons for this anomaly cannot be explained from this analysis. A miss-alignment of the sensor has been ruled out by a recent examination of the sensor. The scatterplot in Figure 79 does not suggest that the high AOAs in these sectors are caused by un-organized turbulent flow, as the vertical turbulence intensity is mostly below 0.4 in the region of interest at H18E.

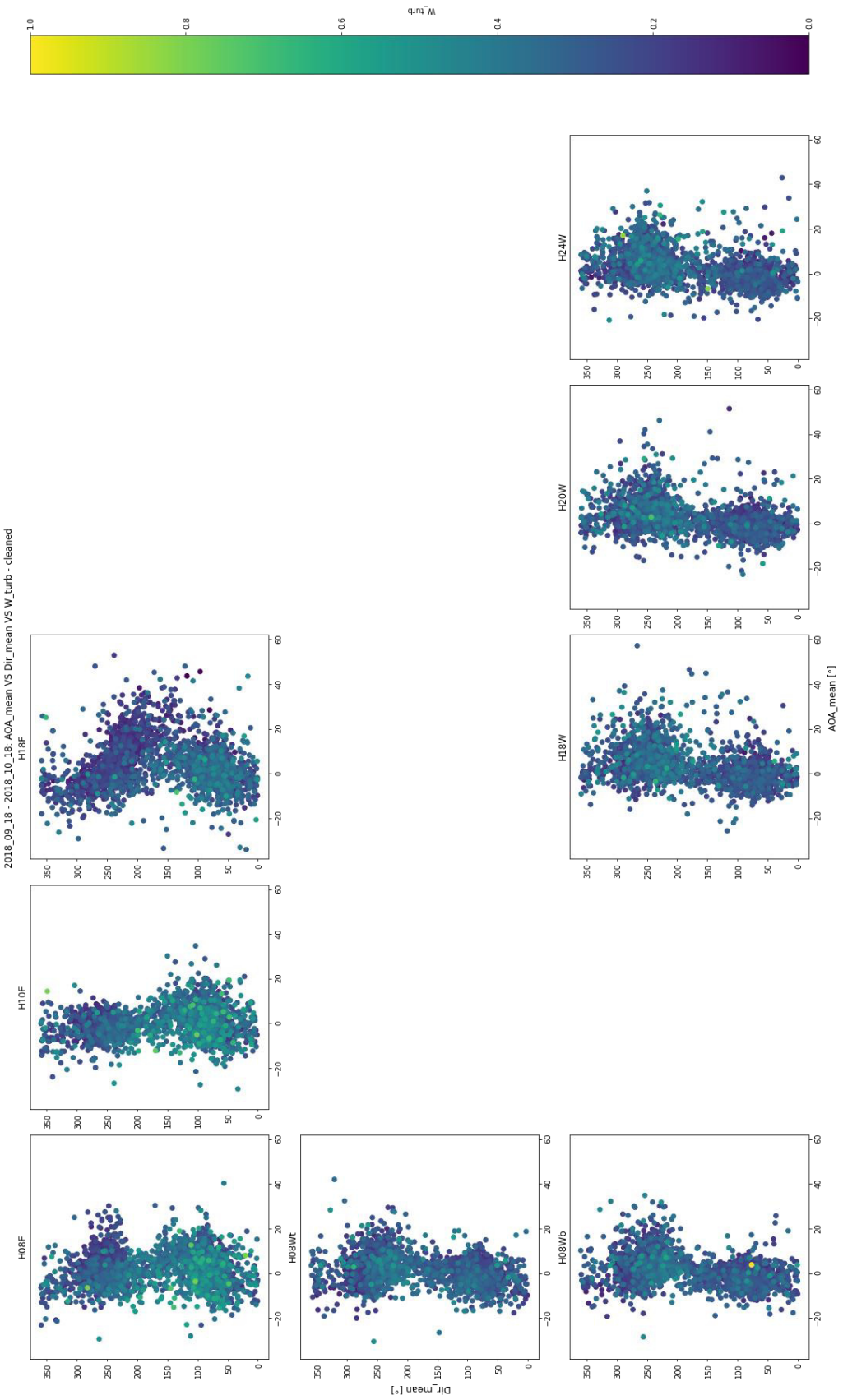


Figure 79: Scatterplots on mean angle of attack and vertical turbulence intensity from different directions per anemometer, arranged by position on the bridge

The author therefore hypothesizes, that the narrow cross-section of the fjord at the point where the bridge crosses the fjord acts as a bottleneck for the wind flow into and out of the fjord, which creates an updraft and re-circulation region just east of the bridge, which gets captured by the anemometer at H18E, but not the other anemometers. Figure 80 and Figure 81 illustrate how such a flow could look like for north-easterly wind.

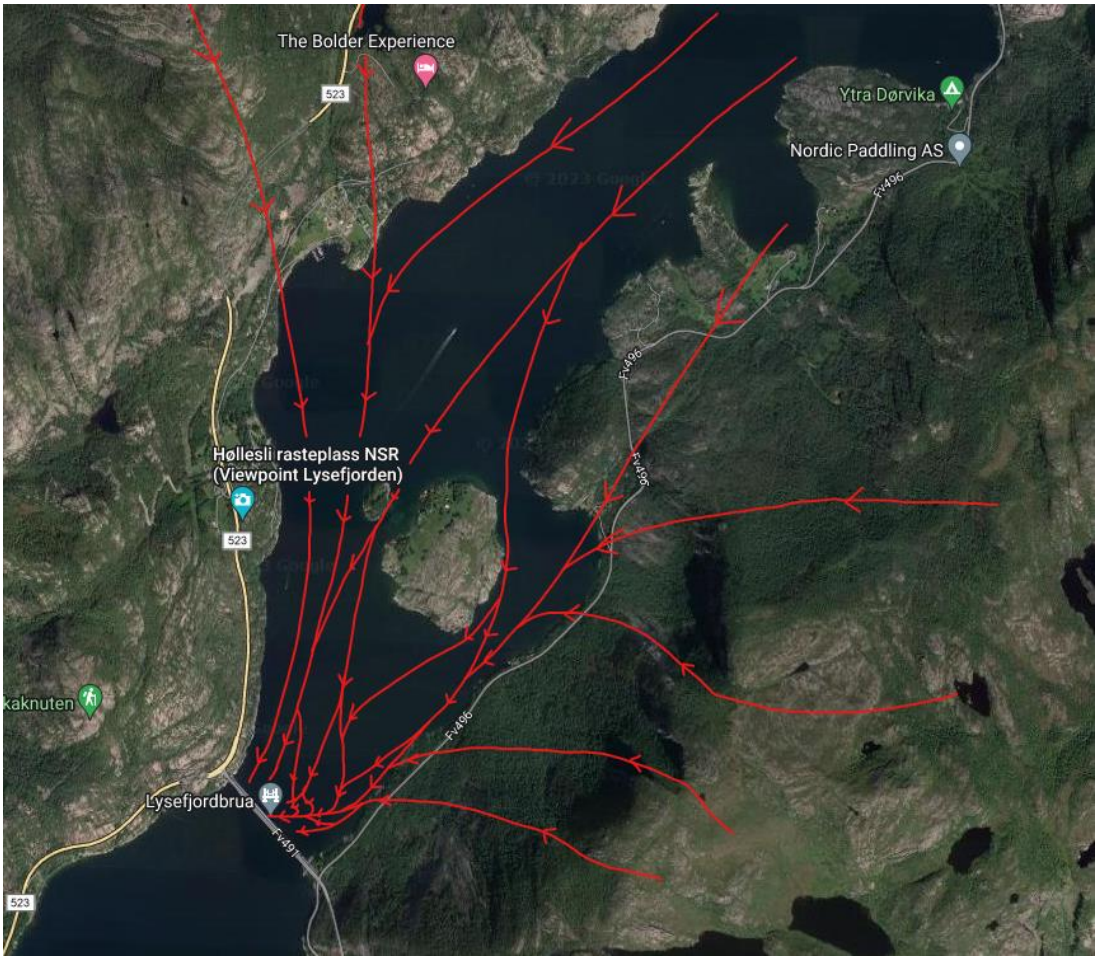


Figure 80: Hypothesis on wind flow causing high angles of attack at H18E from east direction, top view. Adapted from [49]

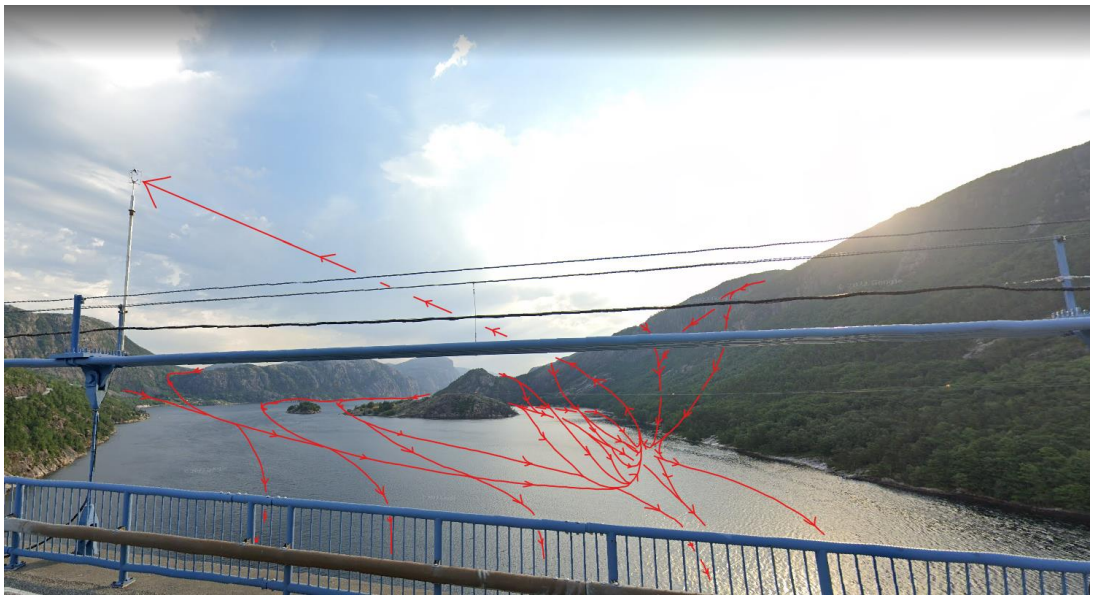


Figure 81: Hypothesis on windflow causing high angles of attack at H18E from east direction, view from the West. Adapted from [50]

Note that the high AOAs from the sector between 135° to 180° are not well explained by this hypothesis, as the wind flow must cross the bridge or come from the direction of the south tower. Another hypothesis is that a downdraft from the hills southeast of the bridge hits the bridge deck and is reflected upwards through the anemometer at H18E. Additional anemometers at H20E and H24E could help in examining this anomaly. Another approach could be a CFD simulation similar to [18], to examine the origin of this anomaly.

Figure 82 shows the correlations between *H_mean* and the absolute *AOA_mean*.

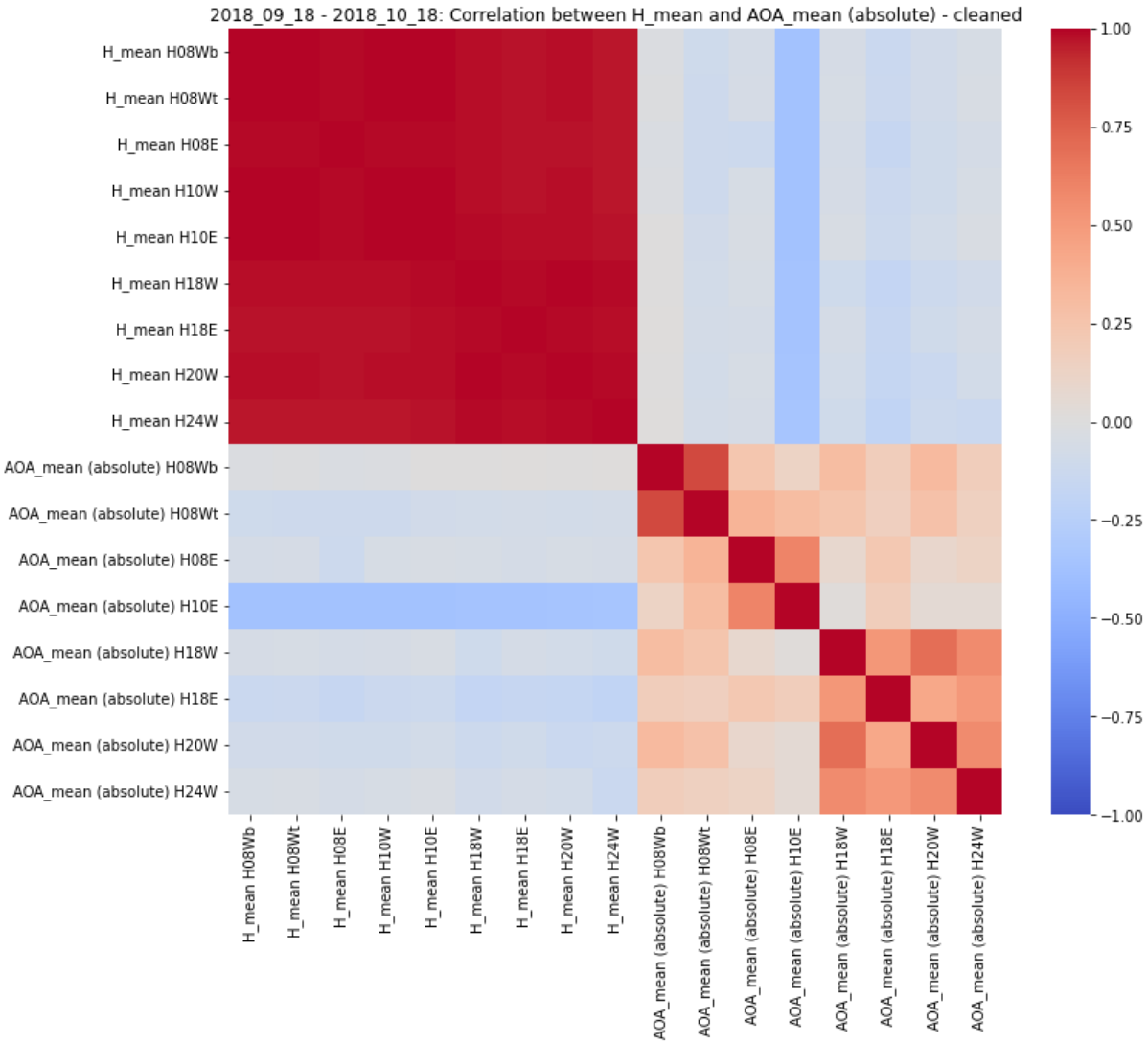


Figure 82: Correlation matrix between mean horizontal wind speed and absolute mean angle of attack per anemometer

From Figure 82 we see that the absolute mean AOA is not correlated very much across the anemometers. The absolute mean AOA at H08Wb and H08t are correlated relatively well with a correlation coefficient of about 0.8. However, their correlation to H08E across the bridge deck is relatively low at about 0.25, which is not significantly higher than the correlation to the other anemometers along the span of the bridge. Similar observations can be made for the other anemometers. The correlation to anemometers on the same side of the bridge is slightly higher, compared to the correlation to anemometers across the bridge deck. For anemometers on the same side of the bridge deck the further away another

anemometer is, the lower their correlation. The correlation to *H_mean* is slightly negative, mostly between -0.25 and 0, except for H10E, where it is about -0.5.

Figure 83 shows the correlation between absolute *AOA_mean* and *H_turb*.

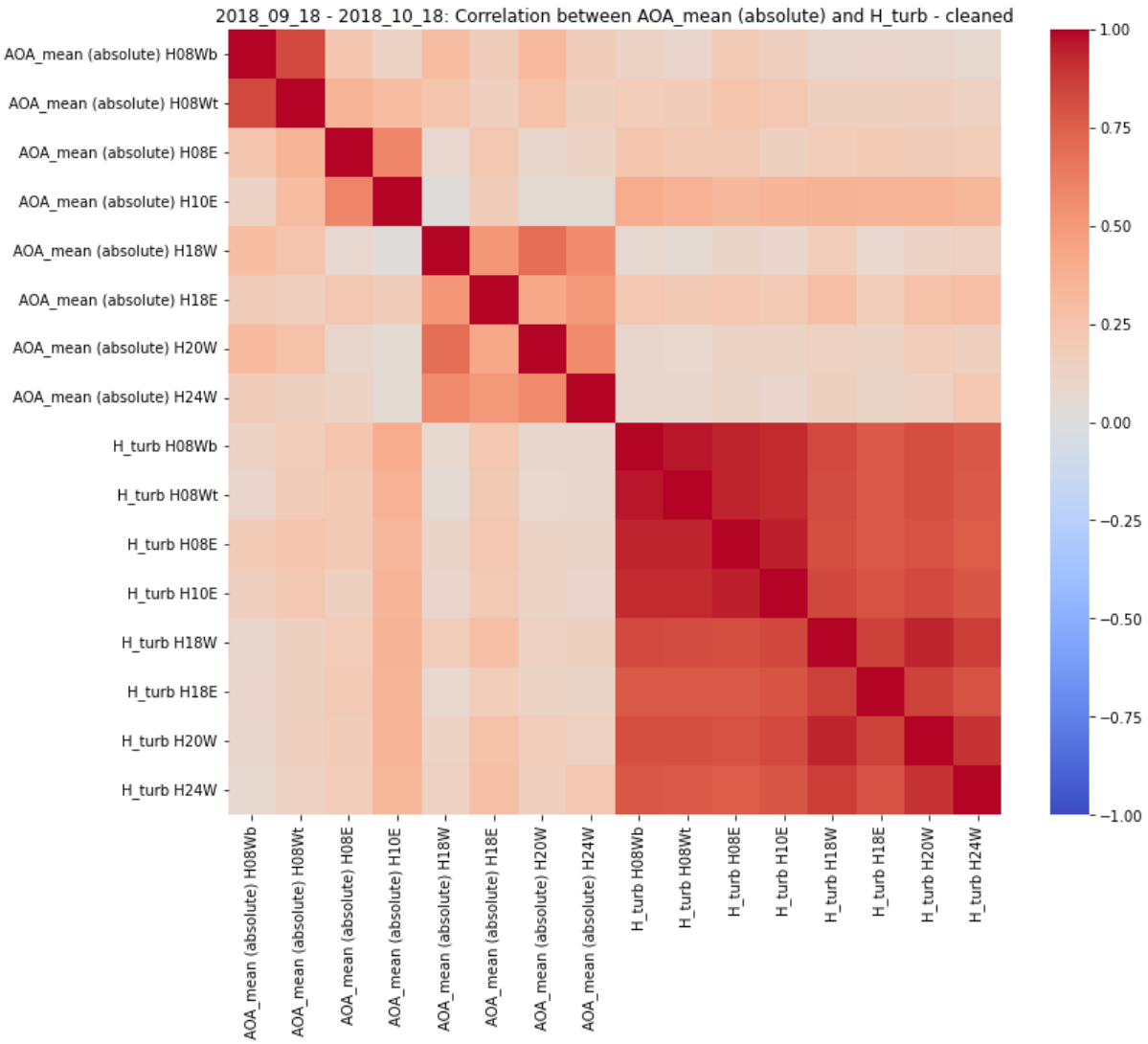


Figure 83: Correlation matrix between absolute mean angle of attack and horizontal turbulence intensity per anemometer

From Figure 83 we see that the correlation between absolute mean AOA and horizontal turbulence intensity is relatively low, but slightly positive between 0 and 0.25 for most anemometers, except for H10E again, where the correlation is slightly higher at about 0.35. Note that correlation does not imply causation and therefore it is not possible to explain this anomaly from these correlation matrices.

4.2 Bridge response

In the following the code of this work is used to examine the response of the bridge to traffic and wind.

4.2.1 Traffic response

Note that the naming error for accelerometer pair H24 mentioned in *1.2.3 Instrumentation Layout* might be carried over in some of the illustrations below. Any mentioning of H20 in reference to an accelerometer is to be interpreted as H24.

Figure 84 shows the maximum vertical acceleration for each 10-minute data package at the centreline of the bridge over the month-long period of recorded data.

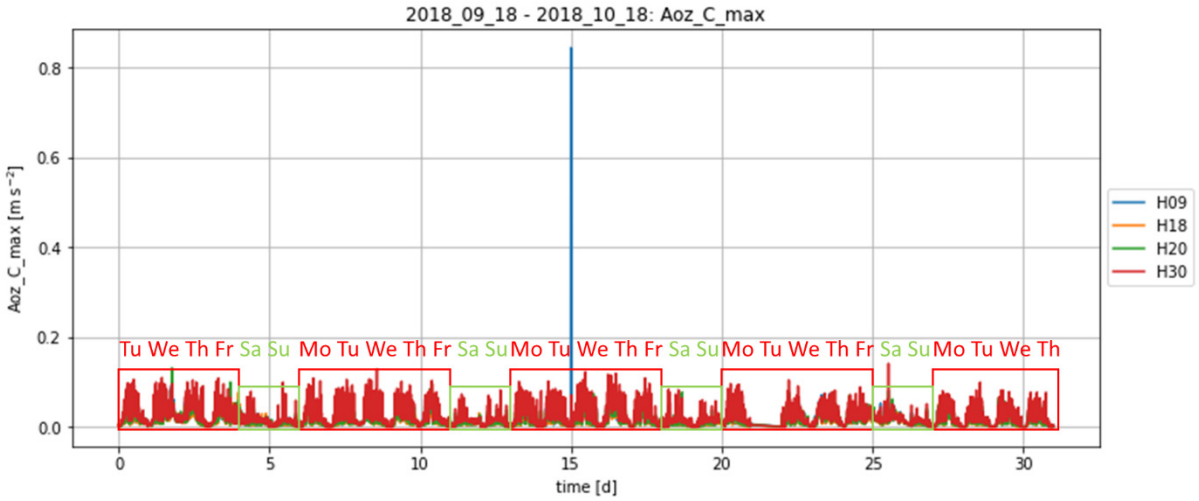


Figure 84: Maximum central vertical accelerations measured at the Lysefjord bridge during 30-day period showing periodic response to traffic

A clear periodic behaviour is visible with a period of one day. This is because more traffic is crossing the bridge during the day than at night. Another clear observation is that after five days of higher maximum accelerations two days of lower maximum accelerations follow. This can respectively be linked to workdays, during which heavy trucks from the Norsk Spennbetong AS concrete plant or NCC Helle sandtak sand pit cross the bridge, and weekend days, during which the traffic primarily consists of lighter cars. As expected by this hypothesis, the first day of data, 18th September 2018, is a Tuesday while the last day, 18th October 2018, is a Thursday. Note that there is an outlier at exactly 15 days at H09, caused by the east accelerometer, which has not been recognised as invalid as it is still inside the operational limits of the accelerometer of ±5g. [23]

4.2.2 Lateral wind response

In Figure 85 to Figure 92 the lateral wind response of the bridge, measured by the standard deviation of acceleration in x-direction *Aox_C_std* is visualised and analysed.

Figure 85 shows a scatterplot of *Aox_C_std* over the mean horizontal wind component perpendicular to the primary bridge axis *Vx_mean* with the respective turbulence intensity *Vx_turb* on the color-axis.

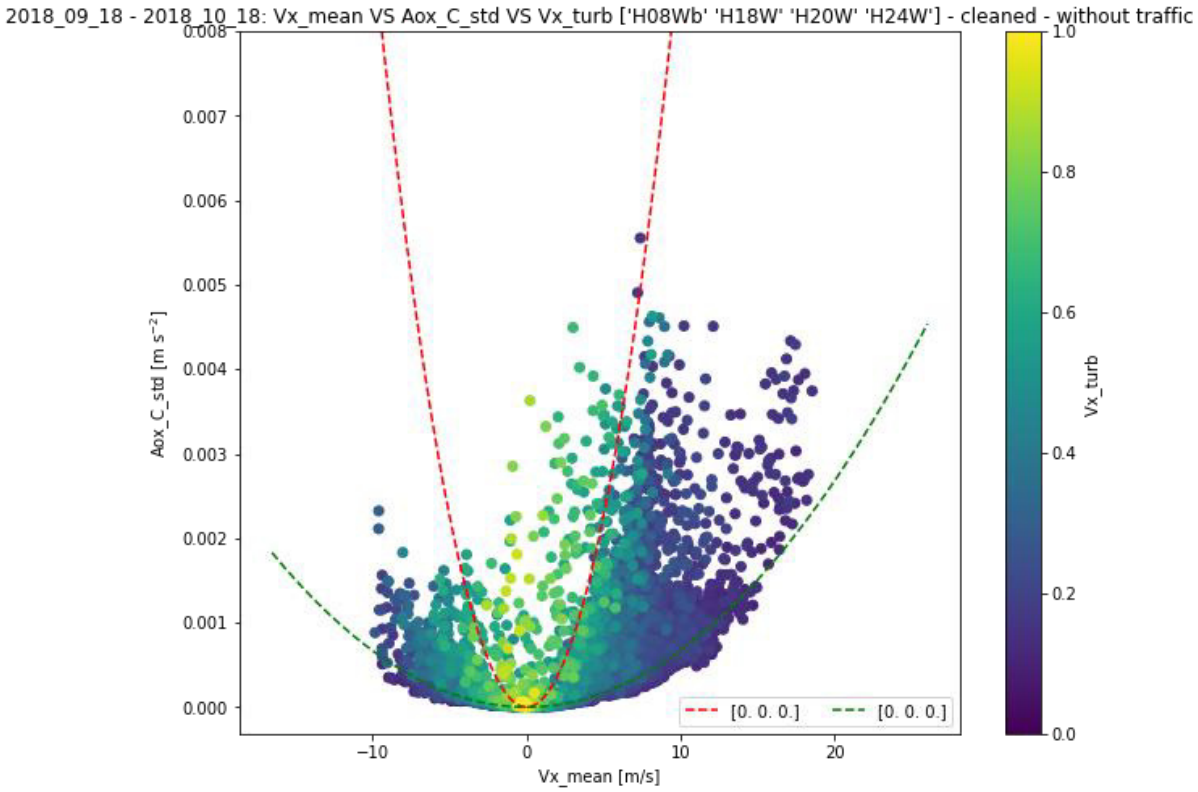


Figure 85: Lateral bridge response and turbulence intensity at different mean wind speeds

Note that positive *Vx* corresponds to wind from south-westerly direction and a negative *Vx* to wind from north-easterly direction, matching the bridge coordinate system in Figure 24. A red parabola through the origin is matched to datapoints with *Vx_turb* > 0.6 and a green parabola through the origin is matched to datapoints with *Vx_turb* < 0.1. These parabolas create an envelope into which most of the datapoints fall. Note how higher standard deviations in the accelerations up to about 0.005 m/s² can occur even at mean wind speeds below 10 m/s, especially under higher turbulence intensity above 0.5.

By normalizing the standard deviation of acceleration by the turbulence intensity we try to reduce the factor of turbulence intensity, expecting to fold the data in on itself. Figure 86 and Figure 87 showcase this for south-westerly and north-easterly wind conditions respectively, split across the accelerometer pairs at H09, H18, H24 and H30. The wind measurements in these plots are taken from the respective downwind anemometer at midspan H18.

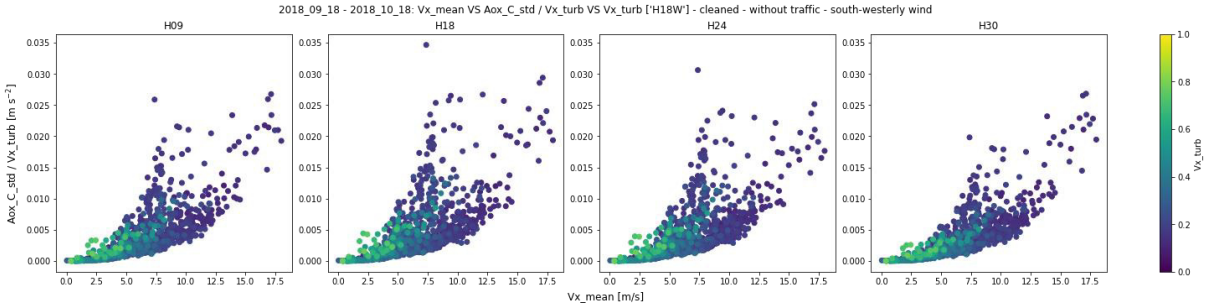


Figure 86: Lateral bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for south-westerly winds per accelerometer pair

As expected, the data is folded in on itself and bound by an upper and lower parabola shape. However, there is still a spread occurring with higher wind speeds.

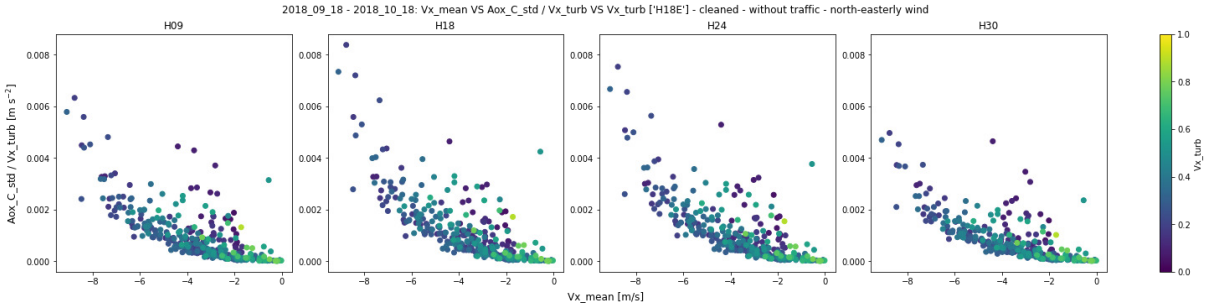


Figure 87: Lateral bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for north-easterly winds per accelerometer pair

Similar observations are to be made for the north-easterly winds, although the distribution of datapoints is more chaotic, especially at wind speeds between -2 m/s and -5 m/s. We can assume that this is due to other effects the terrain has on the wind flow from that direction, especially on the vertical wind component, like the anomaly at H18E discussed in *4.1.3 Angle of attack*.

Figure 88 shows the lateral bridge response at different mean AOAs with the mean horizontal wind speed decoded on the color-axis.

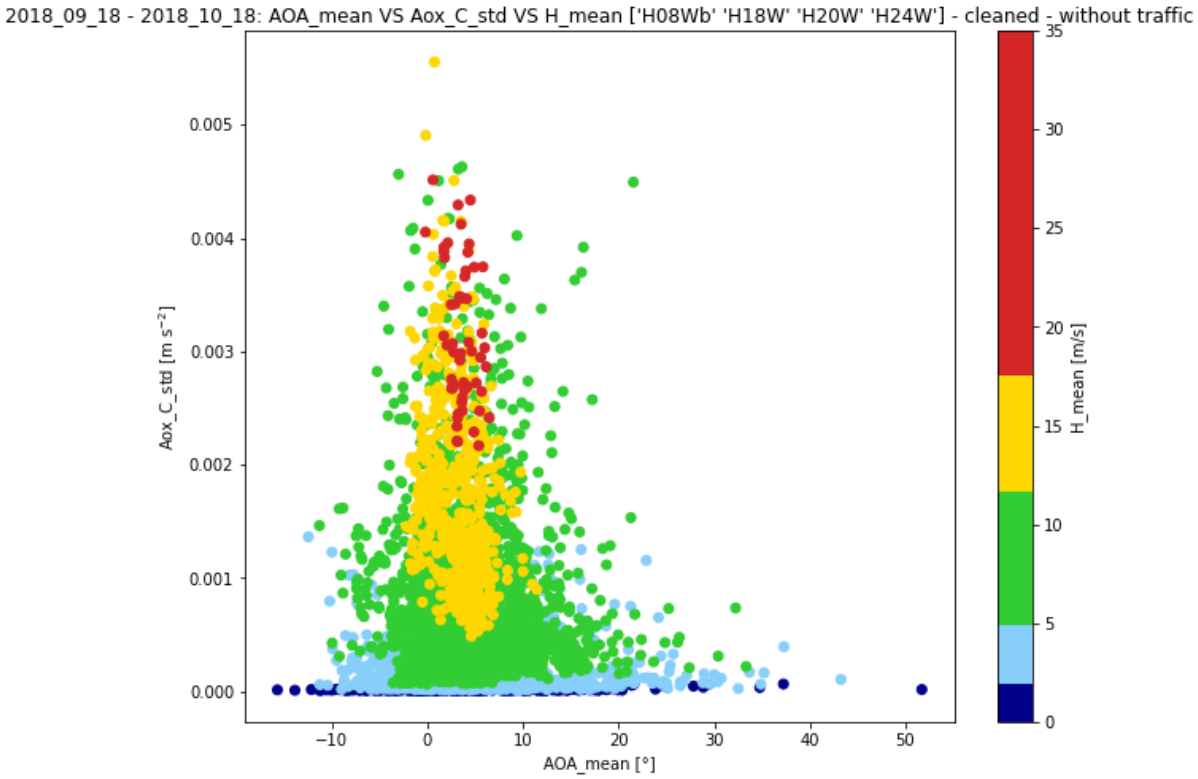


Figure 88: Lateral bridge response and mean wind speed at different angles of attack

From Figure 88 we again see that larger lateral responses above 0.002 m/s² can be observed even at wind speeds below 11 m/s at AOAs between -5° to +20°. Wind speeds above 11 m/s occur only in a relatively narrow AOA band between -1° and +10°. As discussed in *1.3.2 Wind load coefficients* and showcased in Figure 31, the linearity assumption for the drag coefficient is accurate at about -8° to 0°. We see that most of the datapoints at wind speeds above 4 m/s causing large lateral responses are above 0°. This suggests that the linearity assumption for the drag coefficient is not suitable to predict large lateral responses, as they mostly occur at higher AOAs, even at lower wind speeds, where the linearity assumption underestimates the lateral wind response.

Normalizing the lateral bridge response by turbulence intensity again results in Figure 89.

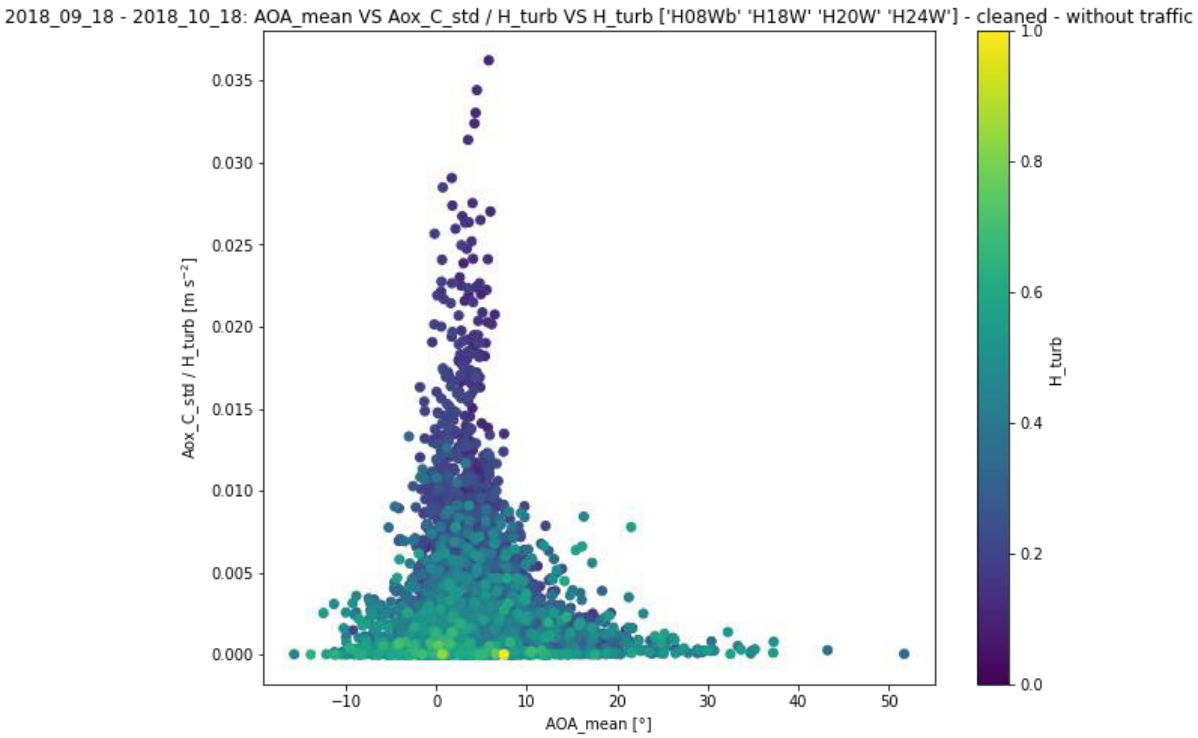


Figure 89: Lateral bridge response normalized by turbulence intensity and turbulence intensity at different angles of attack

We see that some of the higher bridge responses at lower mean wind speeds and at angles of attack further out from 0° are brought down in the normalized scatterplot, as they were more correlated with higher turbulence intensity.

Figure 90 shows the correlation between *H_mean* and *Aox_C_std*.

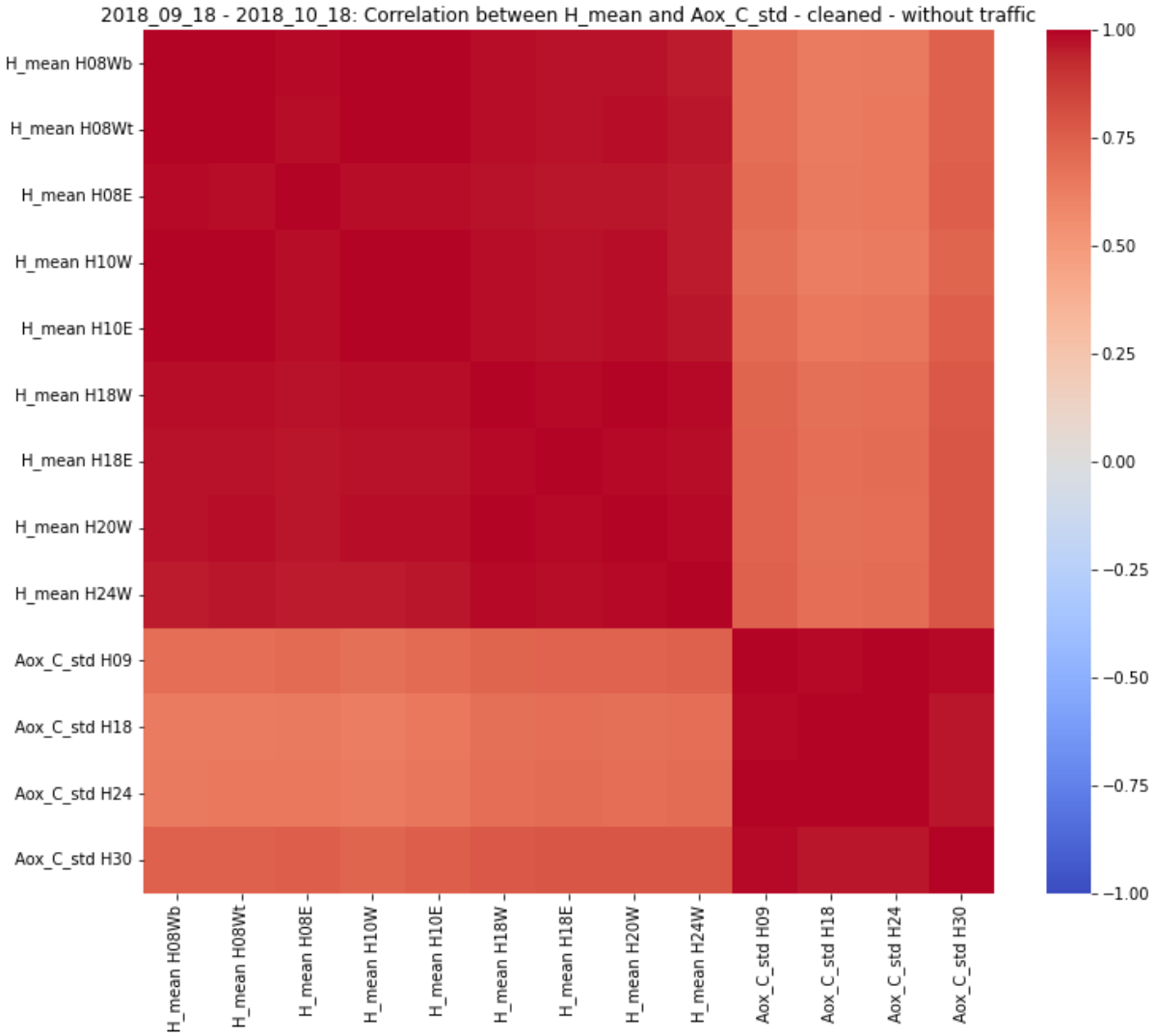


Figure 90: Correlation between mean wind speed and lateral bridge response per anemometer and accelerometer

From Figure 90 we see that the correlation coefficient between the accelerometers on the lateral response is close to 1, displayed as homogeneous dark red color in the lower right quadrant. The correlation between accelerometers H18 and H24 is slightly higher compared to the correlations with and between the other accelerometers, as they are located closer together. The correlation to the mean horizontal wind velocity is also relatively high at about 0.5 to 0.75. Counterintuitively the correlation of lateral response at midspan H18 and H24 to mean horizontal wind velocity is slightly lower compared to the lateral response measured by accelerometers at H09 and H30 closer to the towers. This suggests that the lateral response at the midspan of the bridge, where the main cables are at the lowest point and the hanger length is the

shortest, is less sensitive to higher wind speeds, compared to further out towards the towers. However, this does not account for other factors on the lateral response, explaining how an eigenmode shape like the first and second symmetric horizontal eigenmode shapes *HS1* and *HS2* identified by E. Cheynet in [20, p. 126] which are depicted in Figure 30, can occur. Also note that the data might be a combination of symmetric and asymmetric eigenmode shapes. In the later the acceleration at midspan is close to zero and higher at H09 and H30.

Figure 91 shows the correlation between *H_turb* and *Aox_C_std*.

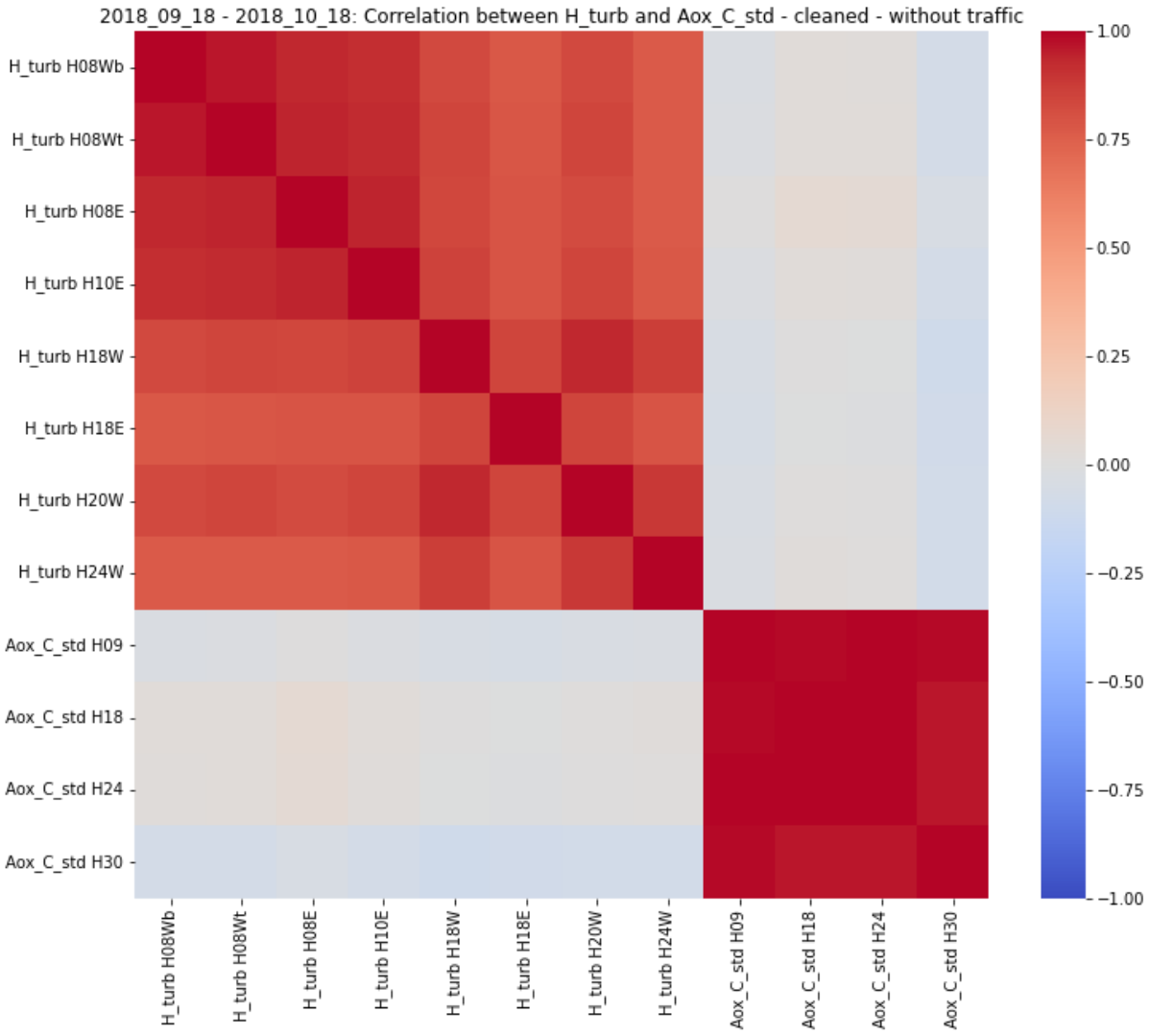


Figure 91: Correlation turbulence intensity and lateral bridge response per anemometer and accelerometer

From Figure 91 we see that the correlation between the lateral response and horizontal turbulence intensity is close to 0, slightly negative at H09 and H30. No significant assumption can be drawn from these correlations.

Figure 92 shows the correlation between absolute *AOA_mean* and *Aox_C_std*.

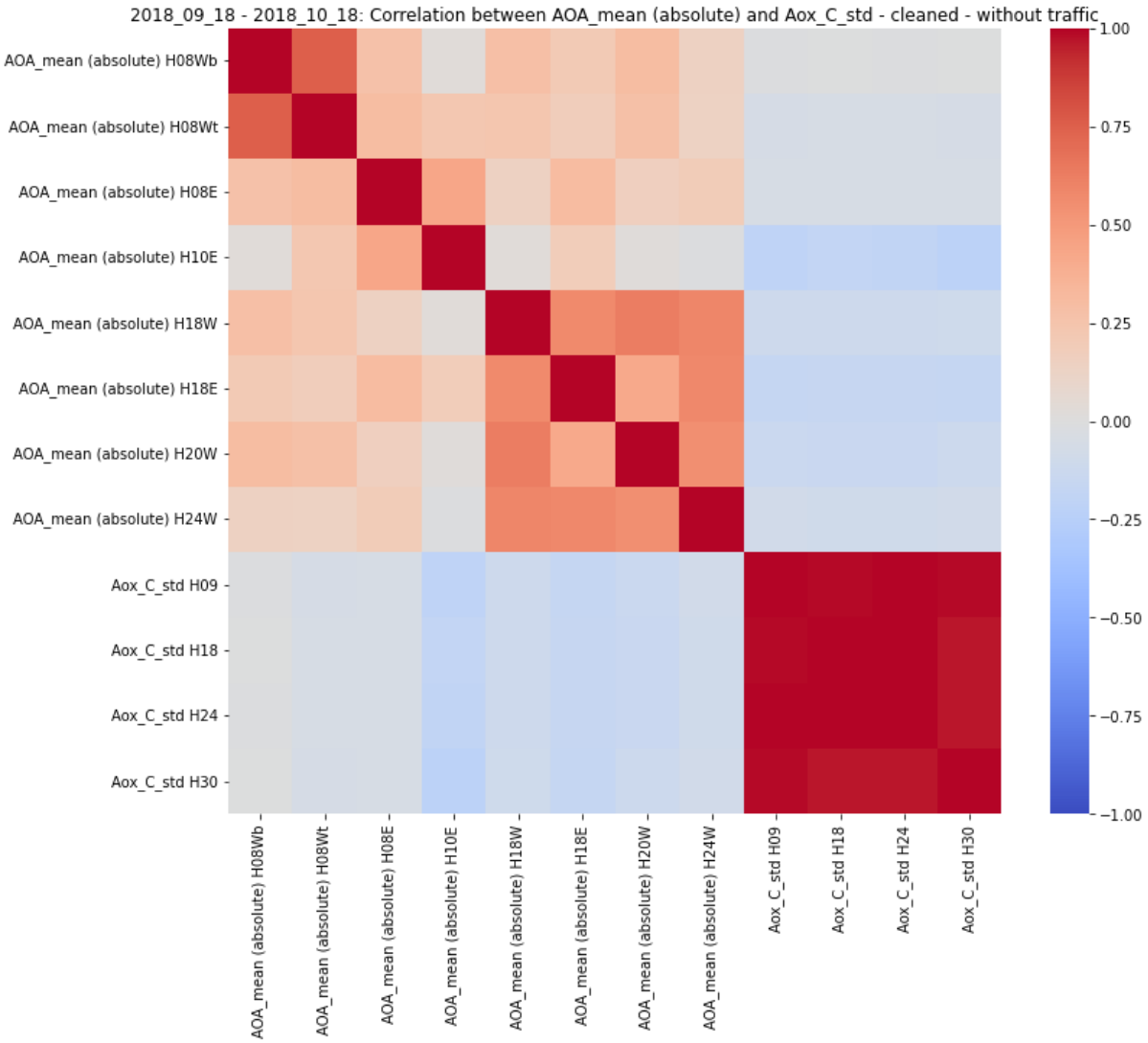


Figure 92: Correlation between absolute mean angle of attack and lateral bridge response per anemometer and accelerometer

From Figure 92 we see that the correlation between the lateral response and absolute mean AOA is close to 0, or slightly negative for anemometers at H10E, H18E and H20W, but below 0.25. This coincides with the anomaly detected in Figure 82 and discussed in *4.1.3 Angle of attack*. No further significant assumptions can be drawn from these correlations.

4.2.3 Vertical wind response

In Figure 93 to Figure 101 the vertical wind response of the bridge, measured by the standard deviation of acceleration in z-direction *Aoz_C_std* is visualised, analysed and compared to the lateral wind response discussed in *4.2.2 Lateral wind response*. Figure 85 shows a scatterplot of *Aoz_C_std* over the mean horizontal wind component perpendicular to the primary bridge axis *Vx_mean* with the respective turbulence intensity *Vx_turb* on the color-axis, similar to Figure 85.

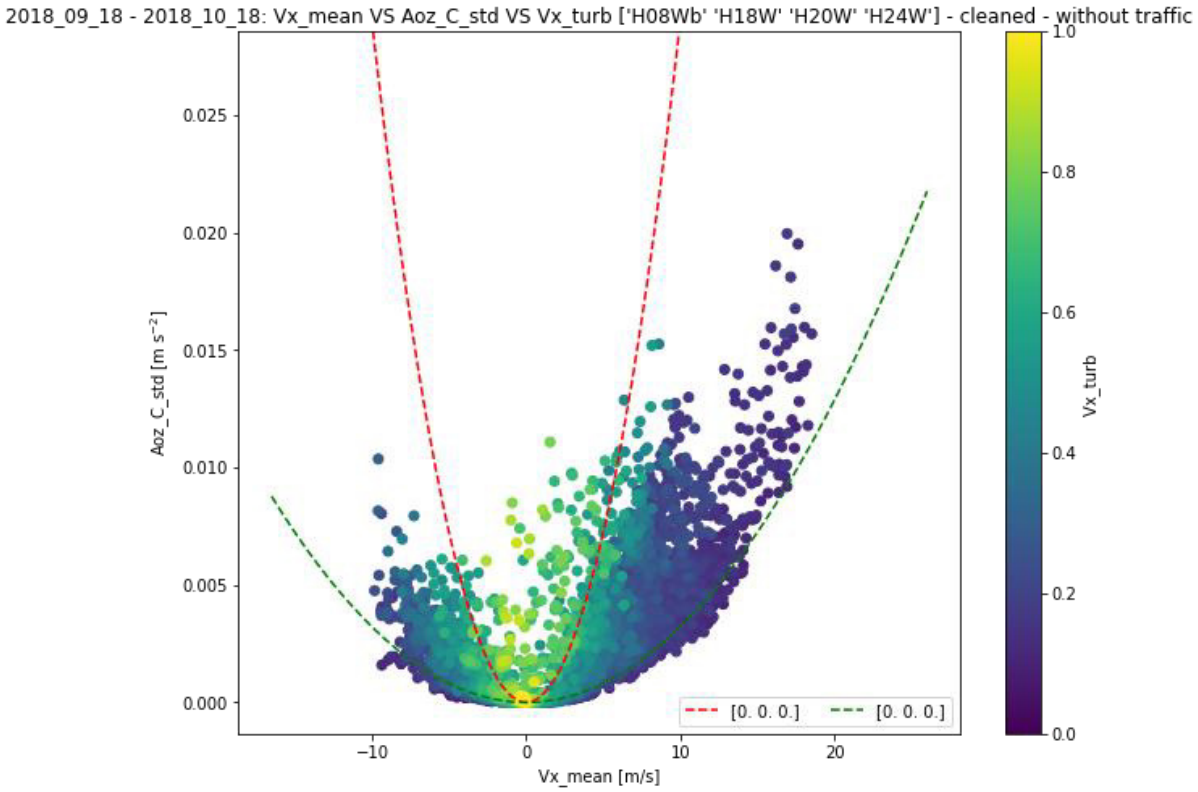


Figure 93: Vertical bridge response and turbulence intensity at different mean wind speeds

The highest standard deviation of vertical acceleration is measured at about 0.02 m/s² at a perpendicular wind speed of about 18 m/s and low turbulence intensity. This is about 3.5 to 4 times higher than the highest measured standard deviation of lateral acceleration. At about 8 m/s *Aoz_C_std* is measured at about 0.015 m/s² at a turbulence intensity of about 0.6. At about 2 m/s *Aoz_C_std* is measured at about 0.011 m/s² at a turbulence intensity of about 0.8.

Figure 94 and Figure 95 showcase *Aoz_C_std / Vx_turb* for south-westerly and north-easterly wind conditions respectively, split across the accelerometer pairs at H09, H18, H24 and H30, similar to Figure 86 and Figure 87.

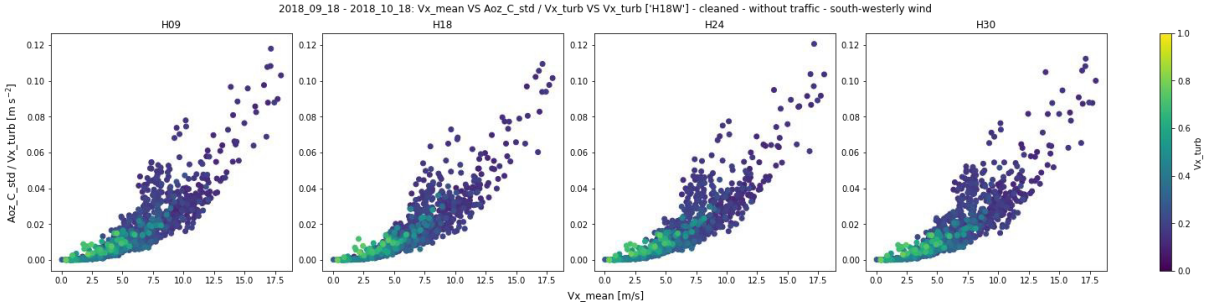


Figure 94: Vertical bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for south-westerly winds per accelerometer pair

As expected, the data is folded in on itself and bound by an upper and lower parabola shape. The remaining spread occurring with higher wind speeds is less than for the lateral response in Figure 86. The higher vertical response we are seeing between about 7 m/s and 12 m/s could be explained by vortex shedding, as described in *1.3.3 Vortex shedding*.

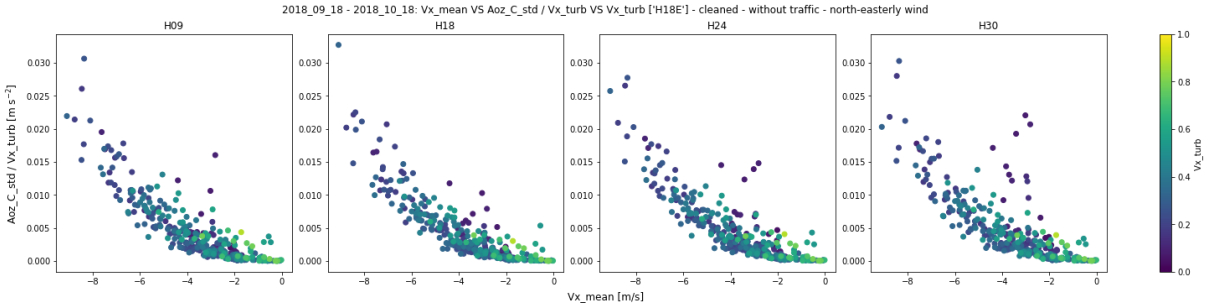


Figure 95: Vertical bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for north-easterly winds per accelerometer pair

As in the case for lateral response in Figure 87, similar observations are again to be made for the north-easterly winds for the vertical response. The distribution of datapoints is again more chaotic, at wind speeds between -2 m/s and -5 m/s, especially towards the south of the bridge at H30.

Figure 96 shows the vertical bridge response at different mean AOAs with the mean horizontal wind speed decoded on the color-axis, similar to Figure 88.

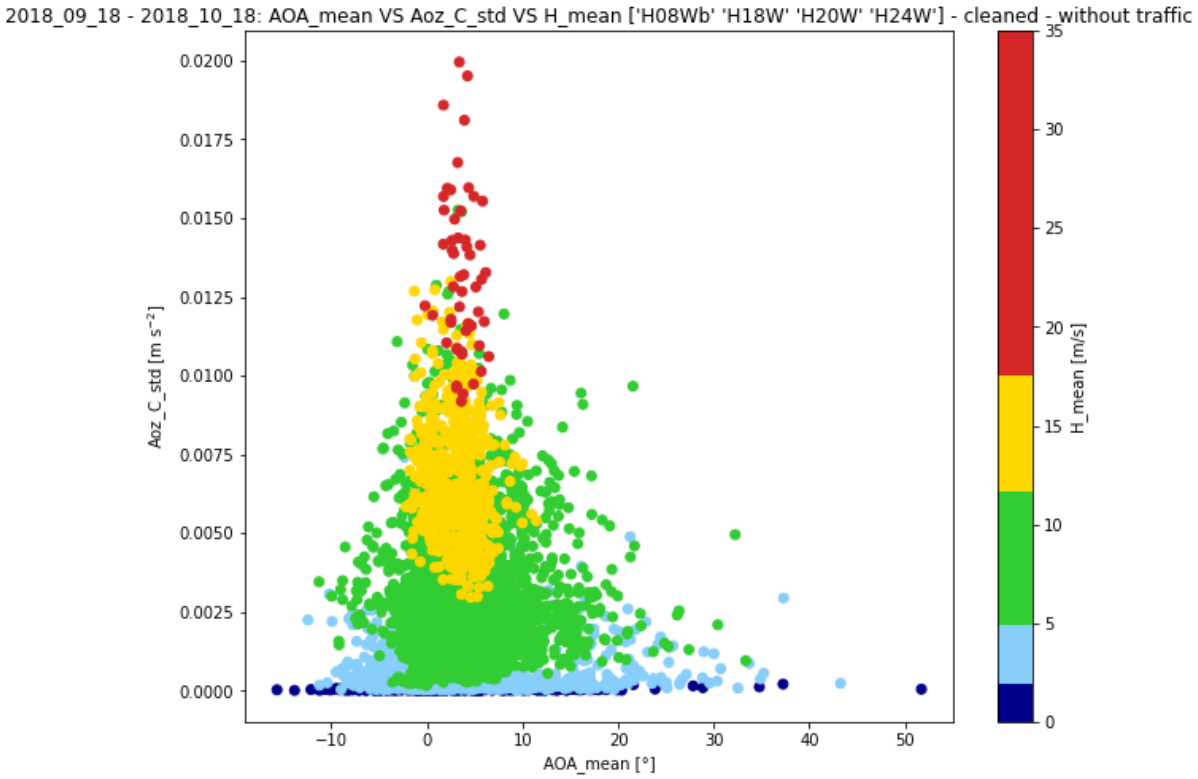


Figure 96: Vertical bridge response and mean wind speed at different angles of attack

From Figure 96 we see that larger vertical responses above 0.0125 m/s² and up to about 0.02 m/s² can be observed primarily at wind speeds above 17 m/s at AOAs between +1° to +8°. Wind speeds between 11 m/s and 17 m/s only result in a response from 0.0025 m/s² up to 0.0125 m/s² at a wider AOA band between -2° and +10°. This contrasts with the lateral wind response, where the wind speed had a lesser impact on the observable wind response. As discussed in *1.3.2 Wind load coefficients* and showcased in Figure 31, the linearity assumption for the lift coefficient is accurate at about -15° to +5°. We see that the datapoints at wind speeds above 11 m/s causing large vertical responses are partially below +5° and partially above +5°.This suggests that the linearity assumption for the lift coefficient is suitable to predict some of the large vertical responses at lower AOAs correctly, but overestimates some of the large vertical responses for AOAs above +5°.

Normalizing the lateral bridge response by turbulence intensity again results in Figure 97, similar to Figure 89.

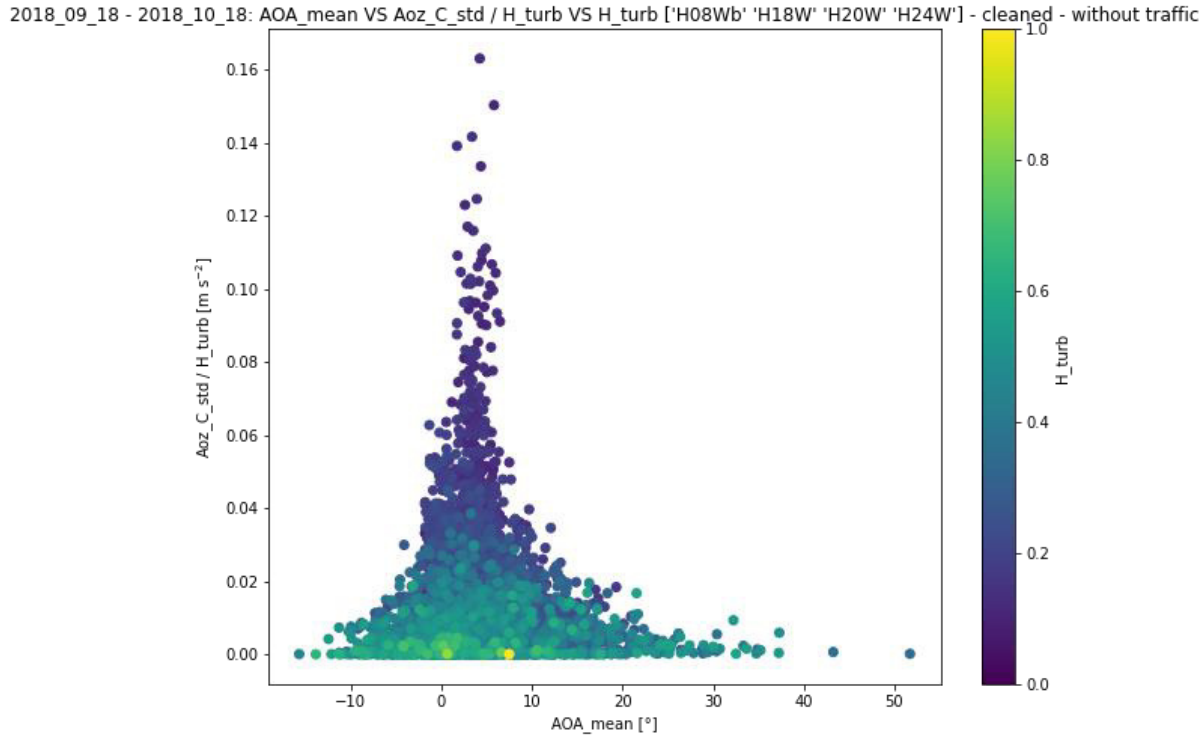


Figure 97: Vertical bridge response normalized by turbulence intensity and turbulence intensity at different angles of attack

Again we see that some of the higher bridge responses at lower mean wind speeds and at angles of attack further out from 0° are brought down in the normalized scatterplot, as they are more correlated with higher turbulence intensity.

Figure 98 shows the correlation between *H_mean* and *Aoz_C_std*, similar to Figure 90.

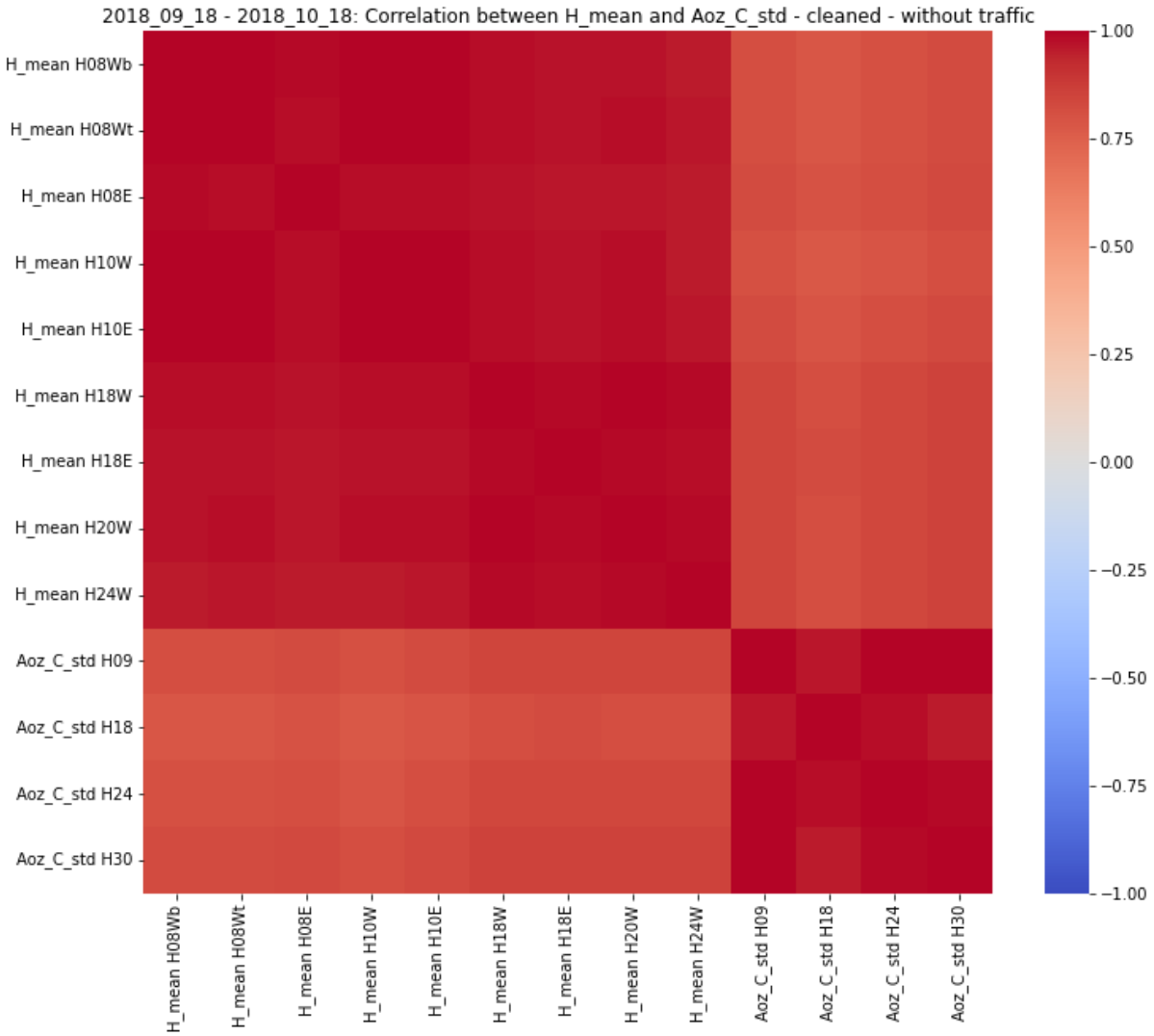


Figure 98: Correlation between mean wind speed and vertical bridge response per anemometer and accelerometer

From Figure 98 we see that the correlation coefficients between all the accelerometer pairs on the vertical response is close to 1, displayed as dark red color in the lower right quadrant. The correlation between accelerometers H09, H24 and H30 is slightly higher compared to the correlations with H18. This is most likely due to the vertical eigenmode shapes depicted in Figure 30, in which the bridge oscillates less at midspan H18, compared to further out towards the north or south, H09 or H24 and H30 respectively. The correlation between H18 and H24 is still higher than the correlation between H18 and H09 or H30, as the former are closer together. The

correlation to the mean horizontal wind velocity is also relatively high at about 0.75 to 0.85. This is higher than the correlation of the lateral response depicted in Figure 90. It is also more homogenous, with a slightly higher correlation to *H_mean* towards the south of the bridge.

Figure 99 shows the correlation between *H_turb* and *Aoz_C_std*, similar to Figure 91.

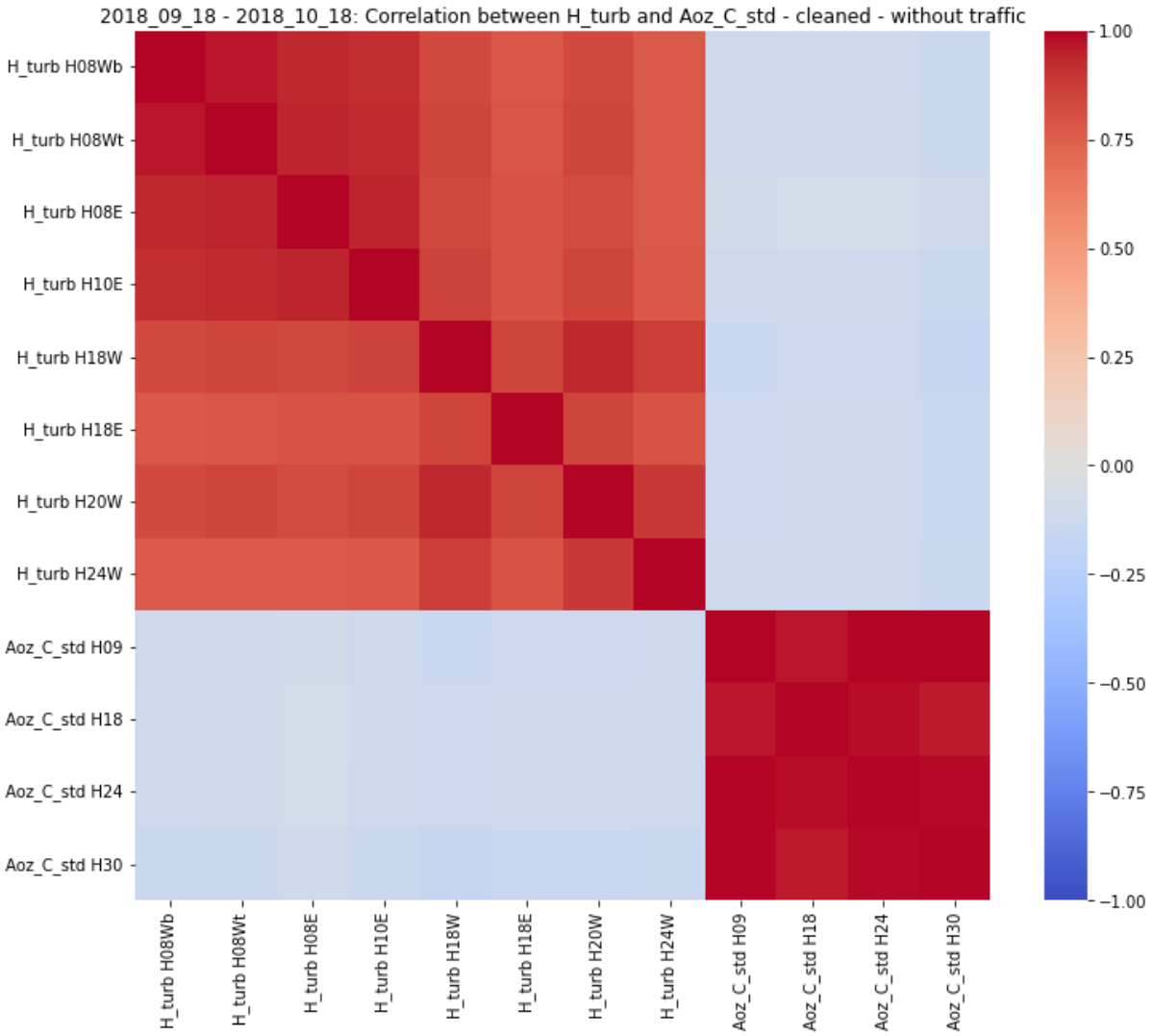


Figure 99: Correlation between turbulence intensity and vertical bridge response per anemometer and accelerometer

From Figure 99 we see that the correlation between the vertical response and horizontal turbulence intensity is homogenously slightly negative at about -0.2, which is in contrast to the lateral response which was less correlated as depicted in Figure 91.

Figure 100 shows the correlation between absolute *AOA_mean* and *Aoz_C_std*, similar to Figure 92.

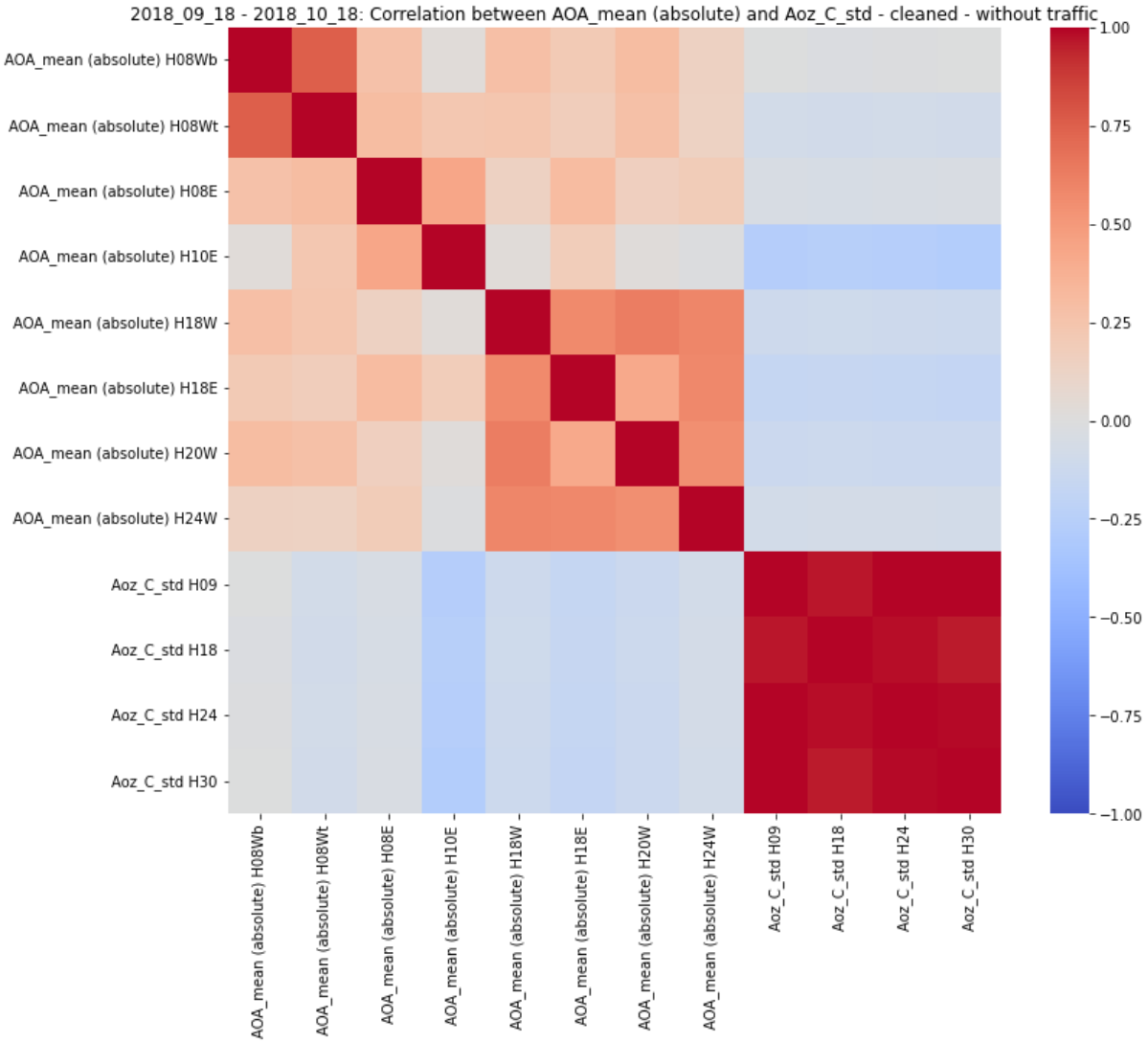


Figure 100: Correlation between absolute mean angle of attack and vertical bridge response per anemometer and accelerometer

From Figure 100 we see that the correlation between the vertical response and absolute mean AOA is close to 0, or slightly negative for anemometers at H10E at about -0.3 as well as H08Wt, H18E and H20W between 0 and -0.25. This, similar to the observation on the lateral response from Figure 92, coincides with the anomaly detected in Figure 82 and discussed in *4.1.3 Angle of attack*. No further significant assumptions can be drawn from these correlations.

Figure 100 shows the correlation between *Aox_C_std* and *Aoz_C_std*.

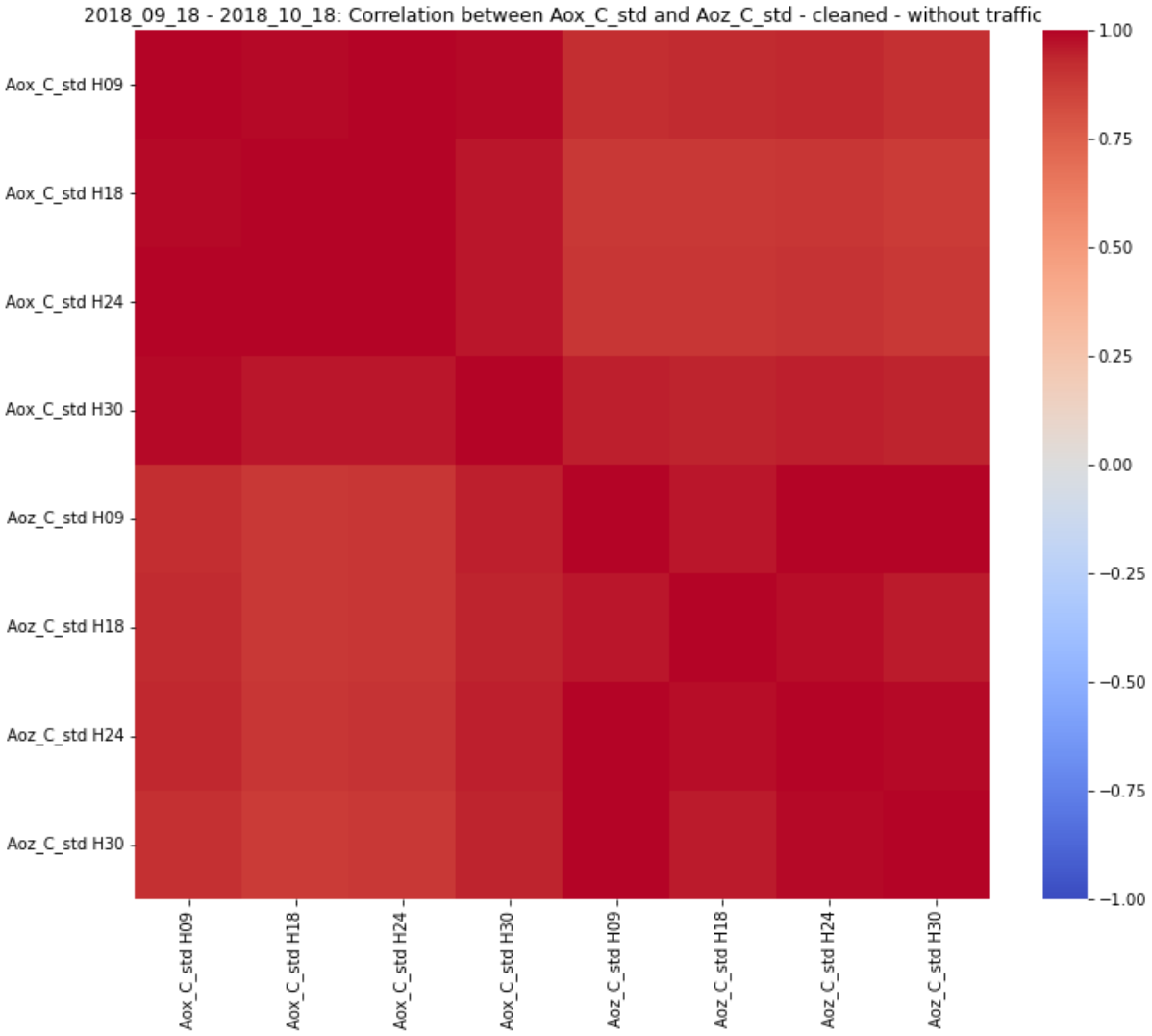


Figure 101: Correlation between lateral and vertical wind response per accelerometer

From Figure 101 we see that the lateral and vertical wind response are well correlated with correlation coefficients above 0.8. The lateral response measured at the outer accelerometers at H09 and H30 is slightly more correlated with the vertical response measured at all accelerometers than the lateral response measured at the inner accelerometers H18 and H24 is.

4.2.4 Torsional wind response

In Figure 102 to Figure 111 the torsional wind response of the bridge, measured by the standard deviation of torsional acceleration *theta_std* is visualised, analysed and compared to the vertical wind response discussed in *4.2.3 Vertical wind response.*

Figure 102 shows a scatterplot of *theta_std* over the mean horizontal wind component perpendicular to the primary bridge axis *Vx_mean* with the respective turbulence intensity *Vx_turb* on the color-axis, similar to Figure 93.

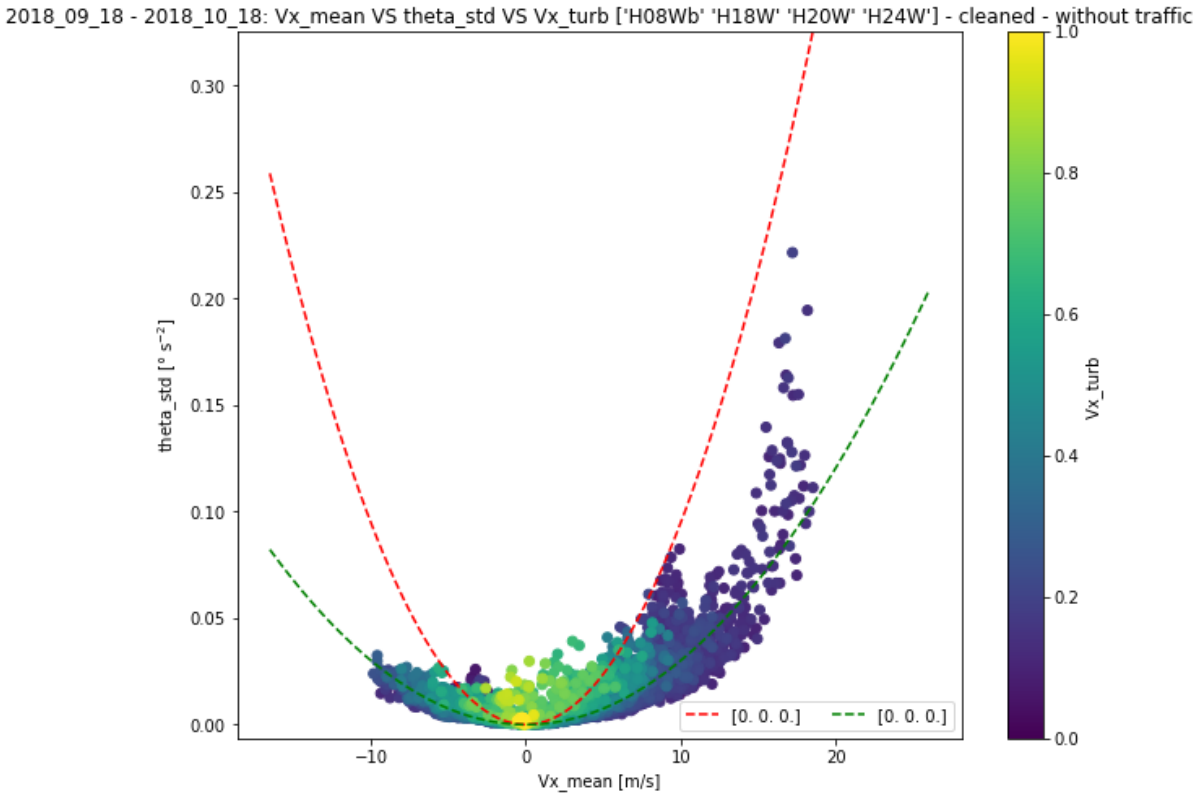


Figure 102: Torsional bridge response and turbulence intensity at different mean wind speeds

The highest standard deviation of torsional acceleration is measured at about 0.22 °/s² at a perpendicular wind speed of about 18 m/s and low turbulence intensity. This is under the same conditions as the highest measured standard deviation of vertical acceleration. At about 10 m/s *theta_std* is measured at about 0.08 °/s² at low turbulence intensity. At about 7 m/s *theta_std* is measured at about 0.05 °/s² at a turbulence intensity of up to about 0.6. We see that theta_std can increase by up to 3 times at about 18 m/s without a significant change in turbulence intensity.

Figure 103 and Figure 104 showcase *theta_std / Vx_turb* for south-westerly and north-easterly wind conditions respectively, split across the accelerometer pairs at H09, H18, H24 and H30, similar to Figure 94 and Figure 95.

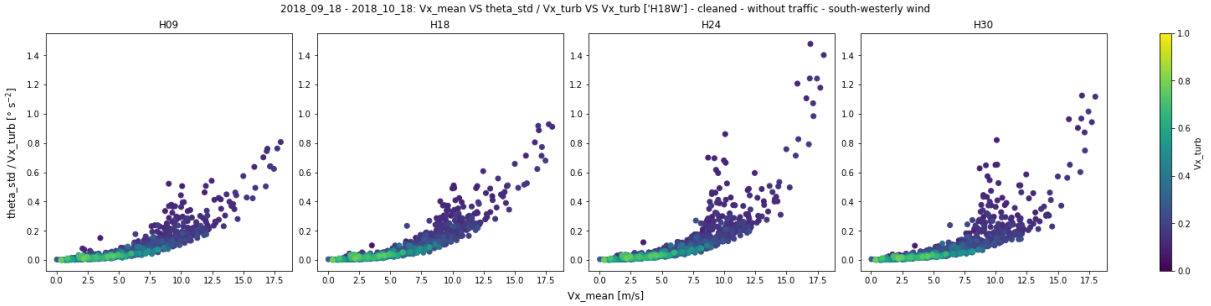


Figure 103: Torsional bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for south-westerly winds per accelerometer pair

As expected, the data is folded in on itself and bound by an upper and lower parabola shape. The remaining spread occurring with higher wind speeds is low, up to 7.5 m/s. Above 7.5 m/s the spread increases.

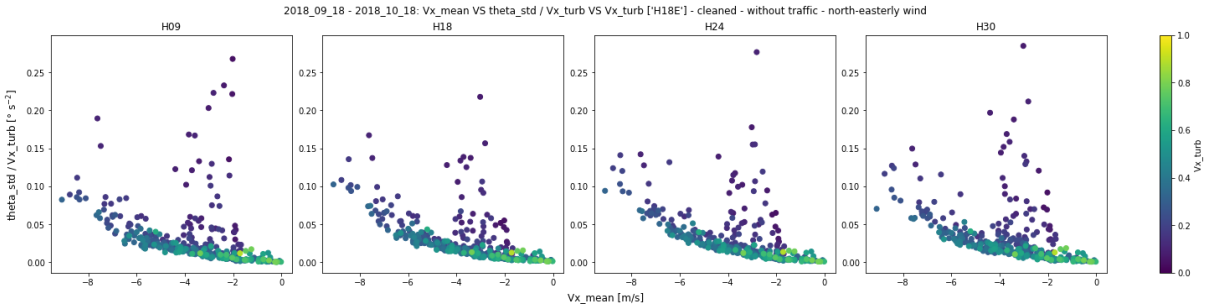


Figure 104: Torsional bridge response normalized by turbulence intensity and turbulence intensity at different mean wind speeds for north-easterly winds per accelerometer pair

As in the case for vertical response in Figure 95, similar observations are again to be made for the north-easterly winds for the torsional response. At wind speeds between -2 m/s and -4 m/s there is a high spread. Between -4 m/s and -6 m/s the spread is low again before increasing again.

Figure 105 shows the vertical bridge response at different mean AOAs with the mean horizontal wind speed decoded on the color-axis, similar to Figure 96.

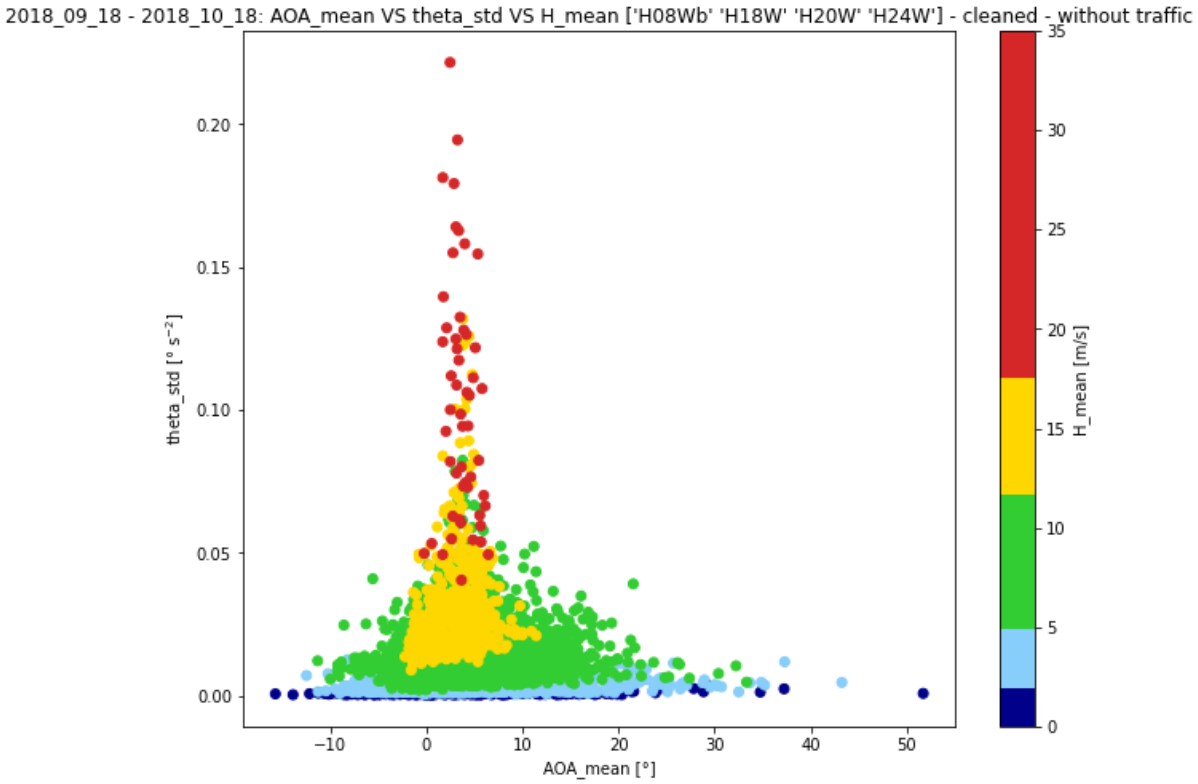


Figure 105: Torsional bridge response and mean wind speed at different angles of attack

From Figure 105 we see that larger torsional responses above 0.08 °/s² and up to about 0.22 °/s² can be observed primarily at wind speeds above 11 m/s at AOAs between +2° to +7°. This is similar to the vertical wind response, where the wind speed has a significant impact on the observable wind response. As discussed in *1.3.2 Wind load coefficients* and showcased in Figure 31, the linearity assumption for the moment coefficient is accurate at about -12° to +5°. We see that the datapoints at wind speeds above 11 m/s causing large torsional responses are mostly just below +5° and only partially above +5°.This suggests that the linearity assumption for the lift coefficient is suitable to predict most of the large torsional responses at lower AOAs correctly, but overestimates some of the large torsional responses for AOAs above 5°.

Normalizing the lateral bridge response by turbulence intensity again results in Figure 106, similar to Figure 97.

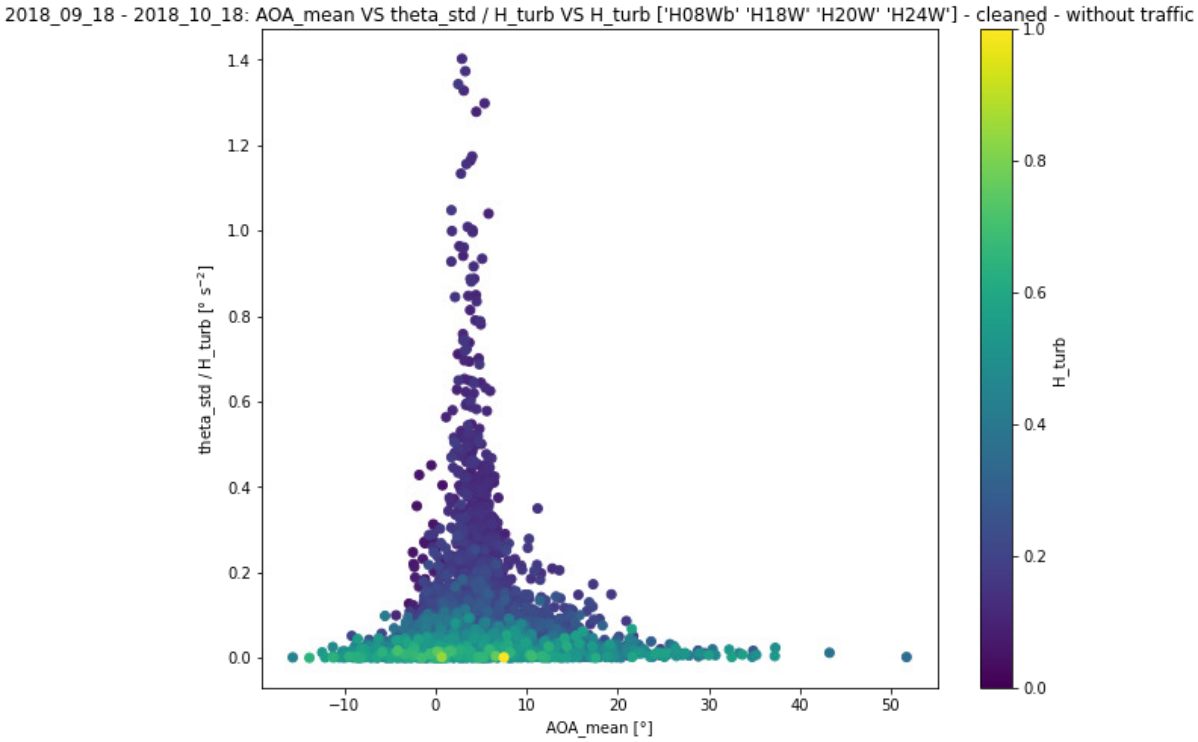


Figure 106: Torsional bridge response normalized by turbulence intensity and turbulence intensity at different angles of attack

In contrast to the lateral and vertical response we do not see as much of an effect from the normalisation, suggesting that high turbulence intensity has a lesser correlation with the torsional wind response.

Figure 107 shows the correlation between *H_mean* and *theta_std*, similar to Figure 98.

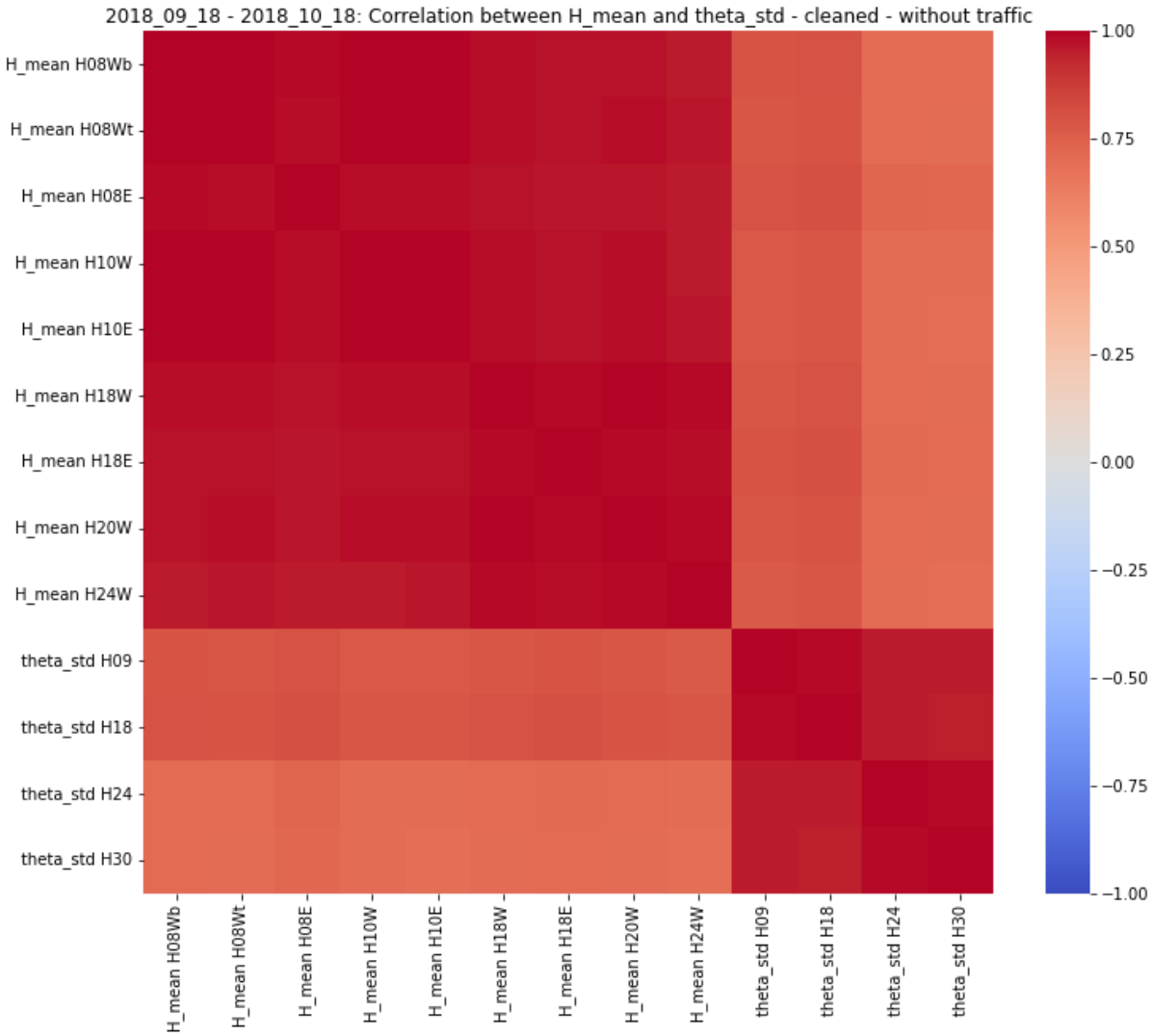


Figure 107: Correlation between mean wind speed and torsional bridge response per anemometer and accelerometer

From Figure 107 we see that the correlation coefficients between all the accelerometer pairs on the tortional response is close to 1, displayed as dark red color in the lower right quadrant. The respective correlation between accelerometers H09 and H18 as well as H24 and H30 is slightly higher compared to the correlations across both groups. This is most likely due to the asymmetric torsional eigenmode shape depicted in Figure 30, in which the north half of the bridge, H09 and H18, oscillates in the opposite direction to the south half, H24 and H30.

There is a positive correlation to the mean horizontal wind velocity, which is slightly higher at about 0.75 for H09 and H18, compared to H24 and H30, where it is about

0.55. The former is comparable to the correlation of the vertical response depicted in Figure 98, while the later is slightly lower, more compareable to the lateral response depicted in Figure 90. However, it is more homogenous across the anemometers.

Figure 108 shows the correlation between *H_turb* and *theta_std*, similar to Figure 99.

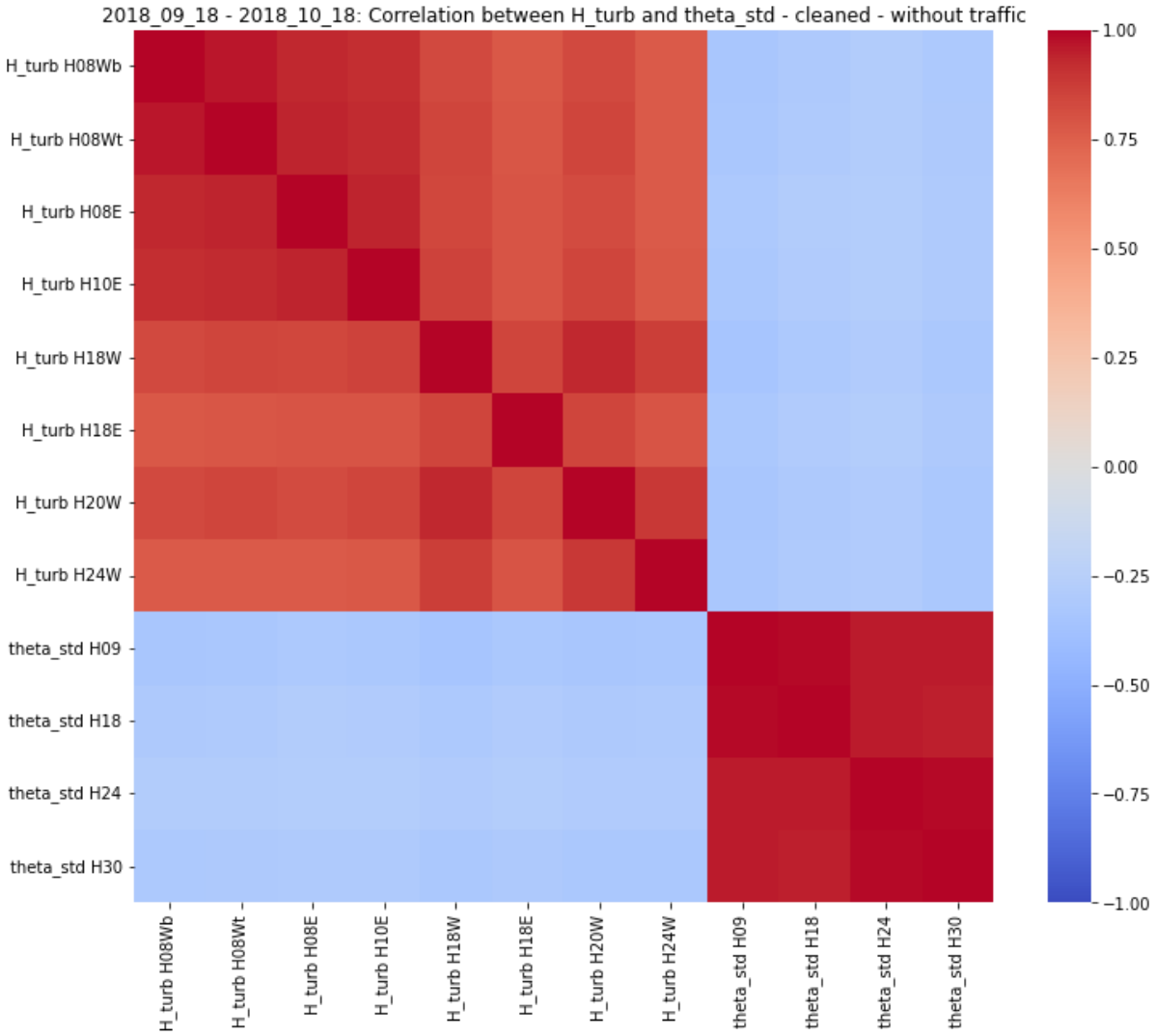


Figure 108: Correlation between turbulence intensity and torsional bridge response per anemometer and accelerometer

From Figure 108 we see that the correlation between the tortional response and horizontal turbulence intensity is homogenously slightly negative at about -0.25, which is similar to the vertical response depicted in Figure 99.

Figure 109 shows the correlation between absolute *AOA_mean* and *theta_std*, similar to Figure 100.

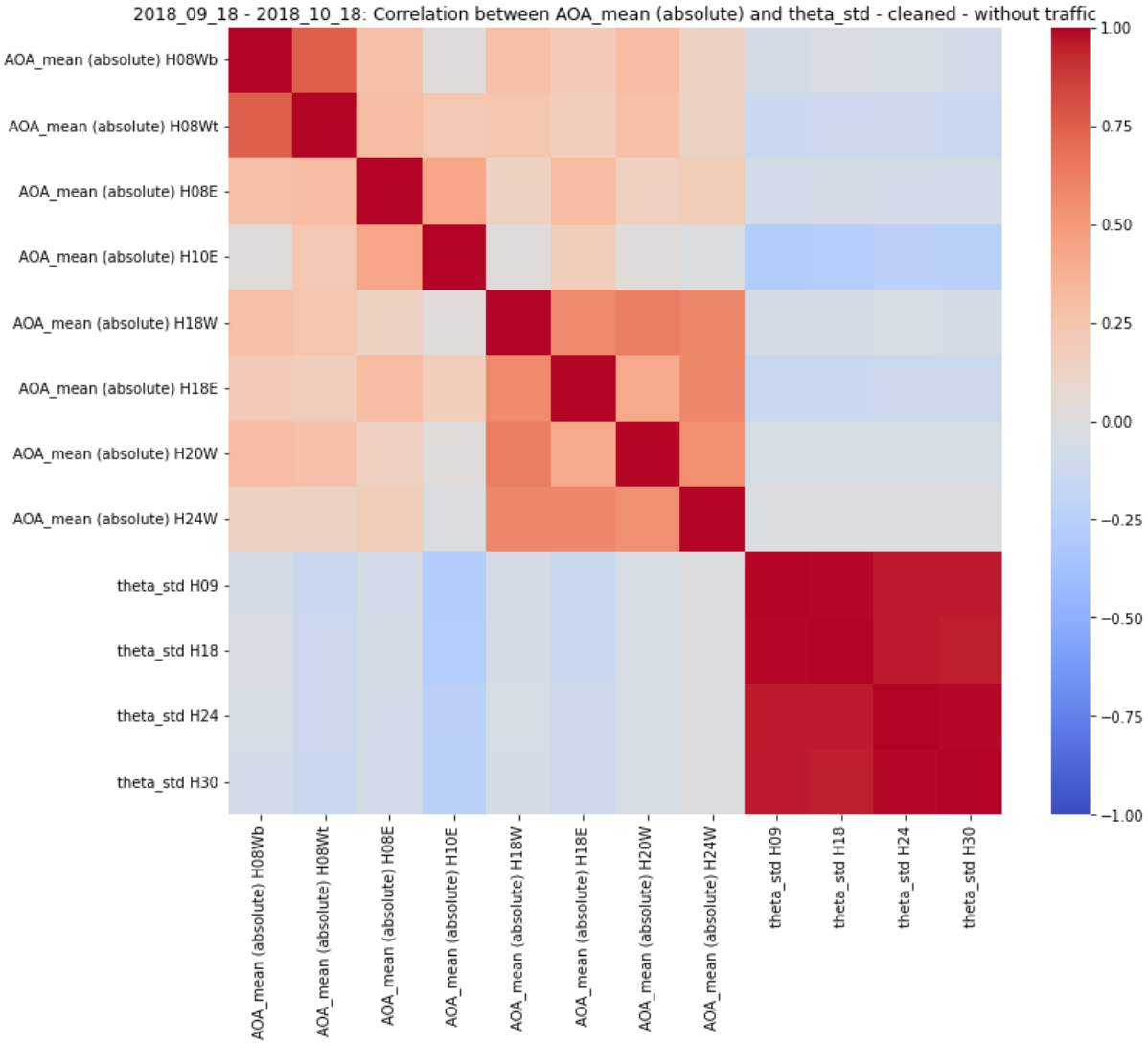


Figure 109: Correlation between absolute mean angle of attack and torsional bridge response per anemometer and accelerometer

From Figure 109 we see that the correlation between the tortional response and absolute mean AOA is close to 0, or slightly negative for anemometers at H10E at about -0.3 as well as H08Wt and H18E between 0 and -0.2. This, similar to the observation on the vertical response from Figure 100, coincides with the anomaly detected in Figure 82 and discussed in *4.1.3 Angle of attack*. No further significant assumptions can be drawn from these correlations.

Figure 110 shows the correlation between *Aox_C_std* and *theta_std*.

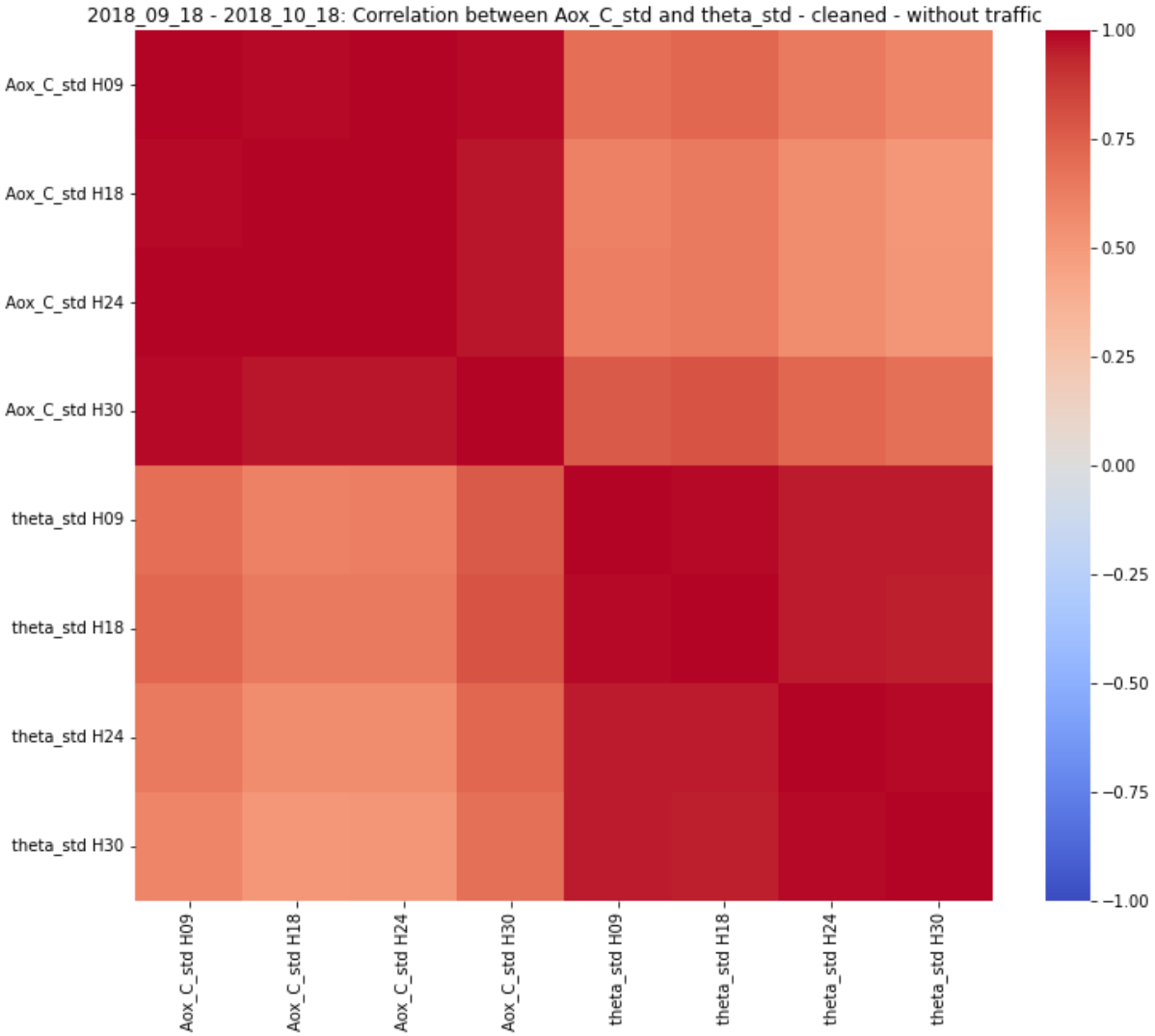


Figure 110: Correlation between lateral and torsional wind response

From Figure 110 we see that the lateral and tortional wind response are somewhat positively correlated with correlation coefficients from about 0.45 to about 0.65, which is lower than the correlation between lateral and vertical wind response. The lateral response measured at the outer accelerometers at H09 and H30 is slightly more correlated with the tortional response measured at all accelerometers than the lateral response measured at the inner accelerometers H18 and H24 is. This is similar to the correlation between lateral and vertical response.

Figure 111 shows the correlation between *Aoz_C_std* and *theta_std*.

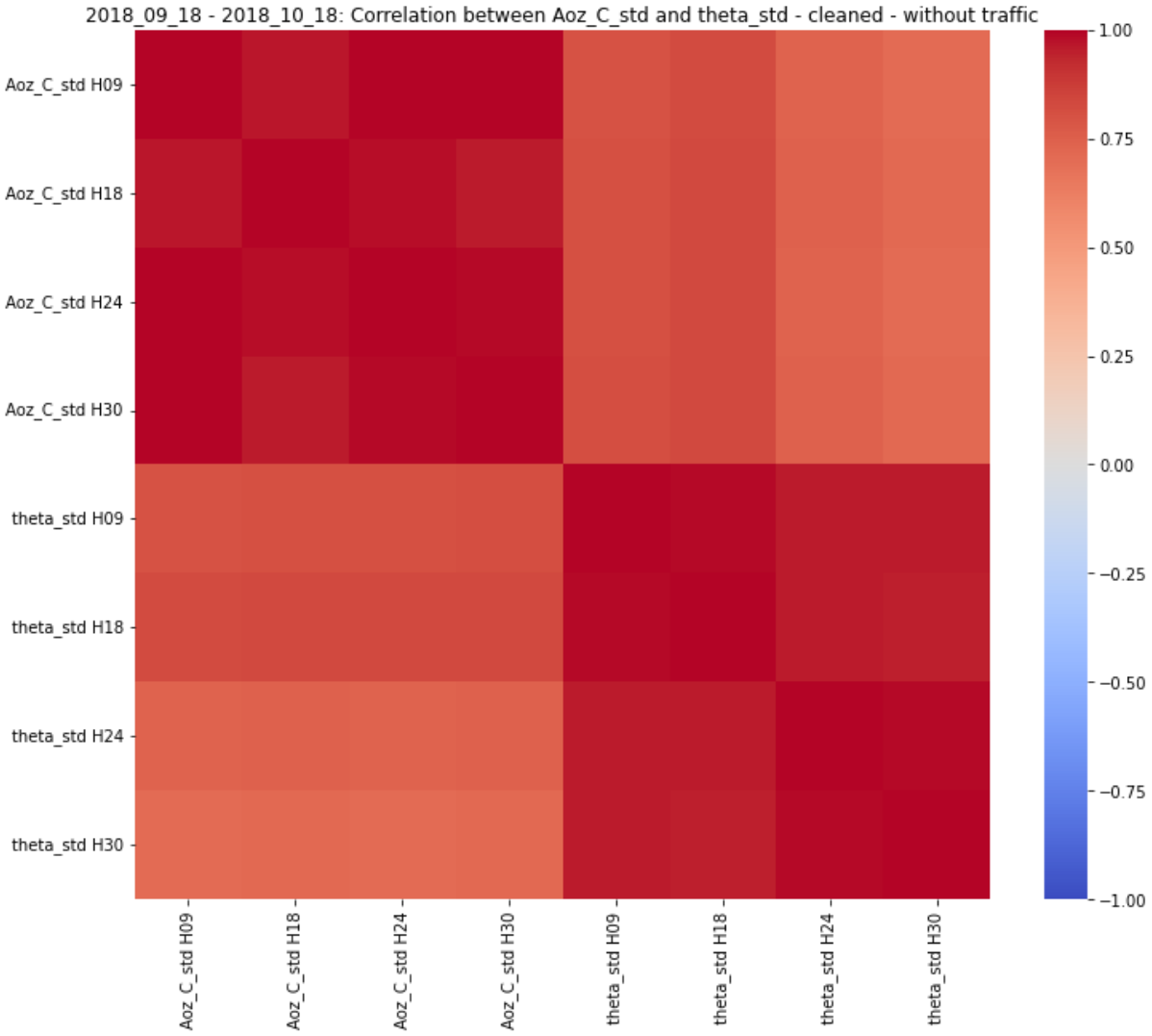


Figure 111: Correlation between vertical and torsional wind response

From Figure 111 we see that the vertical and tortional wind response are more positively correlated with correlation coefficients between about 0.65 and 0.85. The tortional response measured at the northern accelerometers at H09 and H18 is slightly more correlated with the vertical response measured at all accelerometers than the tortional response measured at the southern accelerometers H24 and H30 is.

5. Discussion

The results from this work are summarized and discussed in the sections below.

5.1 Conclusion

The results from analysis confirm the primary wind directions of NNE and SSW. Mean wind speeds reached up to 21.5 m/s and gusts up to 35 m/s in the 30-day period. The directional bandwidth for south-westerly winds is about 15° wider than the one for north-easterly winds. Towards the south of the bridge there is a south tendency for south-westerly winds and north tendency for north-easterly winds. The wind from south-westerly direction is on average about 2.5 m/s higher than the wind from north-easterly direction. The highest wind speeds from south-westerly direction are almost double the highest wind speeds from north-easterly direction.

North-easterly winds are slightly more turbulent than south-westerly winds. Anemometers on the downwind side of the bridge record slightly more turbulent wind compared to their upwind counterparts. Turbulence intensity decrease with higher wind speeds. They are negatively correlated across all anemometers.

Mean AOAs reach from -33° to +53°, but 80% of the data is in a 12° range with a mean of about 3.4° and 0.9° for south-westerly and north-easterly winds respectively. The mean AOA is straightened towards 0° on the downwind side. The linearity assumption discussed in *1.3.2 Wind load coefficients* is accurate enough for negative AOAs most of the time. For positive AOAs it overestimates lift coefficient and moment coefficient 25% of the time and underestimates drag coefficients 75% of the time for winds from south-westerly direction. Downwind anemometers experience less extreme AOAs compared to their upwind counterparts. An anomaly of high mean AOAs occurs at H18E with winds from easterly directions. Absolute mean AOA is not significantly correlated with mean horizontal wind speed or turbulence intensity, across the anemometers, except for H10E where there is a negative correlation with H_mean and positive correlation with H_turb.

The bridge displays a periodic response to the daily traffic with higher maximum vertical accelerations on weekdays due to heavy industrial traffic.

The highest vertical wind response is about 3.5 to 4 times higher than the highest lateral wind response. Most of the larger lateral wind responses are at AOAs above

the suitable range of the linearity assumption, where it underestimates the drag coefficient. Note that some of these larger responses might be correlated with higher turbulence intensity. The linearity assumption is partially suitable for the prediction of vertical and torsional responses, but overestimates some of the larger responses above +5° AOA.

Vertical wind response has the highest positive correlation to mean horizontal wind speed. Torsional wind response has a slightly lower positive correlation to mean horizontal wind speed, with slightly higher correlation at the northern accelerometers. Lateral wind response has the lowest positive correlation to mean horizontal wind speed with slightly higher correlation at the outer accelerometers.

Torsional wind response has the highest negative correlation to horizontal turbulence intensity. Vertical wind response has a low negative correlation to horizontal turbulence intensity. Lateral wind response has an insignificant negative correlation to horizontal turbulence intensity at the outer accelerometers. Lateral, vertical and torsional wind response have an insignificant negative correlation to absolute mean AOA with an anomaly occurring at H10E, where the correlation is slightly more negative.

Lateral and vertical wind response have a high positive correlation. Vertical and tortional wind response have a slightly lower positive correlation, with slightly lower correlations in the southern accelerometers. Lateral and torsional wind response have the lowest positive correlation, with slightly lower correlations in the inner accelerometers.

Note that correlation does not imply causation.

Creating a large amount of diverse but consistent visualisations for various measurement type combinations is fast and relatively easy with the methods provided by the *BridgeData* class created in this work. This allows to quickly get an overview of the data and phenomena happening at the bridge. It allows to find interesting phenomena and anomalies that might be overlooked in a more directed approach. The flexible and modular design allows to focus observations on desired phenomena, for example by using filters. Interpreting the visualisations to explain some of the phenomena requires extensive background knowledge in the field of wind engineering.

The ability to save the state of a class object is particularly useful, as it allows to quickly come back to a previous analysis without having to re-import the data.

5.2 Comparison to relevant works

Some of the relevant works on data of the Lysefjord Bridge include [2], [3], [7], [20], [24], and [18]. These works were regularly referenced in the creation of this work.

Figure 112 shows a direct comparison on the turbulence intensity for different windspeeds to E. Chynet's analysis in 2016 [20, p. 18]. The same period and same sensors have been used for this visualisation. E. Cheynet's scatterplot has been layered on top of this works scatterplot, matching the scaling of the axis.

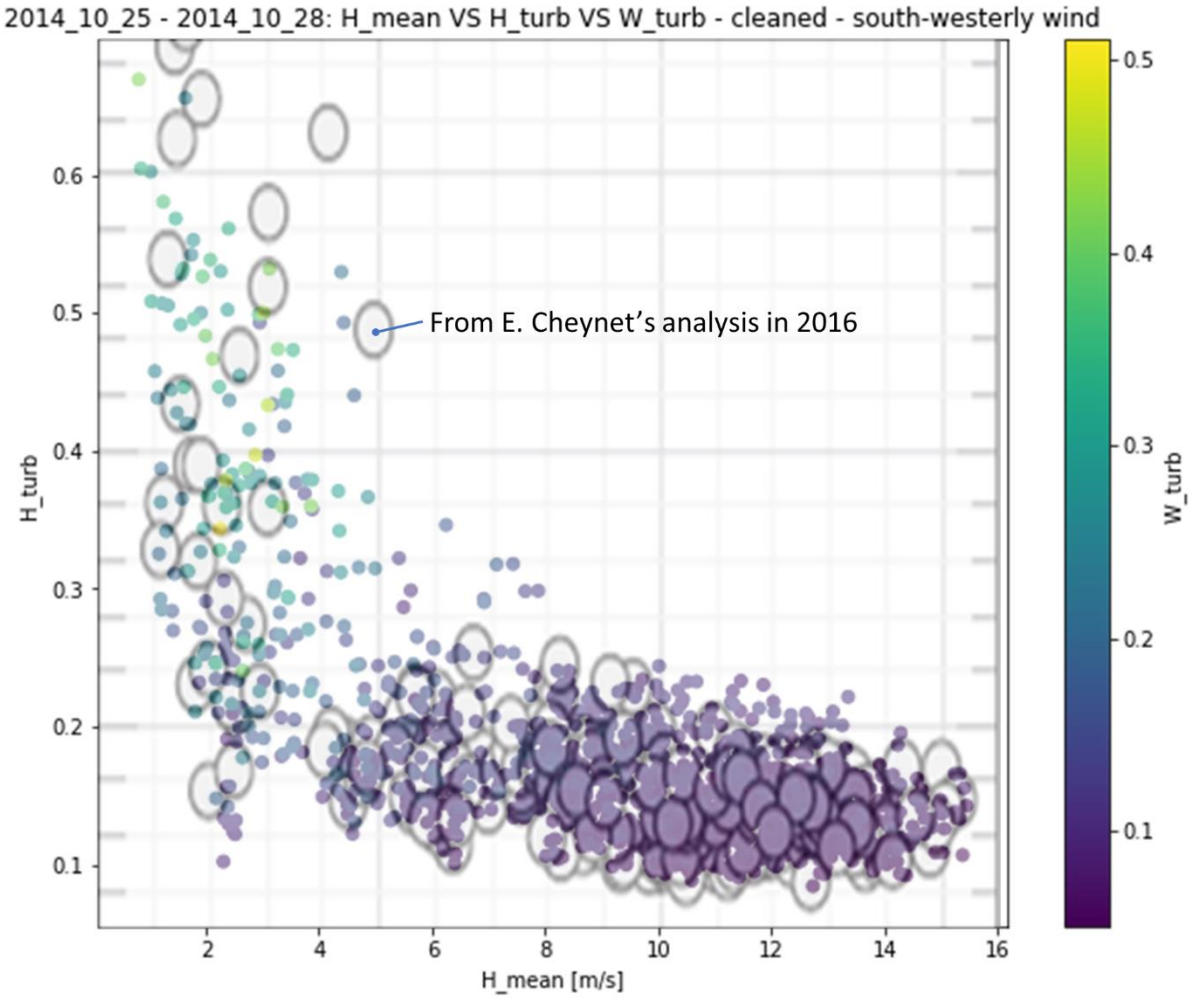


Figure 112: Comparison of turbulence intensity at different wind speeds to E. Cheynet's work in 2016. Adapted from [20, p. 18]

Figure 112 illustrates that a very similar visualisation of the data can be achieved from the same data base. Note that some datapoints might be missing in either scatterplot or have slight variations in their position due to different methods used in

importing and processing of the data, such as the slightly different calculation of the turbulence intensity, as described in *3.1.1 convert_MATLAB*. However, the general structure of the scatterplot is very much comparable, showing the same decrease in turbulence intensity with higher wind speeds.

Similarly, a polar scatterplot from E. Cheynet's work in 2016 [20, p. 136] has been replicated using data from 07/10/2014, as depicted in Figure 113.

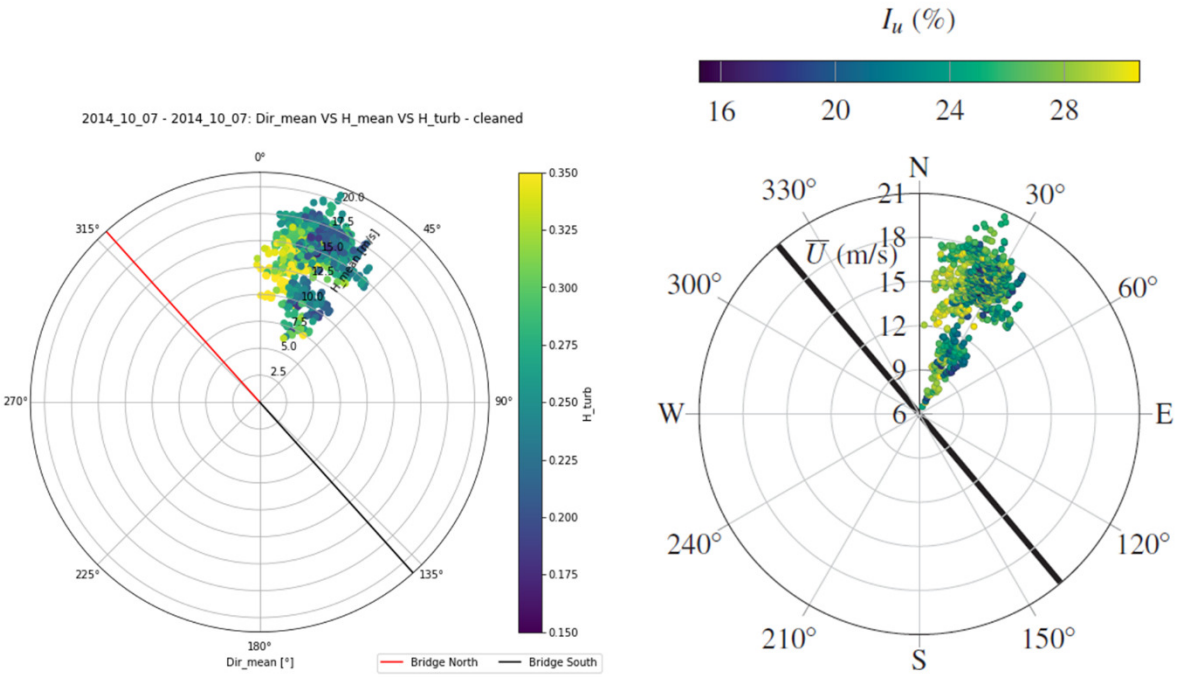


Figure 113: Turbulence visualisation from this work (left) and E. Cheynet's work in 2016 (right) [20, p. 136]

Note that the radial axis starts at 0 m/s in this work, while it starts at 6 m/s in E. Cheynet's work. Apart from slightly different scaling on the radial axis and color axis the polar scatter plots match up very well.

Furthermore E. Cheynet calculates "a mean wind direction of 22° and a standard deviation of almost 9°" [20, p. 135], referring to geographic north. In this work a mean wind direction of about 21° and standard deviation of about 8° is calculated.

Some slight variations might again occur due to different methods in the importing and processing of the data. Also note that differently colored dots on a scatterplot might be plotted on top of each other in a different order. This can be misleading when trying to interpret the color axis. Therefore, lowest color axis values are usually plotted first and higher values afterwards in the colored scatter plots of this work.

However, this can be disabled by setting *plot_in_order_of_color* to *False*. This was done in this comparison to better match E. Cheynet's work.

In contrast to the previously mentioned relevant works, this work did not focus on interpreting results of a particular analysis. However, the author created a toolset to easily and quickly create large amounts of diverse but consistent visualisations for various measurement type combinations, which can help to gain a different perspective on the data. It is possible to perform extensive and detailed analysis on the data using this works code, as demonstrated in *4 Analysis using this works code*. The results from this analysis are in line with the results expected from previous relevant works. Useful insights into wind conditions and wind responses are obtained. This includes statistical descriptions and comparisons of north-easterly and south-westerly wind conditions, correlations of wind responses to wind conditions, evaluation of the linearity assumption described in *1.3.2 Wind load coefficients* and the discovery of previously unknown anomalies at H10E and H18E.

5.3 Limitations of this work

The technical limitations of this work lie in memory-constraints and the disk-space required to process a larger amount of the relatively large MATLAB files. The computational load during the import stage and the creation of some of the plots, especially scatter plots, is therefore quite high with larger datasets and suffer from a lack of parallel processing. Some of the plots lack flexibility as not all the original matplotlib functionality on which they are based on is passed along. The data filtering is somewhat limited compared to some of the available tools used for database manipulations. Some more in-depth tools for specific types of analysis, such as spectral analysis and modal analysis are currently not implemented. The interpretation of the analysis is not very in-depth and might be incomplete or incorrect as the author has no background in wind engineering. This also shows as there is currently no implementation for transforming the coordinate system to mean wind coordinates, which would commonly be used for turbulence-intensity calculations for example. However, the modular design of this works code should allow to easily implement the missing functionality or use third-party libraries in the data processing and analysis.

5.4 Future work

While the author took care to utilise the parallel computation of numpy functionality, attempts on further parallelisation should be made where possible, using *multiprocessing* for example. To further decrease the computational load, a high-performance Python compiler such as *numba* could be utilized. Other methods to increase performance could be *gpu utilisation* or *outsourcing* parts of the process to a server, or high-performance cluster.

Another approach to improve this work lies in implementing additional functionality such as spectral analysis, modal analysis or the transformation to mean wind coordinates. Additionally, machine learning could be used to, for example, perform regression on the wind response and clustering on the wind conditions. B. da Costa gives a great overview of possibilities in [51].

Finally, the anomalies discovered at H18E and H10E could be further examined for example by performing CFD-simulations or LIDAR studies. The suitability of the linearity assumption could be further examined, especially regarding the potential underestimation of the drag coefficient at positive AOAs.

5.5 Authors experience

For the author it was of high importance to create a modular, flexible tool that could be used to perform further analysis than what has been done in this work. The Python and Jupyter Notebook environment have perfectly supported this goal. Style guides and documentations of Python and its libraries such as [35], [36], [37], [41], [42], [43], [44], [45] and [47] were very helpful in building the *BridgeData* class and documenting it properly. Examining the data and creating large amounts of diverse but consistent visualisations for various measurement type combinations has been a very enjoyable process for the author. The previous relevant works on the bridge data have been very helpful to understand the complex wind engineering related phenomena around the Lysefjord Bridge.

The author assumed a role in between the perspective of a wind engineer and a data scientist. This has required a cross-disciplinary mindset, which allowed the author to acquire, utilise and reproduce knowledge from both domains without the need for an in-depth background in either. While this experience has been challenging, the knowledge and skillsets gained from both domains are worth it.

References

[1] B. Elliott, Director, *The Tacoma Narrows Bridge Collapse.* [Film]. United States of America, Washington.1940.

[2] J. Tveiten, “Dynamic analysis of a suspension bridge,” Universitetet i Stavanger - Faculty of Science and Technology, Stavanger, 2012.

[3] J. Wang, E. Cheynet, J. Bogunovic Jakobsen and J. T. Snæbjörnsson, “Time-Domain Analysis of Wind-Induced Response of a Suspension Bridge in Comparison With the Full-Scale Measurements,” in *International Conference on Ocean, Offshore and Arctic Engineering*, Trondheim, 2017.

[4] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9242807,6.0971971,3a,45.6y,314.36h,94.02t/data=!3m6!1e1!3m4!1ssxkD7Fn1ASOaDqRcIDAh_Q!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[5] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9127151,6.0779843,3a,20.4y,42.68h,91.71t/data=!3m6!1e1!3m4!1s7NdvJPV-zhwZB3RQKSEkyg!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[6] Google LLC, “Google Maps,” 2023. [Online]. Available: https://www.google.com/maps/@58.9314358,6.0412977,10.22z/data=!5m1!1e4. [Accessed 21 April 2023].

[7] J. T. Snæbjörnsson, J. Bogunovic Jakobsen, E. Cheynet and J. Wang, “Full-scale monitoring of wind and suspension bridge response,” in *First Conference of Computational Methods in Offshore Technology*, Stavanger, 2017.

[8] Google LLC, “Google Maps,” 2023. [Online]. Available: https://www.google.com/maps/dir/NCC+Helle+sandtak,+4110+Forsand/Norsk+Spennbetong+AS,+Myrbakken,+Forsand/Lysefjord+Bridge,+Sekund%C3%A6r+Fylkesvei+491,+Forsand/Stavanger/@58.9513388,5.7813112,11z/data=!4m26!4m25!1m5!1m1!1s0x463a29682eb02777:0x5247ac464b20. [Accessed 29 April 2023].

[9] Google LLC, “Google Maps,” 2023. [Online]. Available: https://www.google.de/maps/dir/NCC+Helle+sandtak,+Forsand/Bj%C3%B8rn+Hansen+AS+Betongelement,+Myrbakken+51,+4110+Forsand/Lauvvik+ferjekai,+4308+Sandnes/Sandnes/Stavanger/@58.9028614,5.8356285,11.75z/data=!4m32!4m31!1m5!1m1!1s0x463a29682eb02777:0x5247ac464. [Accessed 26 May 2023].

[10] Lysefjorden Utvikling AS, “LYSEFJORDEN 365: EXPLORE THE LYSEFJORD BY FERRY,” 2023. [Online]. Available: https://lysefjorden365.com/ferry/#touristferry. [Accessed 26 May 2023].

[11] Windfinder, “Annual wind and weather statistics for Forsand/Lysefjorden,” February 2017. [Online]. Available: https://www.windfinder.com/windstatistics/forsand_lysefjorden. [Accessed 18 May 2023].

[12] Windfinder, “Annual wind and weather statistics for Liarvatnet,” April 2023. [Online]. Available: https://www.windfinder.com/windstatistics/liarvatnet. [Accessed 18 May 2023].

[13] Windfinder, “Annual wind and weather statistics for Sandnes/Hanafjellet,” July 2019. [Online]. Available: https://www.windfinder.com/windstatistics/sandnes_harrafjellet. [Accessed 18 May 2023].

[14] Windfinder, “Annual wind and weather statistics for Stavanger Airport, Sola,” April 2023. [Online]. Available: https://www.windfinder.com/windstatistics/stavanger_sola. [Accessed 18 May 2023].

[15] Windfinder, “Annual wind and weather statistics for Stokkavika/Idse,” May 2020. [Online]. Available: https://www.windfinder.com/windstatistics/stokkavika_idse. [Accessed 18 May 2023].

[16] Windfinder, “Monthly wind speed statistics and directions for Meling/Forsand,” February 2017. [Online]. Available: https://www.windfinder.com/windstatistics/meling_forsand. [Accessed 18 May 2023].

[17] Windfinder, “Map,” 2023. [Online]. Available: https://www.windfinder.com/#11/58.9700/5.8413/rain/spot. [Accessed 18 May 2023].

[18] E. Cheynet, S. Liu, M. C. Ong, J. Bogunovic Jakobsen, J. T. Snæbjörnsson and I. Gatin, “The influence of terrain on the mean wind flow characteristics in a fjord,” *Journal of Wind Engineering & Industrial Aerodynamics,* vol. 205, 2020.

[19] G. Box, “Robustness in the strategy of scientific model building,” in *Robustness in Statistics*, Launer, 1979.

[20] E. Cheynet, “Wind-induced vibrations of a suspension bridge - A case study in full-scale,” Universitetet i Stavanger, Faculty of Science and Technology, Department of Mechanical and Structural Engineering and Materials Science, Stavanger, 2016.

[21] Gill Instruments Limited, “WindMaster Pro Datasheet iss 8,” 17 August 2022. [Online]. Available: https://gillinstruments.com/compare-3-axis-anemometers/windmaster-3axis/. [Accessed 23 January 2023].

[22] Vaisala, “Vaisala Weather Transmitter WXT520 USER'S GUIDE,” 10 October 2012. [Online]. Available: https://www.vaisala.com/sites/default/files/documents/M210906EN-C.pdf. [Accessed 2023 April 29].

[23] Canterbury Seismic Instruments Ltd., “CUSP-Me Specifications,” 10 May 2017. [Online]. Available: https://csi.net.nz/images/csi%20cusp-me%20specification.pdf. [Accessed 13 June 2023].

[24] E. Cheynet, N. Daniotti, J. Bogunovic Jakobsen and J. T. Snæbjörnsson, “Improved long-span bridge modeling using data-driven identification of vehicle-induced vibrations,” *Structural Control and Health Monitoring,* vol. 27, no. 9, 2020.

[25] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9236844,6.0982108,3a,75y,315.26h,100.8t/data=!3m6!1e1!3m4!1scoKmBZSqSjDXvS-3fd9cDA!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[26] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9236062,6.0983437,3a,75y,181.17h,104.8t/data=!3m6!1e1!3m4!1s9lHm91zk5zxTWddkSLifdg!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[27] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9232071,6.0990383,3a,75y,289.7h,99.29t/data=!3m6!1e1!3m4!1stG-LiVwBm7KJh1WMyvCC6A!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[28] COMSOL, “Eigenfrequency Analysis,” 08 May 2018. [Online]. Available: https://www.comsol.com/multiphysics/eigenfrequency-analysis. [Accessed 08 06 2023].

[29] COMSOL, “Mode Superposition,” 08 May 2018. [Online]. Available: https://www.comsol.com/multiphysics/mode-superposition?parent=structural-mechanics-0182-222. [Accessed 08 June 2023].

[30] SOH Wind Engineering LLC, “LYSEFJORD BRIDGE, NORWAY - Static wind tunnel tests,” WILLISTON, USA, 2021.

[31] S. O. Hansen, B. I. Robin George Srouji and K. Berntsen, “Vortex-induced vibrations of streamlined single box girder bridge decks,” in *14th International Conference on Wind Engineering*, Porto Alegre, 2015.

[32] E. Hjorth-Hansen, E. Strømmen, J. Bogunovic Jakobsen, H.-P. Brathaug and E. Solheim, “Wind tunnel investigations for a proposed suspension bridge across the hardanger fjord,” in *2nd European Conference on Structural Dynamics*, Trondheim, 1993.

[33] TIOBE Software BV, “TIOBE Index for May 2023,” 2 May 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 23 May 2023].

[34] R. Scarlett, “Why Python keeps growing, explained,” GitHub, 02 March 2023. [Online]. Available: https://github.blog/2023-03-02-why-python-keeps-growing-explained/. [Accessed 23 May 2023].

[35] M. Cone, “Markdown Guide Basic Syntax,” 2023. [Online]. Available: https://www.markdownguide.org/basic-syntax/. [Accessed 31 May 2023].

[36] Python Software Foundation, “Documentation » Python HOWTOs » Functional Programming HOWTO,” Sphinx, 17 May 2023. [Online]. Available: https://docs.python.org/3/howto/functional.html. [Accessed 17 May 2023].

[37] Python Software Foundation, “Documentation » The Python Tutorial » 9. Classes,” Sphinx, 17 May 2023. [Online]. Available: https://docs.python.org/3/tutorial/classes.html. [Accessed 17 May 2023].

[38] E. Byeon, “Speed Testing Pandas vs. Numpy,” Towards Data Science, 14 December 2020. [Online]. Available: https://towardsdatascience.com/speed-testing-pandas-vs-numpy-ffbf80070ee7. [Accessed 22 May 2023].

[39] N. McCullum, “NumPy Arrays vs. Pandas Series: A Performance Comparison,” 6th January 2021. [Online]. Available: https://www.nickmccullum.com/numpy-arrays-pandas-series-performance-comparison/#:~:text=what%20we%20discussed%3A-,A%20numpy%20array%20is%20a%20grid%20of%20values%20that%20belong,function%20of%20the%20Pandas%20library.. [Accessed 2022 May 2023].

[40] G. Balaraman, “Numpy Vs Pandas Performance Comparison,” 14 March 2017. [Online]. Available: http://gouthamanbalaraman.com/blog/numpy-vs-pandas-comparison.html. [Accessed 22 May 2023].

[41] G. v. Rossum, B. Warsaw and N. Coghlan, “PEP 8 – Style Guide for Python Code,” 30 April 2023. [Online]. Available: https://peps.python.org/pep-0008/. [Accessed 17 May 2023].

[42] D. Goodger and G. v. Rossum, “PEP 257 – Docstring Conventions,” 06 June 2022. [Online]. Available: https://peps.python.org/pep-0257/. [Accessed 17 May 2023].

[43] numpydoc maintainers, “NumPy Style Guide,” Sphinx, 2023. [Online]. Available: https://numpydoc.readthedocs.io/en/latest/format.html. [Accessed 17 May 2023].

[44] pandas, “pandas docstring guide,” Sphinx, 2023. [Online]. Available: https://pandas.pydata.org/docs/development/contributing_docstring.html. [Accessed 17 May 2023].

[45] NumPy Developers, “numpy.arctan2,” Sphinx, 2022. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html. [Accessed 22 May 2023].

[46] Norwegian Meteorological Institute and the Norwegian Broadcasting Corporation, “Yr,” 2018. [Online]. Available: https://www.yr.no/en/statistics/graph/5-44560/Norway/Rogaland/Sola/Sola?q=2018. [Accessed 11 May 2023].

[47] The SciPy community, “Statistical functions scipy.stats.kurtosis,” Sphinx, 2023. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kurtosis.html. [Accessed 29 May 2023].

[48] SKYbrary Aviation Safety, “Stall Definition,” 2023. [Online]. Available: https://www.skybrary.aero/articles/stall#:~:text=Description,is%20typically%20around%2015%C2%B0.. [Accessed 30 May 2023].

[49] Google LLC, “Google Maps,” 2023. [Online]. Available: https://www.google.de/maps/@58.9353105,6.1070676,4334m/data=!3m1!1e3. [Accessed 05 May 2023].

[50] Google LLC, “Google Street View,” July 2022. [Online]. Available: https://www.google.de/maps/@58.9236844,6.0982108,3a,75y,34.14h,94.34t/data=!3m6!1e1!3m4!1scoKmBZSqSjDXvS-3fd9cDA!2e0!7i16384!8i8192. [Accessed 11 May 2023].

[51] B. M. da Costa, “Trial lecture: Applications of machine learning algorithms in wind engineering,” University of Stavanger, Stavanger.

[52] Windfinder, “Annual wind and weather statistics for Jørpeland,” August 2014. [Online]. Available: https://www.windfinder.com/windstatistics/joerpeland. [Accessed 18 May 2023].

[53] Windfinder, “Annual wind and weather statistics for Hundvåg,” October 2018. [Online]. Available: https://www.windfinder.com/windstatistics/hundvag. [Accessed 18 May 2023].

Appendix

A1: Complete Code

➔ The code itself is provided as an *.ipynb* file. The data is to be obtained from UiS.

➔ Below you will find a *.pdf* conversion of the Jupyter Notebook. Note that this does not reflect the complete code as some of the longer lines are truncated.

Analysis of wind and response measurement data from a suspension bridge

This Jupyter Notebook is accompanying documentation to the **MASTER'S THESIS** *Analysis of wind and response measurement data from a suspension bridge* by *René König*.

Universitetet i Stavanger (UiS)

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation: *Computational Engineering*

The *spring* semester, 2023

Open

Author: *René König, 268127*

Supervisor at UiS: *Professor Aksel Hiorth*

Co-supervisor: *Professor Jasna Bogunovic Jakobsen*

External supervisor(s): *Bernardo Morais da Costa, Statens vegvesen*

Thesis title: *Analysis of wind and response measurement data from a suspension bridge*

Credits (ECTS): *30*

Keywords: *Python, Data Science, Software Development, Suspension Bridge, Wind Engineering, Wind Response, Full-Scale Monitoring, Anemometers, Accelerometers*

Stavanger, *14.06.2023*

Abstract

The Lysefjord Bridge is a suspension bridge at the entrance to the Lysefjord in south western Norway at which full-scale measurements on wind conditions and bridge response are collected using anemometers and accelerometers. In this work Python is used to develop a toolset for analysing the wind and response measurement data from the Lysefjord Bridge. The functionality is provided through different methods compiled in a class. This includes methods for importing and combining data from multiple days, re-arranging and interpreting the data, feature engineering, data cleaning, filtering and various types of visualisations. The code is demonstrated in an analysis of 30 days of data. The analysis focuses on the wind conditions for south-westerly and north-easterly winds in terms of wind speeds, primary directions, turbulence intensity and angle of attack as well as the bridges lateral, vertical and torsional wind response. The analysis shows on average slightly higher wind speeds, lower turbulence intensities and higher angles of attack for south-westerly winds, compared to north-easterly winds. Towards the

southern end of the bridge the wind direction has a south tendency for south-westerly winds and north tendency for north-easterly winds. Turbulence intensity is measured slightly higher on the downwind side of the bridge. The angle of attack is straightened towards 0° on the downwind side. Furthermore, the analysis shows that the assumption of a linear correlation between drag coefficient and angle of attack used in the so-called quasi-steady theory of wind loading and the corresponding numerical simulations underestimates most of the larger lateral bridge responses at angles of attack above 0°. The lift and moment coefficients estimated using similar linearity assumptions overestimate some of the larger vertical and torsional bridge responses at angles of attack above +5°.

Libraries used

The libraries below are used in this code.

In [ ]:

```
import numpy as np # Math, handling data arrays
import matplotlib.pyplot as plt # Plots
import matplotlib as mpl # Handle colormaps
from matplotlib.colors import ListedColormap # Create colormaps
import scipy.io # Read matlab files
import scipy.optimize # Curve-fitting
import scipy.stats # Statistics
import seaborn as sn # Heatmaps
from windrose import WindroseAxes # Wind roses
import gc # Garbage collection to free up memory
import dill # Saving and loading data
import warnings
warnings.filterwarnings('ignore') # Clean up the notebook by ignoring the expected w
```

The code was developed under Python version 3.9.6

`BridgeData` class definition

The `BridgeData` class defined below contains most of the functionallity of this work to load, process and analyze data from the Lysefjord Bridge.

In [ ]:

```
class BridgeData:
    """A class for processing wind and response data from a suspensionbridge, as in

    Attributes
    ----------
    file_names : list of strs
        List of Lysefjord Bridge dataExtracted.mat MATLAB file names at `file_path`
    file_path : str
        Defining the filepath to the location of the bridge data relative to this no
    char_lim : int, default 255
        Character limit for filepath when saving figures. A too long fliepath can le
    tower_dist : float, default 446
        Distance between the two towers of the bridge in meters.
    sag : float, default 45
        Sag of the main cables in meters, measured from the top of the towers.
    n_hangers : int, default 35
        Number of hanger pairs (hangers per main cable) between the towers.
    hanger_dist : float, default 12
```

```
    Distance between hanger pairs in y-direction in meters.
first_hanger_y_tower_dist : float, default 19
    Distance of the first hanger pair from the north tower in meters.
last_hanger_y_tower_dist : float, default 19
    Distance of the last hanger pair from the south tower in meters.
mid_span_cable_height_above_deck : float, default = 3
    Height of the main cables above the deck at mid-span in meters.
mid_span_deck_height : float, default 53.37
    Height of the deck above the waterline at mid-span in meters.
...
acc_x_dist_from_centerline : float, default 7.15/2
    Distance from the East and West accelerometers from the centerline of the br
g : float, default 9.818287
    Local gravity constant. Default value for stavanger measured according to 'G
anemo_range : float, default 65
    Anemometer range in +- m/s.
acc_range : float, default 5
    Accelerometer range in +- G.

"""
def __init__(self,file_names,file_path = '../Data/Bridge/dataExtracted/',char_li
             tower_dist = 446, # m
             h_towers = 102.26, # m
             sag = 45, # m
             n_hangers = 35,
             hanger_dist = 12, # m
             first_hanger_y_tower_dist = 19, # m
             last_hanger_y_tower_dist = 19, # m
             mid_span_cable_height_above_deck = 3, # m
             mid_span_deck_height = 53.37, # m
             tower_1_deck_support_height = 44.9, # m
             tower_2_deck_support_height = 52.36, # m
             girder_height = 2.76, # m
             cable_section_weight = 816, # kg/m
             hanger_section_weight = 95, # kg/m
             girder_section_weight = 5350, # kg/m
             cable_x_dist = 10.5, # m assumption
             deck_width = 9.5, # m assumption
             girder_width = 12.3, # m
             acc_x_dist_from_centerline = 7.15/2, # m
             g = 9.818287,
             anemo_range = 65,
             acc_range = 5
             ):
    """Initializes the class.

    Parameters
    ----------
    file_names : list of strs
        List of Lysefjord Bridge dataExtracted.mat MATLAB file names at `file_pa
    file_path : str
        Defining the filepath to the location of the bridge data relative to thi
    char_lim : int, default 255
        Character limit for filepath when saving figures. A too long fliepath ca
    tower_dist : float, default 446
        Distance between the two towers of the bridge in meters.
    sag : float, default 45
        Sag of the main cables in meters, measured from the top of the towers.
    n_hangers : int, default 35
        Number of hanger pairs (hangers per main cable) between the towers.
    hanger_dist : float, default 12
        Distance between hanger pairs in y-direction in meters.
    first_hanger_y_tower_dist : float, default 19
        Distance of the first hanger pair from the north tower in meters.
```

```
        last_hanger_y_tower_dist : float, default 19
            Distance of the last hanger pair from the south tower in meters.
        mid_span_cable_height_above_deck : float, default = 3
            Height of the main cables above the deck at mid-span in meters.
        mid_span_deck_height : float, default 53.37
            Height of the deck above the waterline at mid-span in meters.
        ...
        acc_x_dist_from_centerline : float, default 7.15/2
            Distance from the East and West accelerometers from the centerline of th
        g : float, default 9.818287
            Local gravity constant. Default value for stavanger measured according t
        anemo_range : float, default 65
            Anemometer range in +- m/s.
        acc_range : float, default 5
            Accelerometer range in +- G.
        """
        self.file_names = file_names
        self.file_path = file_path
        self.char_lim = char_lim

        self.tower_dist = tower_dist
        self.h_towers = h_towers
        self.sag = sag
        self.n_hangers = n_hangers
        self.hanger_dist = hanger_dist
        self.first_hanger_y_tower_dist = first_hanger_y_tower_dist
        self.last_hanger_y_tower_dist = last_hanger_y_tower_dist
        self.mid_span_cable_height_above_deck = mid_span_cable_height_above_deck
        self.mid_span_deck_height = mid_span_deck_height
        self.tower_1_deck_support_height = tower_1_deck_support_height
        self.tower_2_deck_support_height = tower_2_deck_support_height
        self.girder_height = girder_height
        self.cable_section_weight = cable_section_weight
        self.hanger_section_weight = hanger_section_weight
        self.girder_section_weight = girder_section_weight
        self.deck_width = deck_width
        self.girder_width = girder_width
        self.cable_x_dist = cable_x_dist
        self.acc_x_dist_from_centerline = acc_x_dist_from_centerline
        self.g = g
        self.anemo_range = anemo_range
        self.acc_range = acc_range
        self.acc_unit = 'µG'

    # Pre-processing
    def convert_MATLAB(self,data,delete_data_after_import=True,ignore_nans=True,rena
        """Convert Lysefjord Bridge dataExtracted.mat files imported via `scipy.io.l

        Parameters
        ----------
        data : dict of {str : ndarray}
            Dictionary with variable names as keys, and loaded matrices as values fr
        delete_data_after_import : bool, default True
            Delete original `data` from memory after import to free up memory.
        ignore_nans : bool, default True
            Ignores nans when calculating mean, std, min and max, using the remainin
        rename_H20_acc : bool, default True
            Rename accelerometers H20 to H24, as there seems to be a systematic erro
        replace_invalid : bool, default True
            Replace invalid sensor readings (outside of technical sensor range) with
        full_detail : bool, default False
            Keep full detail of `data`, sampled at 50Hz instead of resampling to eve
        H ... TopBottom : bool, default True
            Select which part of the `data` should be imported or generated during i
```

```
Returns
-------
anemo_names : list of str
    List of anemometer names.
acc_names : list of str
    List of accelerometer names.
time_array : array_like
t_array : array_like
anemo : dict of {str : array_like}
    Data from anemometers, 2d arrays sensor_id x measurement_id, grouped by
acc : dict of {str : array_like}
    Data from accelerometers, 2d arrays sensor_id x measurement_id, grouped
weather : dict of {str : array_like}
    Data from weatherstation, 1d arrays measurement_id, grouped by measureme
"""
# Convert time data.
time = data['time'].T[0]
t = data['t'][0]

time_array = time
t_array = np.repeat(t,time_array.shape[0])

# Get names of anemometers and accelerometers.
anemo_names = []
acc_names = []
for ind,anemo in enumerate(data['AnemoName'][0]):
    anemo_names.append(anemo[0])
for ind,acc in enumerate(data['AccelName'][0]):
    if rename_H20_acc:
        if acc[0] == 'H20':
            acc_names.append('H24')
        else:
            acc_names.append(acc[0])
    else:
        acc_names.append(acc[0])
anemo_names = np.array(anemo_names)
acc_names = np.array(acc_names)

anemo_shape = (len(anemo_names),len(time))
acc_shape = (len(acc_names),len(time))

# Initialize dicts for data from anemometers, weather and accelerometers.
anemo = {}
weather = {}
acc = {}

if replace_invalid:
    # Replace data out of technical range of sensors with nan.
    data['H'][(data['H']>self.anemo_range)|(data['H']<0)] = np.nan
    data['W'][(data['W']>self.anemo_range)|(data['W']<-self.anemo_range)] =
    data['Dir'] = np.where((data['Dir']>360)|(data['Dir']<0),np.nan,data['Di

    data['Aox_W'][(data['Aox_W']>self.acc_range*1e6)|(data['Aox_W']<-self.ac
    data['Aoy_W'][(data['Aoy_W']>self.acc_range*1e6)|(data['Aoy_W']<-self.ac
    data['Aoz_W'][(data['Aoz_W']>(self.acc_range+1)*1e6)|(data['Aoz_W']<-(se

    data['Aox_E'][(data['Aox_E']>self.acc_range*1e6)|(data['Aox_E']<-self.ac
    data['Aoy_E'][(data['Aoy_E']>self.acc_range*1e6)|(data['Aoy_E']<-self.ac
    data['Aoz_E'][(data['Aoz_E']>(self.acc_range+1)*1e6)|(data['Aoz_E']<-(se

# Decompose horizontal wind vector into vx and vy component
# according to the bridge coordinate system.
```

```
        vx = -np.sin(np.deg2rad(data['Dir']))*data['H']
        vy = -np.cos(np.deg2rad(data['Dir']))*data['H']


        if full_detail:
            # Retain full detail of the data, sampled at 50Hz instead of every 10min
            # Note that the data is described as _mean, although no mean is calculat
            # This enables use of functions orig. designed for the downsampled data.
            self.full_detail = True
            anemo_shape = (len(anemo_names),len(t)*len(time))
            acc_shape = (len(acc_names),len(t)*len(time))

            # Transform time arrays.
            time_array = np.linspace(time[0],time[-1],len(t)*len(time))
            t_array = np.tile(t,len(time))

            # Convert/create numpy arrays from original data
            # organized in dictionaries by type of measurement.
            if H:
                anemo['H_mean'] = data['H'].reshape(anemo_shape)
            if W:
                anemo['W_mean'] = data['W'].reshape(anemo_shape)

            if Vx or Dir or Upwind:
                anemo['Vx_mean'] = vx.reshape(anemo_shape)
            if Vy or Dir or Upwind:
                anemo['Vy_mean'] = vy.reshape(anemo_shape)

            if Dir or Upwind:
                anemo['Dir_mean'] = data['Dir'].reshape(anemo_shape)

            if Tv:
                anemo['Tv_mean'] = data['Tv'].reshape(anemo_shape)

            if AOA:
                anemo['AOA_mean'] = np.rad2deg(np.arctan(data['W']/data['H'])).resha

            if AOI:
                anemo['AOI_mean'] = np.where((0<data['Dir'])&(data['Dir']<=90),90-da


            if Aox_W:
                acc['Aox_W_mean'] = data['Aox_W'].reshape(acc_shape)

            if Aoy_W:
                acc['Aoy_W_mean'] = data['Aoy_W'].reshape(acc_shape)

            if Aoz_W:
                acc['Aoz_W_mean'] = data['Aoz_W'].reshape(acc_shape)

            if Aox_E:
                acc['Aox_E_mean'] = data['Aox_E'].reshape(acc_shape)

            if Aoy_E:
                acc['Aoy_E_mean'] = data['Aoy_E'].reshape(acc_shape)

            if Aoz_E:
                acc['Aoz_E_mean'] = data['Aoz_E'].reshape(acc_shape)

            if centr:
                acc['Aox_C_mean'] = ((data['Aox_W']+data['Aox_E'])/2).reshape(acc_sh
                acc['Aoy_C_mean'] = ((data['Aoy_W']+data['Aoy_E'])/2).reshape(acc_sh
                acc['Aoz_C_mean'] = ((data['Aoz_W']+data['Aoz_E'])/2).reshape(acc_sh
```

```
            if theta:
                acc['theta_mean'] = np.rad2deg(np.arctan(((data['Aoz_W'] - data['Aoz

            if T:
                weather['T_mean'] = data['T'].flatten()

            if P:
                weather['P_mean'] = data['P'].flatten()

            if Hum:
                weather['Hum_mean'] = data['Hum'].flatten()

        else:
            self.full_detail = False

            # Convert/create numpy arrays from original data
            # organized in dictionaries by type of measurement.
            if ignore_nans:
                # Use np.nanmean etc. instead of np.mean to ignore nans.
                if H:
                    anemo['H_mean'] = np.nanmean(data['H'],axis=2)
                    anemo['H_std'] = np.nanstd(data['H'],axis=2)
                    anemo['H_min'] = np.nanmin(data['H'],axis=2)
                    anemo['H_max'] = np.nanmax(data['H'],axis=2)

                if W:
                    anemo['W_mean'] = np.nanmean(data['W'],axis=2)
                    anemo['W_std'] = np.nanstd(data['W'],axis=2)
                    anemo['W_min'] = np.nanmin(data['W'],axis=2)
                    anemo['W_max'] = np.nanmax(data['W'],axis=2)

                if Vx or Dir or Upwind:
                    anemo['Vx_mean'] = np.nanmean(vx,axis=2)
                    anemo['Vx_std'] = np.nanstd(vx,axis=2)
                    anemo['Vx_min'] = np.nanmin(vx,axis=2)
                    anemo['Vx_max'] = np.nanmax(vx,axis=2)

                if Vy or Dir or Upwind:
                    anemo['Vy_mean'] = np.nanmean(vy,axis=2)
                    anemo['Vy_std'] = np.nanstd(vy,axis=2)
                    anemo['Vy_min'] = np.nanmin(vy,axis=2)
                    anemo['Vy_max'] = np.nanmax(vy,axis=2)

                if Dir or Upwind:
                    # Calculate mean of direction from Vx_mean and Vy_mean
                    # instead of original data to deal with 360-0 discontinuity.
                    anemo['Dir_mean'] = (np.rad2deg(np.arctan2(anemo['Vx_mean'],anem
                    #vv Physically irrelevant because of 360-0 discontinuity, kept f
                    anemo['Dir_std'] = (np.rad2deg(np.arctan2(anemo['Vx_std'],anemo[
                    anemo['Dir_min'] = (np.rad2deg(np.arctan2(anemo['Vx_min'],anemo[
                    anemo['Dir_max'] = (np.rad2deg(np.arctan2(anemo['Vx_max'],anemo[
                    #^^ Physically irrelevant because of 360-0 discontinuity, kept f

                if Tv:
                    anemo['Tv_mean'] = np.nanmean(data['Tv'],axis=2)
                    anemo['Tv_std'] = np.nanstd(data['Tv'],axis=2)
                    anemo['Tv_min'] = np.nanmin(data['Tv'],axis=2)
                    anemo['Tv_max'] = np.nanmax(data['Tv'],axis=2)

                if AOA:
                    anemo['AOA_mean'] = np.nanmean(np.rad2deg(np.arctan(data['W']/da
                    anemo['AOA_std'] = np.nanstd(np.rad2deg(np.arctan(data['W']/data
                    anemo['AOA_min'] = np.nanmin(np.rad2deg(np.arctan(data['W']/data
                    anemo['AOA_max'] = np.nanmax(np.rad2deg(np.arctan(data['W']/data
```

```
    if AOI:
        anemo['AOI_mean'] = np.nanmean(np.where((0<data['Dir'])&(data['D
        anemo['AOI_std'] = np.nanstd(np.where((0<data['Dir'])&(data['Dir
        anemo['AOI_min'] = np.nanmin(np.where((0<data['Dir'])&(data['Dir
        anemo['AOI_max'] = np.nanmax(np.where((0<data['Dir'])&(data['Dir


    if Aox_W:
        acc['Aox_W_mean'] = np.nanmean(data['Aox_W'],axis=2)
        acc['Aox_W_std'] = np.nanstd(data['Aox_W'],axis=2)
        acc['Aox_W_min'] = np.nanmin(data['Aox_W'],axis=2)
        acc['Aox_W_max'] = np.nanmax(data['Aox_W'],axis=2)
        acc['Aox_W_max_by_std'] = acc['Aox_W_max']/acc['Aox_W_std']

    if Aoy_W:
        acc['Aoy_W_mean'] = np.nanmean(data['Aoy_W'],axis=2)
        acc['Aoy_W_std'] = np.nanstd(data['Aoy_W'],axis=2)
        acc['Aoy_W_min'] = np.nanmin(data['Aoy_W'],axis=2)
        acc['Aoy_W_max'] = np.nanmax(data['Aoy_W'],axis=2)
        acc['Aoy_W_max_by_std'] = acc['Aoy_W_max']/acc['Aoy_W_std']

    if Aoz_W:
        acc['Aoz_W_mean'] = np.nanmean(data['Aoz_W'],axis=2)
        acc['Aoz_W_std'] = np.nanstd(data['Aoz_W'],axis=2)
        acc['Aoz_W_min'] = np.nanmin(data['Aoz_W'],axis=2)
        acc['Aoz_W_max'] = np.nanmax(data['Aoz_W'],axis=2)
        acc['Aoz_W_max_by_std'] = acc['Aoz_W_max']/acc['Aoz_W_std']

    if Aox_E:
        acc['Aox_E_mean'] = np.nanmean(data['Aox_E'],axis=2)
        acc['Aox_E_std'] = np.nanstd(data['Aox_E'],axis=2)
        acc['Aox_E_min'] = np.nanmin(data['Aox_E'],axis=2)
        acc['Aox_E_max'] = np.nanmax(data['Aox_E'],axis=2)
        acc['Aox_E_max_by_std'] = acc['Aox_E_max']/acc['Aox_E_std']

    if Aoy_E:
        acc['Aoy_E_mean'] = np.nanmean(data['Aoy_E'],axis=2)
        acc['Aoy_E_std'] = np.nanstd(data['Aoy_E'],axis=2)
        acc['Aoy_E_min'] = np.nanmin(data['Aoy_E'],axis=2)
        acc['Aoy_E_max'] = np.nanmax(data['Aoy_E'],axis=2)
        acc['Aoy_E_max_by_std'] = acc['Aoy_E_max']/acc['Aoy_E_std']

    if Aoz_E:
        acc['Aoz_E_mean'] = np.nanmean(data['Aoz_E'],axis=2)
        acc['Aoz_E_std'] = np.nanstd(data['Aoz_E'],axis=2)
        acc['Aoz_E_min'] = np.nanmin(data['Aoz_E'],axis=2)
        acc['Aoz_E_max'] = np.nanmax(data['Aoz_E'],axis=2)
        acc['Aoz_E_max_by_std'] = acc['Aoz_E_max']/acc['Aoz_E_std']

    if centr:
        acc['Aox_C_mean'] = np.nanmean((data['Aox_W']+data['Aoy_E'])/2,a
        acc['Aox_C_std'] = np.nanstd((data['Aox_W']+data['Aox_E'])/2,axi
        acc['Aox_C_min'] = np.nanmin((data['Aox_W']+data['Aox_E'])/2,axi
        acc['Aox_C_max'] = np.nanmax((data['Aox_W']+data['Aox_E'])/2,axi
        acc['Aox_C_max_by_std'] = acc['Aox_C_max']/acc['Aox_C_std']

        acc['Aoy_C_mean'] = np.nanmean((data['Aoy_W']+data['Aoy_E'])/2,a
        acc['Aoy_C_std'] = np.nanstd((data['Aoy_W']+data['Aoy_E'])/2,axi
        acc['Aoy_C_min'] = np.nanmin((data['Aoy_W']+data['Aoy_E'])/2,axi
        acc['Aoy_C_max'] = np.nanmax((data['Aoy_W']+data['Aoy_E'])/2,axi
        acc['Aoy_C_max_by_std'] = acc['Aoy_C_max']/acc['Aoy_C_std']

        acc['Aoz_C_mean'] = np.nanmean((data['Aoz_W']+data['Aoz_E'])/2,a
        acc['Aoz_C_std'] = np.nanstd((data['Aoz_W']+data['Aoz_E'])/2,axi
```

```
            acc['Aoz_C_min'] = np.nanmin((data['Aoz_W']+data['Aoz_E'])/2,axi
            acc['Aoz_C_max'] = np.nanmax((data['Aoz_W']+data['Aoz_E'])/2,axi
            acc['Aoz_C_max_by_std'] = acc['Aoz_C_max']/acc['Aoz_C_std']

        if theta:
            acc['theta_mean'] = np.nanmean(np.rad2deg(np.arctan(((data['Aoz_
            acc['theta_std'] = np.nanstd(np.rad2deg(np.arctan(((data['Aoz_W'
            acc['theta_min'] = np.nanmin(np.rad2deg(np.arctan(((data['Aoz_W'
            acc['theta_max'] = np.nanmax(np.rad2deg(np.arctan(((data['Aoz_W'
            acc['theta_max_by_std'] = acc['theta_max']/acc['theta_std']

        if T:
            weather['T_mean'] = np.nanmean(data['T'],axis=1)
            weather['T_std'] = np.nanstd(data['T'],axis=1)
            weather['T_min'] = np.nanmin(data['T'],axis=1)
            weather['T_max'] = np.nanmax(data['T'],axis=1)

        if P:
            weather['P_mean'] = np.nanmean(data['P'],axis=1)
            weather['P_std'] = np.nanstd(data['P'],axis=1)
            weather['P_min'] = np.nanmin(data['P'],axis=1)
            weather['P_max'] = np.nanmax(data['P'],axis=1)

        if Hum:
            weather['Hum_mean'] = np.nanmean(data['Hum'],axis=1)
            weather['Hum_std'] = np.nanstd(data['Hum'],axis=1)
            weather['Hum_min'] = np.nanmin(data['Hum'],axis=1)
            weather['Hum_max'] = np.nanmax(data['Hum'],axis=1)


    else:
        if H:
            anemo['H_mean'] = np.mean(data['H'],axis=2)
            anemo['H_std'] = np.std(data['H'],axis=2)
            anemo['H_min'] = np.min(data['H'],axis=2)
            anemo['H_max'] = np.max(data['H'],axis=2)

        if W:
            anemo['W_mean'] = np.mean(data['W'],axis=2)
            anemo['W_std'] = np.std(data['W'],axis=2)
            anemo['W_min'] = np.min(data['W'],axis=2)
            anemo['W_max'] = np.max(data['W'],axis=2)

        if Vx or Dir or Upwind:
            anemo['Vx_mean'] = np.mean(vx,axis=2)
            anemo['Vx_std'] = np.std(vx,axis=2)
            anemo['Vx_min'] = np.min(vx,axis=2)
            anemo['Vx_max'] = np.max(vx,axis=2)

        if Vy or Dir or Upwind:
            anemo['Vy_mean'] = np.mean(vy,axis=2)
            anemo['Vy_std'] = np.std(vy,axis=2)
            anemo['Vy_min'] = np.min(vy,axis=2)
            anemo['Vy_max'] = np.max(vy,axis=2)

        if Dir or Upwind:
            # Calculate mean of direction from Vx_mean and Vy_mean
            # instead of original data to deal with 360-0 discontinuity.
            anemo['Dir_mean'] = (np.rad2deg(np.arctan2(anemo['Vx_mean'],anem
            #vv Physically irrelevant because of 360-0 discontinuity, kept f
            anemo['Dir_std'] = (np.rad2deg(np.arctan2(anemo['Vx_std'],anemo[
            anemo['Dir_min'] = (np.rad2deg(np.arctan2(anemo['Vx_min'],anemo[
            anemo['Dir_max'] = (np.rad2deg(np.arctan2(anemo['Vx_max'],anemo[
            #^^ Physically irrelevant because of 360-0 discontinuity, kept f
```

```
    if Tv:
        anemo['Tv_mean'] = np.mean(data['Tv'],axis=2)
        anemo['Tv_std'] = np.std(data['Tv'],axis=2)
        anemo['Tv_min'] = np.min(data['Tv'],axis=2)
        anemo['Tv_max'] = np.max(data['Tv'],axis=2)

    if AOA:
        anemo['AOA_mean'] = np.mean(np.rad2deg(np.arctan(data['W']/data[
        anemo['AOA_std'] = np.std(np.rad2deg(np.arctan(data['W']/data['H
        anemo['AOA_min'] = np.min(np.rad2deg(np.arctan(data['W']/data['H
        anemo['AOA_max'] = np.max(np.rad2deg(np.arctan(data['W']/data['H
    if AOI:
        anemo['AOI_mean'] = np.mean(np.where((0<data['Dir'])&(data['Dir'
        anemo['AOI_std'] = np.std(np.where((0<data['Dir'])&(data['Dir']<
        anemo['AOI_min'] = np.min(np.where((0<data['Dir'])&(data['Dir']<
        anemo['AOI_max'] = np.max(np.where((0<data['Dir'])&(data['Dir']<


    if Aox_W:
        acc['Aox_W_mean'] = np.mean(data['Aox_W'],axis=2)
        acc['Aox_W_std'] = np.std(data['Aox_W'],axis=2)
        acc['Aox_W_min'] = np.min(data['Aox_W'],axis=2)
        acc['Aox_W_max'] = np.max(data['Aox_W'],axis=2)
        acc['Aox_W_max_by_std'] = acc['Aox_W_max']/acc['Aox_W_std']

    if Aoy_W:
        acc['Aoy_W_mean'] = np.mean(data['Aoy_W'],axis=2)
        acc['Aoy_W_std'] = np.std(data['Aoy_W'],axis=2)
        acc['Aoy_W_min'] = np.min(data['Aoy_W'],axis=2)
        acc['Aoy_W_max'] = np.max(data['Aoy_W'],axis=2)
        acc['Aoy_W_max_by_std'] = acc['Aoy_W_max']/acc['Aoy_W_std']

    if Aoz_W:
        acc['Aoz_W_mean'] = np.mean(data['Aoz_W'],axis=2)
        acc['Aoz_W_std'] = np.std(data['Aoz_W'],axis=2)
        acc['Aoz_W_min'] = np.min(data['Aoz_W'],axis=2)
        acc['Aoz_W_max'] = np.max(data['Aoz_W'],axis=2)
        acc['Aoz_W_max_by_std'] = acc['Aoz_W_max']/acc['Aoz_W_std']

    if Aox_E:
        acc['Aox_E_mean'] = np.mean(data['Aox_E'],axis=2)
        acc['Aox_E_std'] = np.std(data['Aox_E'],axis=2)
        acc['Aox_E_min'] = np.min(data['Aox_E'],axis=2)
        acc['Aox_E_max'] = np.max(data['Aox_E'],axis=2)
        acc['Aox_E_max_by_std'] = acc['Aox_E_max']/acc['Aox_E_std']

    if Aoy_E:
        acc['Aoy_E_mean'] = np.mean(data['Aoy_E'],axis=2)
        acc['Aoy_E_std'] = np.std(data['Aoy_E'],axis=2)
        acc['Aoy_E_min'] = np.min(data['Aoy_E'],axis=2)
        acc['Aoy_E_max'] = np.max(data['Aoy_E'],axis=2)
        acc['Aoy_E_max_by_std'] = acc['Aoy_E_max']/acc['Aoy_E_std']

    if Aoz_E:
        acc['Aoz_E_mean'] = np.mean(data['Aoz_E'],axis=2)
        acc['Aoz_E_std'] = np.std(data['Aoz_E'],axis=2)
        acc['Aoz_E_min'] = np.min(data['Aoz_E'],axis=2)
        acc['Aoz_E_max'] = np.max(data['Aoz_E'],axis=2)
        acc['Aoz_E_max_by_std'] = acc['Aoz_E_max']/acc['Aoz_E_std']

    if centr:
        acc['Aox_C_mean'] = np.mean((data['Aox_W']+data['Aox_E'])/2,axis
        acc['Aox_C_std'] = np.std((data['Aox_W']+data['Aox_E'])/2,axis=2
```

```
                acc['Aox_C_min'] = np.min((data['Aox_W']+data['Aox_E'])/2,axis=2
                acc['Aox_C_max'] = np.max((data['Aox_W']+data['Aox_E'])/2,axis=2
                acc['Aox_C_max_by_std'] = acc['Aox_C_max']/acc['Aox_C_std']

                acc['Aoy_C_mean'] = np.mean((data['Aoy_W']+data['Aoy_E'])/2,axis
                acc['Aoy_C_std'] = np.std((data['Aoy_W']+data['Aoy_E'])/2,axis=2
                acc['Aoy_C_min'] = np.min((data['Aoy_W']+data['Aoy_E'])/2,axis=2
                acc['Aoy_C_max'] = np.max((data['Aoy_W']+data['Aoy_E'])/2,axis=2
                acc['Aoy_C_max_by_std'] = acc['Aoy_C_max']/acc['Aoy_C_std']

                acc['Aoz_C_mean'] = np.mean((data['Aoz_W']+data['Aoz_E'])/2,axis
                acc['Aoz_C_std'] = np.std((data['Aoz_W']+data['Aoz_E'])/2,axis=2
                acc['Aoz_C_min'] = np.min((data['Aoz_W']+data['Aoz_E'])/2,axis=2
                acc['Aoz_C_max'] = np.max((data['Aoz_W']+data['Aoz_E'])/2,axis=2
                acc['Aoz_C_max_by_std'] = acc['Aoz_C_max']/acc['Aoz_C_std']

            if theta:
                acc['theta_mean'] = np.mean(np.rad2deg(np.arctan(((data['Aoz_W']
                acc['theta_std'] = np.std(np.rad2deg(np.arctan(((data['Aoz_W'] -
                acc['theta_min'] = np.min(np.rad2deg(np.arctan(((data['Aoz_W'] -
                acc['theta_max'] = np.max(np.rad2deg(np.arctan(((data['Aoz_W'] -
                acc['theta_max_by_std'] = acc['theta_C_max']/acc['theta_C_std']


            if T:
                weather['T_mean'] = np.mean(data['T'],axis=1)
                weather['T_std'] = np.std(data['T'],axis=1)
                weather['T_min'] = np.min(data['T'],axis=1)
                weather['T_max'] = np.max(data['T'],axis=1)

            if P:
                weather['P_mean'] = np.mean(data['P'],axis=1)
                weather['P_std'] = np.std(data['P'],axis=1)
                weather['P_min'] = np.min(data['P'],axis=1)
                weather['P_max'] = np.max(data['P'],axis=1)

            if Hum:
                weather['Hum_mean'] = np.mean(data['Hum'],axis=1)
                weather['Hum_std'] = np.std(data['Hum'],axis=1)
                weather['Hum_min'] = np.min(data['Hum'],axis=1)
                weather['Hum_max'] = np.max(data['Hum'],axis=1)
        if Turb:
            if H:
                anemo['H_turb'] = anemo['H_std']/np.sqrt(np.square(anemo['H_mean
            if W:
                anemo['W_turb'] = np.abs(anemo['W_std']/np.sqrt(np.square(anemo[
            if Vx:
                anemo['Vx_turb'] = np.abs(anemo['Vx_std']/np.sqrt(np.square(anem
            if Vy:
                anemo['Vy_turb'] = np.abs(anemo['Vy_std']/np.sqrt(np.square(anem

    if Upwind:
        # Create boolean arrays determening if a sensor is up- or downwind
        # at each datapoint, depending on direction measurement.
        anemo['Upwind'] = np.zeros(anemo_shape)
        for ind,anemo_name in enumerate(anemo_names):
            if anemo_name.endswith('W') or anemo_name.endswith('Wb') or anemo_na
                anemo['Upwind'][ind] = np.where(anemo['Dir_mean'][ind]>=180,1,0)
            elif anemo_name.endswith('E') or anemo_name.endswith('Eb') or anemo_
                anemo['Upwind'][ind] = np.where(anemo['Dir_mean'][ind]<180,1,0)
        anemo['Downwind'] = (~anemo['Upwind'].astype(bool)).astype(int)

    if Hanger_num or y_pos:
        anemo['Hanger_num'] = np.zeros(anemo_shape)
```

```
            acc['Hanger_num'] = np.zeros(acc_shape)
            for ind,anemo_name in enumerate(anemo_names):
                if anemo_name.startswith('H'):
                    anemo['Hanger_num'][ind] = np.repeat(int(anemo_name[1:3]),anemo_
                elif anemo_name.startswith('S'):
                    anemo['Hanger_num'][ind] = np.repeat(int(self.n_hangers+1),anemo
            for ind,acc_name in enumerate(acc_names):
                if acc_name.startswith('H'):
                    acc['Hanger_num'][ind] = np.repeat(int(acc_name[1:3]),acc_shape[
                elif acc_name.startswith('S'):
                    acc['Hanger_num'][ind] = np.repeat(int(self.n_hangers+1),acc_sha

        if y_pos:
            anemo['y_pos'] = np.where(anemo['Hanger_num']==0,self.tower_dist,np.wher
            acc['y_pos'] = np.where(acc['Hanger_num']==0,self.tower_dist,np.where(ac

        if WestEast:
            anemo['West'] = np.zeros(anemo_shape)
            anemo['East'] = np.zeros(anemo_shape)
            for ind,anemo_name in enumerate(anemo_names):
                if anemo_name.endswith('W') or anemo_name.endswith('Wt') or anemo_na
                    anemo['West'][ind] = np.repeat(True,anemo_shape[1])
                    anemo['East'][ind] = np.repeat(False,anemo_shape[1])
                elif anemo_name.endswith('E') or anemo_name.endswith('Et') or anemo_
                    anemo['West'][ind] = np.repeat(False,anemo_shape[1])
                    anemo['East'][ind] = np.repeat(True,anemo_shape[1])
                else: # Assume West for sensors without declaration as they might be
                    anemo['West'][ind] = np.repeat(True,anemo_shape[1])
                    anemo['East'][ind] = np.repeat(False,anemo_shape[1])
            # Not applicable for accelerometers.
            acc['West'] = np.zeros(acc_shape)
            acc['East'] = np.zeros(acc_shape)

        if TopBottom:
            anemo['Top'] = np.zeros(anemo_shape)
            anemo['Bottom'] = np.zeros(anemo_shape)
            for ind,anemo_name in enumerate(anemo_names):
                if anemo_name.endswith('Wb') or anemo_name.endswith('Eb') or anemo_n
                    anemo['Top'][ind] = np.repeat(False,anemo_shape[1])
                    anemo['Bottom'][ind] = np.repeat(True,anemo_shape[1])
                elif anemo_name.endswith('Wt') or anemo_name.endswith('Et'):
                    anemo['Top'][ind] = np.repeat(True,anemo_shape[1])
                    anemo['Bottom'][ind] = np.repeat(False,anemo_shape[1])
                else: # Assume Bottom for sensors without declaration as they might
                    anemo['Top'][ind] = np.repeat(False,anemo_shape[1])
                    anemo['Bottom'][ind] = np.repeat(True,anemo_shape[1])
            # Not applicable for accelerometers.
            acc['Top'] = np.zeros(acc_shape)
            acc['Bottom'] = np.zeros(acc_shape)


        if delete_data_after_import:
            # Delete original data after import to free up memory.
            del data
            gc.collect()

        return anemo_names, acc_names, time_array, t_array, anemo, acc, weather

    def define_units(self,accs_in_SI_units=True):
        """Method to convert units and define units of all measurements in a diction

        Parameters
        ----------
        accs_in_SI_units : bool, default True
```

```
            Whether to have measurements from accelerometers in m/s (SI-unit) or µG


        """
        self.units = {}
        for key in self.anemo.keys():
            if key.startswith('H_') or key.startswith('W_') or key.startswith('Vx_')
                self.units[key]='[m/s]'
            elif key.startswith('Dir_') or key.startswith('AOA_') or key.startswith(
                self.units[key]='[°]'
            elif key.startswith('Tv_'):
                self.units[key]='[°C]'
            elif key.startswith('Upwind') or key.startswith('Downwind') or key.start
                self.units[key]=''
            elif key.startswith('y_pos'):
                self.units[key]='[m]'
            if key.endswith('_turb'):
                self.units[key]=''

        for key in self.weather.keys():
            if key.startswith('T'):
                self.units[key]='[°C]'
            elif key.startswith('P'):
                self.units[key]='[hPa]'
            elif key.startswith('Hum'):
                self.units[key]='[%]'

        for key in self.acc.keys():
            if key.startswith('Ao') and not key.endswith('_by_std'):
                if accs_in_SI_units:
                    if self.acc_unit == 'µG':
                        self.acc[key] = self.acc[key]*(1e-6*self.g)
                    self.units[key]='[m s$^{-2}$]'
                else:
                    if self.acc_unit == 'SI':
                        self.acc[key] = self.acc[key]/(1e-6*self.g)
                    self.units[key]='[µG]'
            elif key.startswith('theta') and not key.endswith('_by_std'):
                self.units[key]='[° s$^{-2}$]'
            elif key.endswith('_by_std'):
                self.units[key]=''


        if accs_in_SI_units:
            self.acc_unit = 'SI'
        else:
            self.acc_unit = 'µG'

    def load_data(self,print_data_structure=False,delete_data_after_import=True,igno
        """Wrapper-method to import and combine multiple days of data using `convert

        Convert Lysefjord Bridge dataExtracted.mat files imported via `scipy.io.load

        Parameters
        ----------
        print_data_structure : bool, default False
            Print the structure of the original data. Primarily used for developemen
        delete_data_after_import : bool, default True
            Delete original data from memory after import to free up memory
        ignore_nans : bool, default True
            Ignores nans when calculating mean, std, min and max, using the remainin
        rename_H20_acc : bool, default True
            Rename accelerometers H20 to H24, as there seems to be a systematic erro
        replace_invalid : bool, default True
```

```
        Replace invalid sensor readings (outside of technical sensor range) with
    full_detail : bool, default False
        Keep full detail of data, sampled at 50Hz instead of resampling to every
    H ... TopBottom : bool, default True
        Select which part of the data should be imported or generated during imp

    See Also
    --------
    `convert_MATLAB`
    `define_units`
    """
    # Initialize with data of the first day in the list of file_names.
    print('Day',1,':',self.file_names[0][:-4][14:])
    temp_data = scipy.io.loadmat(self.file_path+self.file_names[0])
    if print_data_structure:
        print(temp_data)
    self.anemo_names, self.acc_names, self.time_array, self.t_array, self.anemo,
    del temp_data
    gc.collect()

    # Batch-processing of further days of data, if any,
    # with temporary variables being created for each day
    # and deleted afterwards to free up memory for processing of the next day.
    for day in range(1,len(self.file_names)):
        print('Day',day+1,':',self.file_names[day][:-4][14:])
        temp_data = scipy.io.loadmat(self.file_path+self.file_names[day])
        temp_anemo_names, temp_acc_names, temp_time_array, temp_t_array, temp_an

        del temp_data
        gc.collect()

        anemo_names,ind = np.unique(np.append(self.anemo_names,temp_anemo_names)
        self.anemo_names = anemo_names[np.argsort(ind)]

        acc_names,ind = np.unique(np.append(self.acc_names,temp_acc_names), retu
        self.acc_names = acc_names[np.argsort(ind)]

        for key in temp_anemo.keys():
            if key in self.anemo.keys():
                self.anemo[key] = np.hstack((self.anemo[key],temp_anemo[key]))
            else:
                self.anemo[key] = np.hstack((np.zeros_like(self.time_array),temp

        for key in temp_acc.keys():
            if key in self.acc.keys():
                self.acc[key] = np.hstack((self.acc[key],temp_acc[key]))
            else:
                self.acc[key] = np.hstack((np.zeros_like(self.time_array),temp_a

        for key in temp_weather.keys():
            if key in self.weather.keys():
                self.weather[key] = np.hstack((self.weather[key],temp_weather[ke
            else:
                self.weather[key] = np.hstack((np.zeros_like(self.time_array),te

        self.time_array = np.hstack((self.time_array, temp_time_array))
        self.t_array = np.hstack((self.t_array, temp_t_array))

        del temp_anemo_names, temp_acc_names, temp_time_array, temp_t_array, tem
        gc.collect()

    self.define_units(accs_in_SI_units)

# Processing
```

```
    def find_invalid_sensors(self,data,threshold=0.5,lp_cutoff=np.inf,hp_cutoff=-np.
        """Helper-method for data cleaning finding invalid sensors in `data`.

        Parameters
        ----------
        data : array_like
            Measurement data in which invalid sensors are to be found.
        threshold : float, default 0.5
            Fraction of data from a specific sensor that is allowed to be invalid be
        lp_cutoff : float or int, default np.inf
            Lowpass cutoff point: If a value is higher than `lp_cutoff`it is interpr
        hp_cutoff : float or int, default -np.inf
            Highpass cutoff point: If a value is lower than `hp_cutoff`it is interpr
        zeros : bool, default False
            Zeros are interpreted as invalid values.
        nans : bool, default True
            nans are interpreted as invalid values.
        lowpass : bool, default False
            Values above `lp_cutoff` are interpreted as invalid values.
        highpass : bool, default False
            Values below `hp_cutoff` are interpreted as invalid values.

        Returns
        -------
        invalid_sensor_id : array_like
            ids of invalid sensors.
        ok_sensor_id : array_like
            ids of ok sensors.

        See Also
        --------
        `find_common_ok_sensors`
        `remove_invalid_sensors`
        """
        invalid_sensor_id = np.array([])

        if zeros:
            potentially_invalid_sensors, count = np.unique(np.where(data==0)[0],retu
            invalid_sensor_id = np.unique(np.append(invalid_sensor_id,potentially_in

        if nans:
            potentially_invalid_sensors, count = np.unique(np.where(np.isnan(data))[
            invalid_sensor_id = np.unique(np.append(invalid_sensor_id,potentially_in

        if lowpass:
            potentially_invalid_sensors, count = np.unique(np.where(data>lp_cutoff)[
            invalid_sensor_id = np.unique(np.append(invalid_sensor_id,potentially_in

        if highpass:
            potentially_invalid_sensors, count = np.unique(np.where(data<hp_cutoff)[
            invalid_sensor_id = np.unique(np.append(invalid_sensor_id,potentially_in

        invalid_sensor_id = np.array([int(i) for i in invalid_sensor_id]).astype(int
        ok_sensor_id = np.array(list(set(np.arange(data.shape[0]))-(set(invalid_sens

        return invalid_sensor_id, ok_sensor_id

    def find_common_ok_sensors(self,ok_sensor_ids_s1,ok_sensor_ids_s2):
        """Method to find common ok sensors.

        Usefull for preparation of post-processing.

        Parameters
        ----------
```

```
        ok_sensor_ids_s1 : list or array_like of ints
            ids of ok sensors from measurement type 1.
        ok_sensor_ids_s2 : list or array_like of ints
            ids of ok sensors from measurement type 2.

        Returns
        -------
        common_sensor_id : list
            Sensor ids of ok sensors in both sets of measurements.
        s1_ind : list
            Sensor indices of common ok sensors for measurement type 1.
        s2_ind : list
            Sensor indices of common ok sensors for measurement type 2.

        See Also
        --------
        `find_invalid_sensors`
        `get_ok_sensor_ind`
        """
        s1_ind = []
        for ind, sensor_id in enumerate(ok_sensor_ids_s1):
            if sensor_id in ok_sensor_ids_s2:
                s1_ind.append(ind)
        s2_ind = []
        for ind, sensor_id in enumerate(ok_sensor_ids_s2):
            if sensor_id in ok_sensor_ids_s1:
                s2_ind.append(ind)

        common_sensor_id = list(set(set(ok_sensor_ids_s1) & set(ok_sensor_ids_s2)))
        return common_sensor_id, s1_ind, s2_ind

    def remove_invalid_sensors(self,data,ok_sensor_id):
        """Helper-method for data cleaning removing invalid sensors from `data`.

        Parameters
        ----------
        data : array_like
            Measurement data in which invalid sensors are to be removed.
        ok_sensor_id : array_like
            Index of ok sensors in measurement data.

        Returns
        -------
        data_cleaned : array_like
            `data` without invalid sensors.

        See Also
        --------
        `find_invalid_sensors`
        `find_common_ok_sensors`
        """
        if len(ok_sensor_id)>0:
            data_cleaned = data[ok_sensor_id]
        else:
            data_cleaned = []
        return data_cleaned

    def idx_data(self,data,nans=False,zeros=False,lp_cutoff=np.inf,hp_cutoff=-np.inf
        """Helper-method indexing data.

        Parameters
        ----------
        data : array_like
            Measurement data in which data is to be indexed.
```

```
        lp_cutoff : float or int, default np.inf
            Lowpass cutoff point: If a value is higher than `lp_cutoff`it is indexed
        hp_cutoff : float or int, default -np.inf
            Highpass cutoff point: If a value is lower than `lh_cutoff`it is indexed
        nans : bool, default False
            Nans are indexed.
        zeros : bool, default False
            Zeros are indexed.
        lowpass : bool, default False
            Values above `lp_cutoff` are indexed.
        highpass : bool, default False
            Values below `hp_cutoff` are indexed.

        Returns
        -------
        idx : array_like
            Indices of data.
        """
        if len(data):
            idx = np.array([])
            if nans:
                idx = np.union1d(idx,np.argwhere(np.any(np.isnan(data[..., :]), axis
            if zeros:
                idx = np.union1d(idx,np.argwhere(np.any(data[..., :] == 0, axis=0)))
            if lowpass:
                idx = np.union1d(idx,np.argwhere(np.any(data[..., :] > lp_cutoff, ax
            if highpass:
                idx = np.union1d(idx,np.argwhere(np.any(data[..., :] < hp_cutoff, ax
            idx = np.unique(idx).astype(int)
            return idx
        else:
            raise ValueError('Data is empty.')

    def clean_data(self,delete_original_data=False,threshold=0.5,detailed_report=Fal
        """Wrapper-method for data cleaning using `find_invalid_sensors`, `remove_in

        Notes
        -----
        If `remove_invalid` is `True` in `load_data`, `clean_data` might not find mo

        Parameters
        ----------
        delete_original_data : bool, default False
            Delete the original (un-cleaned) `data` to free up memory.
        threshold : float, default 0.5
            Fraction of data from a specific sensor that is allowed to be invalid be
        detailed_report : bool, default False
            Print out a detailed report of the data cleaning. Used for de-bugging an

        See Also
        --------
        `find_invalid_sensors`
        `find_common_ok_sensors`
        `get_ok_sensor_ind`
        `remove_invalid_sensors`
        `idx_data`
        """
        # Initialize dicts for cleaned data.
        self.anemo_cleaned = {}
        self.anemo_invalid_sensor_id = {}
        self.anemo_ok_sensor_id = {}
        self.acc_cleaned = {}
        self.acc_invalid_sensor_id = {}
        self.acc_ok_sensor_id = {}
```

```
        self.weather_cleaned = {}
        if self.acc_unit == 'µG':
            acc_thresh_unit_fact = 1e6
        else:
            acc_thresh_unit_fact = self.g

        # Anemometers
        for key in self.anemo.keys():
            # Find invalid sensors in each key.
            if key.startswith('H_m'):
                self.anemo_invalid_sensor_id[key], self.anemo_ok_sensor_id[key] = se
            elif key.startswith('Vx_m') or key.startswith('Vy_m') or key.startswith(
                self.anemo_invalid_sensor_id[key], self.anemo_ok_sensor_id[key] = se
            elif key.startswith('Dir_m'):
                self.anemo_invalid_sensor_id[key], self.anemo_ok_sensor_id[key] = se
            elif key.endswith('_std') or key.endswith('_turb'):
                self.anemo_invalid_sensor_id[key], self.anemo_ok_sensor_id[key] = se
            else:
                self.anemo_invalid_sensor_id[key], self.anemo_ok_sensor_id[key] = se

            # Combine invalid sensors from keys of same sensor.
            if not key.endswith('_turb'): # Not applicable for turbulence calculatio
                if key[:key.find('_')] not in self.anemo_invalid_sensor_id:
                    self.anemo_invalid_sensor_id[key[:key.find('_')]] = self.anemo_i
                else:
                    self.anemo_invalid_sensor_id[key[:key.find('_')]] = np.unique(np

            if len(self.anemo_ok_sensor_id[key]) == 0:
                print(key,'all sensors deemed invalid')
                try:
                    del self.anemo_cleaned[key], self.anemo_invalid_sensor_id[key],
                except:
                    print(key, 'already deleted')
            elif len(self.anemo_invalid_sensor_id[key]) == 0:
                if detailed_report:
                    print(key,'all sensors deemed ok')
            else:
                if detailed_report:
                    print(key,'ok sensors:',self.anemo_names[self.anemo_ok_sensor_id
                    print(key,'invalid sensors:',self.anemo_names[self.anemo_invalid

        # Remove invalid sensors for all keys of same sensor.
        for key in self.anemo.keys():

            self.anemo_invalid_sensor_id[key] = self.anemo_invalid_sensor_id[key[:ke

            if key.startswith('AOA') or key.endswith('_turb'): # AOA and turbulence
                self.anemo_invalid_sensor_id[key] = np.unique(np.append(self.anemo_i
                self.anemo_invalid_sensor_id[key] = np.unique(np.append(self.anemo_i
            elif key.startswith('Vx') or key.startswith('Vy') or key.startswith('Dir
                self.anemo_invalid_sensor_id[key] = np.unique(np.append(self.anemo_i

            self.anemo_ok_sensor_id[key] = np.array(list(set(np.arange(self.anemo[ke
            self.anemo_cleaned[key] = self.remove_invalid_sensors(self.anemo[key],se

            if len(self.anemo_ok_sensor_id[key]) == 0:
                print(key,'all sensors deemed invalid')
                try:
                    del self.anemo_cleaned[key], self.anemo_invalid_sensor_id[key],
                except:
                    print(key, 'already deleted')
            elif len(self.anemo_invalid_sensor_id[key]) == 0:
                if detailed_report:
                    print(key,'all sensors deemed ok')
```

```
            else:
                if detailed_report:
                    print(key,'ok sensors:',self.anemo_names[self.anemo_ok_sensor_id
                    print(key,'invalid sensors:',self.anemo_names[self.anemo_invalid

        # Accelerometers
        for key in self.acc.keys():
            # Find invalid sensors in each key.
            if key.endswith('_max'):
                if key.startswith('Aoz_'):
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
                elif key.startswith('Ao'):
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
                else:
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
            elif key.endswith('_min'):
                if key.startswith('Aoz_'):
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
                elif key.startswith('Ao'):
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
                else:
                    self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = se
            else:
                self.acc_invalid_sensor_id[key], self.acc_ok_sensor_id[key] = self.f

            # Combine invalid sensors from keys of same sensor.
            if key.startswith('Aox_W') or key.startswith('Aoy_W') or key.startswith(
                if 'W_all' not in self.acc_invalid_sensor_id:
                    self.acc_invalid_sensor_id['W_all'] = self.acc_invalid_sensor_id
                else:
                    self.acc_invalid_sensor_id['W_all'] = np.unique(np.append(self.a
            elif key.startswith('Aox_E') or key.startswith('Aoy_E') or key.startswit
                if 'E_all' not in self.acc_invalid_sensor_id:
                    self.acc_invalid_sensor_id['E_all'] = self.acc_invalid_sensor_id
                else:
                    self.acc_invalid_sensor_id['E_all'] = np.unique(np.append(self.a
            elif key.startswith('Aox_C') or key.startswith('Aoy_C') or key.startswit
                if 'C_all' not in self.acc_invalid_sensor_id:
                    self.acc_invalid_sensor_id['C_all'] = self.acc_invalid_sensor_id
                else:
                    self.acc_invalid_sensor_id['C_all'] = np.unique(np.append(self.a

            if len(self.acc_ok_sensor_id[key]) == 0:
                print(key,'all sensors deemed invalid')
                try:
                    del self.acc_cleaned[key], self.acc_invalid_sensor_id[key], self
                except:
                    print(key, 'already deleted')
            elif len(self.acc_invalid_sensor_id[key]) == 0:
                if detailed_report:
                    print(key,'all sensors deemed ok')
            else:
                if detailed_report:
                    print(key,'ok sensors:',self.acc_names[self.acc_ok_sensor_id[key
                    print(key,'invalid sensors:',self.acc_names[self.acc_invalid_sen

        # Remove invalid sensors for all keys of same sensor.
        for key in self.acc.keys():
            if key.startswith('Aox_W') or key.startswith('Aoy_W') or key.startswith(
                self.acc_invalid_sensor_id[key] = self.acc_invalid_sensor_id['W_all'
            elif key.startswith('Aox_E') or key.startswith('Aoy_E') or key.startswit
                self.acc_invalid_sensor_id[key] = self.acc_invalid_sensor_id['E_all'
            elif key.startswith('Aox_C') or key.startswith('Aoy_C') or key.startswit
                self.acc_invalid_sensor_id[key] = self.acc_invalid_sensor_id['C_all'
```

```
            if key.startswith('Aox_C'): #A ox_C dependent on Aox_W and Aox_E.
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval
            elif key.startswith('Aoy_C'): # Aoy_C dependent on Aoy_W and Aoy_E.
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval
            if key.startswith('Aoz_C') or key.startswith('theta'): # Aoz_C and theta
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval
                self.acc_invalid_sensor_id[key] = np.unique(np.append(self.acc_inval

            self.acc_ok_sensor_id[key] = np.array(list(set(np.arange(self.acc[key].s
            self.acc_cleaned[key] = self.remove_invalid_sensors(self.acc[key],self.a

            if len(self.acc_ok_sensor_id[key]) == 0:
                print(key,'all sensors deemed invalid')
                try:
                    del self.acc_cleaned[key], self.acc_invalid_sensor_id[key], self
                except:
                    print(key, 'already deleted')
            elif len(self.acc_invalid_sensor_id[key]) == 0:
                if detailed_report:
                    print(key,'all sensors deemed ok')
            else:
                if detailed_report:
                    print(key,'ok sensors:',self.acc_names[self.acc_ok_sensor_id[key
                    print(key,'invalid sensors:',self.acc_names[self.acc_invalid_sen

        # Find outliers in remaining sensors.
        idx = np.array([])

        # Anemometers
        for key in self.anemo_cleaned.keys():
            if key.startswith('Dir_m'):
                idx = np.union1d(idx,self.idx_data(self.anemo_cleaned[key],nans=True
            elif key.startswith('H_m'):
                idx = np.union1d(idx,self.idx_data(self.anemo_cleaned[key],nans=True
            elif key.startswith('Vx_m') or key.startswith('Vy_m') or key.startswith(
                idx = np.union1d(idx,self.idx_data(self.anemo_cleaned[key],nans=True
            elif key.endswith('_std') or key.endswith('_turb'):
                idx = np.union1d(idx,self.idx_data(self.anemo_cleaned[key],nans=True
            else:
                idx = np.union1d(idx,self.idx_data(self.anemo_cleaned[key],nans=True
            if detailed_report:
                print(key,' cummulative data loss:',np.round(100*(len(idx)/self.time

        # Accelerometers
        for key in self.acc_cleaned.keys():
            if key.endswith('_max'):
                if key.startswith('Aoz_'):
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
                elif key.startswith('Ao'):
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
                else:
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
            elif key.endswith('_min'):
                if key.startswith('Aoz_'):
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
                elif key.startswith('Ao'):
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
                else:
                    idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=Tr
            else:
                idx = np.union1d(idx,self.idx_data(self.acc_cleaned[key],nans=True,z
```

```
        idx = idx.astype(int)

        # Remove outliers from all sensors.
        self.time_array_cleaned = np.delete(self.time_array,idx,axis=0)
        self.t_array_cleaned = np.delete(self.t_array,idx,axis=0)

        for key in self.weather.keys():
            self.weather_cleaned[key] = np.delete(self.weather[key],idx,axis=0)

        for key in self.anemo_cleaned.keys():
            self.anemo_cleaned[key] = np.delete(self.anemo_cleaned[key],idx,axis=1)

        for key in self.acc_cleaned.keys():
            self.acc_cleaned[key] = np.delete(self.acc_cleaned[key],idx,axis=1)

        print('Total data loss:',np.round(100*(1-self.time_array_cleaned.shape[0]/se

        # Delete original data after cleaning to free up memory.
        if delete_original_data:
            del self.anemo, self.acc, self.weather, self.time_array, self.t_array, i
            gc.collect()

    def get_ok_sensor_ind(self,sensor_names=[],sensor_type='anemo',key='H_mean'):
        """Method to get sensor indices of ok sensors for a given `key`.

        Parameters
        ----------
        sensor_names : list or str, optional
            List of sensor names to find in ok sensors of type `sensor_type` for `ke
        sensor_type : {'anemo', 'acc'}
            Type of the sensors.
        key : str, default 'H_mean'
            Key of measurement type from sensor.

        Returns
        -------
        s_ind : list
            Sensor indices of ok sensors for a given `key`.

        Notes
        -----
        Requires `anemo_ok_sensor_id` or `acc_ok_sensor_id`.
        """
        s_ind = []
        if len(sensor_names):
            if type(sensor_names)!=list:
                if type(sensor_names)==str:
                    sensor_names=[sensor_names]
                else:
                    raise TypeError('sensor_names must be list or str.')
            if sensor_type == 'anemo':
                for sensor in sensor_names:
                    if sensor in self.anemo_names[self.anemo_ok_sensor_id[key]]:
                        s_ind.append(np.where(self.anemo_names[self.anemo_ok_sensor_
                    else:
                        raise ValueError(sensor+' not in ok anemometers of key '+key
            elif sensor_type == 'acc':
                for sensor in sensor_names:
                    if sensor in self.acc_names[self.acc_ok_sensor_id[key]]:
                        s_ind.append(np.where(self.acc_names[self.acc_ok_sensor_id[k
                    else:
                        raise ValueError(sensor+' not in ok accelerometers of key '+
            else:
                raise ValueError('sensor_type must be "anemo" or "acc".')
```

```
        else:
            raise ValueError('No sensor_names provided')
        return(s_ind)

    def find_traffic(self,traffic_thresh=8,cleaned=True):
        """Method to find traffic dominated vibrations in accelerometers data using

        Parameters
        ----------
        traffic_thresh : float, default 8
            Threshold for Aoz_C_max_by_std to use for filtering. Recommended 6 ... 8
        cleaned : bool, default True
            Create the array `traffic_cleaned` for the cleaned data.

        Notes
        -----
        Requires `acc['Aoz_C_max_by_std']`.
        """
        if 'Aoz_C_max_by_std' in self.acc.keys():
            self.traffic = np.zeros(self.time_array.shape[0])
            idx = self.idx_data(self.acc['Aoz_C_max_by_std'],zeros=False,lp_cutoff=t
            self.traffic[idx] = 1

            if cleaned:
                self.traffic_cleaned = np.zeros(self.time_array_cleaned.shape[0])
                idx_cleaned = self.idx_data(self.acc_cleaned['Aoz_C_max_by_std'],zer
                self.traffic_cleaned[idx_cleaned] = 1
        else:
            raise KeyError('find_traffic requires Aoz_C_max_by_std.')

    def feature_time(self,cleaned=True):
        """Method to create time arrays in days, hours, minutes and seconds, startin

        Parameters
        ----------
        cleaned : bool, default True
            Create the time arrays for the cleaned data aswell.
        """
        self.days = self.time_array-self.time_array[0]
        self.hours = self.days*24
        self.minutes = self.hours*60
        self.seconds = self.minutes*60

        if cleaned:
            self.days_cleaned = self.time_array_cleaned-self.time_array_cleaned[0]
            self.hours_cleaned = self.days_cleaned*24
            self.minutes_cleaned = self.hours_cleaned*60
            self.seconds_cleaned = self.minutes_cleaned*60

    def filter_data(self,data,prior_idxs=None,nans=False,zeros=False,lp_cutoff=np.in
        """Method to filter data.

        Parameters
        ----------
        data : array_like
            Measurement data in which data is to be indexed.
        prior_idxs : array_like, optional
            Indices of prior filterings with which new indices will be combined.
        lp_cutoff : float or int, default np.inf
            Lowpass cutoff point: If a value is lower than `lp_cutoff`it is indexed
        hp_cutoff : float or int, default -np.inf
            Highpass cutoff point: If a value is higher than `lh_cutoff`it is indexe
        nans : bool, default False
            Nans are indexed.
```

```
        zeros : bool, default False
            Zeros are indexed.
        lowpass : bool, default False
            Values above `lp_cutoff` are indexed.
        highpass : bool, default False
            Values below `hp_cutoff` are indexed.
        method : {'all','any'}, default 'all'
            Method to use for filtering. Index a datapoint when the filtercondition
        mode : {'and','or'}, default 'and'
            Mode in which indices are to be combined. 'and' if all filterconditions
        prior_mode : {'and','or'}, default 'and'
            Mode in which indices are to be combined with indices of prior filtering

        Returns
        -------
        idx : array_like
            Indices of filtered data.

        See Also
        --------
        `idx_data`
        """
        if len(data):
            if mode == 'or':
                idx = np.array([])
            elif mode == 'and':
                idx = np.arange(np.shape(data)[-1])
            else:
                if type(mode)==str:
                    raise ValueError('Incompatible mode. mode needs to be "and" or "
                else:
                    raise TypeError('mode needs to be a string, either "and" or "or"

            if len(np.shape(data))==1:
                if mode == 'or':
                    if nans:
                        idx = np.union1d(idx,np.where(np.isnan(data)))
                    if zeros:
                        idx = np.union1d(idx,np.where(data[..., :] == 0))
                    if lowpass:
                        idx = np.union1d(idx,np.where(data[..., :] < lp_cutoff))
                    if highpass:
                        idx = np.union1d(idx,np.where(data[..., :] > hp_cutoff))
                elif mode == 'and':
                    if nans:
                        idx = np.intersect1d(idx,np.where(np.isnan(data)))
                    if zeros:
                        idx = np.intersect1d(idx,np.where(data[..., :] == 0))
                    if lowpass:
                        idx = np.intersect1d(idx,np.where(data[..., :] < lp_cutoff))
                    if highpass:
                        idx = np.intersect1d(idx,np.where(data[..., :] > hp_cutoff))
            else:
                if method == 'all':
                    if mode == 'or':
                        if nans:
                            idx = np.union1d(idx,np.argwhere(np.all(np.isnan(data[..
                        if zeros:
                            idx = np.union1d(idx,np.argwhere(np.all(data[..., :] ==
                        if lowpass:
                            idx = np.union1d(idx,np.argwhere(np.all(data[..., :] < l
                        if highpass:
                            idx = np.union1d(idx,np.argwhere(np.all(data[..., :] > h
                    elif mode == 'and':
```

```
                    if nans:
                        idx = np.intersect1d(idx,np.argwhere(np.all(np.isnan(dat
                    if zeros:
                        idx = np.intersect1d(idx,np.argwhere(np.all(data[..., :]
                    if lowpass:
                        idx = np.intersect1d(idx,np.argwhere(np.all(data[..., :]
                    if highpass:
                        idx = np.intersect1d(idx,np.argwhere(np.all(data[..., :]

                elif method == 'any':
                    if mode == 'or':
                        if nans:
                            idx = np.union1d(idx,np.argwhere(np.any(np.isnan(data[..
                        if zeros:
                            idx = np.union1d(idx,np.argwhere(np.any(data[..., :] ==
                        if lowpass:
                            idx = np.union1d(idx,np.argwhere(np.any(data[..., :] < l
                        if highpass:
                            idx = np.union1d(idx,np.argwhere(np.any(data[..., :] > h
                    elif mode == 'and':
                        if nans:
                            idx = np.intersect1d(idx,np.argwhere(np.any(np.isnan(dat
                        if zeros:
                            idx = np.intersect1d(idx,np.argwhere(np.any(data[..., :]
                        if lowpass:
                            idx = np.intersect1d(idx,np.argwhere(np.any(data[..., :]
                        if highpass:
                            idx = np.intersect1d(idx,np.argwhere(np.any(data[..., :]
                else:
                    if type(method)==str:
                        raise ValueError('Incompatible method. method needs to be "a
                    else:
                        raise TypeError('method needs to be a string, either "any" o

            if type(prior_idxs)!=type(None):
                if prior_mode == 'or':
                    idx = np.union1d(idx, np.array(prior_idxs).astype(int))
                elif prior_mode == 'and':
                    idx = np.intersect1d(idx,np.array(prior_idxs).astype(int))
                else:
                    if type(prior_mode)==str:
                        raise ValueError('Incompatible prior_mode. prior_mode needs
                    else:
                        raise TypeError('prior_mode needs to be a string, either "an

            idx = np.unique(idx).astype(int)
            return idx
        else:
            raise ValueError('data is empty.')

    # Post-processing
    def bridge_layout(self,b_anemo_height_above_deck=6,t_anemo_height_above_deck=10,
        """Method to create a bridge layout showing the sensor positions for the sel

        Parameters
        ----------
        b_anemo_height_above_deck : float, default 6
            Height above the bridge deck in meters, at which the bottom row anemomet
        t_anemo_height_above_deck : float, default 10
            Height above the bridge deck in meters, at which the top row anemometers
        d_acc_height_above_deck : float, default -0.3
            Default height above the bridge deck in meters, at which the acceleromet
        top_view : bool, default True
            Create a top-view layout of the bridge.
```

```
        side_view : bool, default True
            Create a side-view layout of the bridge.
        show_cables : bool, default True
            Show the main supporting cables of the brige.
        show_deck : bool, default True
            Show the deck of the brige.
        show_girder : bool, default True
            Show the bridge girder.
        show_hangers : bool, default True
            Show the hangers on which the deck is suspended.
        show_towers : bool, default True
            Show the north- and south-tower of the bridge.
        show_COMs : bool, default False
            Show the center of mass for each element of the bridge model.
        show_joints : bool, default False
            Show the joints between elements of the bridge model.
        show_sensors : bool, default True
            Show the locations of anemometers and accelerometers that were active on
        show_water : bool, default True
            Show the waterline in the side-view layout.
        invert_xaxis : bool, default True
            Invert the x-axis to start at hanger 1 (at the north tower) from the lef
        title_suffix : str, optional
            Add a suffix to the title.
        save : bool, default False:
            Save the plot in ../images/. Note: Requires the prior existence of that
        """
        # Calculate y_ and z_ positions of cables and deck.
        y_ = np.zeros(self.n_hangers+2)
        y_[1:-1] = np.linspace(self.first_hanger_y_tower_dist,self.tower_dist-self.l
        y_[-1] = self.tower_dist
        z_cable = np.zeros_like(y_)
        z_cable[0] = z_cable[-1] = self.h_towers
        z_cable[len(z_cable)//2] = self.h_towers - self.sag

        def parabola(x,a,h,k):
            y = a*np.square(x-h)+k
            return y

        h_cable = self.tower_dist/2
        k_cable = self.h_towers - self.sag

        popt_cable, pcov_cable = scipy.optimize.curve_fit(parabola,[y_[0],y_[(self.n
        z_cable = parabola(y_,popt_cable[0],h_cable,k_cable)

        z_deck = np.zeros_like(z_cable)
        z_deck[(self.n_hangers+2)//2] = self.mid_span_deck_height
        z_deck[0] = self.tower_1_deck_support_height+self.girder_height
        z_deck[-1] = self.tower_2_deck_support_height+self.girder_height

        h_deck = y_[(self.n_hangers+2)//2]
        k_deck = z_deck[(self.n_hangers+2)//2]

        popt_deck, pcov_deck = scipy.optimize.curve_fit(parabola,[y_[0],y_[(self.n_h
        z_deck = parabola(y_,popt_deck[0],popt_deck[1],popt_deck[2])

        # Get sensor positions from sensor names.
        anemo_hanger_num = np.zeros(len(self.anemo_names))
        acc_hanger_num = np.zeros(len(self.acc_names))
        x_anemometer = np.zeros(len(self.anemo_names))
        anemo_height_above_deck = np.zeros(len(self.anemo_names))
        acc_height_above_deck = np.zeros(len(self.acc_names))

        for ind,anemo_name in enumerate(self.anemo_names):
```

```
        if anemo_name.startswith('H'):
            anemo_hanger_num[ind] = int(anemo_name[1:3])
        elif anemo_name.startswith('N'):
            anemo_hanger_num[ind] = 0
            anemo_height_above_deck[ind] = self.h_towers-self.tower_2_deck_suppo
            x_anemometer[ind] = -self.cable_x_dist/2
        elif anemo_name.startswith('S'):
            anemo_hanger_num[ind] = self.n_hangers+1
            anemo_height_above_deck[ind] = self.h_towers-self.tower_1_deck_suppo
            x_anemometer[ind] = -self.cable_x_dist/2
        if anemo_name.endswith('W') or anemo_name.endswith('Wb') or anemo_name.e
            x_anemometer[ind] = -self.cable_x_dist/2
        elif anemo_name.endswith('E') or anemo_name.endswith('Eb') or anemo_name
            x_anemometer[ind] = self.cable_x_dist/2
        if anemo_name.endswith('W') or anemo_name.endswith('Wb') or anemo_name.e
            anemo_height_above_deck[ind] = b_anemo_height_above_deck
        elif anemo_name.endswith('Wt') or anemo_name.endswith('Et'):
            anemo_height_above_deck[ind] = t_anemo_height_above_deck

    for ind,acc_name in enumerate(self.acc_names):
        if acc_name.startswith('H'):
            acc_hanger_num[ind] = int(acc_name[1:3])
            acc_height_above_deck[ind] = d_acc_height_above_deck
        elif acc_name.startswith('N'):
            acc_hanger_num[ind] = 0
            acc_height_above_deck[ind] = self.h_towers-self.tower_2_deck_support
        elif acc_name.startswith('S'):
            acc_hanger_num[ind] = self.n_hangers+1
            acc_height_above_deck[ind] = self.h_towers-self.tower_1_deck_support

    y_anemometer = y_[np.subtract(self.n_hangers+1, anemo_hanger_num).astype(int
    y_accelerometer = y_[np.subtract(self.n_hangers+1, acc_hanger_num).astype(in

    z_anemometer = np.interp(y_anemometer,y_,z_deck) + anemo_height_above_deck
    z_accelerometer = np.interp(y_accelerometer,y_,z_deck) + acc_height_above_de

    # Calculate COMs, joint positions, lengths and masses.
    hanger_COM = [y_[1:-1],(z_cable[1:-1]+z_deck[1:-1])/2]
    hanger_top_joint = [y_[1:-1],z_cable[1:-1]]
    hanger_bottom_joint = [y_[1:-1],z_deck[1:-1]]
    hanger_length = z_cable[1:-1]-z_deck[1:-1]
    hanger_mass = self.hanger_section_weight*hanger_length

    cable_segment_COM = np.zeros((2,self.n_hangers+1))
    cable_segment_length = np.zeros(self.n_hangers+1)

    deck_segment_COM = np.zeros((2,self.n_hangers+1))
    deck_segment_length = np.zeros(self.n_hangers+1)

    for ind in range(self.n_hangers+1):
        cable_segment_COM[0][ind] = (y_[ind+1]+y_[ind])/2
        cable_segment_COM[1][ind] = (z_cable[ind+1]+z_cable[ind])/2
        cable_segment_length[ind] = np.linalg.norm(x=[np.array([y_[ind+1],z_cabl

        deck_segment_COM[0][ind] = (y_[ind+1]+y_[ind])/2
        deck_segment_COM[1][ind] = ((z_deck[ind+1]-(self.girder_height/2))+(z_de
        deck_segment_length[ind] = np.linalg.norm(x=[np.array([y_[ind+1],z_deck[

    cable_segment_south_joint = [y_[:-1],z_cable[:-1]]
    cable_segment_north_joint = [y_[1:],z_cable[1:]]
    cable_segment_mass = self.cable_section_weight*cable_segment_length

    deck_segment_south_joint = [y_[:-1],z_deck[:-1]]
    deck_segment_north_joint = [y_[1:],z_deck[1:]]
```

```
        deck_segment_mass = self.girder_section_weight*deck_segment_length

        tot_cable_length = np.sum(cable_segment_length)
        tot_deck_length = np.sum(deck_segment_length)
        tot_cable_mass = np.sum(cable_segment_mass)
        tot_deck_mass = np.sum(deck_segment_mass)

        # Create bridge layout.
        if top_view:
            plt.figure(figsize=(12,4))
            if show_cables:
                plt.plot(y_,np.repeat(self.cable_x_dist/2,len(y_)),'-',color='b',lab
                plt.plot(y_,np.repeat(-self.cable_x_dist/2,len(y_)),'-',color='g',la

            if show_deck:
                plt.plot(y_,np.repeat(self.deck_width/2,len(y_)),'-',color='dimgrey'
                plt.plot(y_,np.repeat(0,len(y_)),'--',color='dimgrey',label='centerl
                plt.plot(y_,np.repeat(-self.deck_width/2,len(y_)),'-',color='dimgrey
                plt.fill_between(y_,self.deck_width/2,-self.deck_width/2,color='ligh

            if show_girder:
                plt.plot(y_,np.repeat(self.girder_width/2,len(y_)),'--',color='darkg
                plt.plot(y_,np.repeat(-self.girder_width/2,len(y_)),'--',color='dark

            if show_hangers:
                plt.plot(y_,np.repeat(self.cable_x_dist/2,len(y_)),'.',color='b',lab
                plt.plot(y_,np.repeat(-self.cable_x_dist/2,len(y_)),'.',color='g',la

            if show_towers:
                plt.plot([y_[0]],self.cable_x_dist/2,'d',ms=10,color='black',label='
                plt.plot([y_[0]],-self.cable_x_dist/2,'d',ms=10,color='black')
                plt.plot([y_[-1]],self.cable_x_dist/2,'d',ms=10,color='red',label='N
                plt.plot([y_[-1]],-self.cable_x_dist/2,'d',ms=10,color='red')

            if show_COMs:
                plt.plot(hanger_COM[0],np.repeat(self.cable_x_dist/2,len(hanger_COM[
                plt.plot(hanger_COM[0],np.repeat(-self.cable_x_dist/2,len(hanger_COM
                plt.plot(cable_segment_COM[0],np.repeat(self.cable_x_dist/2,len(cabl
                plt.plot(cable_segment_COM[0],np.repeat(-self.cable_x_dist/2,len(cab
                plt.plot(deck_segment_COM[0],np.repeat(0,len(deck_segment_COM[0])),'

            if show_joints:
                plt.plot(hanger_top_joint[0],np.repeat(self.cable_x_dist/2,len(hange
                plt.plot(hanger_top_joint[0],np.repeat(-self.cable_x_dist/2,len(hang
                plt.plot(hanger_bottom_joint[0],np.repeat(self.cable_x_dist/2,len(ha
                plt.plot(hanger_bottom_joint[0],np.repeat(-self.cable_x_dist/2,len(h

                plt.plot(cable_segment_south_joint[0],np.repeat(self.cable_x_dist/2,
                plt.plot(cable_segment_south_joint[0],np.repeat(-self.cable_x_dist/2
                plt.plot(cable_segment_north_joint[0],np.repeat(self.cable_x_dist/2,
                plt.plot(cable_segment_north_joint[0],np.repeat(-self.cable_x_dist/2

                plt.vlines(deck_segment_south_joint[0],-self.deck_width/2,self.deck_
                plt.vlines(deck_segment_north_joint[0],-self.deck_width/2,self.deck_

            if show_sensors:
                plt.plot(y_accelerometer,np.repeat(self.acc_x_dist_from_centerline,l
                plt.plot(y_accelerometer,np.repeat(-self.acc_x_dist_from_centerline,
                plt.plot(y_anemometer,x_anemometer,'^',ms=10,mec='black',color='gray

            if xticks_h_num:
                xtick_labels = list(np.arange(0,self.n_hangers+2,1))[::-1]
                xtick_labels[0] = 'S_Tower'
                xtick_labels[-1] = 'N_Tower'
```

```
                plt.xticks(y_,xtick_labels)
                plt.xlabel('hanger number')
            else:
                plt.xticks(np.arange(0,self.tower_dist+10,10),np.arange(0,self.tower
                plt.xlabel('distance [m]')

            if invert_xaxis:
                plt.xlim(self.tower_dist+1,-1)
            else:
                plt.xlim(-1,self.tower_dist+1)

            plt.yticks(np.arange(-1*np.round(max((self.girder_width,self.cable_x_dis
            plt.ylim(-1.5*np.round(max((self.girder_width,self.cable_x_dist,self.dec
            plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
            plt.grid()
            plt.ylabel('distance from centerline [m]')

            plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14
            if save:
                plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
            plt.show()
            plt.close()
            gc.collect()

        if side_view:
            plt.figure(figsize=(12,4))
            if show_cables:
                if invert_xaxis:
                    plt.plot(y_,z_cable,'-',color='g',label='cable')
                else:
                    plt.plot(y_,z_cable,'-',color='b',label='cable')

            if show_deck:
                plt.plot(y_,z_deck,'-',color='dimgrey',label='deck')

            if show_girder:
                plt.plot(y_,z_deck-self.girder_height,'-',color='darkgrey',label='gi
                plt.fill_between(y_,z_deck,z_deck-self.girder_height,color='darkgrey

            if show_hangers:
                if invert_xaxis:
                    plt.vlines(hanger_bottom_joint[0],hanger_bottom_joint[1],hanger_
                else:
                    plt.vlines(hanger_bottom_joint[0],hanger_bottom_joint[1],hanger_

            if show_towers:
                plt.vlines([y_[0]],[0],[z_cable[0]],color='black',label='South Tower
                plt.vlines([y_[-1]],[0],[z_cable[-1]],color='red',label='North Tower

            if show_water:
                plt.hlines(0,0,self.tower_dist,linestyle='-.',color='blue',label='Wa

            if show_COMs:
                plt.plot(hanger_COM[0],hanger_COM[1],'o',color='m',label='hanger COM
                plt.plot(cable_segment_COM[0],cable_segment_COM[1],'o',color='y',lab
                plt.plot(deck_segment_COM[0],deck_segment_COM[1],'o',color='c',label

            if show_joints:
                plt.plot(hanger_top_joint[0],hanger_top_joint[1],'.',color='m',label
                plt.plot(hanger_bottom_joint[0],hanger_bottom_joint[1],'.',color='m'

                plt.plot(cable_segment_south_joint[0],cable_segment_south_joint[1],'
                plt.plot(cable_segment_north_joint[0],cable_segment_north_joint[1],'
```

```
                plt.plot(deck_segment_south_joint[0],deck_segment_south_joint[1],'.'
                plt.plot(deck_segment_north_joint[0],deck_segment_north_joint[1],'.'

            if show_sensors:
                plt.plot(y_accelerometer,z_accelerometer,'s',ms=10,mec='black',color
                plt.plot(y_anemometer,z_anemometer,'^',ms=10,mec='black',color='gray


            if xticks_h_num:
                xtick_labels = list(np.arange(0,self.n_hangers+2,1))[::-1]
                xtick_labels[0] = 'S_Tower'
                xtick_labels[-1] = 'N_Tower'
                plt.xticks(y_,xtick_labels)
                plt.xlabel('hanger number')
            else:
                plt.xticks(np.arange(0,self.tower_dist+10,10),np.arange(0,self.tower
                plt.xlabel('distance [m]')

            if invert_xaxis:
                plt.xlim(self.tower_dist+1,-1)
            else:
                plt.xlim(-1,self.tower_dist+1)

            plt.yticks(np.arange(0,self.h_towers,10),np.arange(0,self.h_towers,10))
            plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
            plt.grid()
            plt.ylabel('height above water [m]')
            plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14
            if save:
                plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
            plt.show()
            plt.close()
            gc.collect()

    def plot_data(self,x_array,y_arrays,labels=[],title_suffix='',xlabel='x',ylabel=
        """Method to plot Lysefjord Bridge data in a consistent manner.

        Parameters
        ----------
        x_array : array_like
            Array of x-values to be plotted.
        y_arrays : array_like or list of array_likes
            Array(s) of y-values to be plotted.
        labels : list of str, optional
            Labels for the data series.
        title_suffix : str, optional
            Add a suffix to the title.
        xlabel : str, default 'x'
            Label for the x-axis.
        ylabel : str, default 'y'
            Label for the y-axis.
        yunit : str, default '[]'
            Unit of y-axis.
        grid : bool, default True
            Show a grid on the plot.
        legend : bool, default True
            Show a legend on the plot.
        split_sensors : bool, default False
            Split the plot into the different sensors.
        xlim : tuple of (float, float), optional
            Tuple of two values specifying the x-axis limits.
        ylim : tuple of (float, float), optional
            Tuple of two values specifying the y-axis limits.
        axhlines : list of float, optional
```

```
        List of y-values for horizontal lines to be plotted on the plot.
    axhline_colors : list of str, default ['r','r','gold','g']
        List of colors for the horizontal lines.
    yticks : array_like, optional
        Array of y-axis tick values.
    ignored_sensors : list of str, optional
        List of sensors to be ignored in the plot.
    save : bool, default False:
        Save the plot in ../images/. Note: Requires the prior existence of that
    """


    # Manage labels.
    if len(y_arrays.shape)!=1:
        if len(labels)<np.shape(y_arrays)[0]:
            difference = np.shape(y_arrays)[0]-len(labels)
            for i in range(difference):
                labels=np.append(labels,'data '+str(len(labels)+1))

    if self.full_detail:
        ylabel=ylabel.replace('_mean','')

    # Plot single dataseries.
    if len(y_arrays.shape)==1:
        plt.figure(figsize=(12,5))
        plt.plot(x_array,y_arrays,label=labels[0])
        for ind,y in enumerate(axhlines):
            try:
                plt.axhline(y,linestyle='--',label=y,color=axhline_colors[ind])
            except:
                try:
                    plt.axhline(y,linestyle='--',label=y,color=axhline_colors[-1
                except:
                    plt.axhline(y,linestyle='--',label=y)
        if grid:
            plt.grid()
        if legend:
            plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
        plt.xlabel(xlabel)
        plt.ylabel(ylabel+' '+yunit)
        plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14
        plt.xlim(xlim)
        plt.ylim(ylim)
        plt.yticks(yticks)
    elif len(y_arrays)==0:
        raise ValueError('No Data.')
        return

    # Plot multiple dataseries.
    else:
        if split_sensors:
            fig = plt.figure(figsize=(12,7),constrained_layout=True)
            axes = []
            for i in range(np.shape(y_arrays)[0]):
                if labels[i] not in ignored_sensors:
                    if i == 0:
                        axes.append(plt.subplot(len(y_arrays),1,i+1))
                    else:
                        axes.append(plt.subplot(len(y_arrays),1,i+1,sharex=axes[
                    plt.plot(x_array,y_arrays[i],label=labels[i])
                    for ind,y in enumerate(axhlines):
                        try:
                            plt.axhline(y,linestyle='--',label=y,color=axhline_c
                        except:
```

```
                            try:
                                plt.axhline(y,linestyle='--',label=y,color=axhli
                            except:
                                plt.axhline(y,linestyle='--',label=y)
                    if grid:
                        plt.grid()
                    if legend:
                        plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
                    if i == 0:
                        plt.title(self.file_names[0][:-4][14:]+' - '+self.file_n
                    plt.xlim(xlim)
                    plt.ylim(ylim)
                    plt.yticks(yticks)
                fig.supxlabel(xlabel)
                fig.supylabel(ylabel+' '+yunit)
            else:
                plt.figure(figsize=(12,5))
                for i in range(np.shape(y_arrays)[0]):
                    if labels[i] not in ignored_sensors:
                        plt.plot(x_array,y_arrays[i],label=labels[i])
                for ind,y in enumerate(axhlines):
                    try:
                        plt.axhline(y,linestyle='--',label=y,color=axhline_colors[in
                    except:
                        try:
                            plt.axhline(y,linestyle='--',label=y,color=axhline_color
                        except:
                            plt.axhline(y,linestyle='--',label=y)
                if grid:
                    plt.grid()
                if legend:
                    plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
                plt.xlabel(xlabel)
                plt.ylabel(ylabel+' '+yunit)
                plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4
                plt.xlim(xlim)
                plt.ylim(ylim)
                plt.yticks(yticks)
        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()

    def hist(self,data,bins=50,xlabel='data',xunit='[]',mode='relative frequency [%]
        """Method to create histograms on bridge data in a consistent manner.

        Parameters
        ----------
        data : array_like
            Data to create the histogram on.
        bins : int, default 50
            Number of bins.
        xlabel : str, default 'data'
            Label describing the `data`.
        xunit : str, default '[]'
            Unit of `data`.
        mode : {'absolute frequency','relative frequency [%]','probability density'}
            Mode of the histogram. 'absolute frequency' has the bars height equal to
        title_suffix : str, optional
            Add a suffix to the title.
        split_sensors : bool, default False
            Split the histogram into the different sensors.
        sensors : list of str, optional
```

```
    List of sensor labels to place above seperate histograms if `split_senso
ncol : int, default 4
    Number of columns defining the grid to place the histograms on if `split
bridge_model : bool, default False
    Whether to place the histograms on a gird based on their location at the
Hanger_num : list of int, optional
    List of the hanger numbers at which each sensor is located.
West, Top : list of bool, optional
    List of booleans defining whether each sensor is located on the West sid
save : bool, default False
    Save the histogram in ../images/. Note: Requires the prior existence of

See Also
--------
`hist2d`
`windrose`
"""
# Manage labels.
if len(data.shape)!=1:
    if len(sensors)<len(data):
        difference = len(data)-len(sensors)
        for i in range(difference):
            sensors=np.append(sensors,'sensor '+str(len(sensors)+1))

if self.full_detail:
    xlabel=xlabel.replace('_mean','')

if len(np.shape(data))==1 or split_sensors==False:
    # Create one histogram for one or multiple sensors.
    plt.figure(figsize=(10,8))
    if mode == 'absolute frequency':
        plt.hist(data.flatten(),density=False,bins=bins)
    elif mode == 'relative frequency [%]':
        plt.hist(data.flatten(),density=False,weights=np.zeros_like(data.fla
    elif mode == 'probability density':
        plt.hist(data.flatten(),density=True,bins=bins)
    else:
        raise ValueError('Invalid mode. mode needs to be "absolute frequency
    plt.xlabel(xlabel+' '+xunit)
    plt.ylabel(mode)
    plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14

else:
    # Create multiple histograms for multiple sensors.
    if bridge_model:
        # Place histograms on a grid based on the sensors' locations on the

        # Assign a level based on West/East and Top/Bottom location of the s
        lvl = []
        for i in range(np.shape(data)[0]):
            if West[i] == 1:
                if Top[i] == 1:
                    lvl.append(2)
                else:
                    lvl.append(3)
            else:
                if Top[i] == 1:
                    lvl.append(0)
                else:
                    lvl.append(1)

        # Skip empty rows and cols.
        Hanger_nums = np.unique(Hanger_num)
        ncol = len(Hanger_nums)
```

```
                col = []
                unique_lvls = np.unique(lvl)
                nrow = len(unique_lvls)
                row = []
                for i in range(np.shape(data)[0]):
                    col.append(np.where(Hanger_nums==Hanger_num[i])[0][0])
                    row.append(np.where(unique_lvls==lvl[i])[0][0])

                # Plot the histograms.
                fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
                axes = []
                for i in range(np.shape(data)[0]):
                    if i == 0:
                        axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i])))
                    else:
                        axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i]),sha
                    if mode == 'absolute frequency':
                        plt.hist(data[i],density=False,bins=bins)
                    elif mode == 'relative frequency [%]':
                        plt.hist(data[i],density=False,weights=np.zeros_like(data[i]
                    elif mode == 'probability density':
                        plt.hist(data[i],density=True,bins=bins)
                    else:
                        raise ValueError('Invalid mode. mode needs to be "absolute f
                    plt.title(sensors[i])
                fig.supxlabel(xlabel+' '+xunit)
                fig.supylabel(mode)
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][
            else:
                nrow = np.ceil(len(data)/ncol).astype(int)
                fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
                axes = []
                for i in range(np.shape(data)[0]):
                    if i == 0:
                        axes.append(plt.subplot(nrow,ncol,i+1))
                    else:
                        axes.append(plt.subplot(nrow,ncol,i+1,sharex=axes[0],sharey=
                    if mode == 'absolute frequency':
                        plt.hist(data[i],density=False,bins=bins)
                    elif mode == 'relative frequency [%]':
                        plt.hist(data[i],density=False,weights=np.zeros_like(data[i]
                    elif mode == 'probability density':
                        plt.hist(data[i],density=True,bins=bins)
                    else:
                        raise ValueError('Invalid mode. mode needs to be "absolute f
                    plt.title(sensors[i])
                fig.supxlabel(xlabel+' '+xunit)
                fig.supylabel('probability density')
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][

        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()

    def scatterplot(self,data1,data2,data3,label1='data1',label2='data2',label3='dat
        """Method to create scatterplots on bridge data in a consistent manner.

        Parameters:
        -----------
        data1 : array_like
            Data for x-axis.
        data2 : array_like
```

```
    Data for y-axis.
data3 : array_like
    Data for color-axis.
label1 : str, default 'data1'
    Label for x-axis.
label2 : str, default 'data2'
    Label for y-axis.
label3 : str, default 'data3'
    Label for colorbar.
units : list of str, default ['[]','[]','[]']
    Units for x-,y- and color-axis.
title_suffix : str, optional
    Add a suffix to the title.
color : bool, default True
    The scatterplot is colored based on data3.
plot_in_order_of_color : bool, default True
    Plot the datapoints of higher color ontop of datapoints of lower color.
cmap : colormap or str, default 'viridis'
    Colormap to use for the scatterplot.
vmin : float, optional
    Lower limit for the color-axis.
vmax : float, optional
    Upper limit for the color-axis.
curve_fit : bool, default False
    Fit a curve to the scatterplot. Note: Only works for single scatterplot,
func : {'quadratic', 'linear'}
    Function for curve fitting.
upper_lim : float, default np.inf
    Upper limit for data3 values.
lower_lim : float, default -np.inf
    Lower limit for data3 values.
bounds : tuple, default ([-np.inf,-np.inf,-np.inf],[np.inf,np.inf,np.inf])
    Bounds for curve fitting parameters.
res : int, default 1000
    Number of points for the curve fitting.
pred : float, default 1.5
    Value for prediction of curve fitting.
curve_fit_return : bool, default False
    Return the parameters of the curve fit.
split_sensors : bool, default False
    Split the scatterplot into the different sensors.
sensors : list of str, optional
    List of sensor labels to place above seperate scatterplots if `split_sen
ncol : int, default 4
    Number of columns defining the grid to place the scatterplots on if `spl
bridge_model : bool, default False
    Whether to place the scatterplots on a gird based on their location at t
Hanger_num : list of int, optional
    List of the hanger numbers at which each sensor is located.
West, Top : list of bool, optional
    List of booleans defining whether each sensor is located on the West sid
save : bool, default False
    Save the scatterplot in ../images/. Note: Requires the prior existence o

Returns:
--------
popt1, popt2, pcov1, pcov2 : array_like, optional
    If `curve_fit_return` is True, returns the parameters of the curve fit.

See Also
--------
`polar_scatterplot`
"""
# Checking for miss-shaped data.
```

```
        if color:
            if not (data1.shape == data2.shape == data3.shape):
                raise IndexError('data1',data1.shape,', data2',data2.shape,' and dat

        else:
            if not (data1.shape == data2.shape):
                raise IndexError('data1',data1.shape,' and data2',data2.shape,' must

        # Manage labels.
        if len(data1.shape)!=1:
            if len(sensors)<len(data1):
                difference = len(data1)-len(sensors)
                for i in range(difference):
                    sensors=np.append(sensors,'sensor '+str(len(sensors)+1))

        if self.full_detail:
            label1=label1.replace('_mean','')
            label2=label2.replace('_mean','')
            label3=label3.replace('_mean','')

        if vmin == None:
            vmin=np.min(data3)
        if vmax == None:
            vmax=np.max(data3)

        if type(cmap)==str:
            cmap=mpl.colormaps[cmap]

        if len(np.shape(data1))==1 or split_sensors==False:
            # Create one scatterplot for one or multiple sensors.
            plt.figure(figsize=(10,8))
            if color:
                if plot_in_order_of_color:
                    ind=np.argsort(data3.flatten())
                    data1=data1.flatten()[ind]
                    data2=data2.flatten()[ind]
                    data3=data3.flatten()[ind]
                plt.scatter(data1,data2,c=data3,vmin=vmin,vmax=vmax,cmap=cmap)
                plt.colorbar(label=label3+' '+units[2])
                plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4
            else:
                plt.scatter(data1,data2)
                plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4
            plt.xlabel(label1+' '+units[0])
            plt.ylabel(label2+' '+units[1])

            if curve_fit:
                if func=='quadratic':
                    def quadratic_function(x,a,h,k):
                        y = a*(x-h)*(x-h)+k
                        return y
                    popt1,pcov1=scipy.optimize.curve_fit(quadratic_function,data1[da
                    popt2,pcov2=scipy.optimize.curve_fit(quadratic_function,data1[da
                    plt.plot(np.linspace(np.mean(data1)-pred*np.abs(np.mean(data1)-n
                    plt.plot(np.linspace(np.mean(data1)-pred*np.abs(np.mean(data1)-n
                    plt.ylim(np.mean(data2)-pred*np.abs(np.mean(data2)-np.min(data2)
                    plt.legend(bbox_to_anchor=(1, 0), loc='lower right',ncol=2)
                elif func=='linear':
                    def linear_function(x,a,b):
                        y = a*x+b
                        return y
                    popt1,pcov1=scipy.optimize.curve_fit(linear_function,data1[data3
                    popt2,pcov2=scipy.optimize.curve_fit(linear_function,data1[data3
                    plt.plot(np.linspace(np.mean(data1)-pred*np.abs(np.mean(data1)-n
```

```
                    plt.plot(np.linspace(np.mean(data1)-pred*np.abs(np.mean(data1)-n
                    plt.ylim(np.mean(data2)-pred*np.abs(np.mean(data2)-np.min(data2)
                    plt.legend(bbox_to_anchor=(1, 0), loc='lower right',ncol=2)
                else:
                    raise NotImplementedError('Only "quadratic" and "linear" functio
            else:
                # Create multiple scatterplots for multiple sensors.
                if bridge_model:
                    # Place scatterplots on a grid based on the sensors' locations on th
                    # Assign level based on West/East and Top/Bottom location.
                    lvl = []
                    for i in range(np.shape(data1)[0]):
                        if West[i] == 1:
                            if Top[i] == 1:
                                lvl.append(2)
                            else:
                                lvl.append(3)
                        else:
                            if Top[i] == 1:
                                lvl.append(0)
                            else:
                                lvl.append(1)
                    # Skip empty rows and cols.
                    Hanger_nums = np.unique(Hanger_num)
                    ncol = len(Hanger_nums)
                    col = []
                    unique_lvls = np.unique(lvl)
                    nrow = len(unique_lvls)
                    row = []
                    for i in range(np.shape(data1)[0]):
                        col.append(np.where(Hanger_nums==Hanger_num[i])[0][0])
                        row.append(np.where(unique_lvls==lvl[i])[0][0])

                    # Plot the scatterplots.
                    fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
                    axes = []
                    for i in range(np.shape(data1)[0]):
                        if i == 0:
                            axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i])))
                        else:
                            axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i]),sha
                        if color:
                            if plot_in_order_of_color:
                                ind=np.argsort(data3[i])
                                data1[i]=data1[i][ind]
                                data2[i]=data2[i][ind]
                                data3[i]=data3[i][ind]
                            sct = plt.scatter(data1[i],data2[i],c=data3[i],vmin=vmin,vma
                        else:
                            plt.scatter(data1[i],data2[i])
                        plt.title(sensors[i])
                    if color:
                        fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
                        fig.colorbar(sct, ax=axes,label=label3+' '+units[2])
                    else:
                        fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
                    fig.supxlabel(label1+' '+units[0])
                    fig.supylabel(label2+' '+units[1])
                else:
                    nrow = np.ceil(len(data1)/ncol).astype(int)
                    fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
                    axes = []
                    for i in range(np.shape(data1)[0]):
                        if i == 0:
```

```
                    axes.append(plt.subplot(nrow,ncol,i+1))
                else:
                    axes.append(plt.subplot(nrow,ncol,i+1,sharex=axes[0],sharey=
                if color:
                    if plot_in_order_of_color:
                        ind=np.argsort(data3[i])
                        data1[i]=data1[i][ind]
                        data2[i]=data2[i][ind]
                        data3[i]=data3[i][ind]
                    sct = plt.scatter(data1[i],data2[i],c=data3[i],vmin=vmin,vma
                else:
                    plt.scatter(data1[i],data2[i])
                plt.title(sensors[i])
            if color:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
                fig.colorbar(sct, ax=axes,label=label3+' '+units[2])
            else:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
            fig.supxlabel(label1+' '+units[0])
            fig.supylabel(label2+' '+units[1])

        if save:
            if color:
                plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
            else:
                plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
        plt.show()
        plt.close()
        gc.collect()
        if curve_fit:
            if curve_fit_return:
                return popt1, popt2, pcov1, pcov2

    def polar_scatterplot(self,data1,data2,data3,label1='Direction',label2='data2',l
        """Method to create polar scatterplots on bridge data in a consistent manner

        Parameters:
        -----------
        data1 : array_like
            Data for theta-axis.
        data2 : array_like
            Data for r-axis.
        data3 : array_like
            Data for color-axis.
        label1 : str, default 'data1'
            Label for theta-axis.
        label2 : str, default 'data2'
            Label for r-axis.
        label3 : str, default 'data3'
            Label for colorbar.
        units : list of str, default ['[]','[]','[]']
            Units for x-,y- and color-axis.
        title_suffix : str, optional
            Add a suffix to the title.
        bridge_offset : float, default -42
            Offset of the bridge-axis from the geographic north-south-axis in degree
        bridge_north : bool, default False
            Orient the polar scatterplot with 'bridge-north' at the top.
        color : bool, default True
            The polar scatterplot is colored based on data3.
        plot_in_order_of_color : bool, default True
            Plot the datapoints of higher color ontop of datapoints of lower color.
        cmap : colormap or str, default 'viridis'
            Colormap to use for the polar scatterplot.
```

```
        vmin : float, optional
            Lower limit for the color-axis.
        vmax : float, optional
            Upper limit for the color-axis.
        split_sensors : bool, default False
            Split the polar scatterplot into the different sensors.
        sensors : list of str, optional
            List of sensor labels to place above seperate polar scatterplots if `spl
        ncol : int, default 4
            Number of columns defining the grid to place the polar scatterplots on i
        bridge_model : bool, default False
            Whether to place the polar scatterplots on a gird based on their locatio
        Hanger_num : list of int, optional
            List of the hanger numbers at which each sensor is located.
        West, Top : list of bool, optional
            List of booleans defining whether each sensor is located on the West sid
        save : bool, default False
            Save the polar scatterplot in ../images/. Note: Requires the prior exist

        See Also
        --------
        `scatterplot`
        `windrose`
        """
        # Checking for miss-shaped data.
        if color:
            if not (data1.shape == data2.shape == data3.shape):
                raise IndexError('data1',data1.shape,', data2',data2.shape,' and dat
        else:
            if not (data1.shape == data2.shape):
                raise IndexError('data1',data1.shape,' and data2',data2.shape,' must

        # Manage labels.
        if len(data1.shape)!=1:
            if len(sensors)<len(data1):
                difference = len(data1)-len(sensors)
                for i in range(difference):
                    sensors=np.append(sensors,'sensor '+str(len(sensors)+1))

        if self.full_detail:
            label1=label1.replace('_mean','')
            label2=label2.replace('_mean','')
            label3=label3.replace('_mean','')

        if vmin == None:
            vmin=np.min(data3)
        if vmax == None:
            vmax=np.max(data3)

        if type(cmap)==str:
            cmap=mpl.colormaps[cmap]

        if bridge_north:
            bridge_offset=0

        if len(np.shape(data1))==1 or split_sensors==False:
            # Create one polar scatterplot for one or multiple sensors.
            fig = plt.figure(figsize=(10,8))
            ax = fig.add_subplot(projection='polar')
            plt.axvline(np.deg2rad(bridge_offset),color='r',label='Bridge North')
            plt.axvline(np.deg2rad(180+bridge_offset),color='black',label='Bridge So
            if color:
                if plot_in_order_of_color:
                    ind=np.argsort(data3.flatten())
```

```
                data1=data1.flatten()[ind]
                data2=data2.flatten()[ind]
                data3=data3.flatten()[ind]
            sct = ax.scatter(np.deg2rad(data1+bridge_offset),data2,c=data3,vmin=
            fig.colorbar(sct,label=label3+' '+units[2])
            fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][
        else:
            ax.scatter(np.deg2rad(data1+bridge_offset),data2)
            fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][
        ax.set_xlabel(label1+' '+units[0])
        ax.set_ylabel(label2+' '+units[1], rotation=55)
        ax.yaxis.set_label_coords(0.72, 0.8,transform=ax.transAxes)
        ax.set_theta_zero_location('N', offset=0)
        ax.set_theta_direction(-1)
        plt.legend(bbox_to_anchor=(0.9, 0.05), loc='lower right',bbox_transform=

    else:
        # Create multiple polar scatterplots for multiple sensors.
        if bridge_model:
            # Place polar scatterplots on a grid based on the sensors' locations
            # Assign level based on West/East and Top/Bottom location.
            lvl = []
            for i in range(np.shape(data1)[0]):
                if West[i] == 1:
                    if Top[i] == 1:
                        lvl.append(2)
                    else:
                        lvl.append(3)
                else:
                    if Top[i] == 1:
                        lvl.append(0)
                    else:
                        lvl.append(1)
            # Skip empty rows and cols.
            Hanger_nums = np.unique(Hanger_num)
            ncol = len(Hanger_nums)
            col = []
            unique_lvls = np.unique(lvl)
            nrow = len(unique_lvls)
            row = []
            for i in range(np.shape(data1)[0]):
                col.append(np.where(Hanger_nums==Hanger_num[i])[0][0])
                row.append(np.where(unique_lvls==lvl[i])[0][0])

            # Plot the polar scatterplots.
            fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
            axes = []
            for i in range(np.shape(data1)[0]):
                if i == 0:
                    axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i]),pro
                    plt.axvline(np.deg2rad(bridge_offset),color='r',label='Bridg
                    plt.axvline(np.deg2rad(180+bridge_offset),color='black',labe
                else:
                    axes.append(plt.subplot2grid((nrow,ncol),(row[i],col[i]),pro
                    plt.axvline(np.deg2rad(bridge_offset),color='r')
                    plt.axvline(np.deg2rad(180+bridge_offset),color='black')
                if color:
                    if plot_in_order_of_color:
                        ind=np.argsort(data3[i])
                        data1[i]=data1[i][ind]
                        data2[i]=data2[i][ind]
                        data3[i]=data3[i][ind]
                    sct = axes[i].scatter(np.deg2rad(data1[i]+bridge_offset),dat
                else:
```

```
                    axes[i].scatter(np.deg2rad(data1[i]+bridge_offset),data2[i])
                axes[i].set_theta_zero_location('N', offset=0)
                axes[i].set_theta_direction(-1)
                plt.title(sensors[i])
            if color:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
                fig.colorbar(sct, ax=axes,label=label3+' '+units[2])
            else:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
            fig.legend(bbox_to_anchor=(0.9, 0), loc='lower right',bbox_transform
            fig.supxlabel(label1+' '+units[0])
            fig.supylabel(label2+' '+units[1],rotation=67.5)
        else:
            nrow = np.ceil(len(data1)/ncol).astype(int)
            fig = plt.figure(figsize=(5*ncol,5*nrow),constrained_layout=True)
            axes = []
            for i in range(np.shape(data1)[0]):
                if i == 0:
                    axes.append(plt.subplot(nrow,ncol,i+1,projection='polar'))
                    plt.axvline(np.deg2rad(bridge_offset),color='r',label='Bridg
                    plt.axvline(np.deg2rad(180+bridge_offset),color='black',labe
                else:
                    axes.append(plt.subplot(nrow,ncol,i+1,projection='polar',sha
                    plt.axvline(np.deg2rad(bridge_offset),color='r')
                    plt.axvline(np.deg2rad(180+bridge_offset),color='black')
                if color:
                    if plot_in_order_of_color:
                        ind=np.argsort(data3[i])
                        data1[i]=data1[i][ind]
                        data2[i]=data2[i][ind]
                        data3[i]=data3[i][ind]
                    sct = axes[i].scatter(np.deg2rad(data1[i]+bridge_offset),dat
                else:
                    axes[i].scatter(np.deg2rad(data1[i]+bridge_offset),data2[i])
                axes[i].set_theta_zero_location('N', offset=0)
                axes[i].set_theta_direction(-1)
                plt.title(sensors[i])
            if color:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
                fig.colorbar(sct, ax=axes,label=label3+' '+units[2])
            else:
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[
            fig.legend(bbox_to_anchor=(0.9, 0), loc='lower right',bbox_transform
            fig.supxlabel(label1+' '+units[0])
            fig.supylabel(label2+' '+units[1],rotation=67.5)
    if save:
        if color:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
        else:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_name
    plt.show()
    plt.close()
    gc.collect()

def windrose(self,Dir,data,bridge_offset=-42,label='data',unit='[]',title_suffix
    """Method to create wind roses on bridge data in a consistent manner.

    Parameters:
    -----------
    Dir : array_like
        Directional data for theta-axis.
    data : array_like
        Data for wind rose.
    bridge_offset : float, default -42
```

```
    Offset of the bridge-axis from the geographic north-south-axis in degree
label : str, default 'data'
    Label for color-legend.
unit : str, default '[]'
    Unit of `data`.
title_suffix : str, optional
    Add a suffix to the title.
bridge_north : bool, default False
    Orient the wind rose with 'bridge-north' at the top.
bins : array_like, optional
    Bins to use for the wind rose.
nbins : int, default 6
    Number of bins to use for the wind rose if `bins` is not provided.
cmap : colormap or str, default 'viridis'
    Colormap to use for the wind rose.
split_sensors : bool, default False
    Split the wind rose into the different sensors.
sensors : list of str, optional
    List of sensor labels to place above seperate wind roses if `split_senso
ncol : int, default 4
    Number of columns defining the grid to place the wind roses on if `split
bridge_model : bool, default False
    Whether to place the wind roses on a gird based on their location at the
Hanger_num : list of int, optional
    List of the hanger numbers at which each sensor is located.
West, Top : list of bool, optional
    List of booleans defining whether each sensor is located on the West sid
save : bool, default False
    Save the polar scatterplot in ../images/. Note: Requires the prior exist
See Also
--------
`polar_scatterplot`
`hist`
`hist2d`
"""
# Checking for miss-shaped data.
if not (Dir.shape == data.shape):
    raise IndexError('Dir',Dir.shape,' and data',data.shape,' must be of the

# Manage labels.
if len(data.shape)!=1:
    if len(sensors)<len(data):
        difference = len(data)-len(sensors)
        for i in range(difference):
            sensors=np.append(sensors,'sensor '+str(len(sensors)+1))

if self.full_detail:
    label=label.replace('_mean','')

if bridge_north:
    bridge_offset=0

if len(bins)==0:
    bins=np.linspace(np.min(data),np.max(data),nbins)

if type(cmap)==str:
    cmap=mpl.colormaps[cmap]

if len(np.shape(data))==1 or split_sensors==False:
    # Create one wind rose for one or multiple sensors.
    fig = plt.figure(figsize=(10,10))
    ax = WindroseAxes.from_ax(fig=fig)

    # Plot the wind rose.
```

```
            ax.bar(Dir.flatten()+bridge_offset, data.flatten(), normed=True, opening

            # Setup additional elements.
            ax.axvline(np.deg2rad(90-bridge_offset),color='r',label='Bridge North')
            ax.axvline(np.deg2rad(270-bridge_offset),color='black',label='Bridge Sou
            ax.set_ylabel('relative frequency [%]', rotation=55)
            ax.yaxis.set_label_coords(0.72, 0.8,transform=ax.transAxes)
            ax.set_legend(title=label+' '+unit,bbox_to_anchor=(0.9, 0.1),bbox_transf
            fig.legend(bbox_to_anchor=(0.9, 0.1),bbox_transform=fig.transFigure, loc
            plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14

        else:
            # Create multiple wind roses for multiple sensors.
            if bridge_model:
                # Place the wind roses on a grid based on the sensors' locations on

                # Assign a level based on West/East and Top/Bottom location of the s
                lvl = []
                for i in range(np.shape(data)[0]):
                    if West[i] == 1:
                        if Top[i] == 1:
                            lvl.append(2)
                        else:
                            lvl.append(3)
                    else:
                        if Top[i] == 1:
                            lvl.append(0)
                        else:
                            lvl.append(1)

                # Skip empty rows and cols.
                Hanger_nums = np.unique(Hanger_num)
                ncol = len(Hanger_nums)
                col = []
                unique_lvls = np.unique(lvl)
                nrow = len(unique_lvls)
                row = []
                for i in range(np.shape(data)[0]):
                    col.append(np.where(Hanger_nums==Hanger_num[i])[0][0])
                    row.append(np.where(unique_lvls==lvl[i])[0][0])

                # Create the wind rose axes.
                fig = plt.figure(figsize=(6*ncol,6*nrow),constrained_layout=True)
                axes = []
                for i in range(np.shape(data)[0]):
                    temp_ax=plt.subplot2grid((nrow,ncol),(row[i],col[i]))
                    rect=temp_ax.get_position()
                    temp_ax.remove()
                    axes.append(WindroseAxes.from_ax(fig=fig,rect=rect))

                    # Plot the wind roses.
                    axes[i].bar(Dir[i]+bridge_offset, data[i], normed=True, opening=

                    # Setup additional elements.
                    if i == 0:
                        axes[i].axvline(np.deg2rad(90-bridge_offset),color='r',label
                        axes[i].axvline(np.deg2rad(270-bridge_offset),color='black',
                    else:
                        axes[i].axvline(np.deg2rad(90-bridge_offset),color='r')
                        axes[i].axvline(np.deg2rad(270-bridge_offset),color='black')
                    plt.title(sensors[i])
                axes[-1].set_legend(title=label+' '+unit,bbox_to_anchor=(0.9, 0.1),
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][
                fig.legend(bbox_to_anchor=(0.9, 0.1), loc='upper left',bbox_transfor
```

```
                fig.supylabel('relative frequency [%]',rotation=67.5)
            else:
                # Define the plot grid
                nrow = np.ceil(len(data)/ncol).astype(int)

                # Create the wind rose axes.
                fig = plt.figure(figsize=(6*ncol,6*nrow),constrained_layout=True)
                axes = []
                for i in range(np.shape(data)[0]):
                    temp_ax=plt.subplot(nrow,ncol,i+1)
                    rect=temp_ax.get_position()
                    temp_ax.remove()
                    axes.append(WindroseAxes.from_ax(fig=fig,rect=rect))

                    # Plot the wind roses.
                    axes[i].bar(Dir[i]+bridge_offset, data[i], normed=True, opening=

                    # Setup additional elements.
                    if i == 0:
                        axes[i].axvline(np.deg2rad(90-bridge_offset),color='r',label
                        axes[i].axvline(np.deg2rad(270-bridge_offset),color='black',
                    else:
                        axes[i].axvline(np.deg2rad(90-bridge_offset),color='r')
                        axes[i].axvline(np.deg2rad(270-bridge_offset),color='black')
                    plt.title(sensors[i])
                axes[-1].set_legend(title=label+' '+unit,bbox_to_anchor=(0.9, 0.1),
                fig.suptitle(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][
                fig.legend(bbox_to_anchor=(0.9, 0.1), loc='upper left',bbox_transfor
                fig.supylabel('relative frequency [%]',rotation=67.5)

        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()

    def hist2d(self,data1,data2,label1='data1',label2='data2',units=['[]','[]'],titl
        """Method to create 2D histograms on bridge data in a consistent manner.

        Parameters
        ----------
        data1 : array_like
            Data for x-axis of the 2d histogram.
        data2 : array_like
            Data for y-axis of the 2d histogram.
        label1 : str, default 'data1'
            Label describing `data1`.
        label2 : str, default 'data2'
            Label describing `data2`.
        units : list of str, default ['[]','[]']
            Units of `data1` and `data2`.
        title_suffix : str, optional
            Add a suffix to the title.
        bins : tuple of ints, default (20,20)
            Tuple of two values specifying the number of bins in x- and y-direction.
        save : bool, default False
            Save the 2D histogram in ../images/. Note: Requires the prior existence

        See Also
        --------
        `hist`
        `windrose`
        """
        if self.full_detail:
```

```
            label1=label1.replace('_mean','')
            label2=label2.replace('_mean','')

        plt.figure(figsize=(10,8))
        plt.hist2d(data1,data2,bins,density=True,cmin=0,cmax=1)
        plt.xlabel(label1+' '+units[0])
        plt.ylabel(label2+' '+units[1])
        plt.colorbar(label='probability density')
        plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14:]+'
        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()

    def boxplot(self,data,labels,ylabel='data',yunit='[]',title_suffix='',save=False
        """Method to create boxplots on bridge data in a consistent manner.

        Parameters
        ----------
        data : array_like
            Data for the boxplots with first axis as sensor id and second axis as me
        labels : list of str
            Sensor labels for the different boxplots.
        ylabel : str, default 'data'
            Label describing `data`.
        yunit : str, optional
            Unit of `data`.
        title_suffix : str, optional
            Add a suffix to the title.
        save : bool, default False
            Save the boxplot in ../images/. Note: Requires the prior existence of th
        """
        if self.full_detail:
            ylabel=ylabel.replace('_mean','')

        plt.figure(figsize=(10,8))
        plt.boxplot(data.T,vert=True,labels=labels)
        plt.ylabel(ylabel+' '+yunit)
        plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14:]+'
        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()

    def correlation_matrix(self,data1,data2,labels_data1=[],labels_data2=[],title1='
        """Method to create heatmaps of correlation matrices on bridge data in a con

        Parameters
        ----------
        data1 : array_like
            Input data for `data1` to be corelated with itself and `data2`.
        data2 : array_like
            Input data for `data2` to be corelated with itself and `data1`.
        labels_data1 : list of str, optional
            Labels for `data1`.
        labels_data2 : list of str, optional
            Labels for `data2`.
        title1 : str, default 'data1'
            Title describing `data1`.
        title2 : str, default 'data2'
            Title describing `data2`.
        title_suffix : str, optional
```

```
            Add a suffix to the title.
        save : bool, default False
            Save the correlation matrix heatmap in ../images/. Note: Requires the pr
        """
        Correlation_matrix = np.corrcoef(data1,data2)
        cm_labels=[title1+' '+label for label in list(labels_data1)]+[title2+' '+lab
        plt.figure(figsize=(12,10))
        try:
            sn.heatmap(Correlation_matrix,cmap='coolwarm',vmin=-1,vmax=1,xticklabels
        except:
            sn.heatmap(Correlation_matrix,cmap='coolwarm',vmin=-1,vmax=1)
        plt.title(self.file_names[0][:-4][14:]+' - '+self.file_names[-1][:-4][14:]+'
        if save:
            plt.savefig(('../images/'+self.file_names[0][:-4]+'-'+self.file_names[-1
        plt.show()
        plt.close()
        gc.collect()
```

Example of how to use the `BridgeData` class

The code below is an example to showcase how to use the `BridgeData` class. Note that the code generally has to be run from top to bottom, although some code blocks might be able to run on their own, depending on the state of other code blocks. Should you experience errors or unexpected result when experimenting with the code try running the code from the top after you made your changes.

Initialize a `BridgeData` class instance

Set the `file_path` .

In [ ]:
```
file_path = '../Data/Bridge/dataExtracted/' # Adjust filepath to where your data is
```

Set the `file_names` . Un-comment one of the pre-set list of `file_names` or create your own. Make sure the data files are located in the location defined by `file_path`

In [ ]:
```
# file_names = ['dataExtracted_2014_05_22.mat'] # From Etienne´s analysis

# file_names = ['dataExtracted_2014_08_01.mat','dataExtracted_2014_08_02.mat','dataE

# file_names = ['dataExtracted_2014_08_09.mat'] # Up to 25m/s from SSW and NNE

# file_names = ['dataExtracted_2014_08_18.mat','dataExtracted_2014_08_19.mat'] # Up

# file_names = ['dataExtracted_2014_10_07.mat'] # From Etienne´s analysis N-NE up to

# file_names = ['dataExtracted_2014_10_26.mat'] # From Etienne´s analysis S-SW up to

# file_names = ['dataExtracted_2014_10_25.mat','dataExtracted_2014_10_26.mat','dataE

# file_names = ['dataExtracted_2017_02_20.mat','dataExtracted_2017_02_21.mat','dataE

# file_names = ['dataExtracted_2017_04_21.mat','dataExtracted_2017_04_22.mat','dataE

# file_names = ['dataExtracted_2017_04_24.mat'] # Peak of 16m/s
```

```
# file_names = ['dataExtracted_2017_10_01.mat','dataExtracted_2017_10_02.mat','dataE

# file_names = ['dataExtracted_2018_01_14.mat','dataExtracted_2018_01_15.mat'] # Up

# file_names = ['dataExtracted_2018_09_18.mat','dataExtracted_2018_09_19.mat']

# file_names = ['dataExtracted_2018_09_19.mat']

# file_names = ['dataExtracted_2018_09_19.mat','dataExtracted_2018_09_20.mat','dataE

# file_names = ['dataExtracted_2018_09_18.mat','dataExtracted_2018_09_19.mat','dataE

file_names = ['dataExtracted_2018_09_18.mat','dataExtracted_2018_09_19.mat','dataExt

# file_names = ['dataExtracted_2018_10_02.mat','dataExtracted_2018_10_03.mat']

# file_names = ['dataExtracted_2018_10_11.mat','dataExtracted_2018_10_12.mat'] # Up
```

Create an instance of the `BridgeData` class.

In [ ]:
```
LFB = BridgeData(file_names=file_names,file_path=file_path)
```

Load data

Use the `load_data` method to import the MATLAB files defined by `file_names` . refer to the documentation (using `help(BridgeData.load_data)`) to learn more about the different options for the import of the data, such as the `full_detail` mode, which allows acces to the full 50Hz sample rate data.

Caution: It is recommended to only use the `full_detail` mode on **one day** of data that you have previously singled out from a larger dataset for detailed analysis. This is because of the large ammount of data that needs to be processed (4.32 million datapoints per sensor for each day of data).

In [ ]:
```
help(BridgeData.load_data)
```

```
Help on function load_data in module __main__:

load_data(self, print_data_structure=False, delete_data_after_import=True, ignore_na
ns=True, rename_H20_acc=True, replace_invalid=True, full_detail=False, H=True, W=Tru
e, Vx=True, Vy=True, Dir=True, Tv=True, Turb=True, Aox_W=True, Aoy_W=True, Aoz_W=Tru
e, Aox_E=True, Aoy_E=True, Aoz_E=True, T=True, P=True, Hum=True, AOA=True, AOI=True,
Upwind=True, theta=True, centr=True, Hanger_num=True, y_pos=True, accs_in_SI_units=T
rue, WestEast=True, TopBottom=True)
    Wrapper-method to import and combine multiple days of data using `convert_MATLAB
`. Parameters are passed on to this helper-method.

    Convert Lysefjord Bridge dataExtracted.mat files imported via `scipy.io.loadmat`
to a more usable structure of numpy arrays, partially grouped in dictionaries, for f
urther processing. 10-min datapackages are condensed into one sample described by th
e mean, std, min and max of that package, unless `full_detail` is used.

    Parameters
    ----------
    print_data_structure : bool, default False
        Print the structure of the original data. Primarily used for developement an
d bug-fixing
```

```
    delete_data_after_import : bool, default True
        Delete original data from memory after import to free up memory
    ignore_nans : bool, default True
        Ignores nans when calculating mean, std, min and max, using the remaining no
n-nan values in the 10-min data package for the calculations, instead of setting the
value representing the whole 10-min data package to nan. This results in the return
of more non-nan values, decreasing data loss during cleaning, but increasing import
time.
    rename_H20_acc : bool, default True
        Rename accelerometers H20 to H24, as there seems to be a systematic error in
the MATLAB files.
    replace_invalid : bool, default True
        Replace invalid sensor readings (outside of technical sensor range) with nan
during import. Allows to retain more data during data cleaning with sleightly higher
import time.
    full_detail : bool, default False
        Keep full detail of data, sampled at 50Hz instead of resampling to every 10m
in. It is recommended to not use this for more than 1 day of data at a time. Note: T
he values are represented as _mean arrays, although no mean is calculated. Not all f
unctions of this class are designed to work with this form of data, but it allows mo
re detailed insights in special cases
    H ... TopBottom : bool, default True
        Select which part of the data should be imported or generated during import.
Only select the measurement types you are interested in to keep import times lower.

    See Also
    --------
    `convert_MATLAB`
    `define_units`
```

Note that it is alternatively possible to load the data from a previously saved analysis using `dill.load` .

In [ ]:

```
LFB.load_data(print_data_structure=False,delete_data_after_import=True,replace_inval
# LFB = dill.load(open(file_path+file_names[0][:-4]+'-'+file_names[-1][:-4],'rb')) #
```

```
Day 1 : 2018_09_18
Day 2 : 2018_09_19
Day 3 : 2018_09_20
Day 4 : 2018_09_21
Day 5 : 2018_09_22
Day 6 : 2018_09_23
Day 7 : 2018_09_24
Day 8 : 2018_09_25
Day 9 : 2018_09_26
Day 10 : 2018_09_27
Day 11 : 2018_09_28
Day 12 : 2018_09_29
Day 13 : 2018_09_30
Day 14 : 2018_10_01
Day 15 : 2018_10_02
Day 16 : 2018_10_03
Day 17 : 2018_10_04
Day 18 : 2018_10_05
Day 19 : 2018_10_06
Day 20 : 2018_10_07
Day 21 : 2018_10_08
Day 22 : 2018_10_10
Day 23 : 2018_10_11
Day 24 : 2018_10_12
Day 25 : 2018_10_13
```

```
Day 26 : 2018_10_14
Day 27 : 2018_10_15
Day 28 : 2018_10_16
Day 29 : 2018_10_17
Day 30 : 2018_10_18
```

Bridge layout

The names of active anemometers can be obtained from `anemo_names`.

In [ ]:
```
LFB.anemo_names
```

Out[ ]:
```
array(['H08Wb', 'H08Wt', 'H08E', 'H10W', 'H10E', 'H18W', 'H18E', 'H20W',
       'H24W'], dtype='<U5')
```

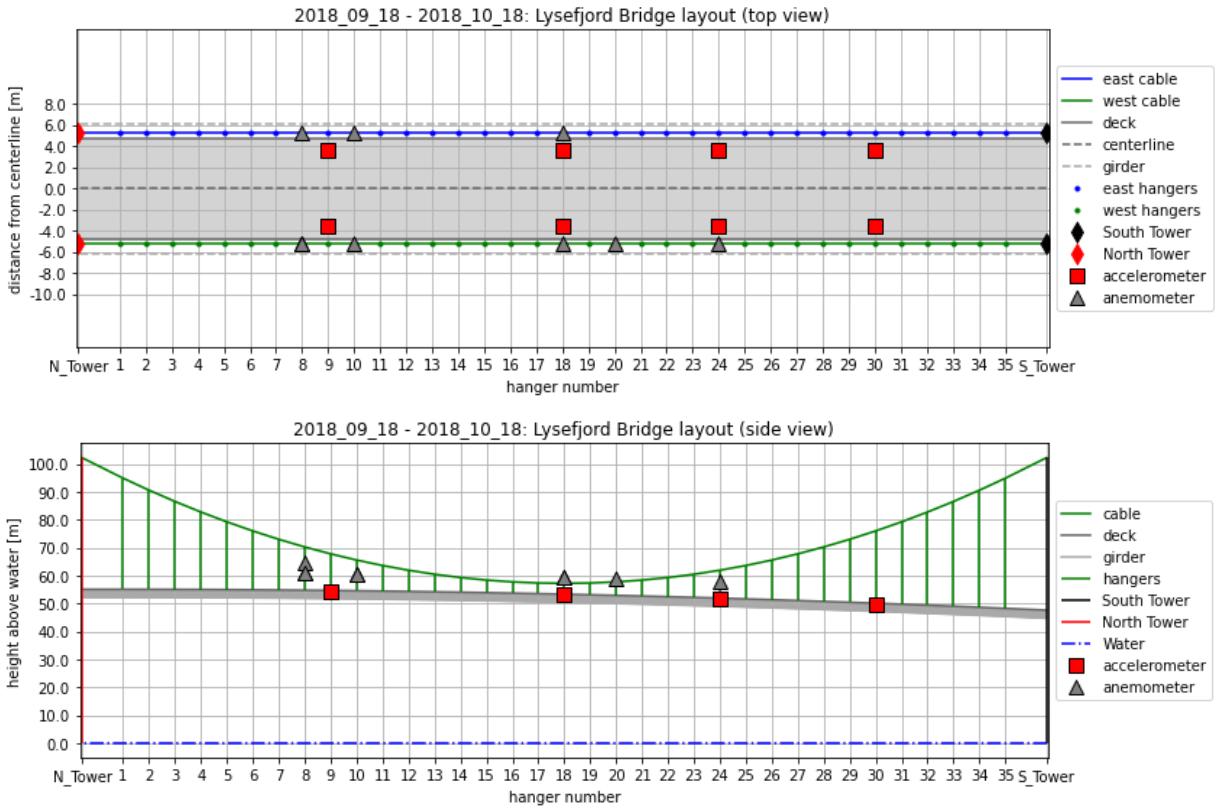
The names of active accelerometers can be obtained from `acc_names`.

In [ ]:
```
LFB.acc_names
```

Out[ ]:
```
array(['H09', 'H18', 'H24', 'H30'], dtype='<U3')
```

The `bridge_layout` method creates a layout of the active instrumentation on the bridge during the selected period.

In [ ]:
```
LFB.bridge_layout()
```



Traffic classification

The `find_traffic` method classifies each datapoint to be traffic- (1) or wind-dominated (0) using the *max_by_std* criteria on vertical accelerations. This can be used to filter the data. Note that this method is not available in `full_detail` mode. Also note that the keyword `cleaned` is set to *False*, as we are dealing with un-processed (un-cleaned) data.

In [ ]:
```
if LFB.full_detail==False:
    LFB.find_traffic(traffic_thresh=8,cleaned=False)
    print('Traffic percentage:',np.round(100*np.count_nonzero(LFB.traffic)/len(LFB.t
```

```
Traffic percentage: 68.24 %
```

Filter

The `filter_data` method allows filtering the data in various ways. Note that multiple filters can be combined in different ways aswell. It is good practise to add a remark to the title of plots that use filtered data using the `title_suffix` keyword. Uncomment one of the filter lines below to see it in effect, or use it as a template to create your own filter.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array)); title_suffix = ''
if LFB.full_detail:
    title_suffix+= ' - full detail'
# filter_idxs = LFB.filter_data(LFB.anemo['H_mean'],prior_idxs=filter_idxs,hp_cutoff
# filter_idxs = LFB.filter_data(LFB.anemo['AOA_mean'],prior_idxs=filter_idxs,hp_cuto
# filter_idxs = LFB.filter_data(LFB.traffic,prior_idxs=filter_idxs,zeros=True,mode='
```

Plot *un-processed* data

Below we will examin how to plot some of the un-processed (un-cleaned) time series data.

Setting your `keys_of_interest` beforehand allows batch-post-processing in the visualisation of data using for-loops over all keys and if-conditions on the name of the keys.

In [ ]:
```
keys_of_interest = ['H_mean','H_max','H_turb','W_mean','W_turb','AOA_mean','AOA_std'
# keys_of_interest = ['H_mean','Aoz_C_mean','Aoz_C_max','Aoz_C_max_by_std','Aoz_C_st
# keys_of_interest =['H_mean','Aoz_C_mean','Aoz_C_max_by_std']
```

Time settings

The method `feature_time` is used to to initialize time series in days, hours, minutes and seconds for your data, starting at the first datapoint. Note that the keyword `cleaned` is set to *False*, as we are dealing with un-processed (un-cleaned) data.

You can set your prefered `timescale` and `time_of_interest` prior to batch-post-processing.

In [ ]:
```
LFB.feature_time(cleaned=False) #Initiate time series for un-cleaned data.

timescale='days' #Change this variable to 'days', 'hours', 'minutes' or 'seconds' to

if timescale=='days':
    plot_time=LFB.days
    plot_time_label = 'time [d]'
elif timescale=='hours':
    plot_time=LFB.hours
    plot_time_label = 'time [h]'
elif timescale=='minutes':
    plot_time=LFB.minutes
    plot_time_label = 'time [min]'
elif timescale=='seconds':
    plot_time=LFB.seconds
```

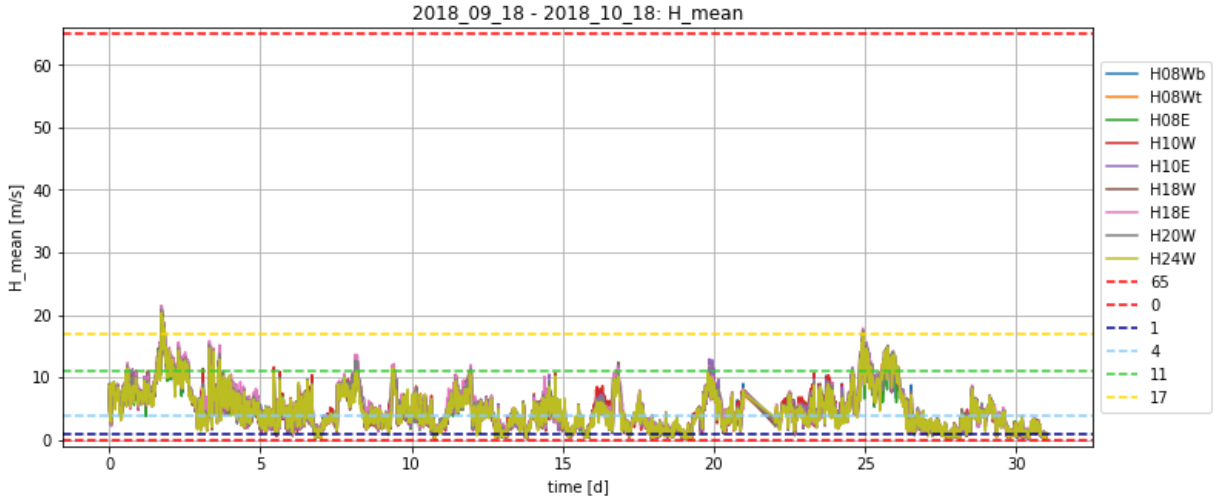
```
    plot_time_label = 'time [s]'

# time_of_interest=(0,24) #use this line instead of the one below to zoom in on deta
time_of_interest=(None,None) #default
```

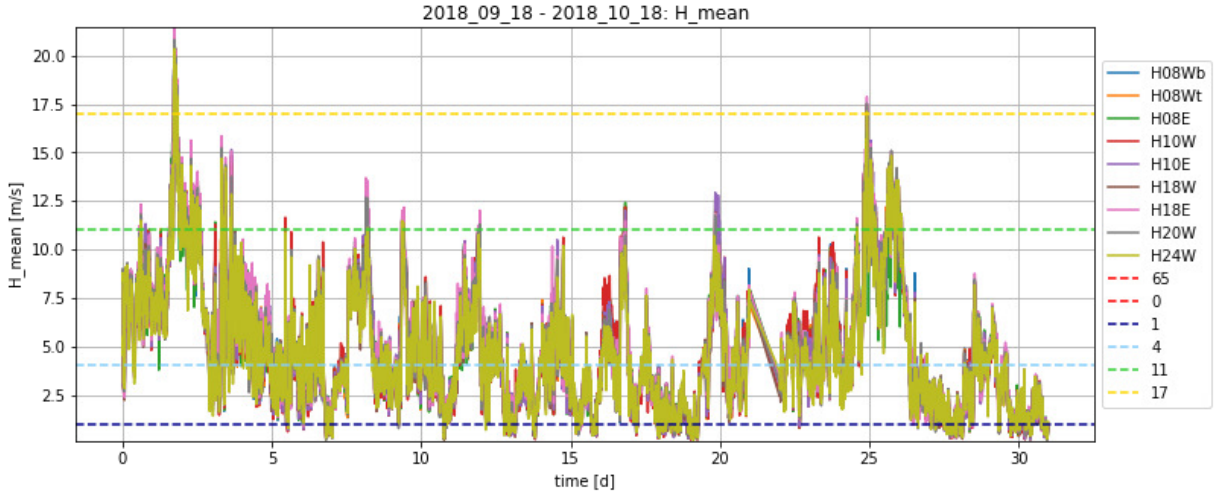
Anemometers

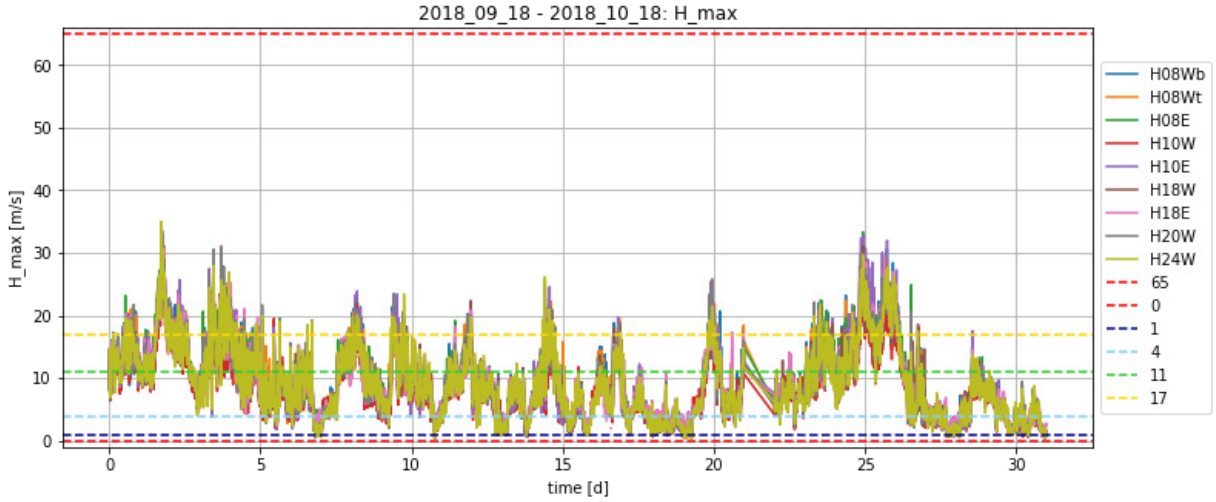
Below is an example for batch post-processing of **anemometer** data using a *for-loop* and *if-statements* on the *keys* previously defined by `keys_of_interest`. This allows to set specific *plot parameters* in the `plot_data` method for different *keys*. The names of anemometers that are not of interest can be given to the `ignored_anemos` keyword as a list.

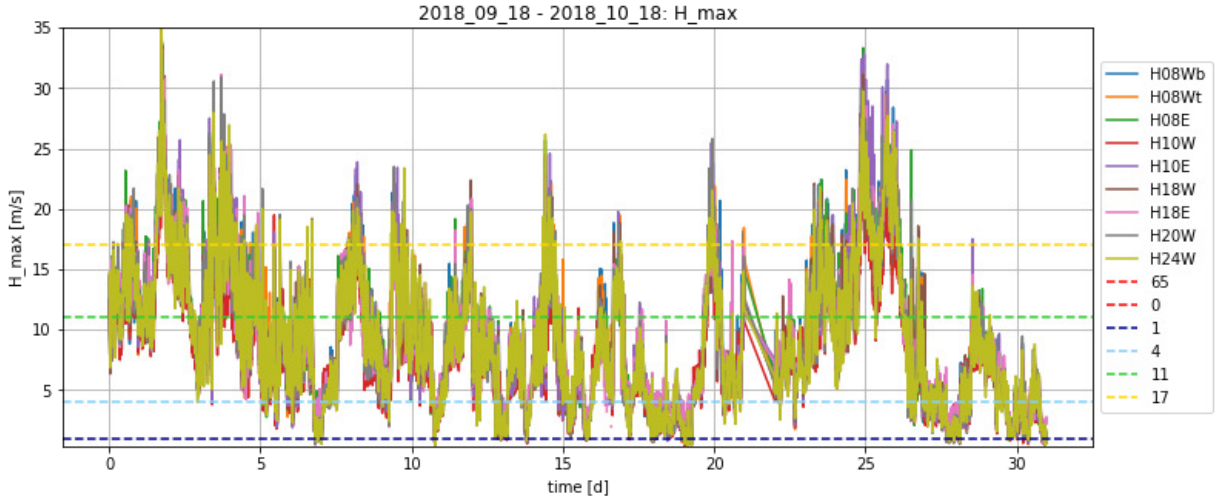
In [ ]:

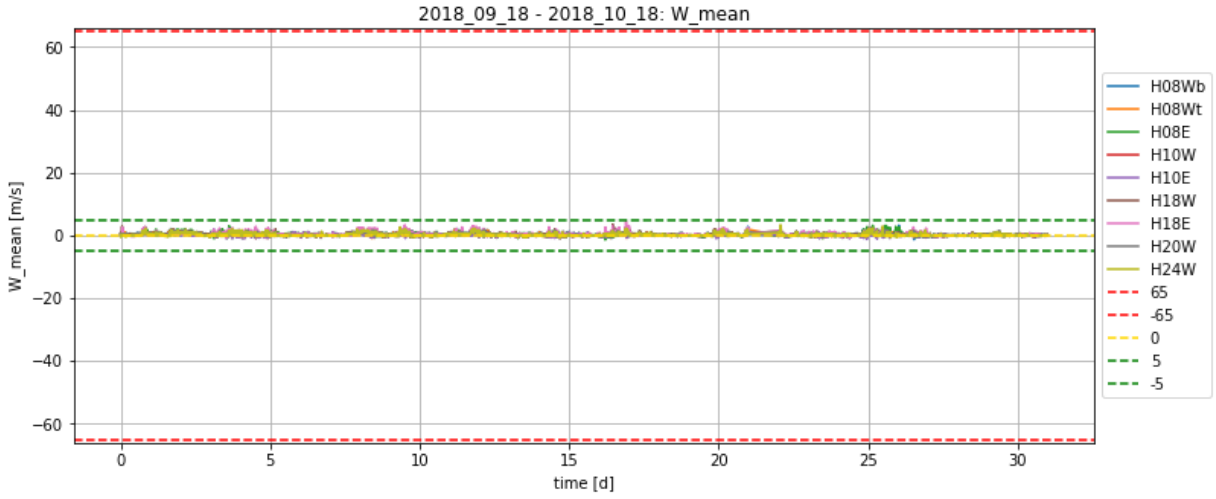
```
ignored_anemos = []
for key in LFB.anemo.keys():
    if key in keys_of_interest:
        if key.startswith('H_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        elif key.startswith('Dir_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        elif key.startswith('W_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        elif key.startswith('Vx_m') or key.startswith('Vy_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        elif key.startswith('AOA_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        elif key.startswith('AOI_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
        else:
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo[key].T[filter_idxs].T,yla
```

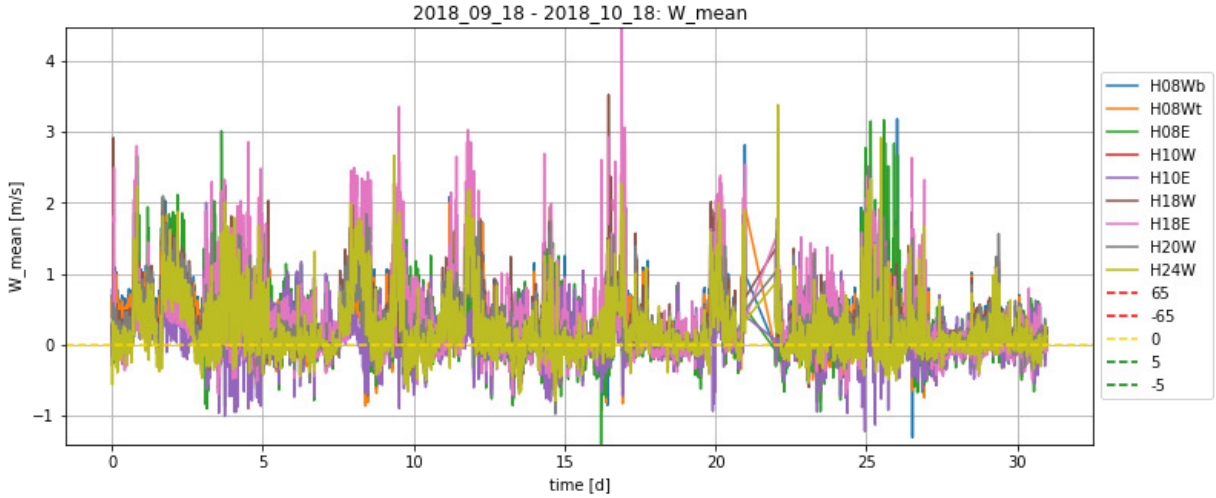


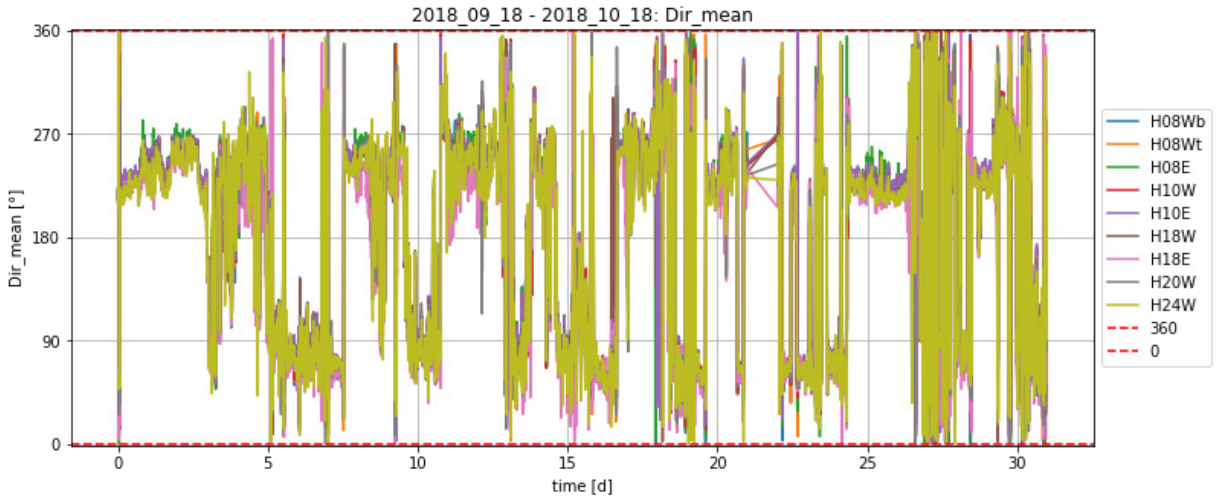

2018_09_18 - 2018_10_18: H_mean
H_mean [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
0
1
4
11
17

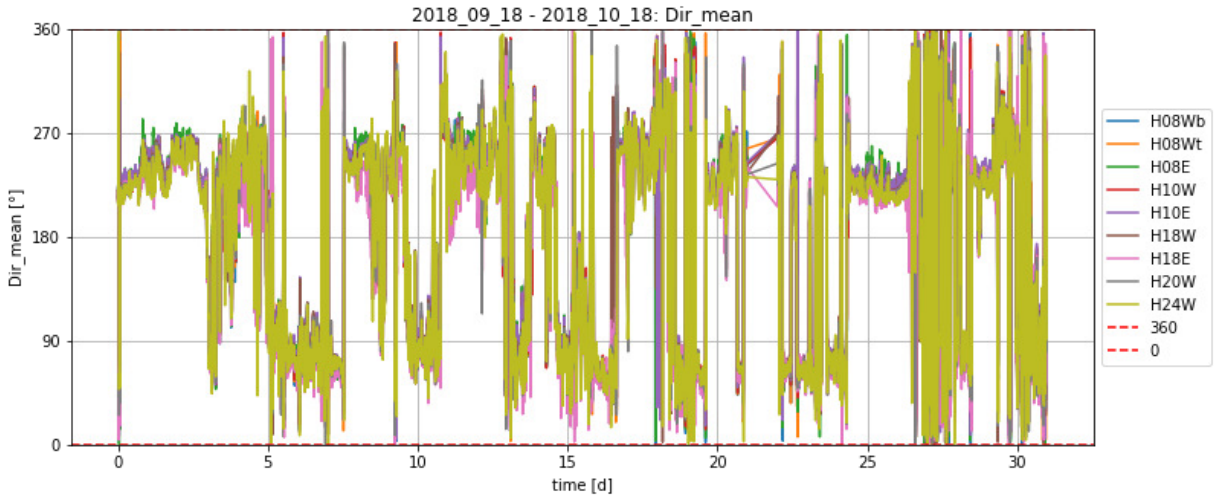

2018_09_18 - 2018_10_18: H_max
H_max [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
0
1
4
11
17

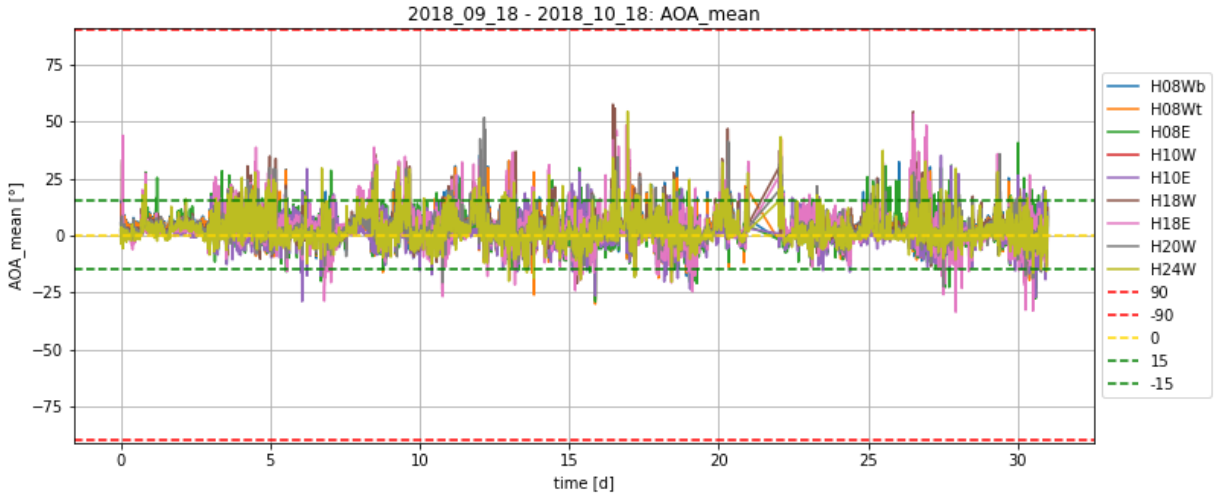

2018_09_18 - 2018_10_18: H_max
H_max [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
0
1
4
11
17

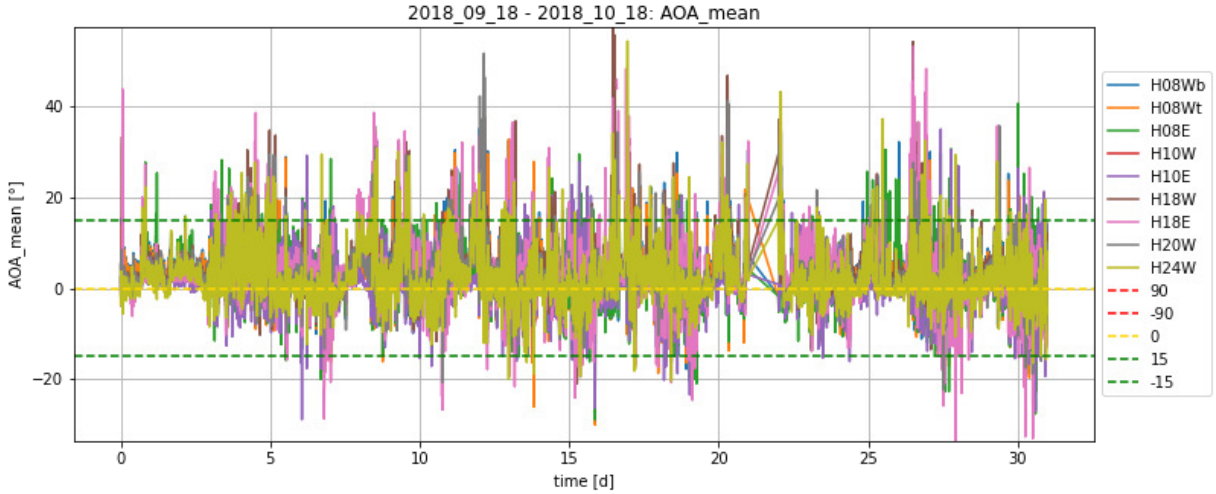

2018_09_18 - 2018_10_18: W_mean
W_mean [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
-65
0
5
-5

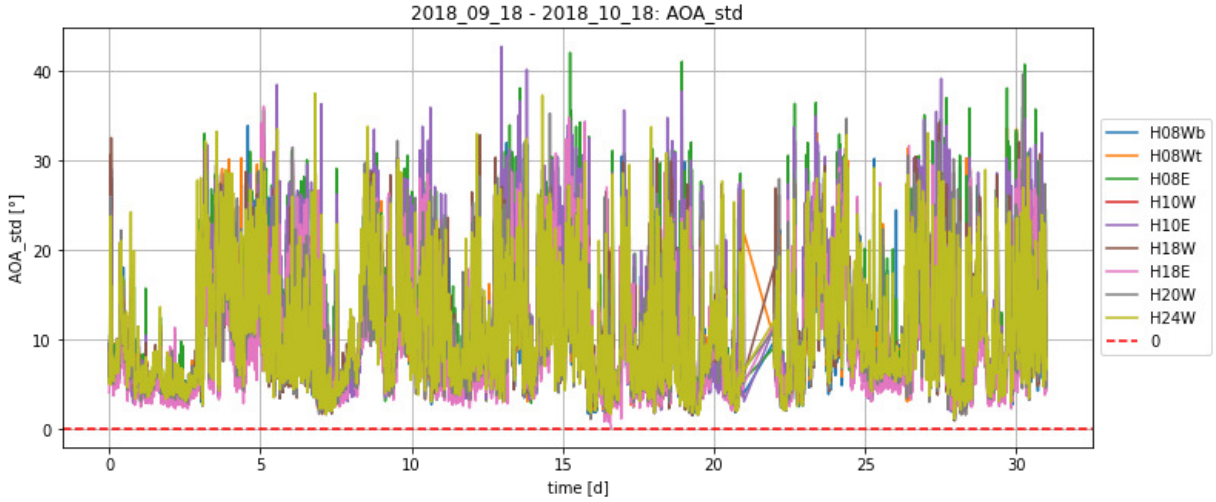

2018_09_18 - 2018_10_18: W_mean
W_mean [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
-65
0
5
-5

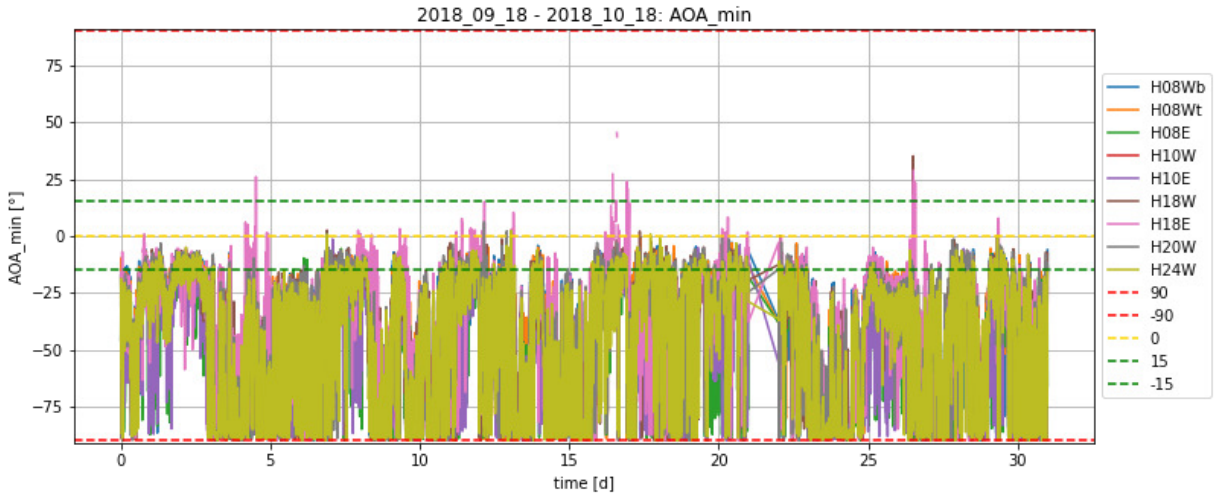

2018_09_18 - 2018_10_18: Dir_mean
Dir_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
360
0

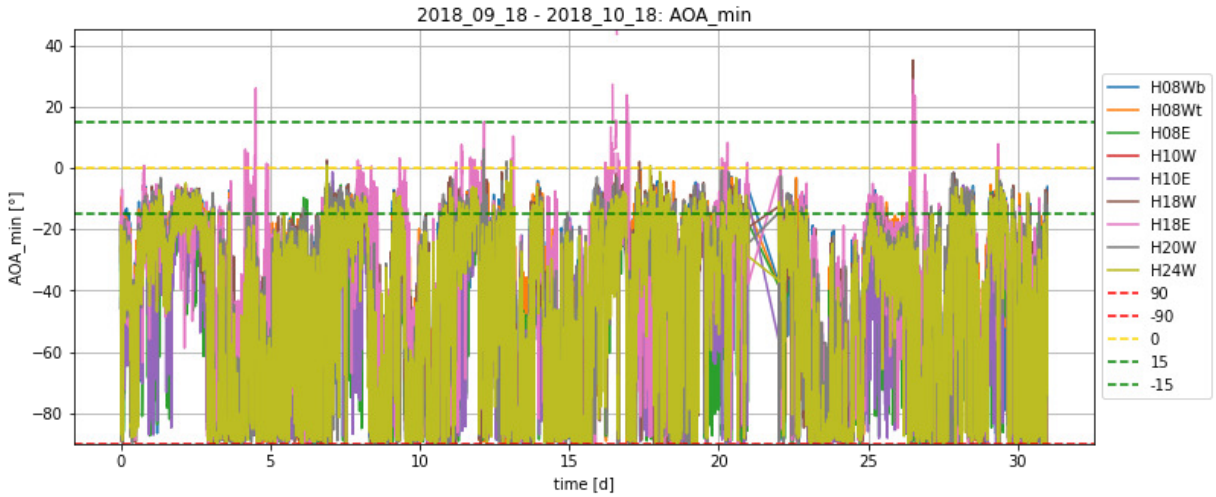

2018_09_18 - 2018_10_18: Dir_mean
Dir_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
360
0

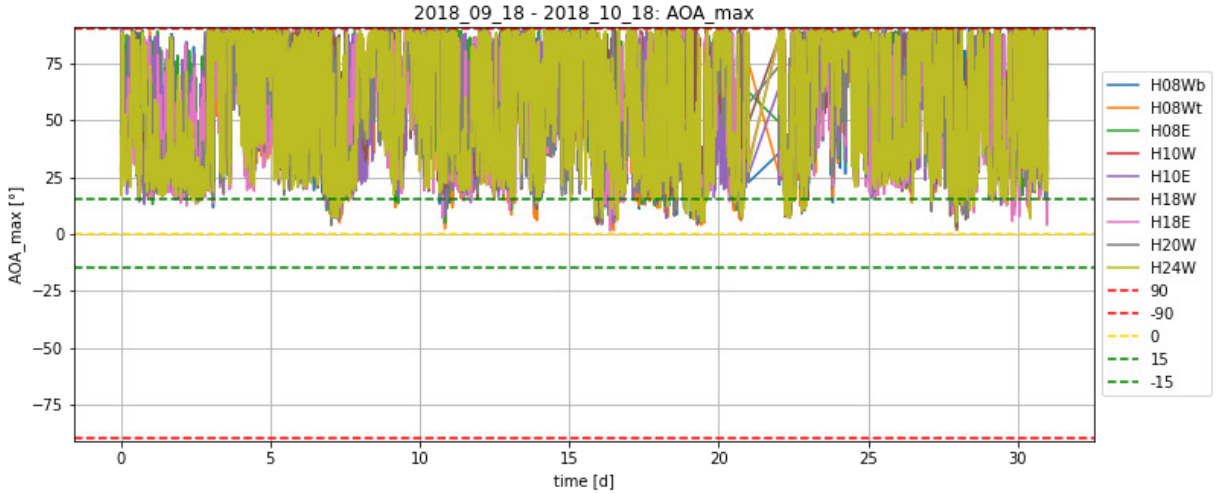

2018_09_18 - 2018_10_18: AOA_mean
AOA_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

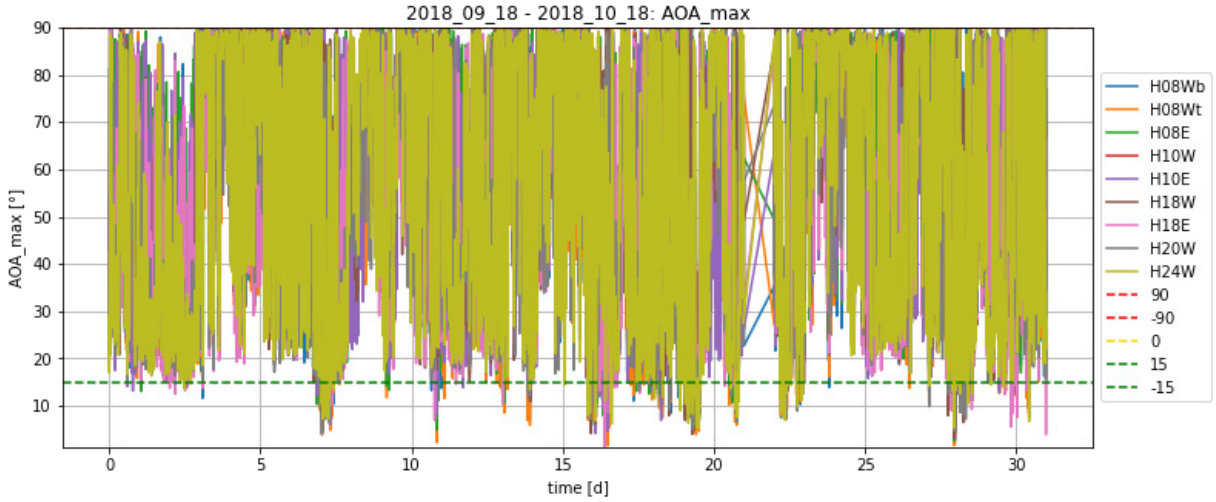

2018_09_18 - 2018_10_18: AOA_mean
AOA_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

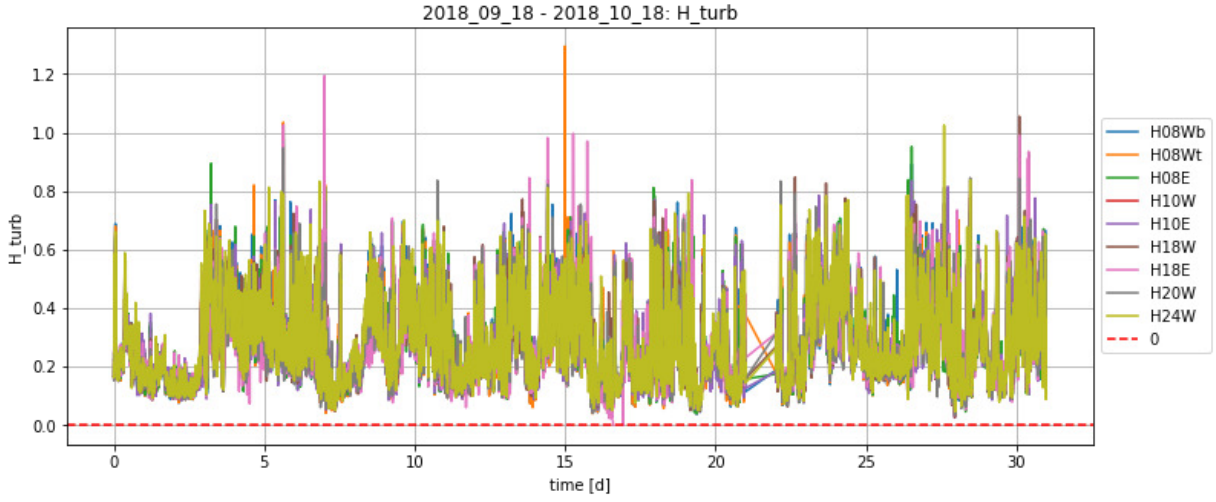

2018_09_18 - 2018_10_18: AOA_std
AOA_std [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
0

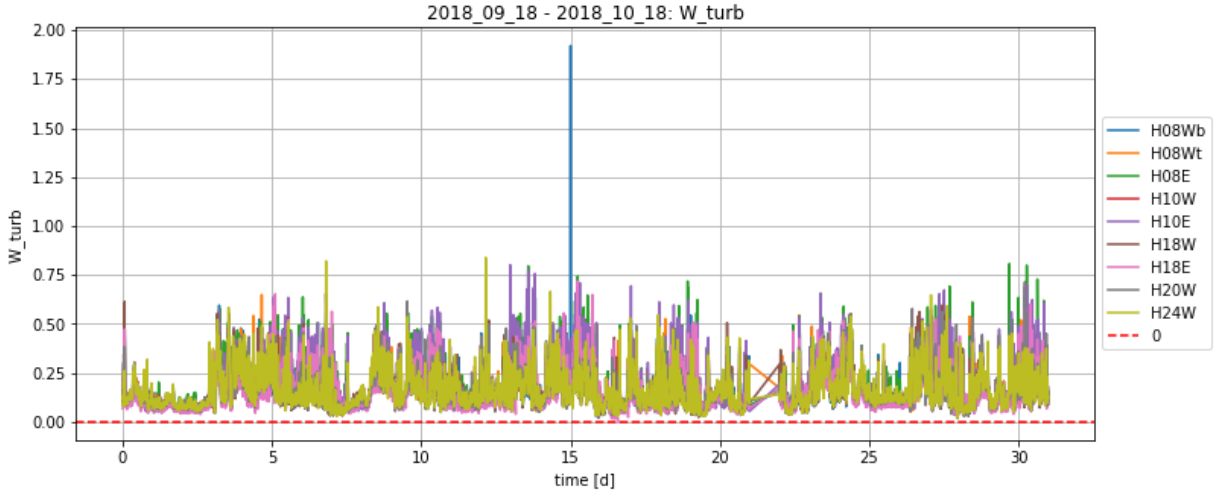

2018_09_18 - 2018_10_18: AOA_min
AOA_min [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

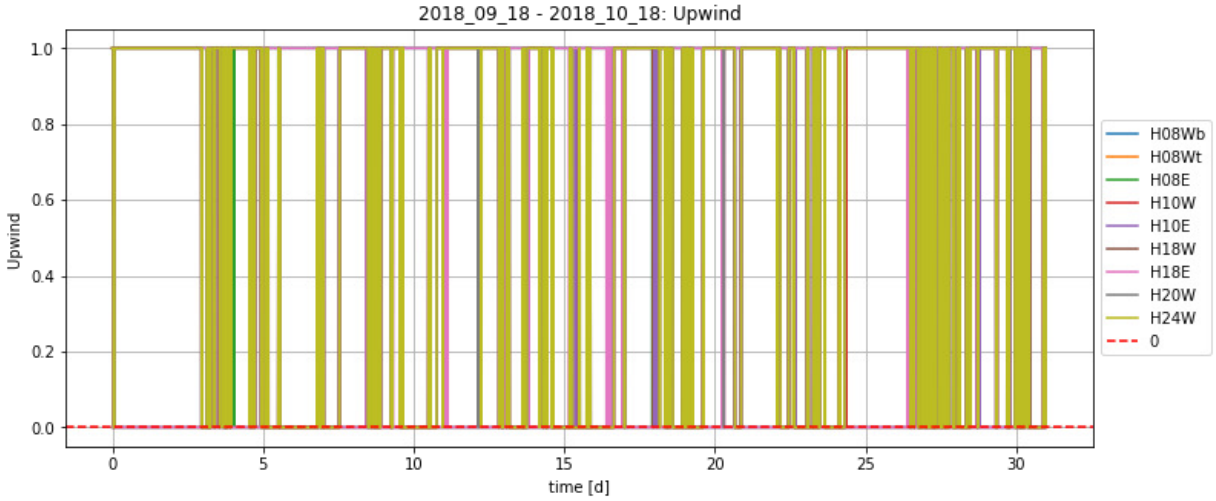

2018_09_18 - 2018_10_18: AOA_min
AOA_min [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: AOA_max
AOA_max [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: AOA_max
AOA_max [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: H_turb
H_turb
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
0


2018_09_18 - 2018_10_18: W_turb
W_turb
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
0

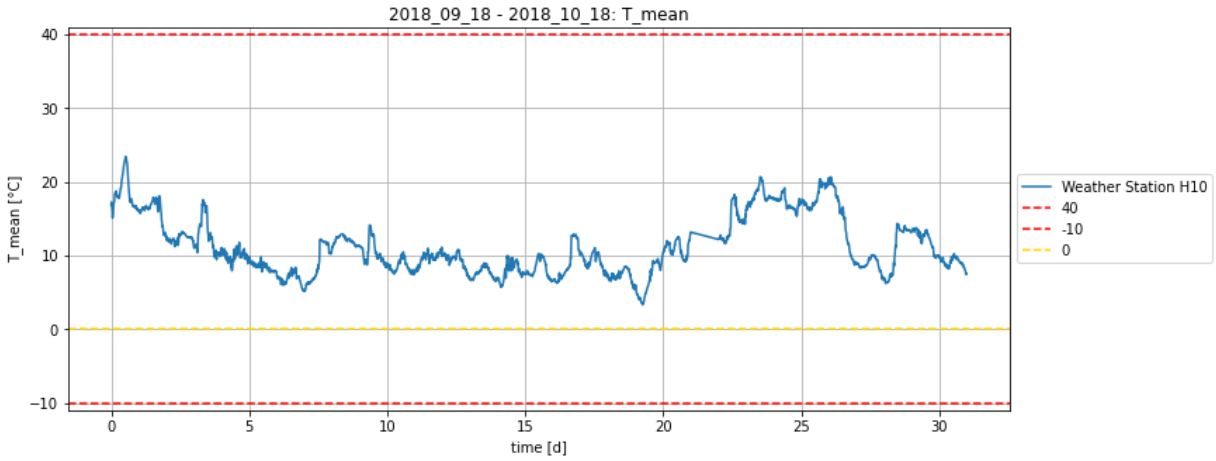


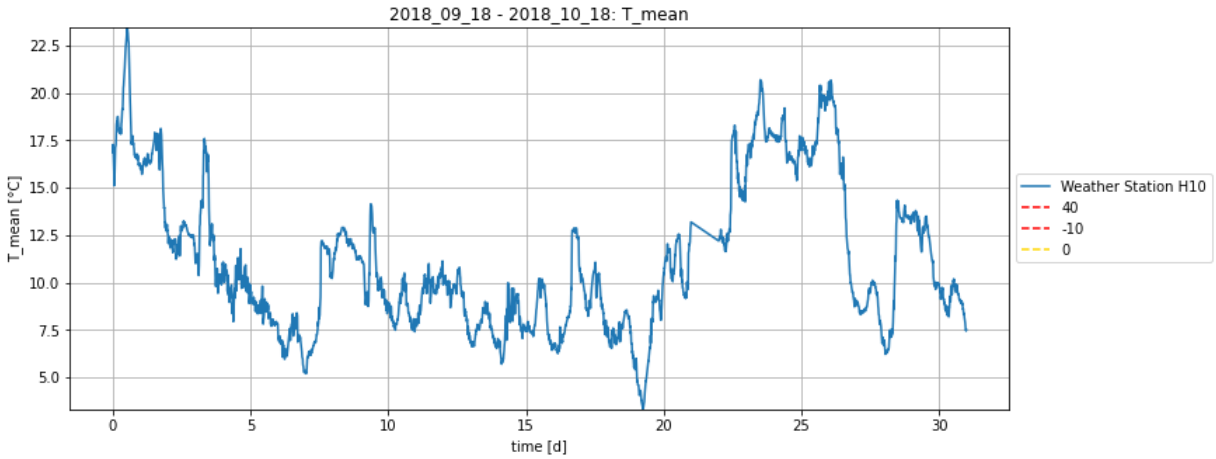
Weather

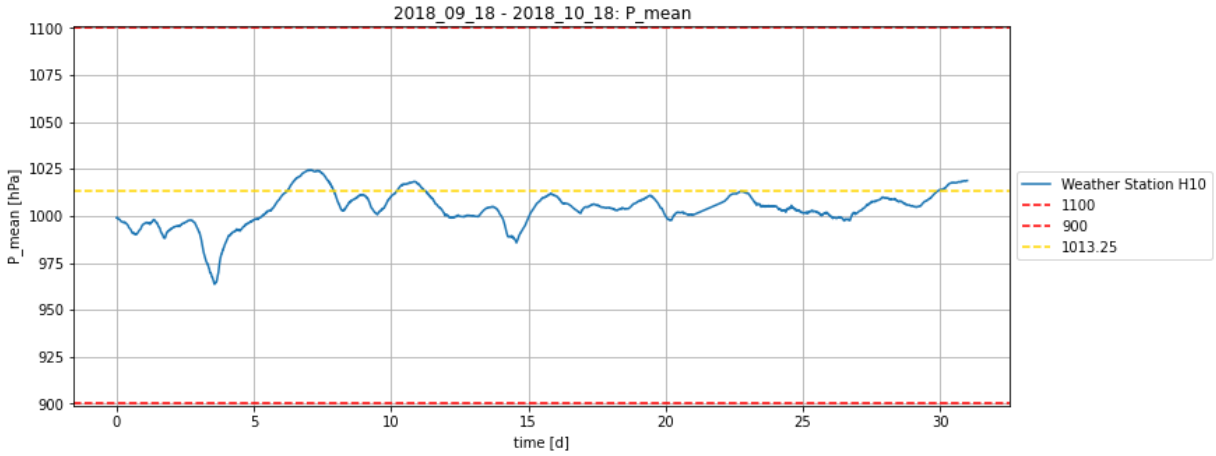
Similarly the `weather` data from the *weather station* on Hanger 10 can be plotted.

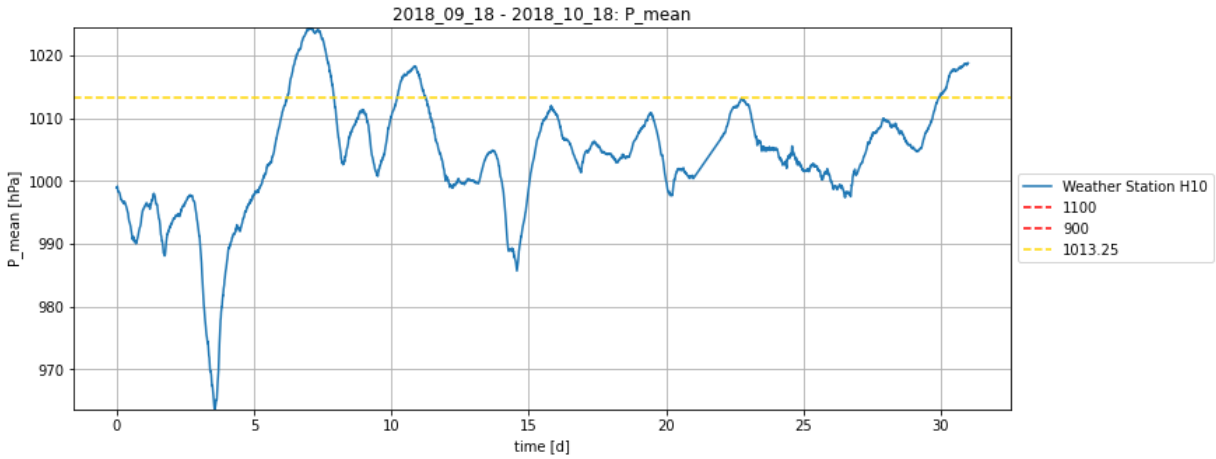
In [ ]:

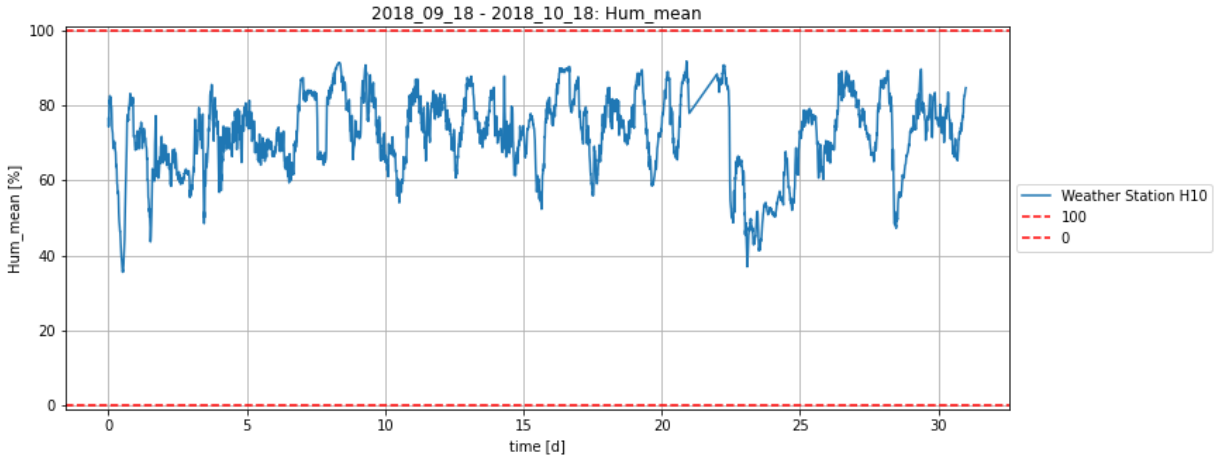
```
for key in LFB.weather.keys():
    if key in keys_of_interest:
        if key.startswith('Hum_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
        elif key.startswith('T_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
        elif key.startswith('P_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
            LFB.plot_data(plot_time[filter_idxs],LFB.weather[key][filter_idxs],ylabe
```

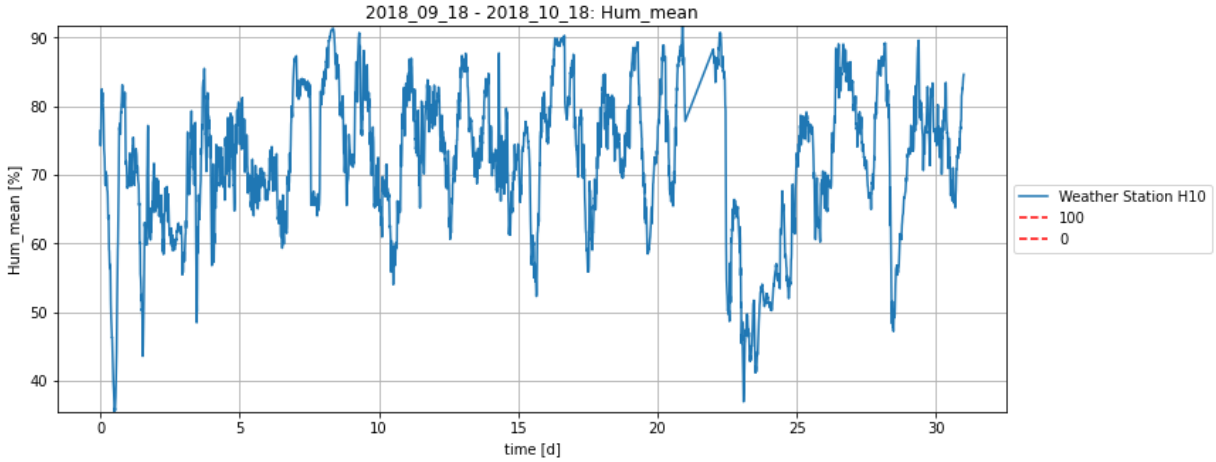












Accelerometers

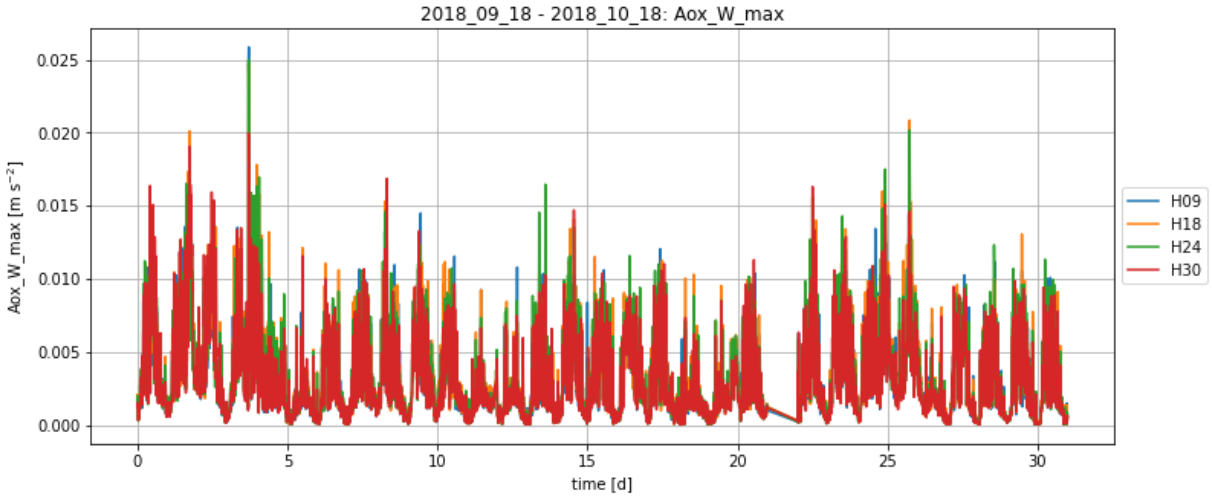
Similarly the data of **accelerometers** can be plotted.

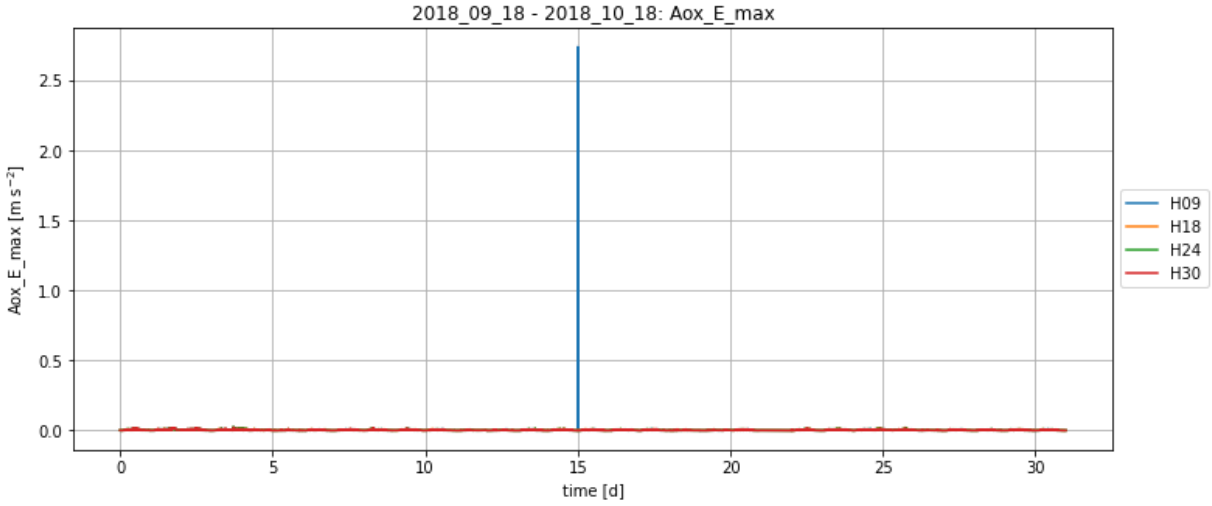
Note how the status of the `full_detail` state of the `BridgeData` class is used to perform additional analysis when using the full 50Hz sample rate data. The `full_detail` mode has to be enabled during the import of the data in the `load_data` method.

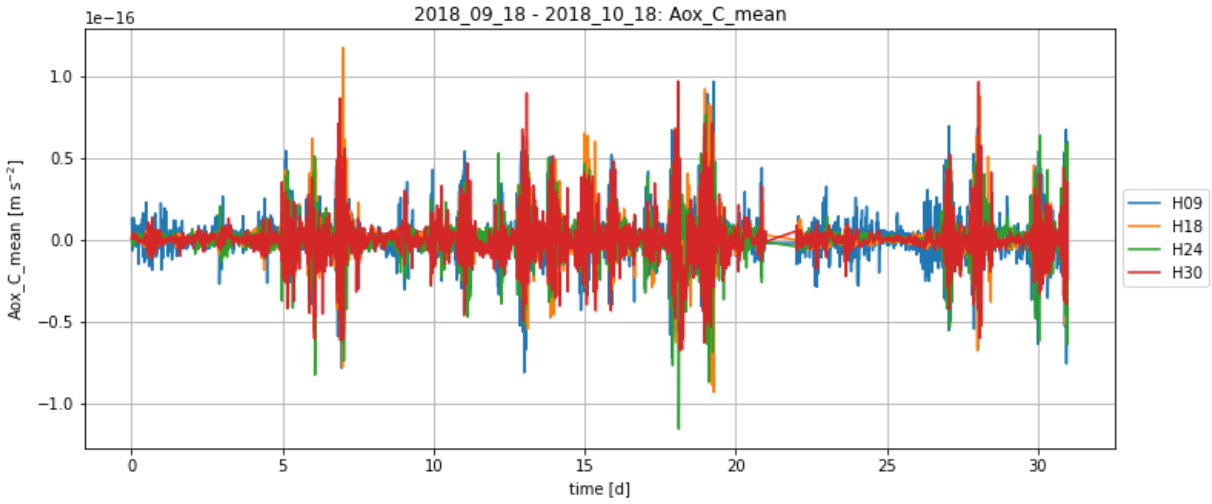
Caution: It is recommended to only use the `full_detail` mode on **one day** of data that you have previously singled out from a larger dataset for detailed analysis. This is because of the large ammount of data that needs to be processed (4.32 million datapoints per sensor for each day of data)..

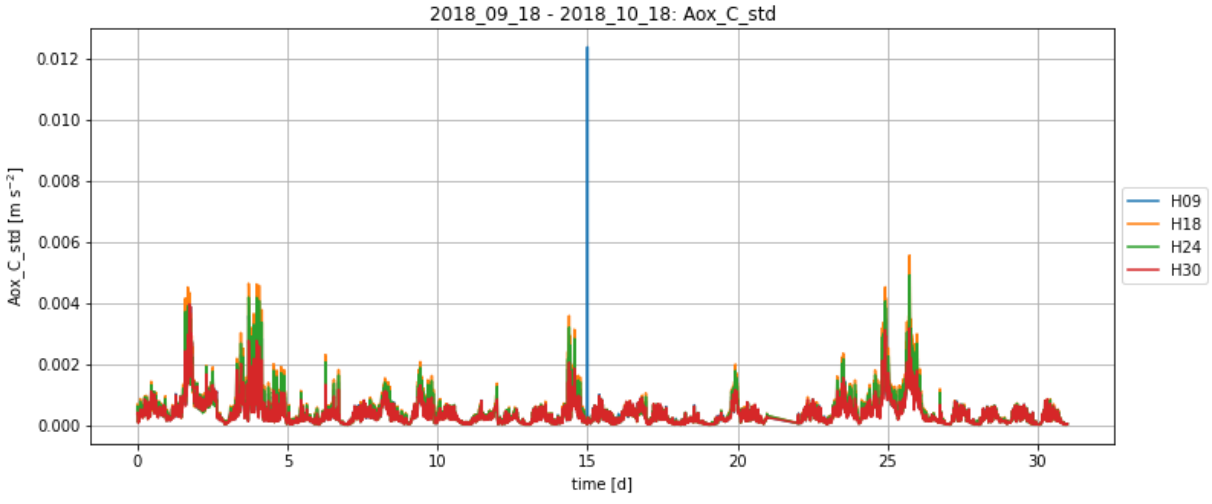
In [ ]:

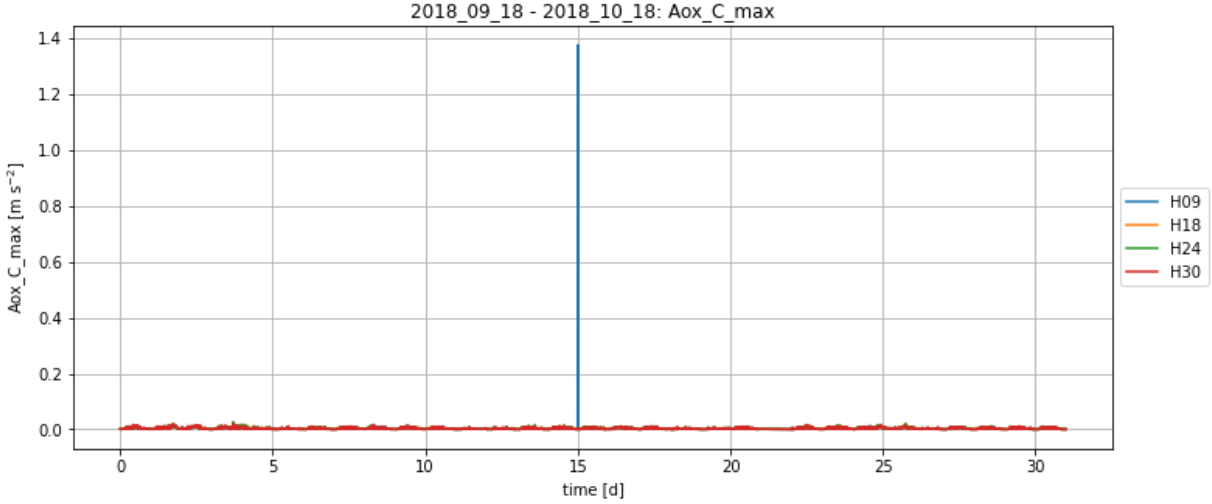
```
ignored_acc = []
for key in LFB.acc.keys():
    if key in keys_of_interest:
        if key == 'Aoz_C_mean':
            if LFB.full_detail:
                LFB.plot_data(plot_time[filter_idxs],LFB.acc[key].T[filter_idxs].T,y
                for sensor_id,sensor in enumerate(LFB.acc_names):
                    if sensor not in ignored_acc:
                        if type(time_of_interest[0]) == type(None) and type(time_of_
                            Aoz_C_max = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                            Aoz_C_std = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                        elif type(time_of_interest[0]) != type(None) and type(time_o
                            Aoz_C_max = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                            Aoz_C_std = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                        elif type(time_of_interest[0]) == type(None) and type(time_o
                            Aoz_C_max = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                            Aoz_C_std = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                        elif type(time_of_interest[0]) != type(None) and type(time_o
                            Aoz_C_max = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                            Aoz_C_std = LFB.acc['Aoz_C_mean'][sensor_id][filter_idxs
                        print(LFB.acc_names[sensor_id])
                        print('Aoz_C_max =',np.round(Aoz_C_max,2),'\nAoz_C_std =',np
            else:
                LFB.plot_data(plot_time[filter_idxs],LFB.acc[key].T[filter_idxs].T,y
        else:
            LFB.plot_data(plot_time[filter_idxs],LFB.acc[key].T[filter_idxs].T,ylabe
```

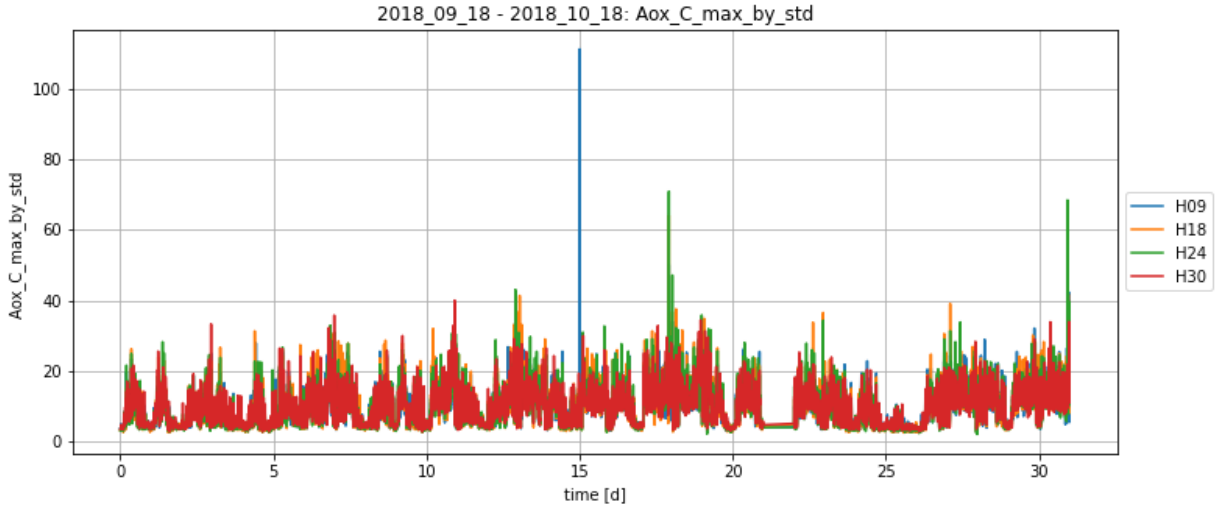


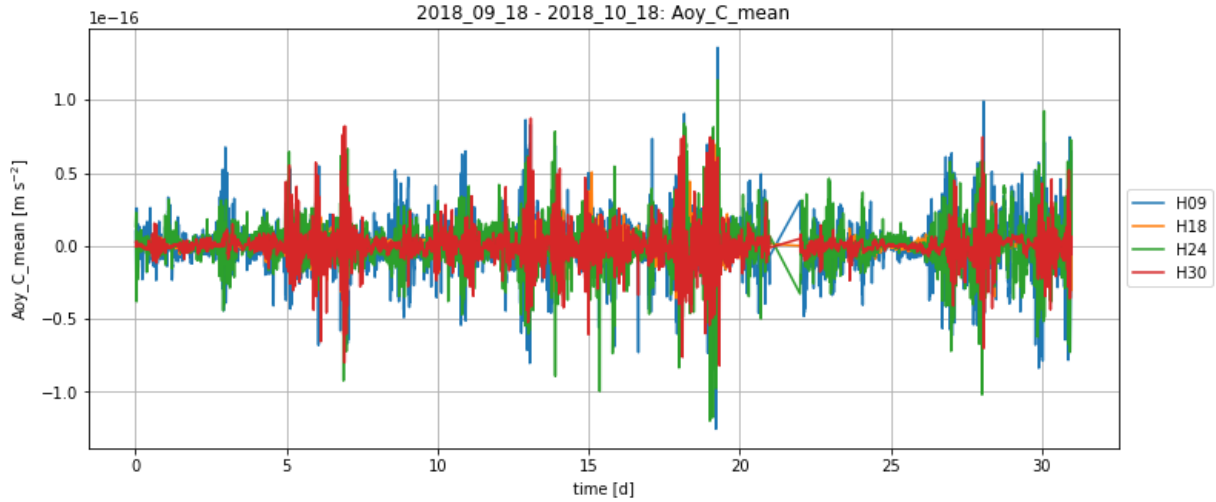

2018_09_18 - 2018_10_18: Aox_E_max
Aox_E_max [m s⁻²]
time [d]
H09
H18
H24
H30

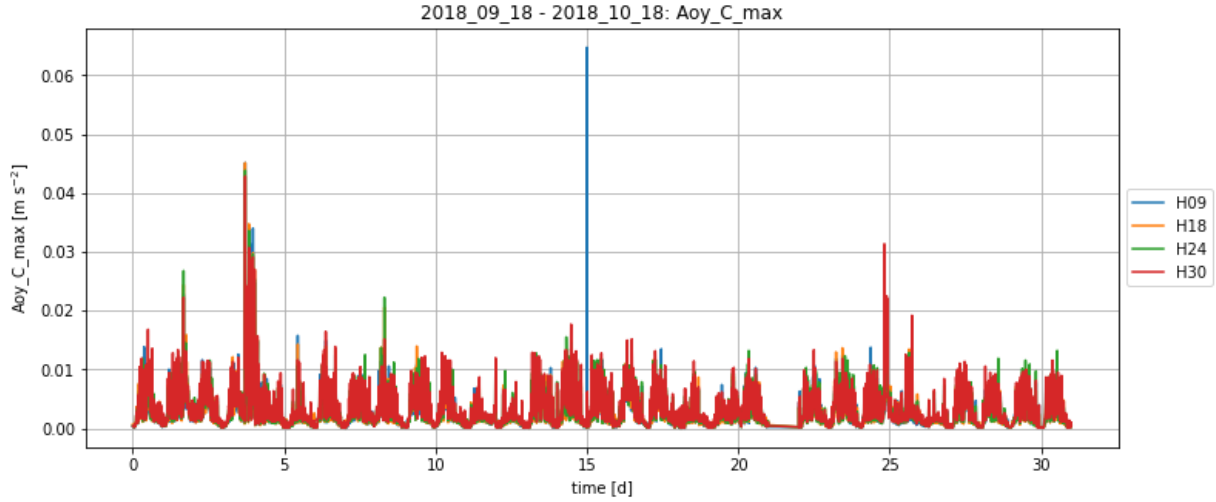

2018_09_18 - 2018_10_18: Aox_C_mean
1e−16
Aox_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

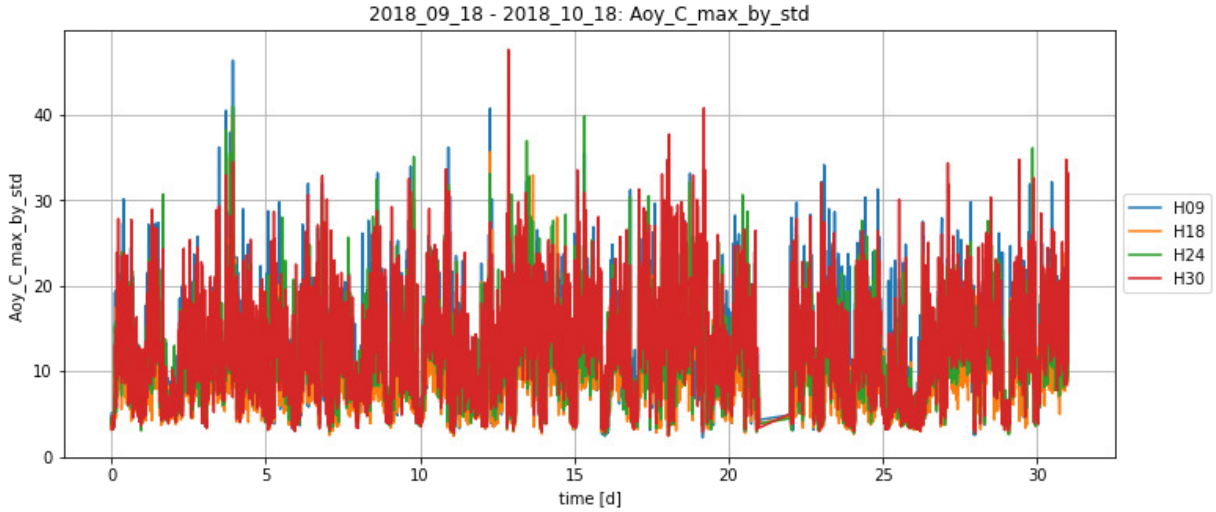

2018_09_18 - 2018_10_18: Aox_C_std
Aox_C_std [m s⁻²]
time [d]
H09
H18
H24
H30

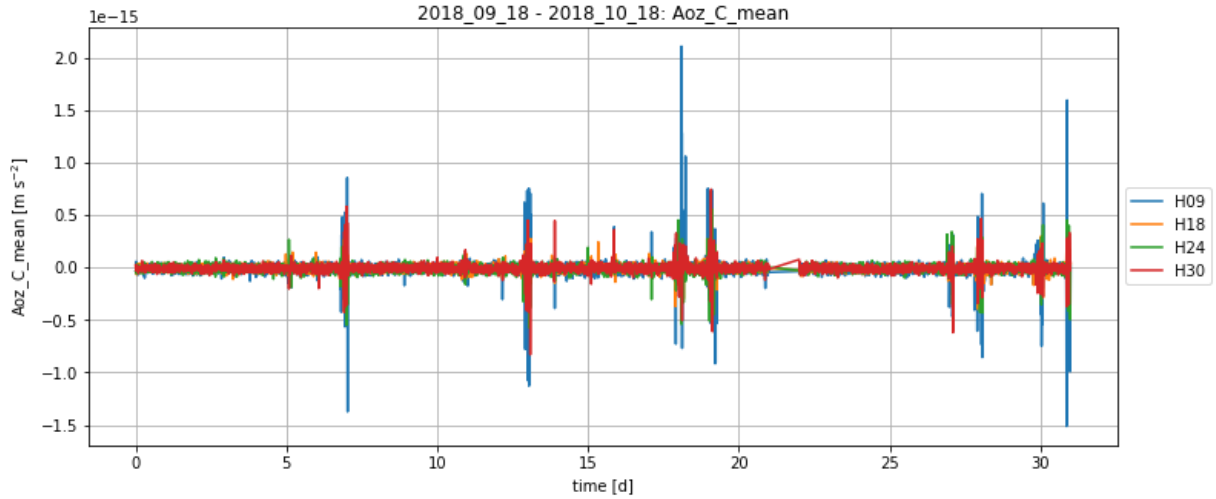

2018_09_18 - 2018_10_18: Aox_C_max
Aox_C_max [m s⁻²]
time [d]
H09
H18
H24
H30

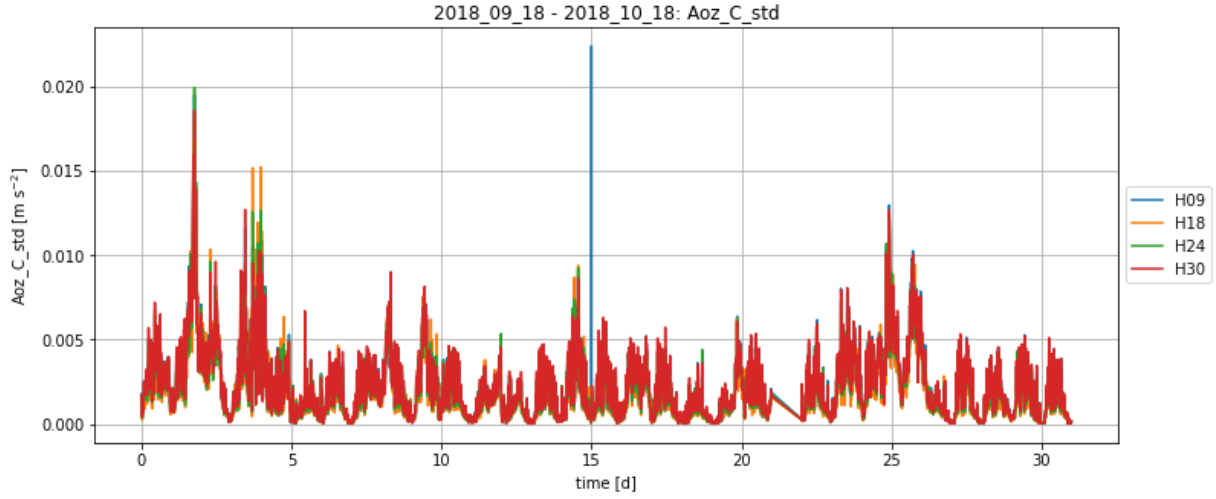

2018_09_18 - 2018_10_18: Aox_C_max_by_std
Aox_C_max_by_std
time [d]
H09
H18
H24
H30

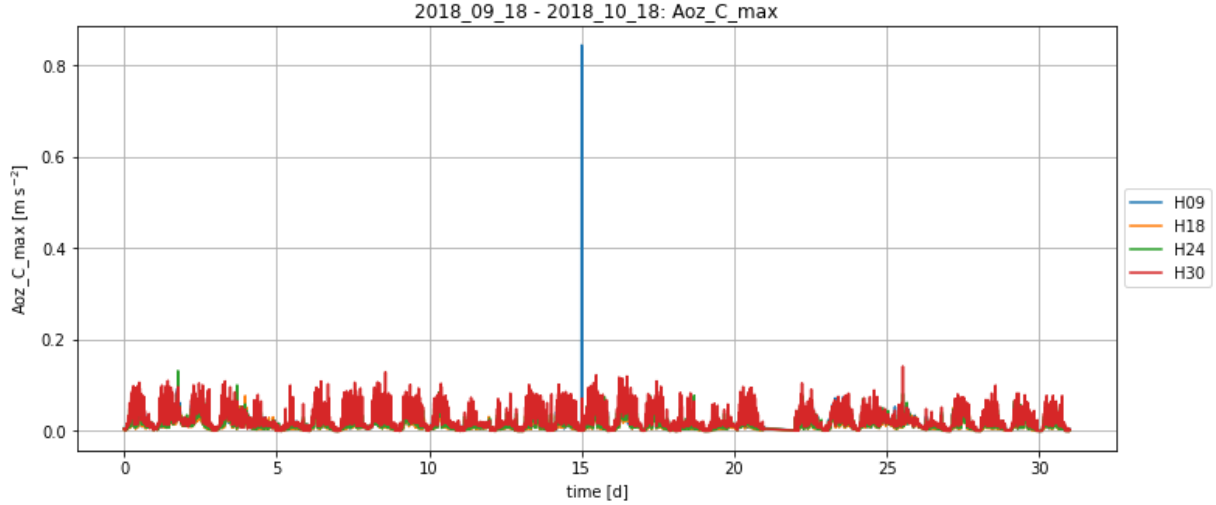

2018_09_18 - 2018_10_18: Aoy_C_mean
1e−16
Aoy_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

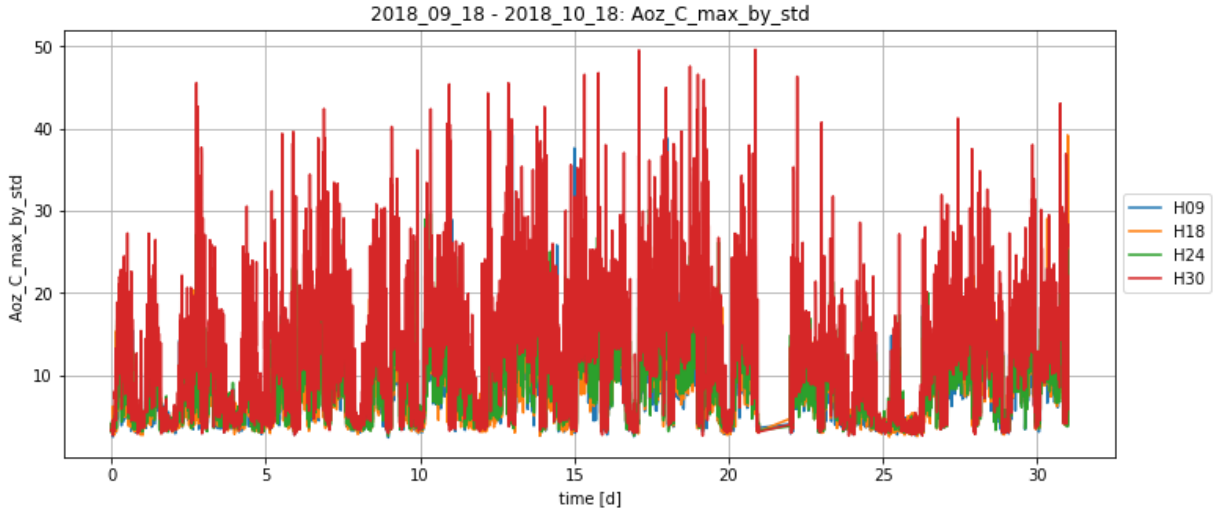

2018_09_18 - 2018_10_18: Aoy_C_max
Aoy_C_max [m s⁻²]
time [d]
H09
H18
H24
H30

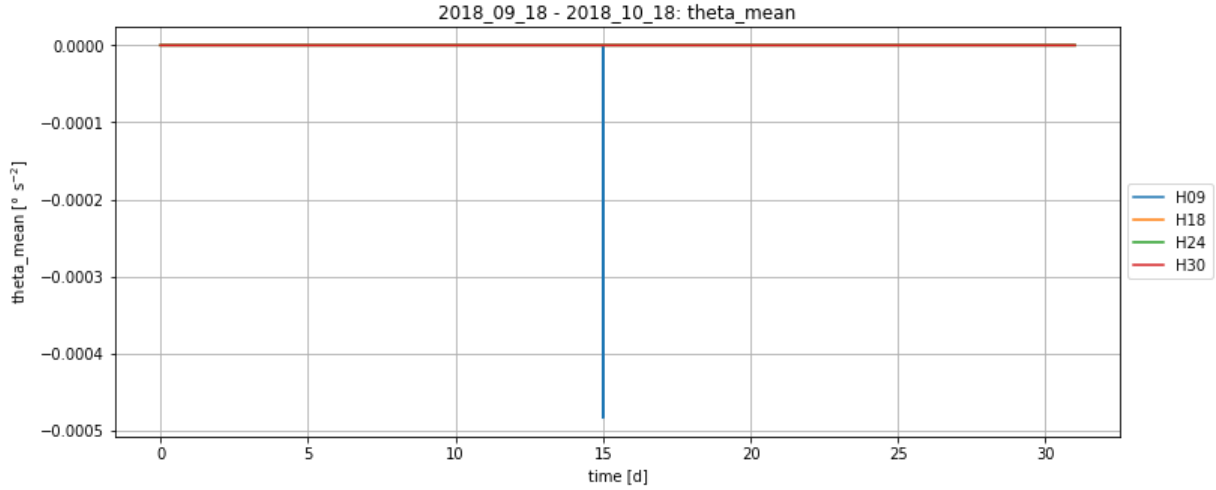

2018_09_18 - 2018_10_18: Aoy_C_max_by_std
Aoy_C_max_by_std
time [d]
H09
H18
H24
H30

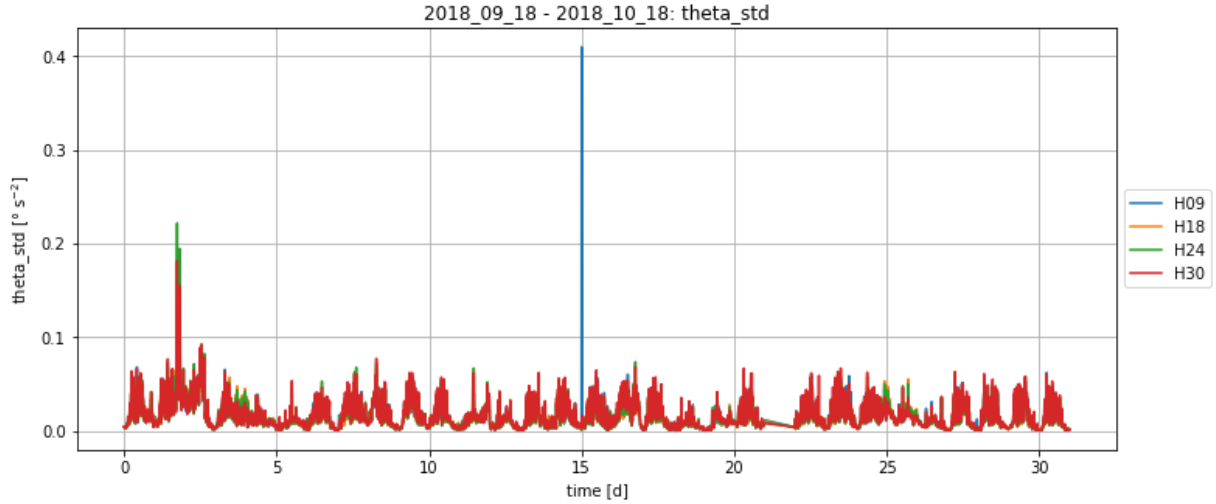

2018_09_18 - 2018_10_18: Aoz_C_mean
1e−15
Aoz_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

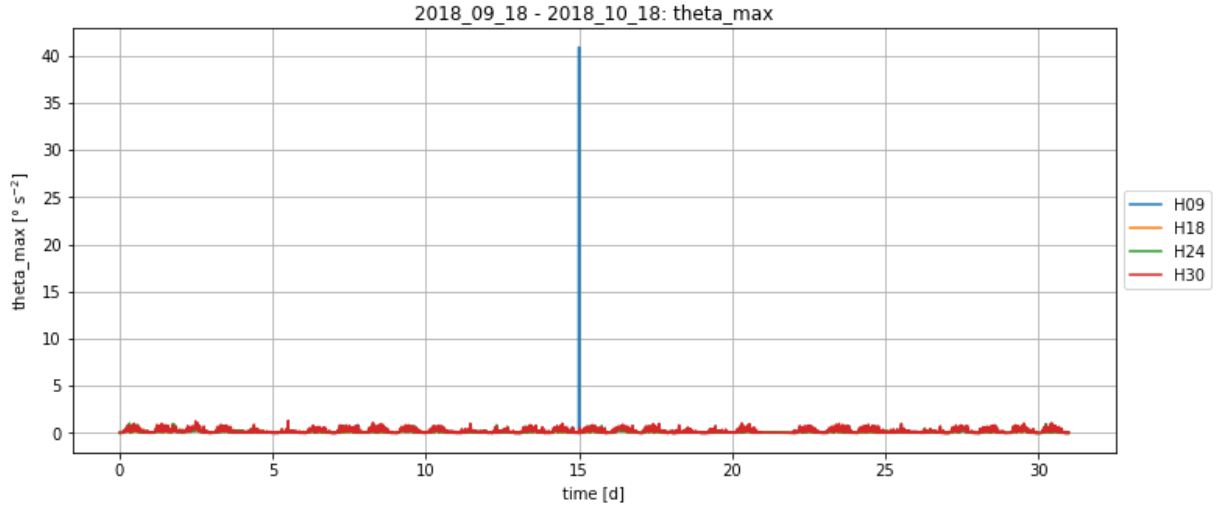

2018_09_18 - 2018_10_18: Aoz_C_std
Aoz_C_std [m s⁻²]
time [d]
H09
H18
H24
H30

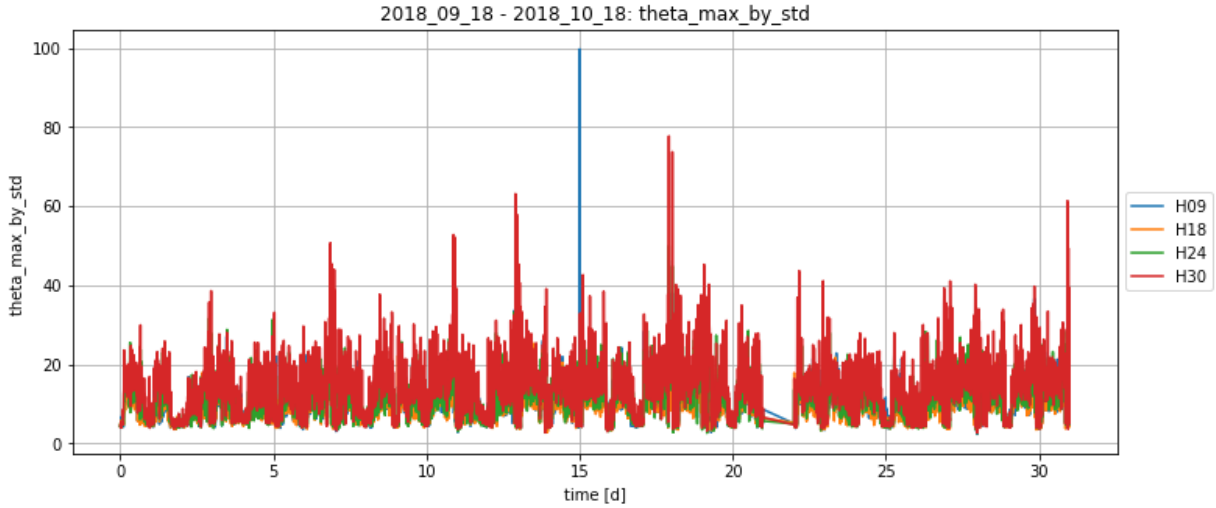

2018_09_18 - 2018_10_18: Aoz_C_max
Aoz_C_max [m s⁻²]
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: Aoz_C_max_by_std
Aoz_C_max_by_std
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: theta_mean
theta_mean [° s⁻²]
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: theta_std
theta_std [° s⁻²]
time [d]
H09
H18
H24
H30

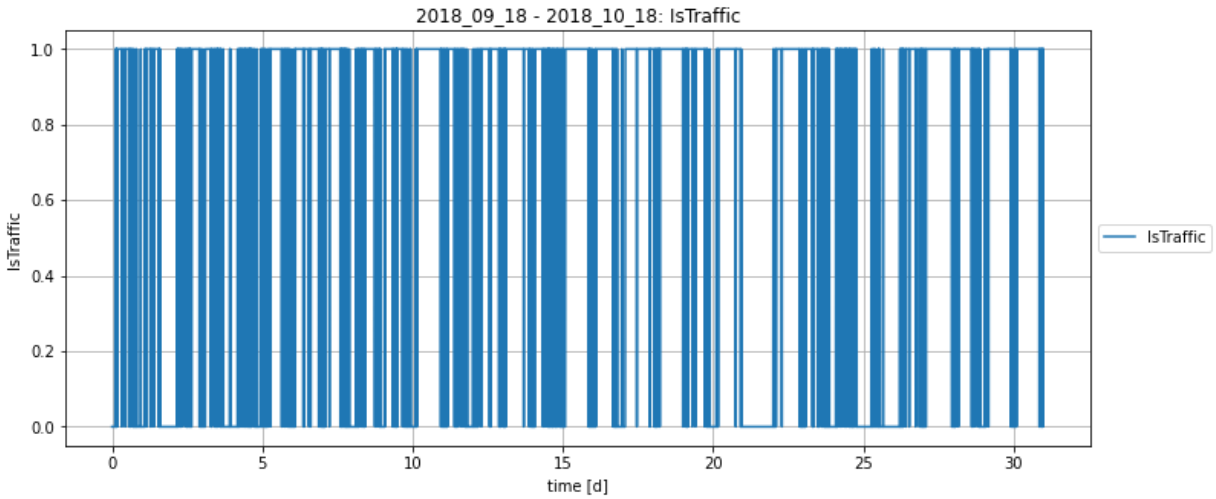

2018_09_18 - 2018_10_18: theta_max
theta_max [° s⁻²]
time [d]
H09
H18
H24
H30



Traffic

Below is a visualisation of the traffic classification previously created using the `find_traffic` method. Note that this method is not available in `full_detail` mode.

In [ ]:
```
if LFB.full_detail==False:
    LFB.plot_data(plot_time,LFB.traffic,ylabel='IsTraffic',yunit='',xlabel=plot_time
```



Clean the data

The `clean_data` method is used to remove 'invalid' sensors, that produce too much invalid data, as defined by the `threshold` keyword and remove invalid data from the remaining sensors. Note that some data cleaning already takes place during the import of the data if the `replace_invalid` keyword in the `load_data` method is set to *True*.

In [ ]:
```
LFB.clean_data(threshold=0.5,detailed_report=True)
```

```
H_mean all sensors deemed ok
H_std all sensors deemed ok
H_min all sensors deemed ok
H_max all sensors deemed ok
W_mean ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_mean invalid sensors: ['H10W']
W_std ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_std invalid sensors: ['H10W']
W_min ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
```

```
W_min invalid sensors: ['H10W']
W_max ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_max invalid sensors: ['H10W']
Vx_mean all sensors deemed ok
Vx_std all sensors deemed ok
Vx_min all sensors deemed ok
Vx_max all sensors deemed ok
Vy_mean all sensors deemed ok
Vy_std all sensors deemed ok
Vy_min all sensors deemed ok
Vy_max all sensors deemed ok
Dir_mean all sensors deemed ok
Dir_std all sensors deemed ok
Dir_min all sensors deemed ok
Dir_max all sensors deemed ok
AOA_mean ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_mean invalid sensors: ['H10W']
AOA_std ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_std invalid sensors: ['H10W']
AOA_min ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_min invalid sensors: ['H10W']
AOA_max ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_max invalid sensors: ['H10W']
H_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
H_turb invalid sensors: ['H10W']
W_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_turb invalid sensors: ['H10W']
Vx_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
Vx_turb invalid sensors: ['H10W']
Vy_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
Vy_turb invalid sensors: ['H10W']
Upwind all sensors deemed ok
Downwind all sensors deemed ok
Hanger_num all sensors deemed ok
y_pos all sensors deemed ok
West all sensors deemed ok
East all sensors deemed ok
Top all sensors deemed ok
Bottom all sensors deemed ok
H_mean all sensors deemed ok
H_std all sensors deemed ok
H_min all sensors deemed ok
H_max all sensors deemed ok
W_mean ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_mean invalid sensors: ['H10W']
W_std ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_std invalid sensors: ['H10W']
W_min ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_min invalid sensors: ['H10W']
W_max ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_max invalid sensors: ['H10W']
Vx_mean all sensors deemed ok
Vx_std all sensors deemed ok
Vx_min all sensors deemed ok
Vx_max all sensors deemed ok
Vy_mean all sensors deemed ok
Vy_std all sensors deemed ok
Vy_min all sensors deemed ok
Vy_max all sensors deemed ok
Dir_mean all sensors deemed ok
Dir_std all sensors deemed ok
Dir_min all sensors deemed ok
Dir_max all sensors deemed ok
AOA_mean ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
```

```
AOA_mean invalid sensors: ['H10W']
AOA_std ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_std invalid sensors: ['H10W']
AOA_min ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_min invalid sensors: ['H10W']
AOA_max ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
AOA_max invalid sensors: ['H10W']
H_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
H_turb invalid sensors: ['H10W']
W_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
W_turb invalid sensors: ['H10W']
Vx_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
Vx_turb invalid sensors: ['H10W']
Vy_turb ok sensors: ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H24W']
Vy_turb invalid sensors: ['H10W']
Upwind all sensors deemed ok
Downwind all sensors deemed ok
Hanger_num all sensors deemed ok
y_pos all sensors deemed ok
West all sensors deemed ok
East all sensors deemed ok
Top all sensors deemed ok
Bottom all sensors deemed ok
Aox_W_mean all sensors deemed ok
Aox_W_std all sensors deemed ok
Aox_W_min all sensors deemed ok
Aox_W_max all sensors deemed ok
Aox_W_max_by_std all sensors deemed ok
Aoy_W_mean all sensors deemed ok
Aoy_W_std all sensors deemed ok
Aoy_W_min all sensors deemed ok
Aoy_W_max all sensors deemed ok
Aoy_W_max_by_std all sensors deemed ok
Aox_E_mean all sensors deemed ok
Aox_E_std all sensors deemed ok
Aox_E_min all sensors deemed ok
Aox_E_max all sensors deemed ok
Aox_E_max_by_std all sensors deemed ok
Aoy_E_mean all sensors deemed ok
Aoy_E_std all sensors deemed ok
Aoy_E_min all sensors deemed ok
Aoy_E_max all sensors deemed ok
Aoy_E_max_by_std all sensors deemed ok
Aox_C_mean all sensors deemed ok
Aox_C_std all sensors deemed ok
Aox_C_min all sensors deemed ok
Aox_C_max all sensors deemed ok
Aox_C_max_by_std all sensors deemed ok
Aoy_C_mean all sensors deemed ok
Aoy_C_std all sensors deemed ok
Aoy_C_min all sensors deemed ok
Aoy_C_max all sensors deemed ok
Aoy_C_max_by_std all sensors deemed ok
Aoz_C_mean all sensors deemed ok
Aoz_C_std all sensors deemed ok
Aoz_C_min all sensors deemed ok
Aoz_C_max all sensors deemed ok
Aoz_C_max_by_std all sensors deemed ok
theta_mean all sensors deemed ok
theta_std all sensors deemed ok
theta_min all sensors deemed ok
theta_max all sensors deemed ok
theta_max_by_std all sensors deemed ok
Hanger_num all sensors deemed ok
```

```
y_pos all sensors deemed ok
West all sensors deemed ok
East all sensors deemed ok
Top all sensors deemed ok
Bottom all sensors deemed ok
Aox_W_mean all sensors deemed ok
Aox_W_std all sensors deemed ok
Aox_W_min all sensors deemed ok
Aox_W_max all sensors deemed ok
Aox_W_max_by_std all sensors deemed ok
Aoy_W_mean all sensors deemed ok
Aoy_W_std all sensors deemed ok
Aoy_W_min all sensors deemed ok
Aoy_W_max all sensors deemed ok
Aoy_W_max_by_std all sensors deemed ok
Aox_E_mean all sensors deemed ok
Aox_E_std all sensors deemed ok
Aox_E_min all sensors deemed ok
Aox_E_max all sensors deemed ok
Aox_E_max_by_std all sensors deemed ok
Aoy_E_mean all sensors deemed ok
Aoy_E_std all sensors deemed ok
Aoy_E_min all sensors deemed ok
Aoy_E_max all sensors deemed ok
Aoy_E_max_by_std all sensors deemed ok
Aox_C_mean all sensors deemed ok
Aox_C_std all sensors deemed ok
Aox_C_min all sensors deemed ok
Aox_C_max all sensors deemed ok
Aox_C_max_by_std all sensors deemed ok
Aoy_C_mean all sensors deemed ok
Aoy_C_std all sensors deemed ok
Aoy_C_min all sensors deemed ok
Aoy_C_max all sensors deemed ok
Aoy_C_max_by_std all sensors deemed ok
Aoz_C_mean all sensors deemed ok
Aoz_C_std all sensors deemed ok
Aoz_C_min all sensors deemed ok
Aoz_C_max all sensors deemed ok
Aoz_C_max_by_std all sensors deemed ok
theta_mean all sensors deemed ok
theta_std all sensors deemed ok
theta_min all sensors deemed ok
theta_max all sensors deemed ok
theta_max_by_std all sensors deemed ok
Hanger_num all sensors deemed ok
y_pos all sensors deemed ok
West all sensors deemed ok
East all sensors deemed ok
Top all sensors deemed ok
Bottom all sensors deemed ok
H_mean  cummulative data loss: 0.39 %
H_std  cummulative data loss: 0.42 %
H_min  cummulative data loss: 0.42 %
H_max  cummulative data loss: 0.42 %
W_mean  cummulative data loss: 0.42 %
W_std  cummulative data loss: 0.42 %
W_min  cummulative data loss: 0.42 %
W_max  cummulative data loss: 0.42 %
Vx_mean  cummulative data loss: 0.42 %
Vx_std  cummulative data loss: 0.42 %
Vx_min  cummulative data loss: 0.42 %
Vx_max  cummulative data loss: 0.42 %
Vy_mean  cummulative data loss: 0.42 %
```

```
Vy_std  cummulative data loss: 0.42 %
Vy_min  cummulative data loss: 0.42 %
Vy_max  cummulative data loss: 0.42 %
Dir_mean  cummulative data loss: 0.42 %
Dir_std  cummulative data loss: 0.42 %
Dir_min  cummulative data loss: 0.42 %
Dir_max  cummulative data loss: 0.42 %
AOA_mean  cummulative data loss: 0.42 %
AOA_std  cummulative data loss: 0.42 %
AOA_min  cummulative data loss: 0.42 %
AOA_max  cummulative data loss: 0.42 %
H_turb  cummulative data loss: 0.42 %
W_turb  cummulative data loss: 0.42 %
Vx_turb  cummulative data loss: 0.42 %
Vy_turb  cummulative data loss: 0.42 %
Upwind  cummulative data loss: 0.42 %
Downwind  cummulative data loss: 0.42 %
Hanger_num  cummulative data loss: 0.42 %
y_pos  cummulative data loss: 0.42 %
West  cummulative data loss: 0.42 %
East  cummulative data loss: 0.42 %
Top  cummulative data loss: 0.42 %
Bottom  cummulative data loss: 0.42 %
Total data loss: 0.42 %
```

Save `BridgeData` state

`dill.dump` can be used to save the `BridgeData` class instance in its current state, including imported and processed data.

In [ ]:
```
# Save the BridgeData class instance (incl. data)
dill.dump(LFB,file=open(LFB.file_path+LFB.file_names[0][:-4]+'-'+LFB.file_names[-1][
```

Load `BridgeData` state

`dill.load` can be used to load the previously saved `BridgeData` class instance in its respective state, at the time of saving, including imported and processed data. This is usefull when returning to the analysis of a larger dataset, as the long import process (`load_data`) can be skipped.

In [ ]:
```
# Re-load the file.
LFB = dill.load(open(file_path+file_names[0][:-4]+'-'+file_names[-1][:-4],'rb'))
```

Traffic classification

The traffic classification needs to be re-done for processed (cleaned) data, by setting the `cleaned` keyword to *True* when using the `find_traffic` method.

In [ ]:
```
if LFB.full_detail==False:
    LFB.find_traffic(traffic_thresh=8,cleaned=True)
    print('Traffic percentage:',np.round(100*np.count_nonzero(LFB.traffic_cleaned)/l
```

```
Traffic percentage: 68.15 %
```

Filter

Similarly to the traffic classification, filtering needs to be re-done for processed (cleaned) data, by providing the `_cleaned` data array to the `filter_data` method.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
if LFB.full_detail:
    title_suffix+= ' - full detail'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'],prior_idxs=filter_idxs
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'][LFB.get_ok_sensor_ind(
```

Plot *cleaned* data

Below you will find examples for plotting the *cleaned* time series data, similar to what was done in the plotting of the un-processed (un-cleaned) data.

Time settings

Similarly to the traffic classification, the time arrays need to be re-initialized for processed (cleaned) data, by setting the `cleaned` keyword to *True* when using the `feature_time` method.

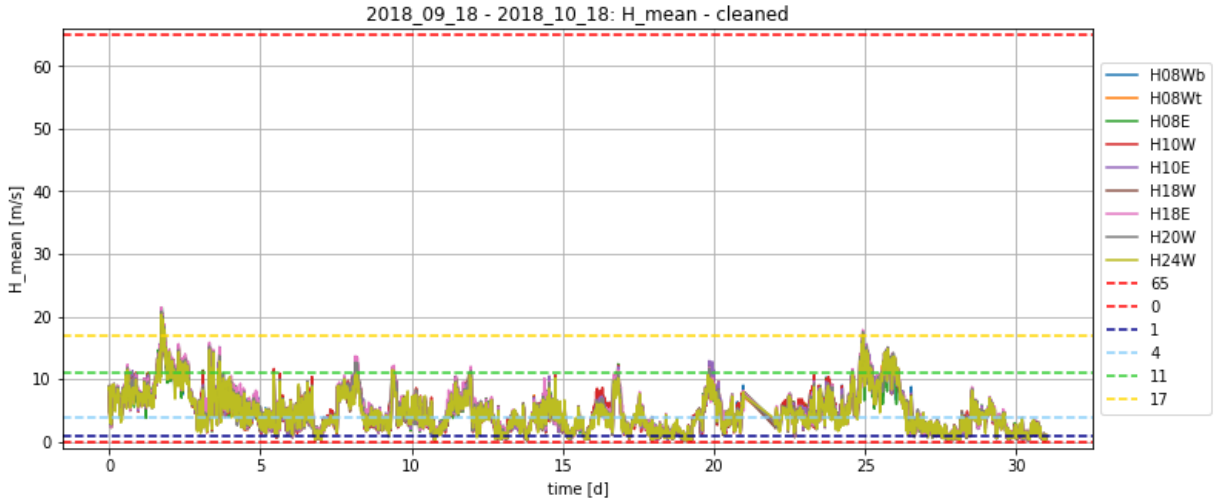
In [ ]:
```
LFB.feature_time(cleaned=True) # Initialize time series for cleaned data.

timescale='days' #Change this variable to 'days', 'hours', 'minutes' or 'seconds' to

if timescale=='days':
    plot_time=LFB.days_cleaned
    plot_time_label = 'time [d]'
elif timescale=='hours':
    plot_time=LFB.hours_cleaned
    plot_time_label = 'time [h]'
elif timescale=='minutes':
    plot_time=LFB.minutes_cleaned
    plot_time_label = 'time [min]'
elif timescale=='seconds':
    plot_time=LFB.seconds_cleaned
    plot_time_label = 'time [s]'

# time_of_interest=(600,610) # Use this line instead of the one below to zoom in on
time_of_interest=(None,None) # Default
```

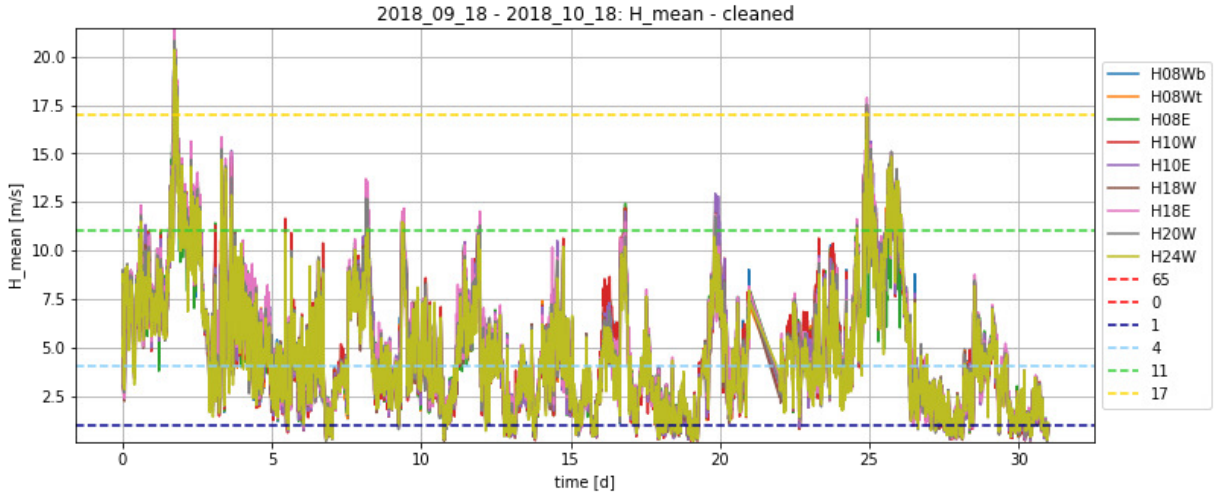
Anemometers

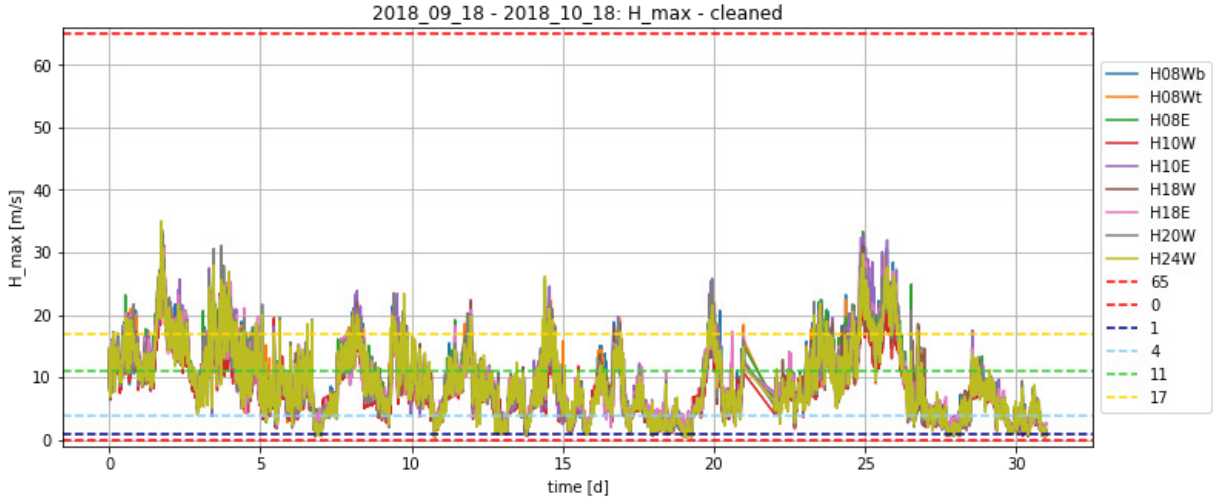
Below is an example for batch post-processing of **anemometer** data using a *for-loop* and *if-statements* on the *keys* previously defined by `keys_of_interest`. This allows to set specific *plot parameters* in the `plot_data` method for different *keys*. The names of anemometers that are not of interest can be given to the `ignored_anemos` keyword as a list.

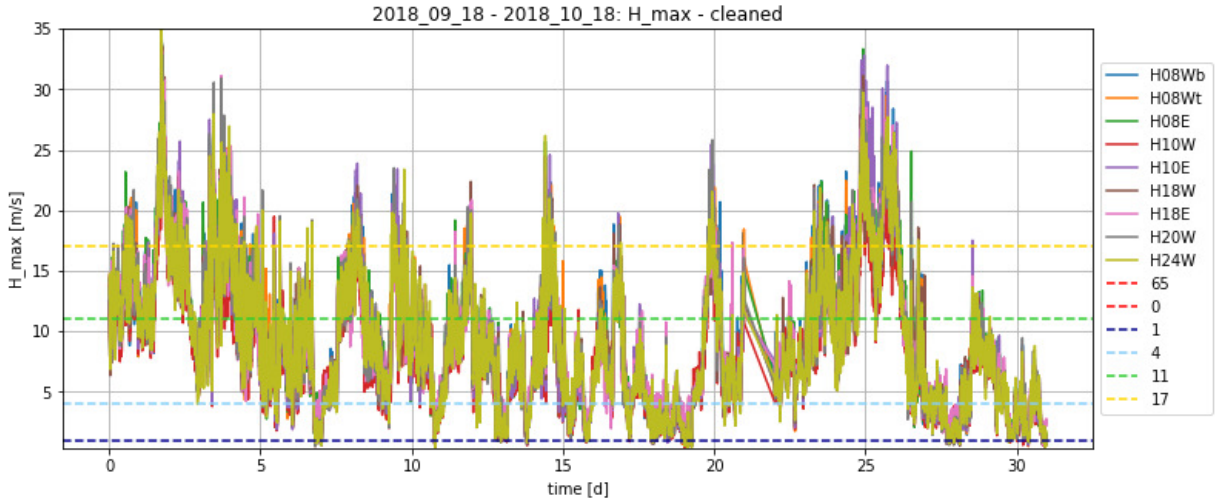
In [ ]:
```
ignored_anemos = []
for key in LFB.anemo_cleaned.keys():
    if key in keys_of_interest:
        if key.startswith('H_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
```

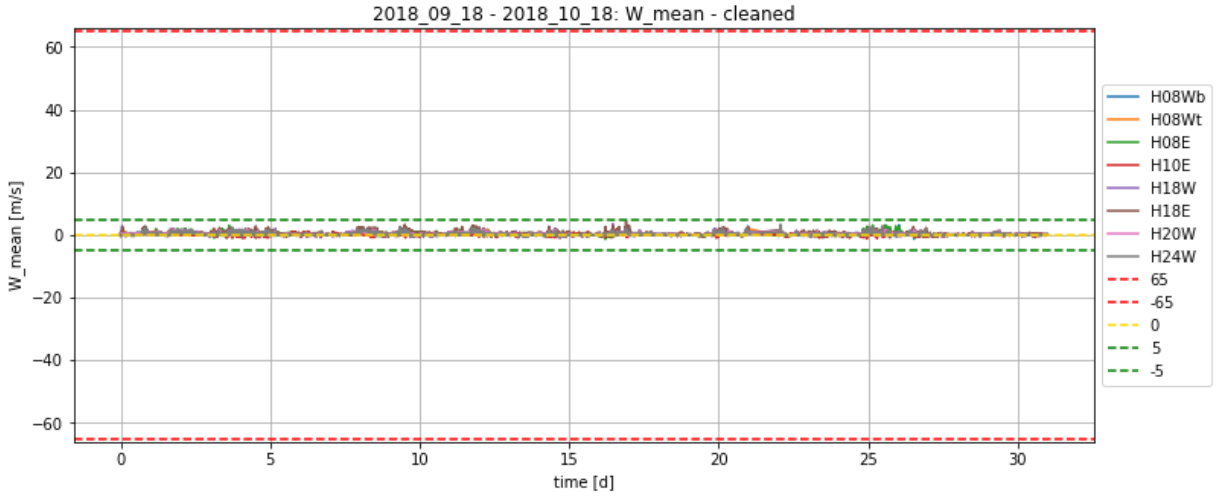
```
        elif key.startswith('Dir_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
        elif key.startswith('W_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
        elif key.startswith('Vx_m') or key.startswith('Vy_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
        elif key.startswith('AOA_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
        elif key.startswith('AOI_m'):
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
        else:
            LFB.plot_data(plot_time[filter_idxs],LFB.anemo_cleaned[key].T[filter_idx
```

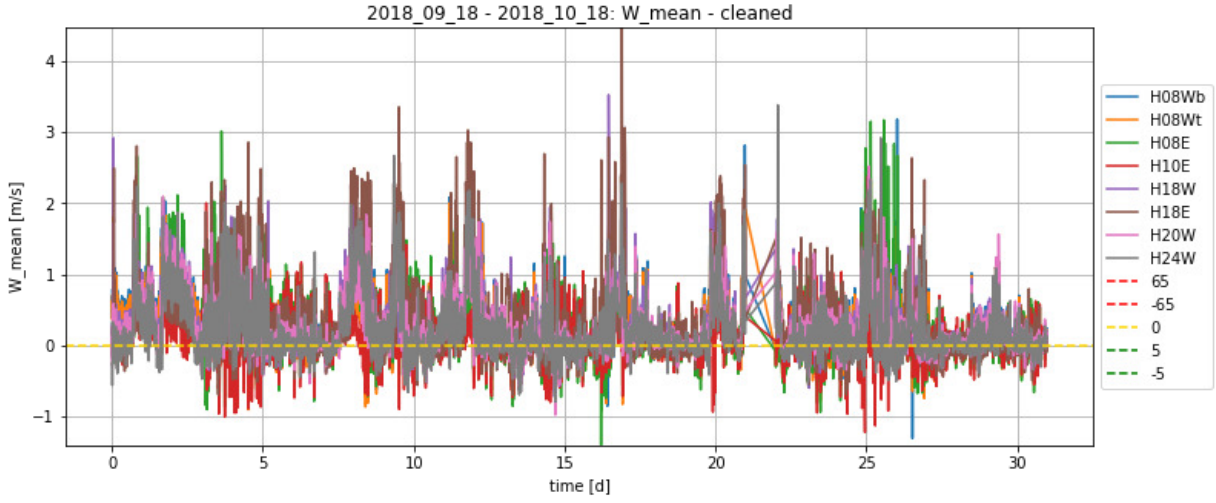


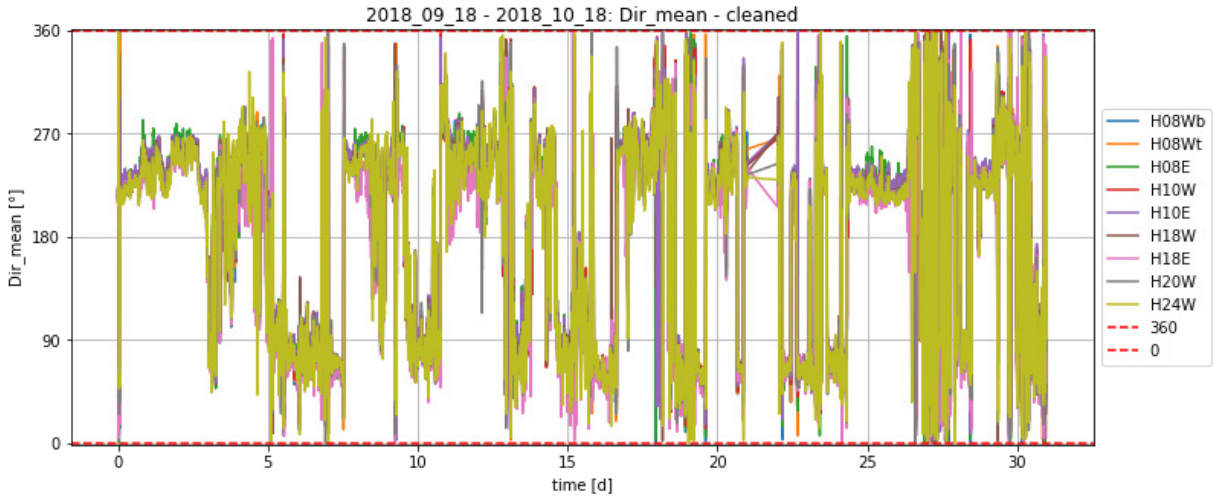


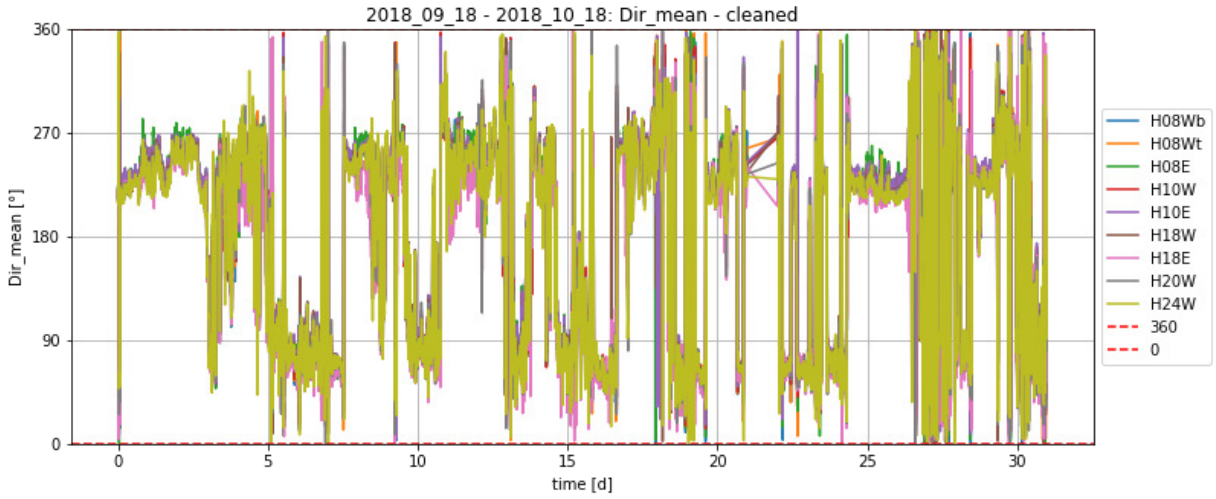

2018_09_18 - 2018_10_18: H_max - cleaned
H_max [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
0
1
4
11
17

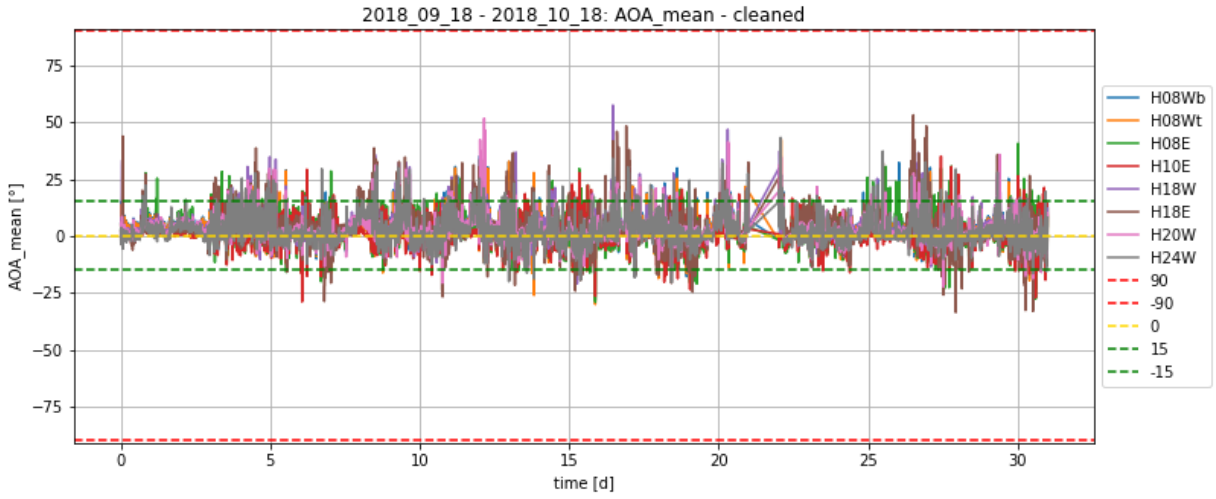

2018_09_18 - 2018_10_18: H_max - cleaned
H_max [m/s]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
65
0
1
4
11
17

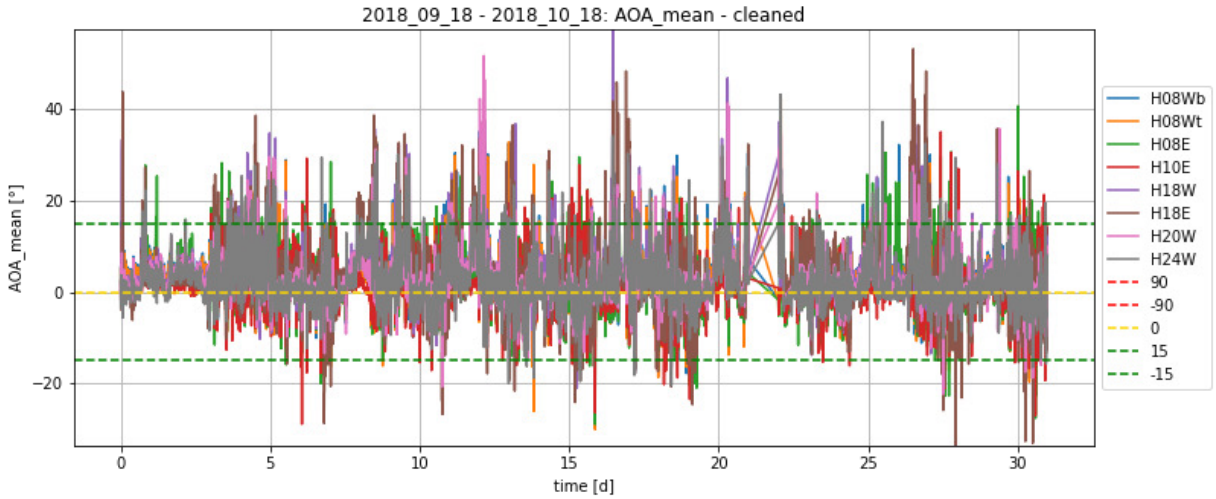

2018_09_18 - 2018_10_18: W_mean - cleaned
W_mean [m/s]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
65
-65
0
5
-5

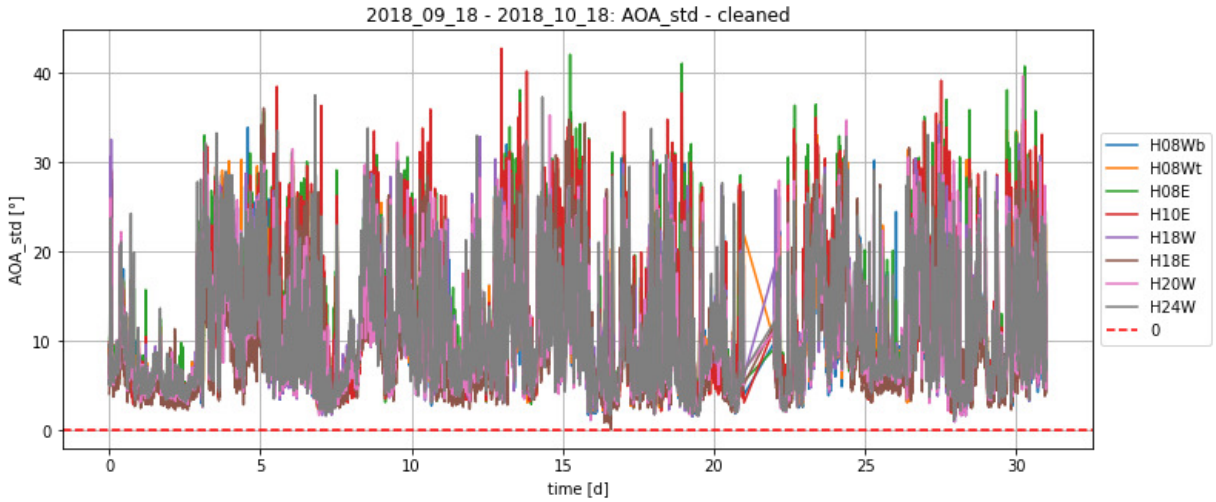

2018_09_18 - 2018_10_18: W_mean - cleaned
W_mean [m/s]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
65
-65
0
5
-5

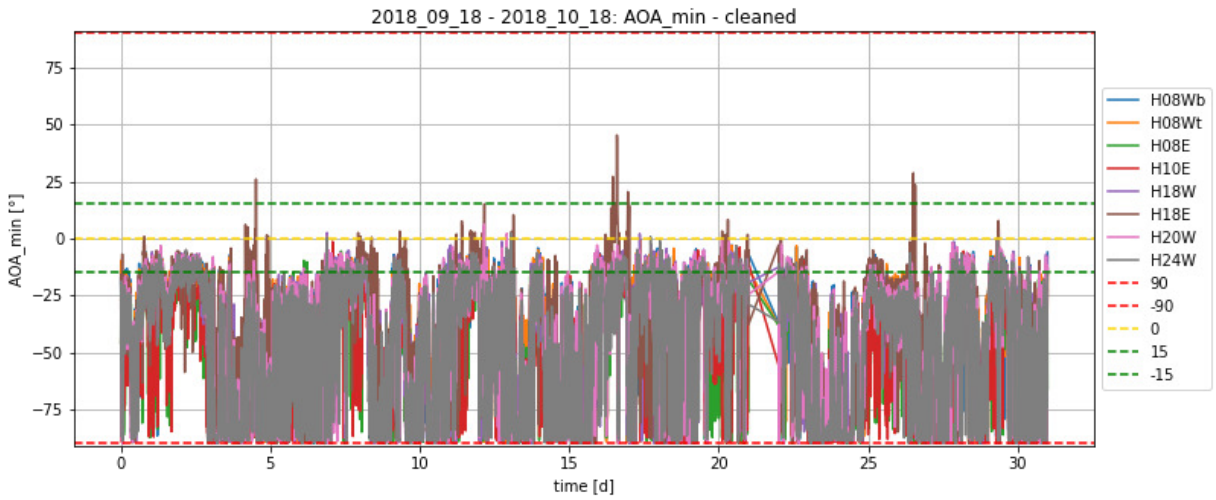

2018_09_18 - 2018_10_18: Dir_mean - cleaned
Dir_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
360
0

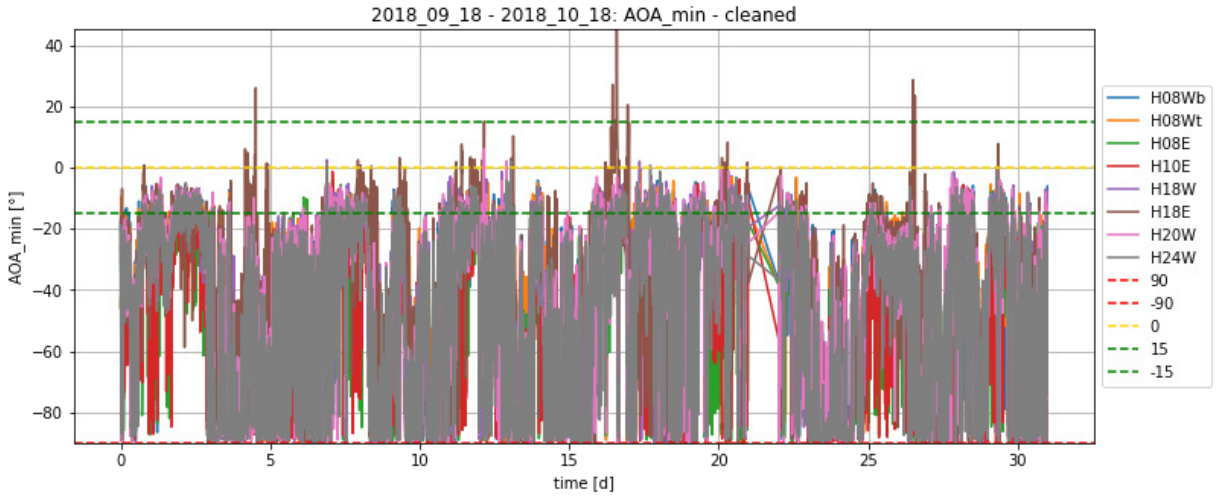

2018_09_18 - 2018_10_18: Dir_mean - cleaned
Dir_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
360
0

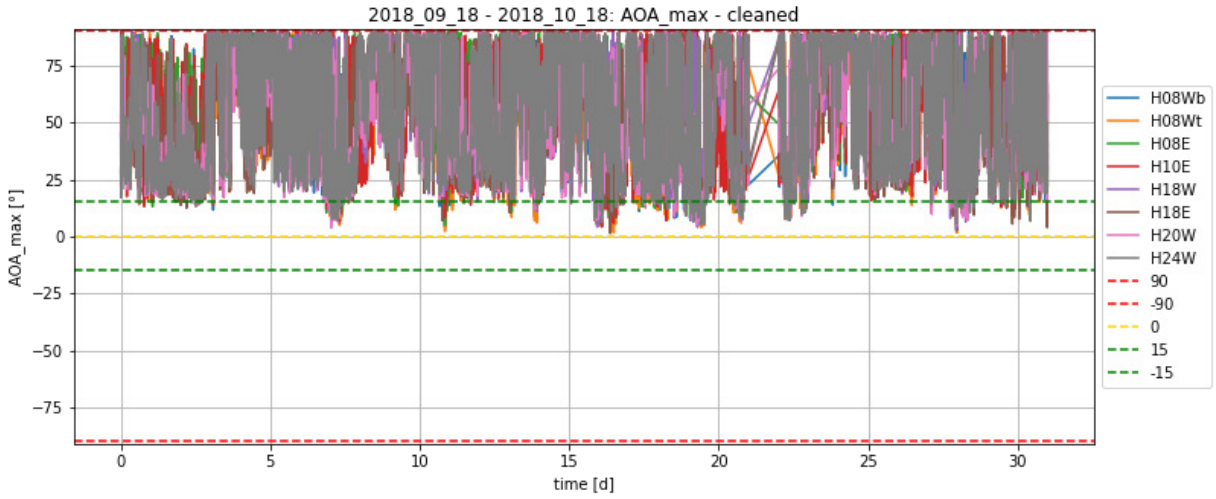

2018_09_18 - 2018_10_18: AOA_mean - cleaned
AOA_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

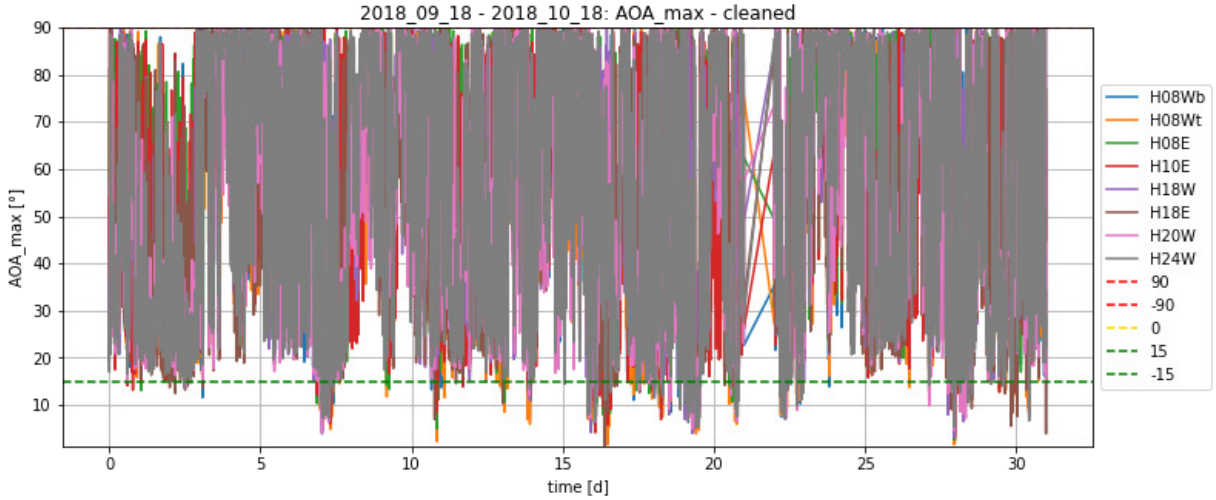

2018_09_18 - 2018_10_18: AOA_mean - cleaned
AOA_mean [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

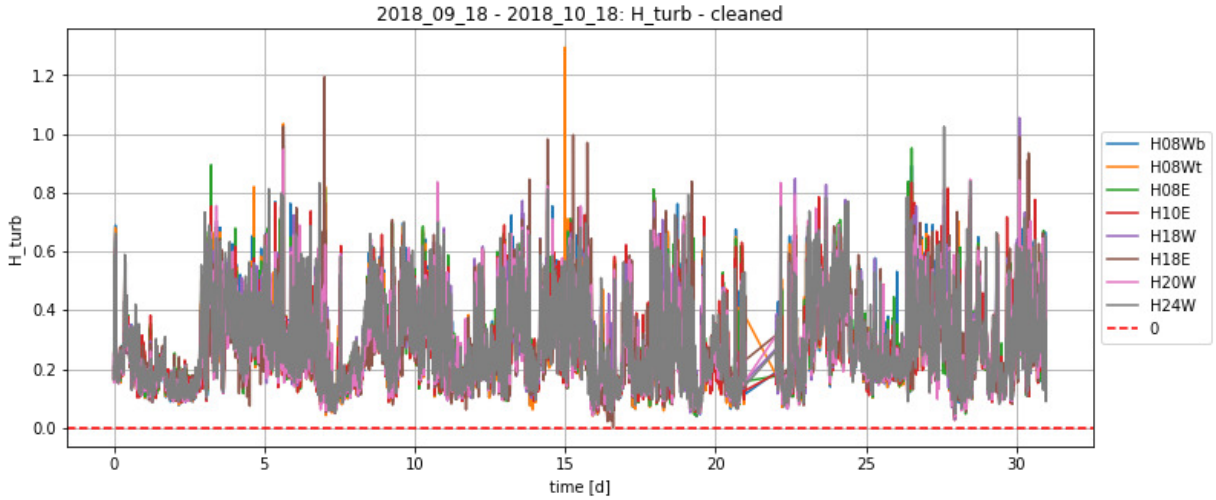

2018_09_18 - 2018_10_18: AOA_std - cleaned
AOA_std [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
0

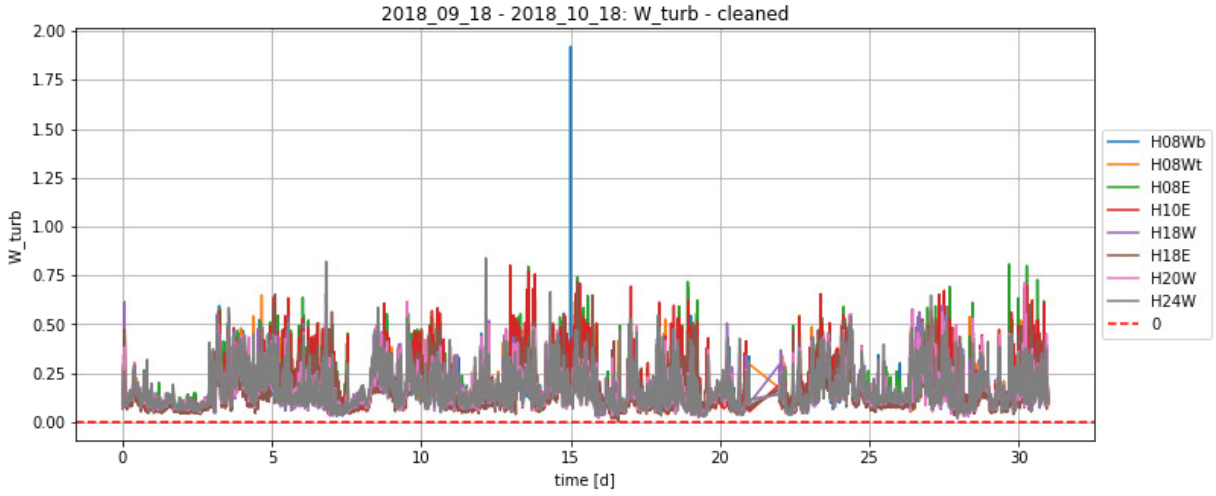

2018_09_18 - 2018_10_18: AOA_min - cleaned
AOA_min [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15

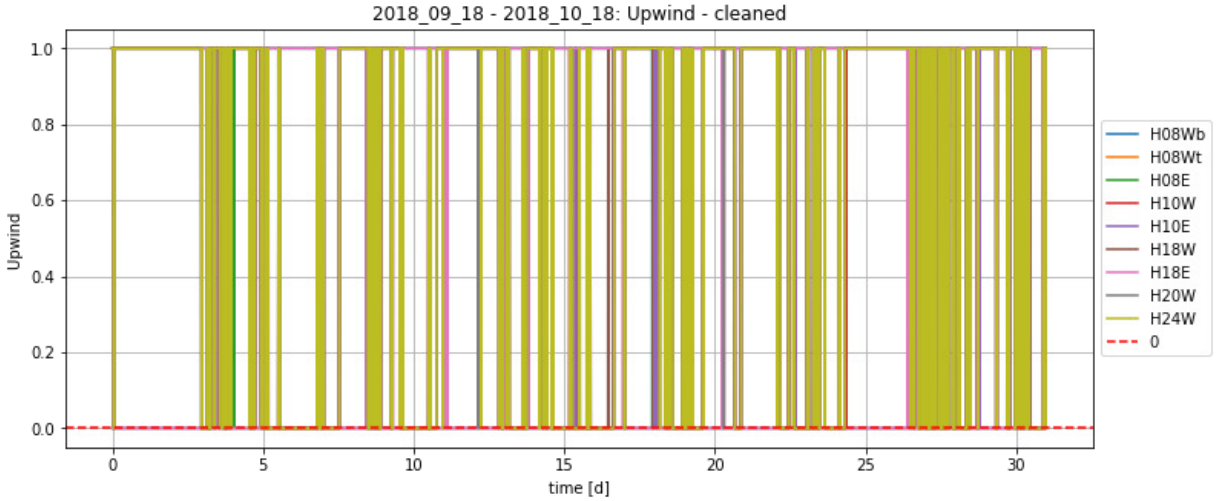

2018_09_18 - 2018_10_18: AOA_min - cleaned
AOA_min [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: AOA_max - cleaned
AOA_max [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: AOA_max - cleaned
AOA_max [°]
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
90
-90
0
15
-15


2018_09_18 - 2018_10_18: H_turb - cleaned
H_turb
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
0


2018_09_18 - 2018_10_18: W_turb - cleaned
W_turb
time [d]
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W
0

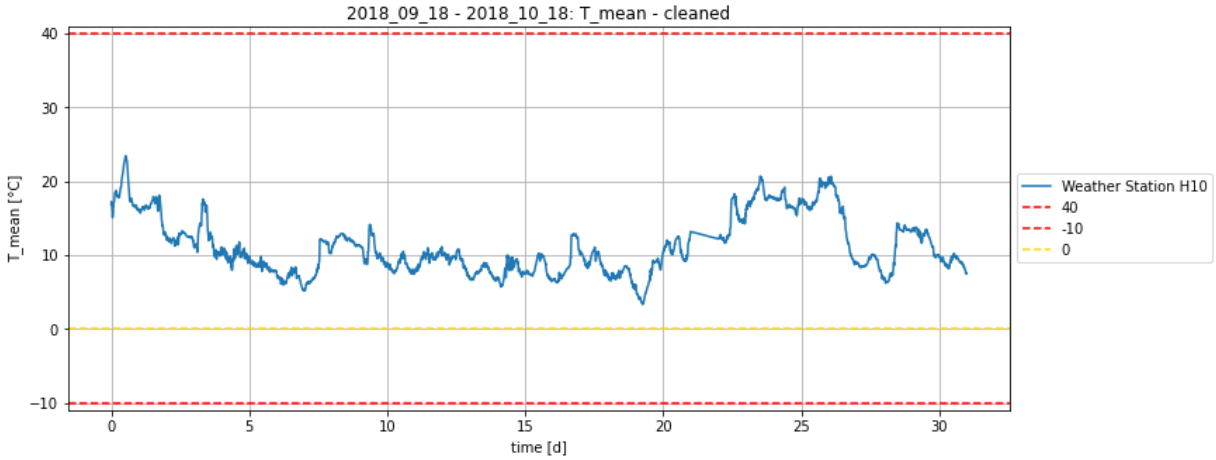

2018_09_18 - 2018_10_18: Upwind - cleaned
Upwind
time [d]
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W
0

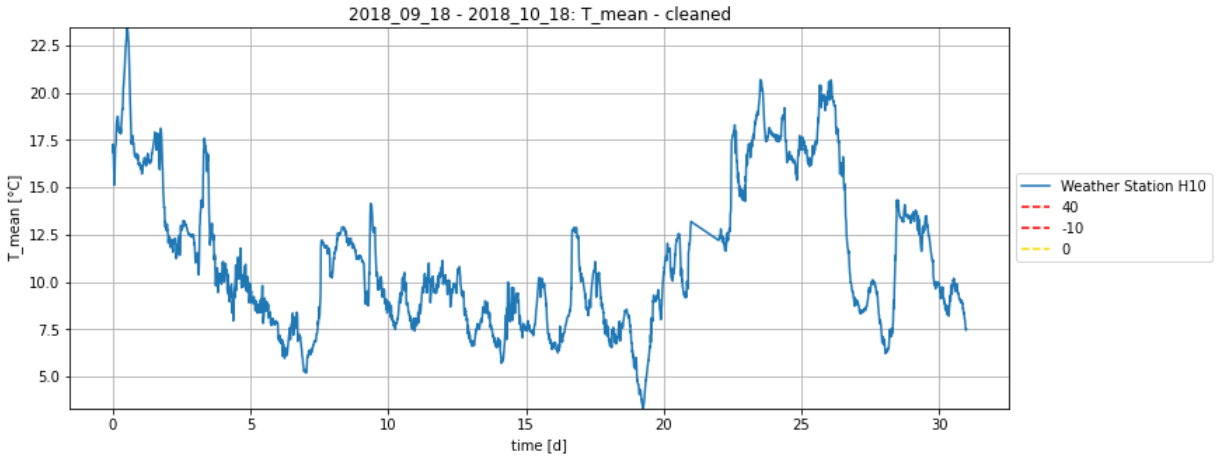
Weather

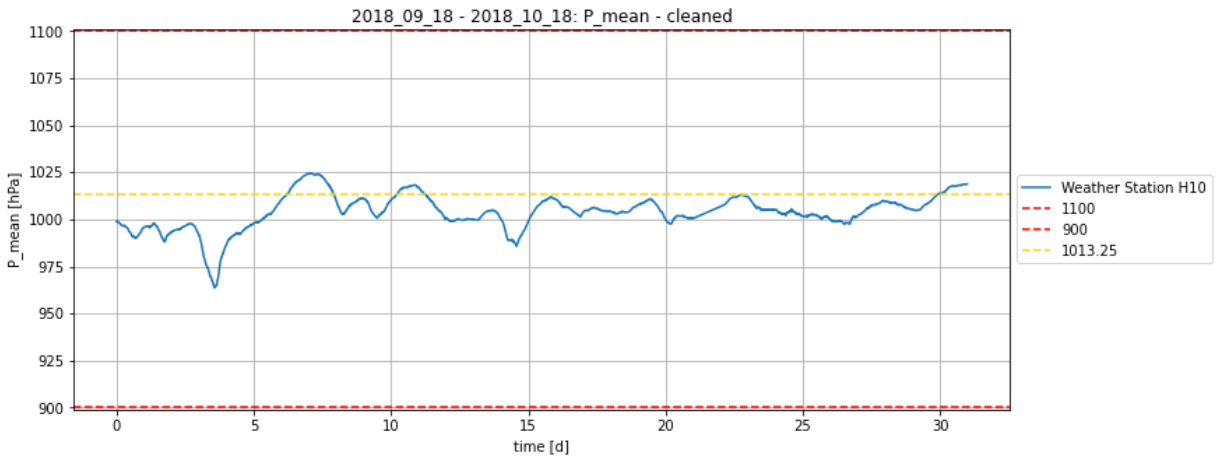
Similarly the `weather` data from the *weather station* on Hanger 10 can be plotted.

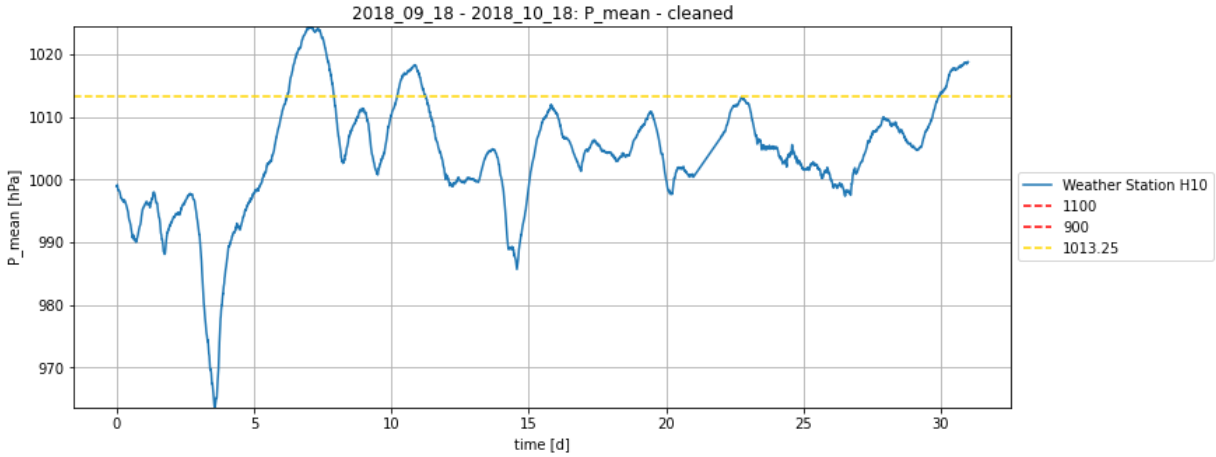
In [ ]:

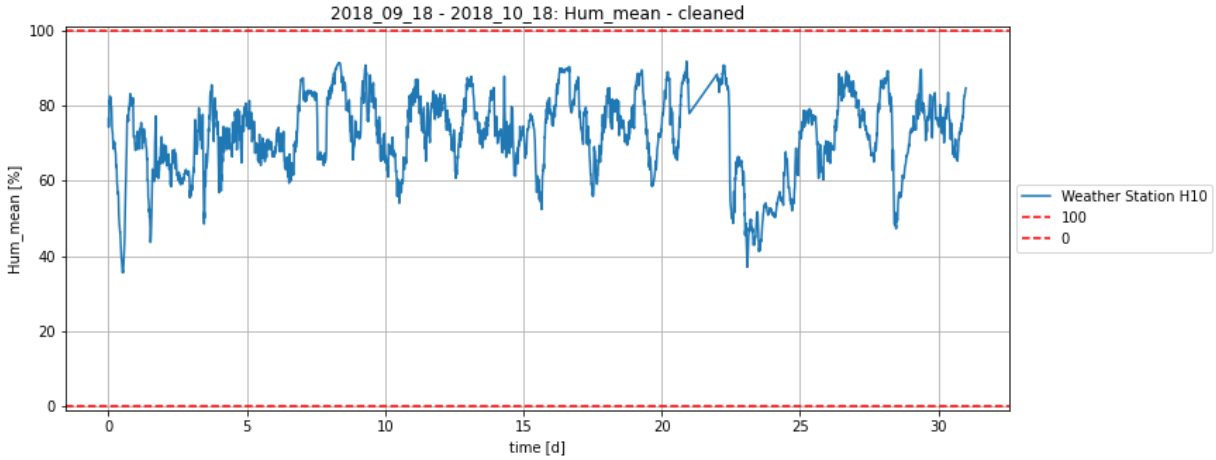
```
for key in LFB.weather_cleaned.keys():
    if key in keys_of_interest:
        if key.startswith('Hum_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
        elif key.startswith('T_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
        elif key.startswith('P_mean'):
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
            LFB.plot_data(plot_time[filter_idxs],LFB.weather_cleaned[key][filter_idx
```

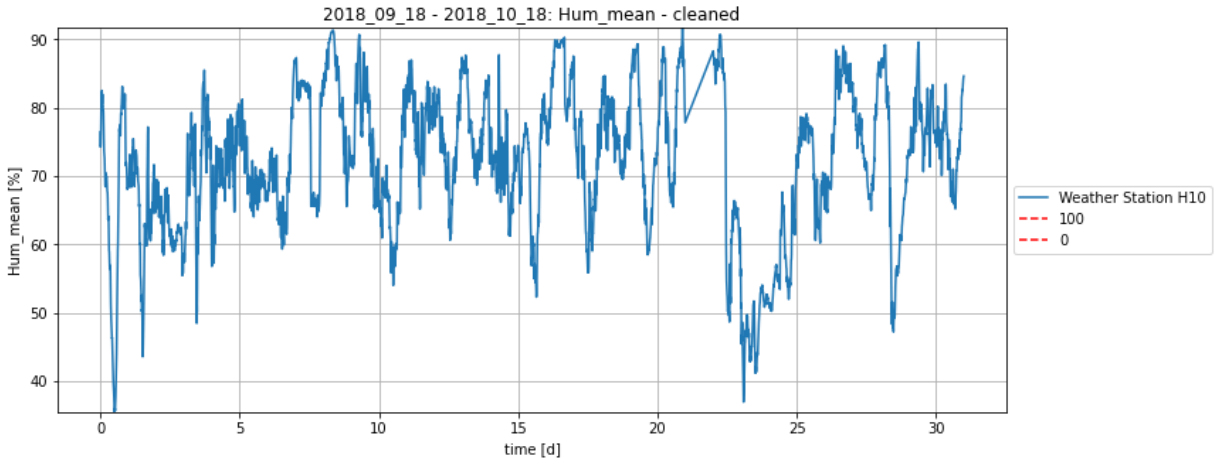










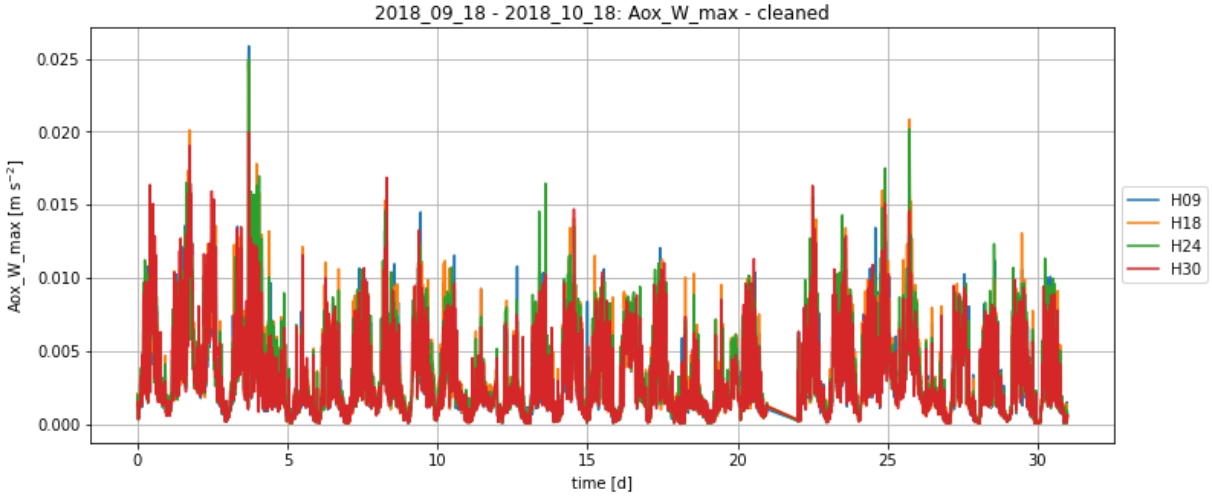


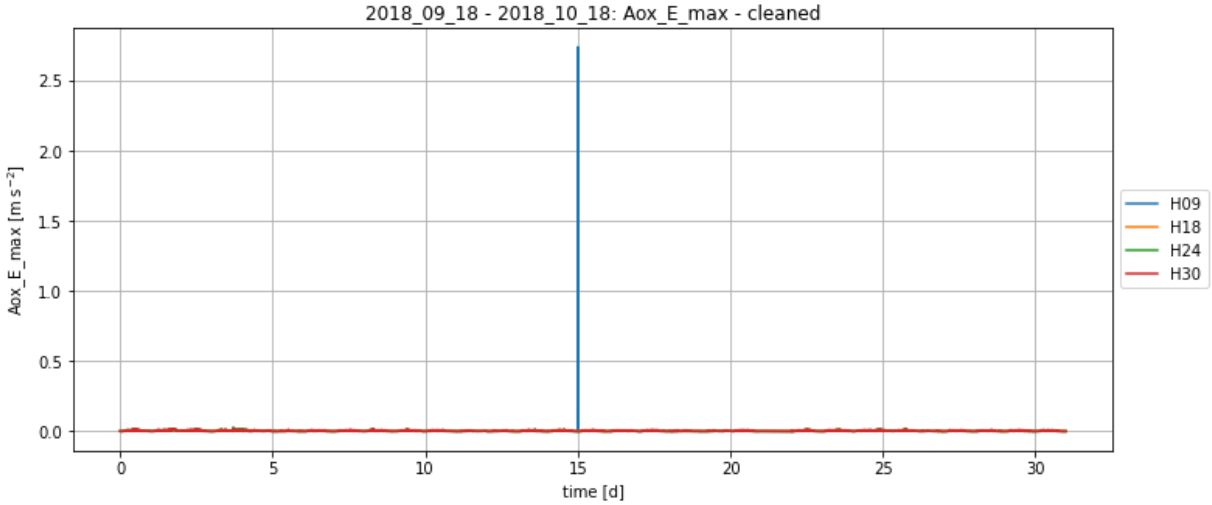
Accelerometers

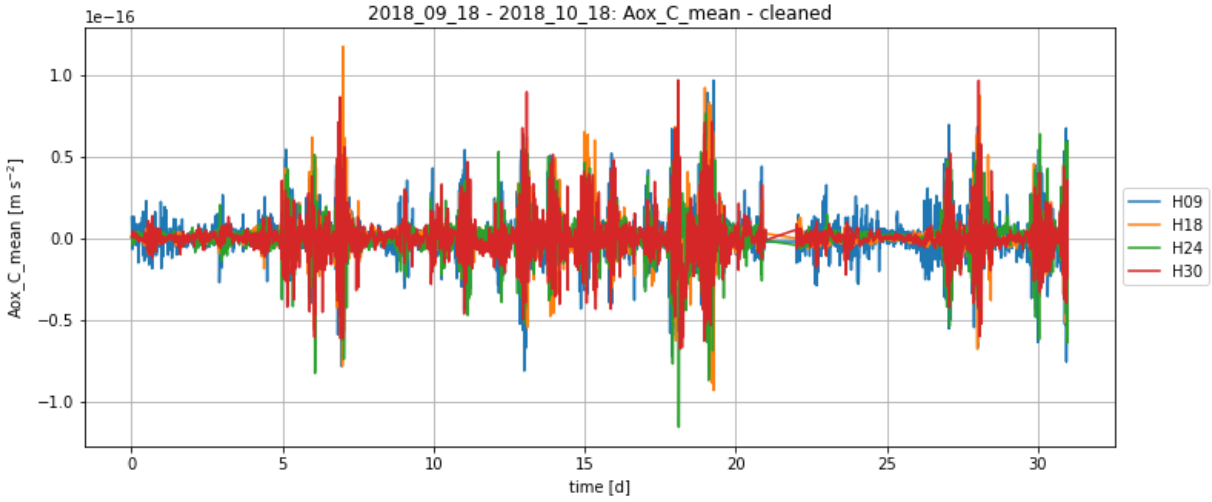
Similarly the **accelerometer** data can be plotted.

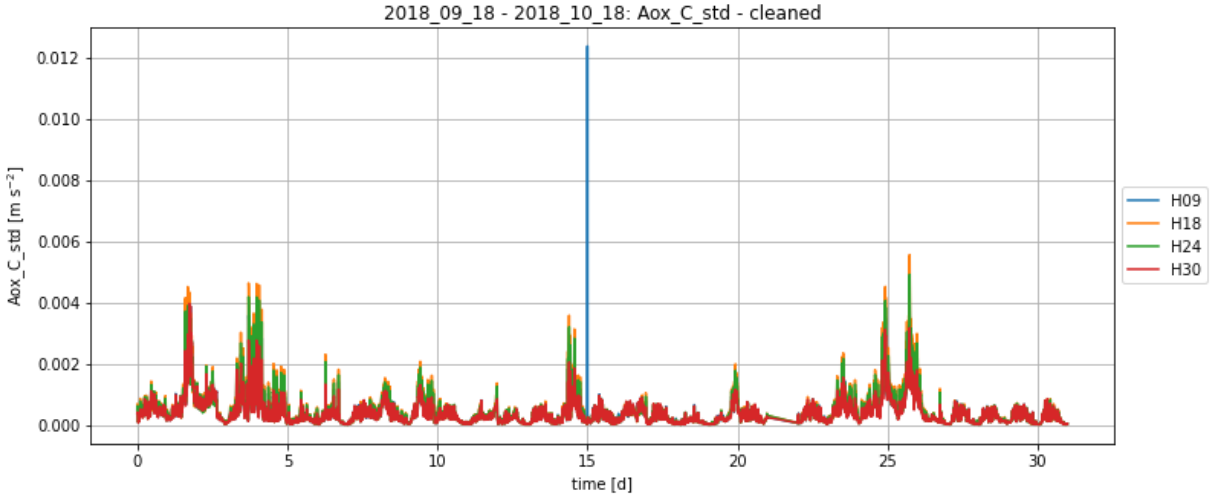
In [ ]:

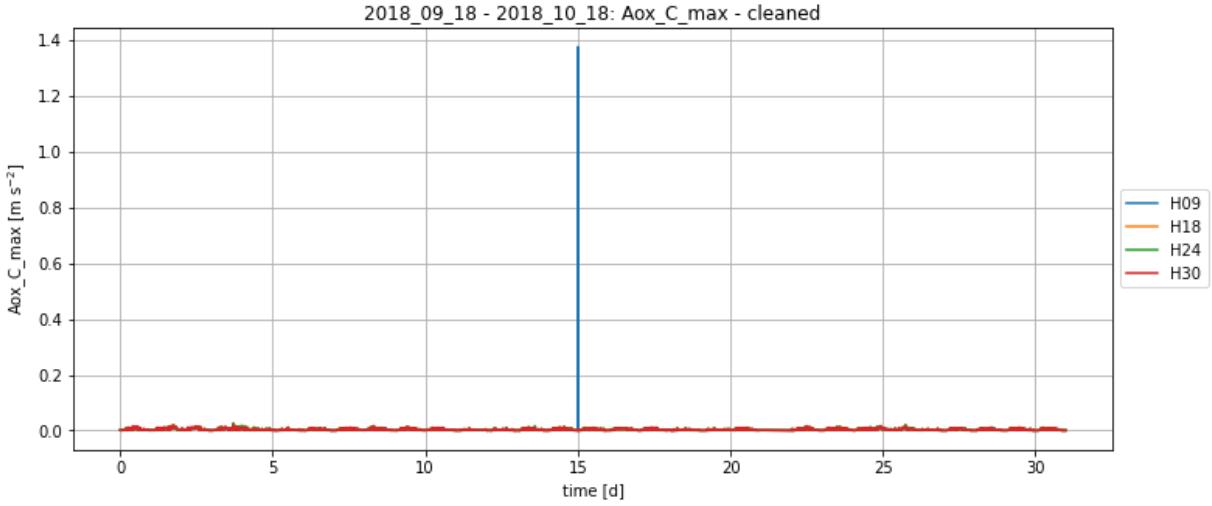
```
ignored_acc=[]
for key in LFB.acc_cleaned.keys():
    if key in keys_of_interest:
        LFB.plot_data(plot_time[filter_idxs],LFB.acc_cleaned[key].T[filter_idxs].T,y
```

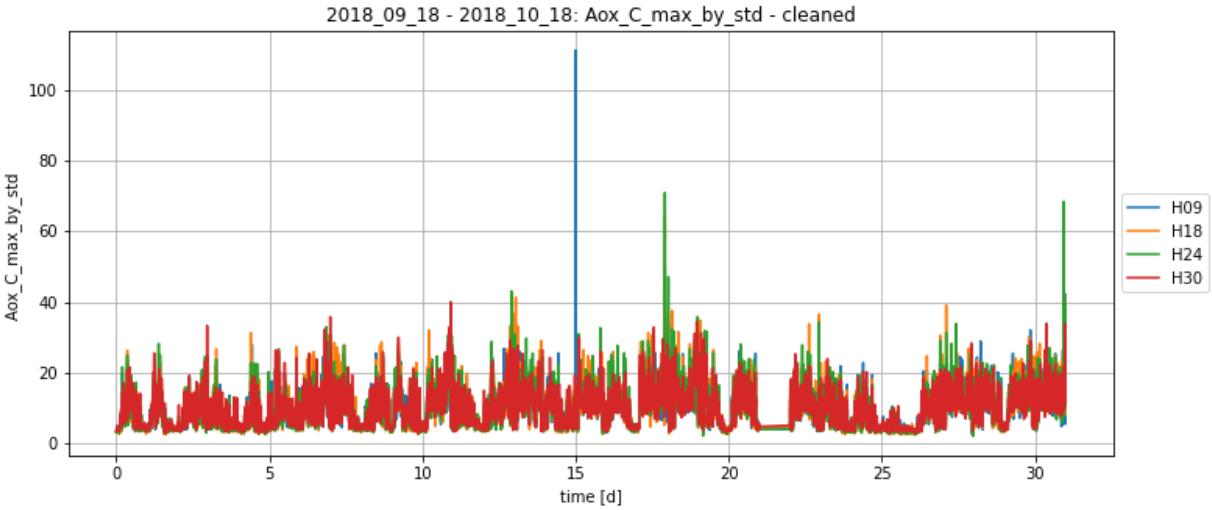

2018_09_18 - 2018_10_18: Aox_W_max - cleaned
Aox_W_max [m s⁻²]
time [d]
H09
H18
H24
H30

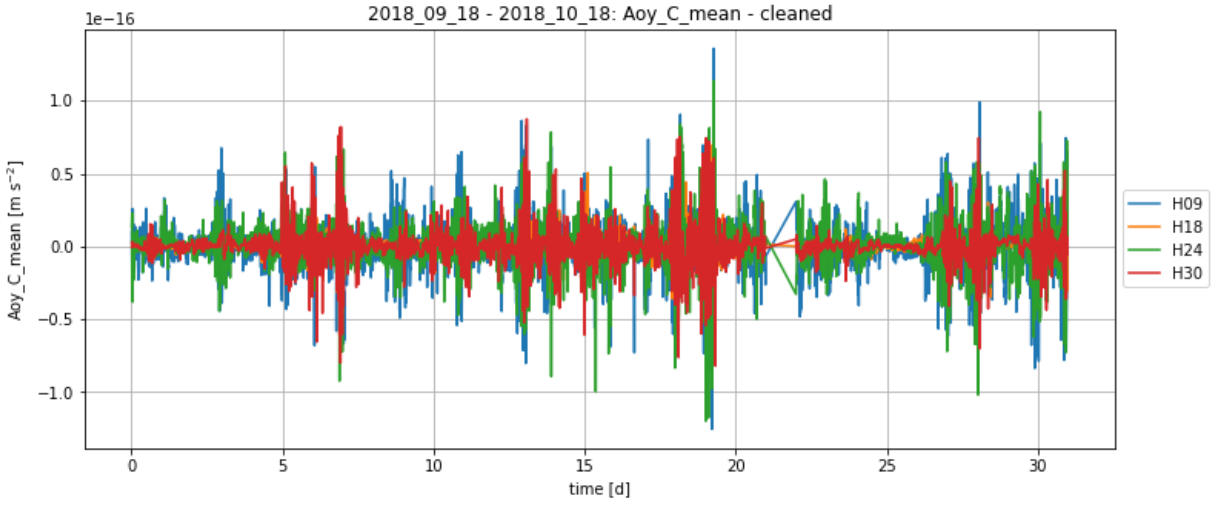

2018_09_18 - 2018_10_18: Aox_E_max - cleaned
Aox_E_max [m s⁻²]
time [d]
H09
H18
H24
H30

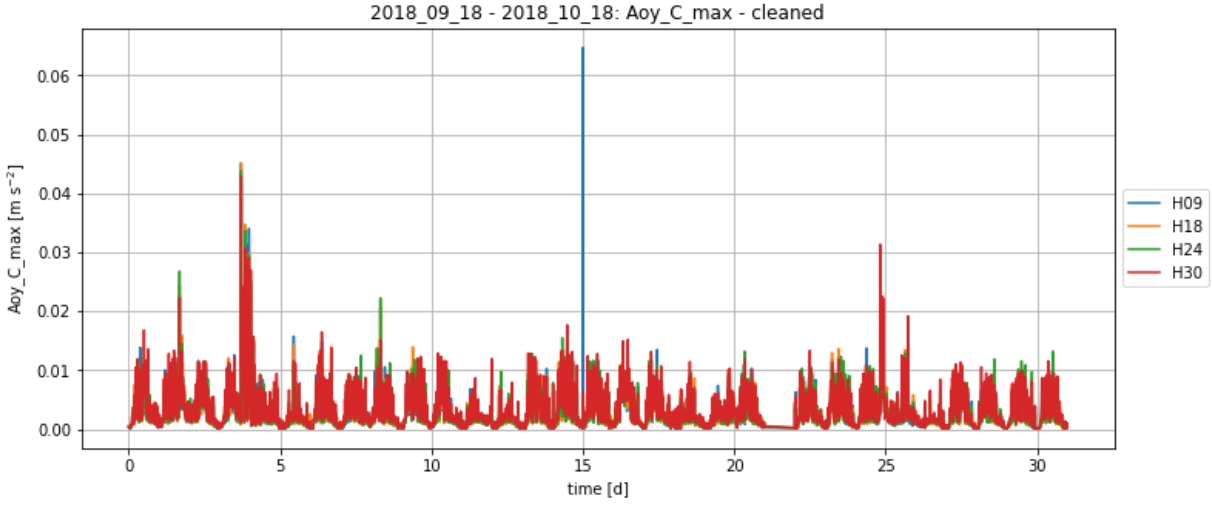

2018_09_18 - 2018_10_18: Aox_C_mean - cleaned
1e−16
Aox_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

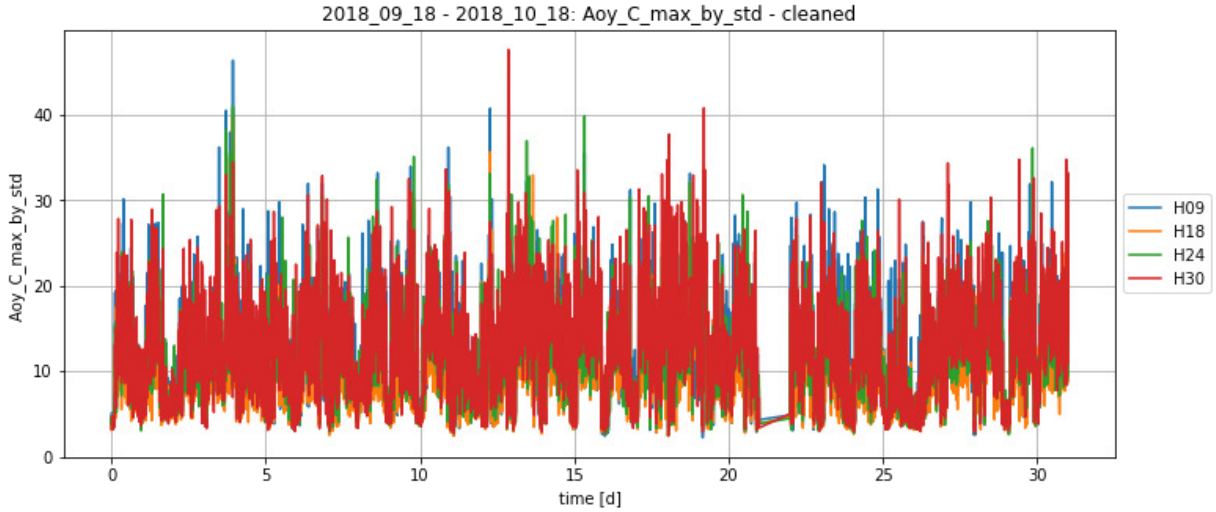

2018_09_18 - 2018_10_18: Aox_C_std - cleaned
Aox_C_std [m s⁻²]
time [d]
H09
H18
H24
H30

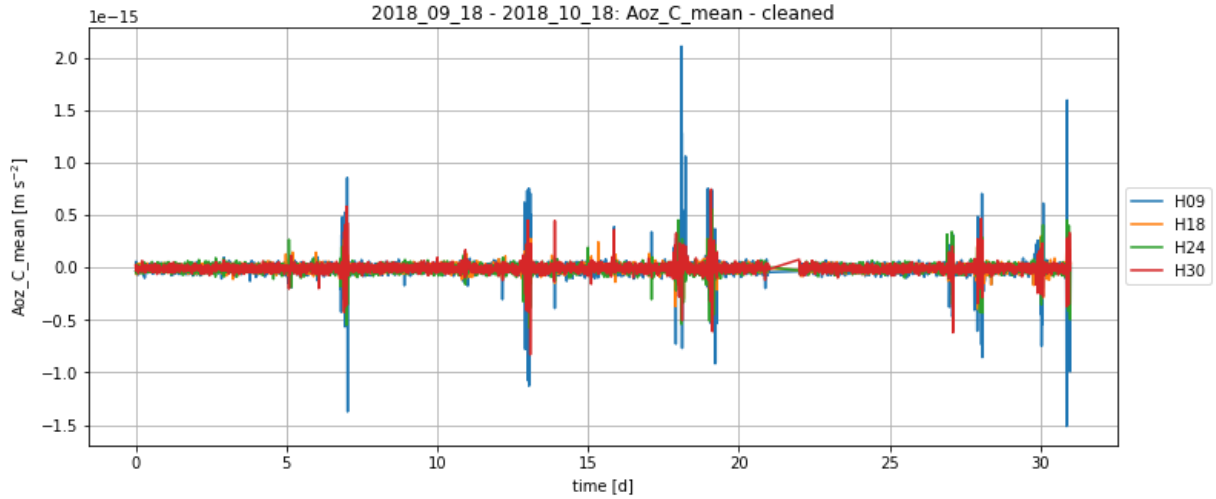

2018_09_18 - 2018_10_18: Aox_C_max - cleaned
Aox_C_max [m s⁻²]
time [d]
H09
H18
H24
H30

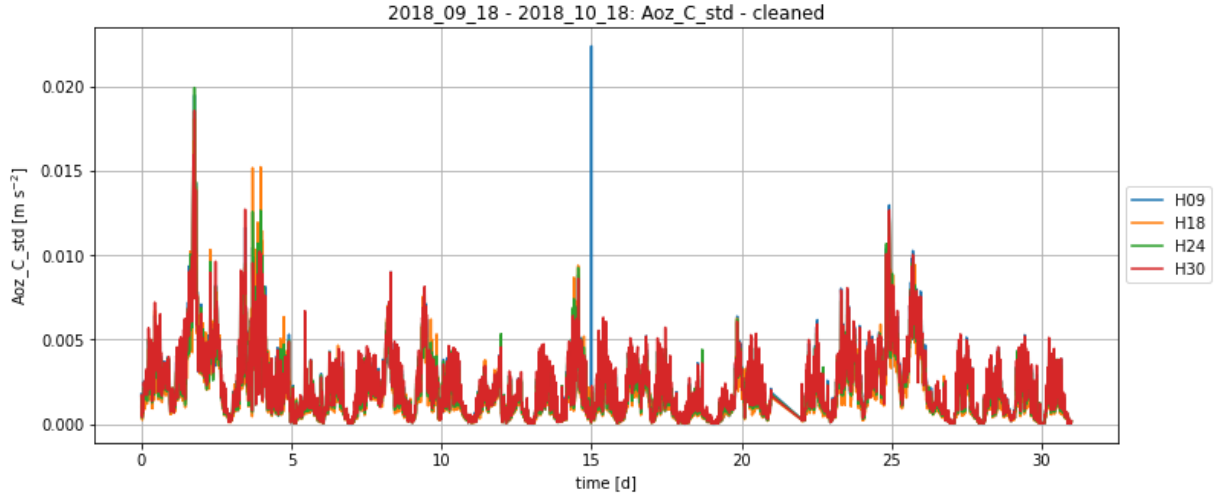

2018_09_18 - 2018_10_18: Aox_C_max_by_std - cleaned
Aox_C_max_by_std
time [d]
H09
H18
H24
H30

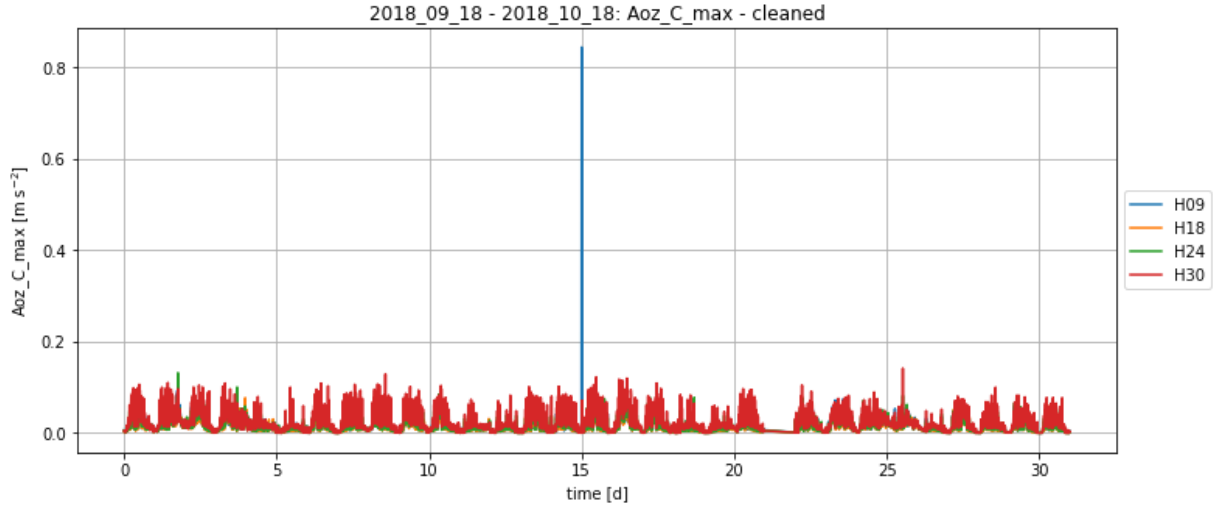

2018_09_18 - 2018_10_18: Aoy_C_mean - cleaned
1e−16
Aoy_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

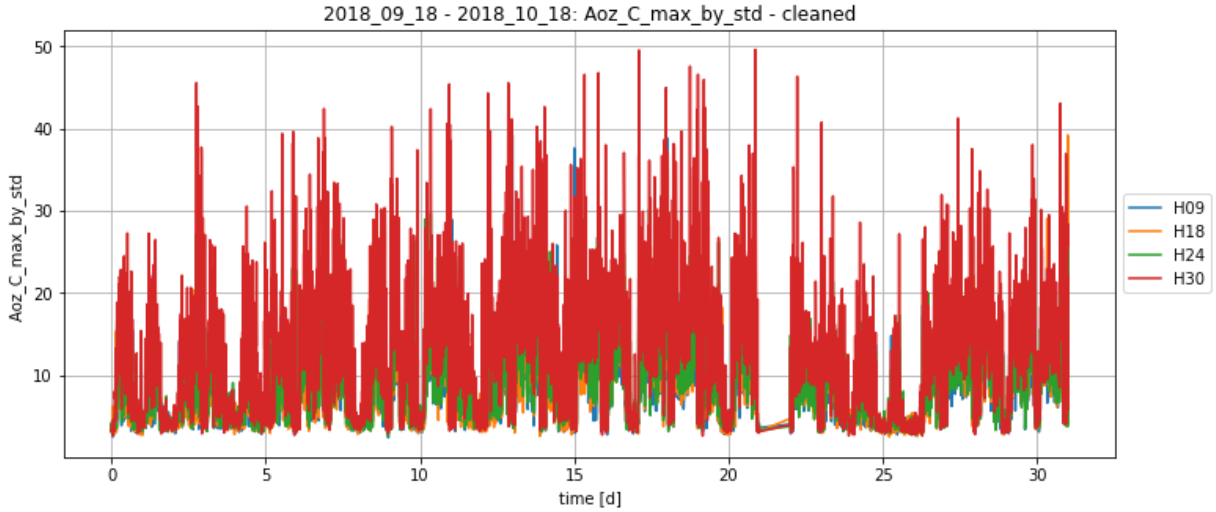

2018_09_18 - 2018_10_18: Aoy_C_max - cleaned
Aoy_C_max [m s⁻²]
time [d]
H09
H18
H24
H30

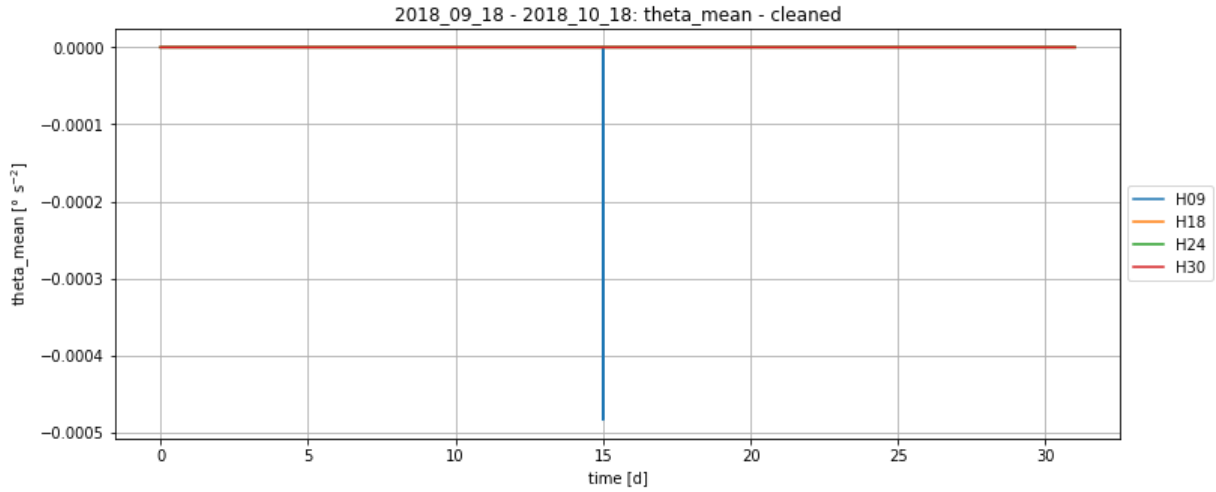

2018_09_18 - 2018_10_18: Aoy_C_max_by_std - cleaned
Aoy_C_max_by_std
time [d]
H09
H18
H24
H30

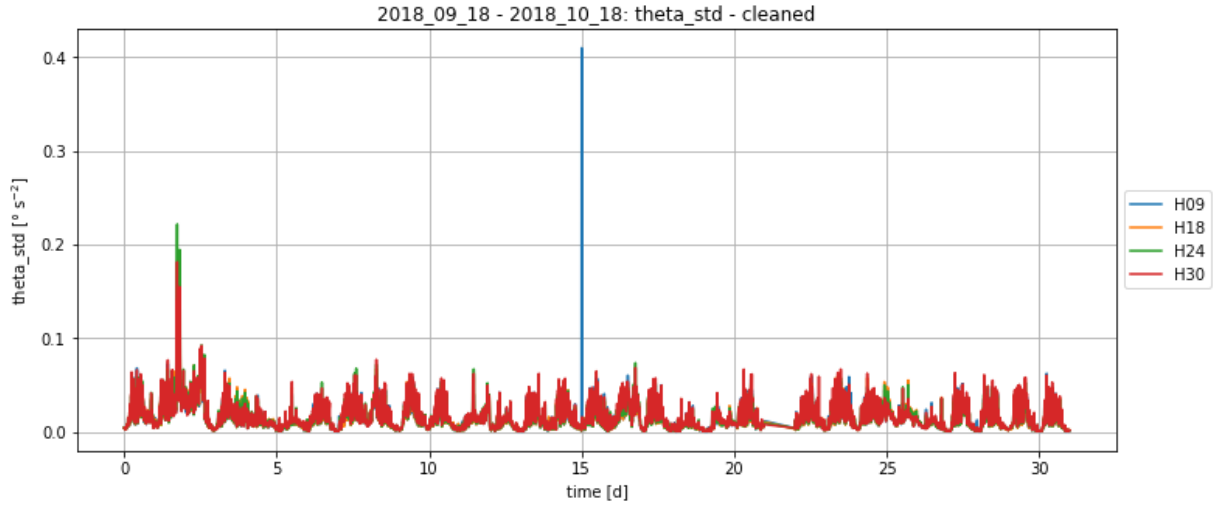

2018_09_18 - 2018_10_18: Aoz_C_mean - cleaned
1e−15
Aoz_C_mean [m s⁻²]
time [d]
H09
H18
H24
H30

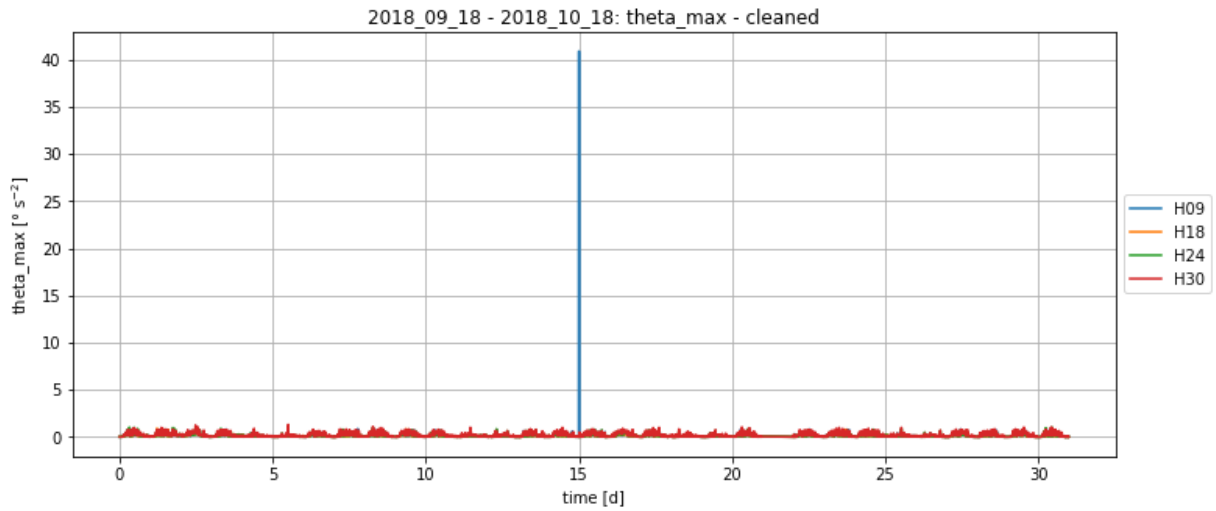

2018_09_18 - 2018_10_18: Aoz_C_std - cleaned
Aoz_C_std [m s⁻²]
time [d]
H09
H18
H24
H30

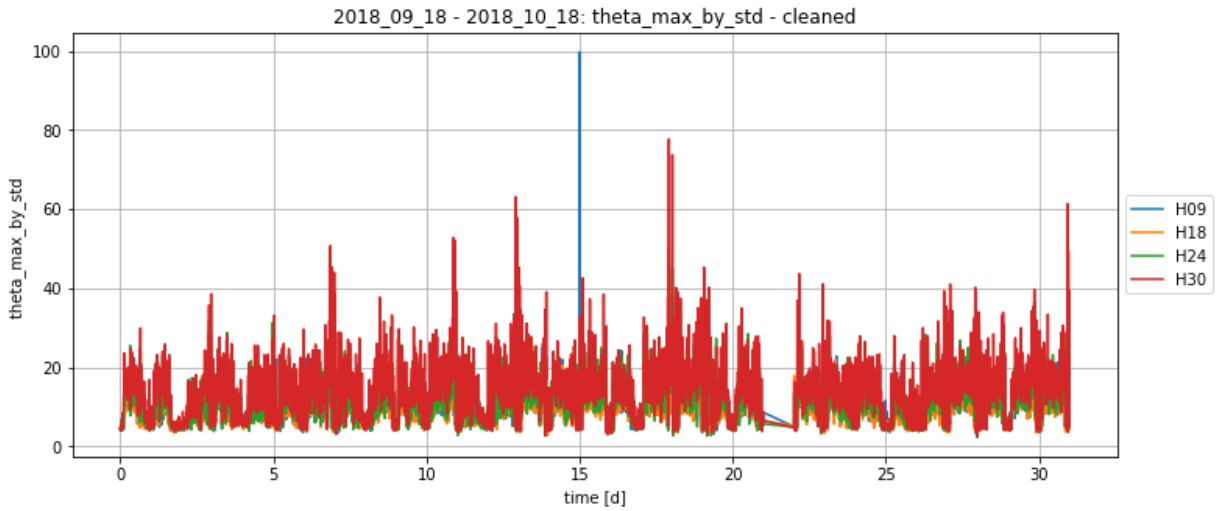

2018_09_18 - 2018_10_18: Aoz_C_max - cleaned
Aoz_C_max [m s⁻²]
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: Aoz_C_max_by_std - cleaned
Aoz_C_max_by_std
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: theta_mean - cleaned
theta_mean [° s⁻²]
time [d]
H09
H18
H24
H30


2018_09_18 - 2018_10_18: theta_std - cleaned
theta_std [° s⁻²]
time [d]
H09
H18
H24
H30

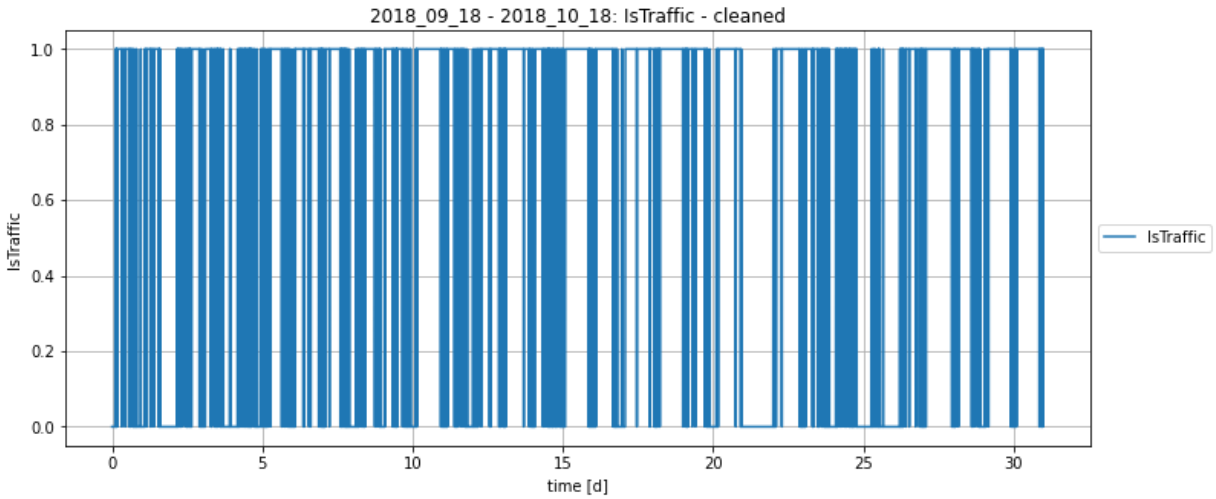

2018_09_18 - 2018_10_18: theta_max - cleaned
theta_max [° s⁻²]
time [d]
H09
H18
H24
H30



Traffic

Below is a visualisation of the traffic classification previously created using the `find_traffic` method. Note that this method is not available in `full_detail` mode.

In [ ]:
```
if LFB.full_detail==False:
    LFB.plot_data(plot_time,LFB.traffic_cleaned,ylabel='IsTraffic',yunit='',xlabel=p
```



Histograms

Below are some examples on creating histograms using the `hist` method. Note that, as in the time series plots above, filters can be applied using the `filter_data` method and `keys_of_interest` can be set.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
```

In [ ]:
```
keys_of_interest = ['H_mean','H_max','W_mean','Vx_mean','Vy_mean','Dir_mean','H_turb
```

Anemometers

Below is an example of histograms on **anemometer** data. The structure of the batch-processing is similar to the time-series plots above.

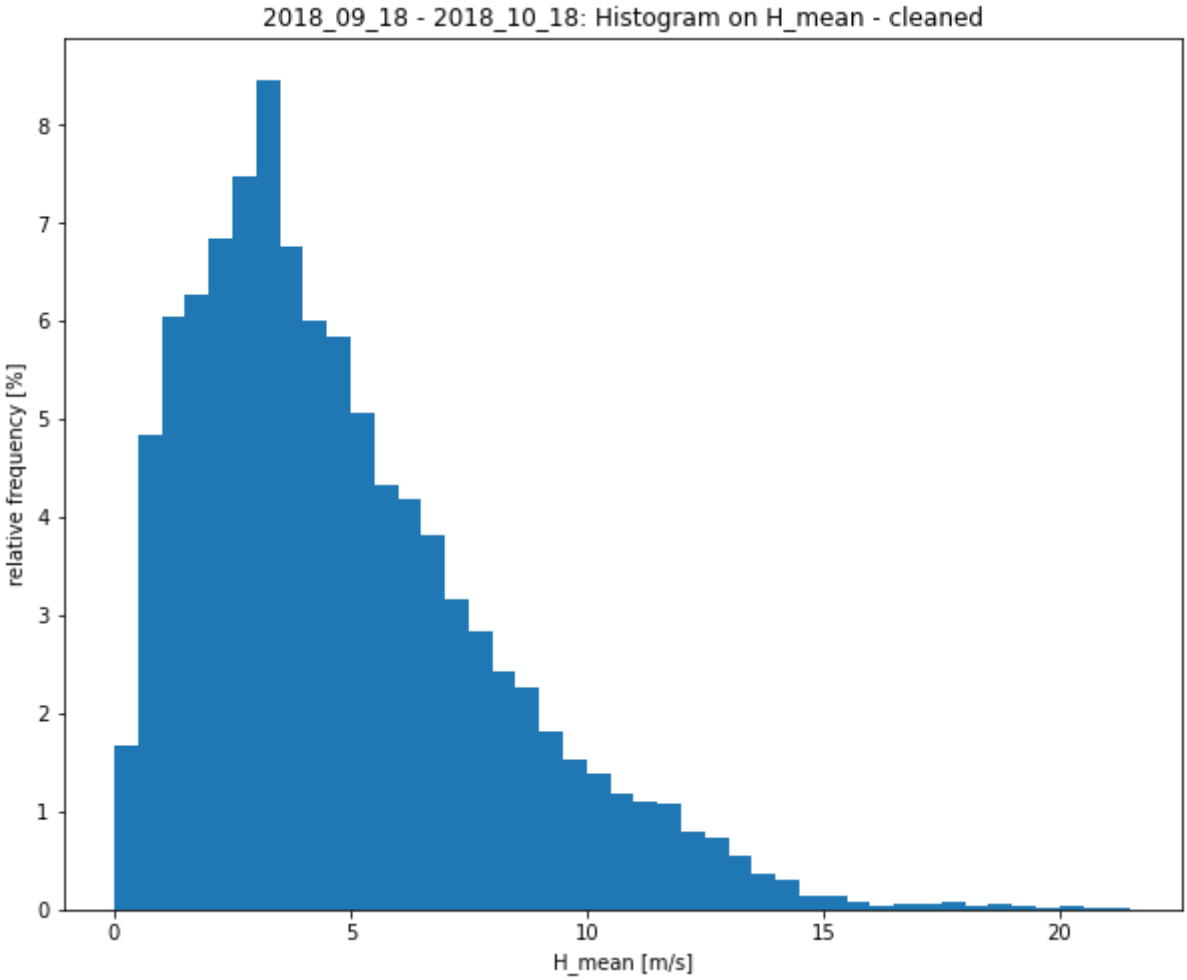
Note how different `bins` are setup for the histograms of different types of measurements (*keys*) for the `hist` method.

Another usefull keyword is the `split_sensors` keyword, which when set to *True* creates a histogram for each *sensor* or combines them into a single histogram when set to *False*.

It is also possible to perform any kind of statistical analysis on the data using standard libraries like *numpy* or *scipy* or other 3rd-party libraries.

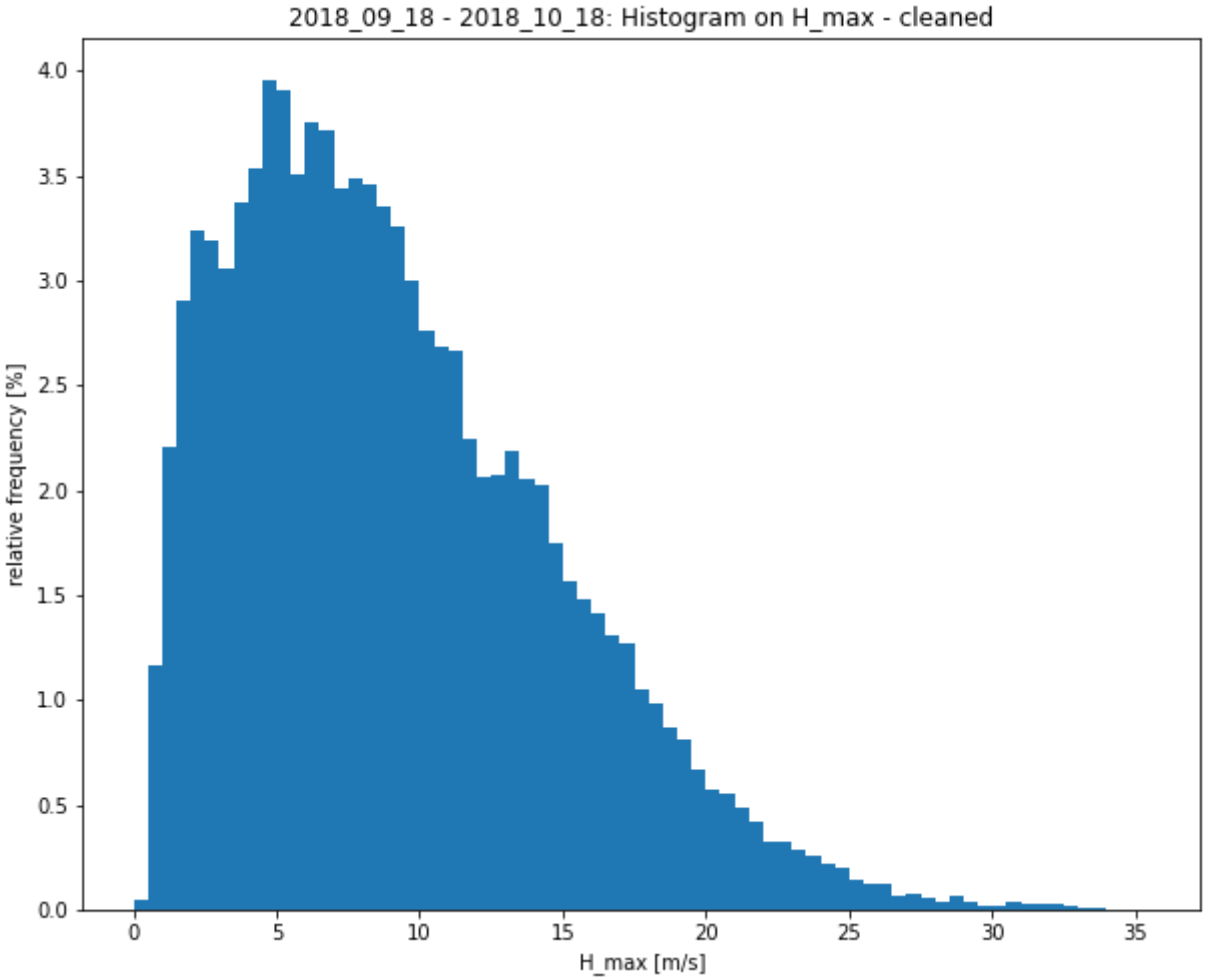
In [ ]:

```
for key in LFB.anemo_cleaned.keys():
    if key in keys_of_interest:
        if key.startswith('Dir_m'): # Adjust the bin width to 5° for directional dat
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('H_m'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.endswith('_turb'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('AOA_m'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        else:
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        print('Statistics for',key,LFB.units[key],title_suffix)
        print('-----------------------------------------------')
        print('Mean:',np.round(np.mean(LFB.anemo_cleaned[key].T[filter_idxs].T),2))
        print('Median:',np.round(np.median(LFB.anemo_cleaned[key].T[filter_idxs].T),
        print('Percentile [10, 25, 75, 90]:',np.round(np.percentile(LFB.anemo_cleane
        print('Minimum:',np.round(np.min(LFB.anemo_cleaned[key].T[filter_idxs].T),2)
        print('Maximum:',np.round(np.max(LFB.anemo_cleaned[key].T[filter_idxs].T),2)
        print('Variance:',np.round(np.var(LFB.anemo_cleaned[key].T[filter_idxs].T),2
        print('Standard deviation:',np.round(np.std(LFB.anemo_cleaned[key].T[filter_
        print('Skewness:',np.round(scipy.stats.skew(LFB.anemo_cleaned[key].T[filter_
        print('Kurtosis:',np.round(scipy.stats.kurtosis(LFB.anemo_cleaned[key].T[fil
```

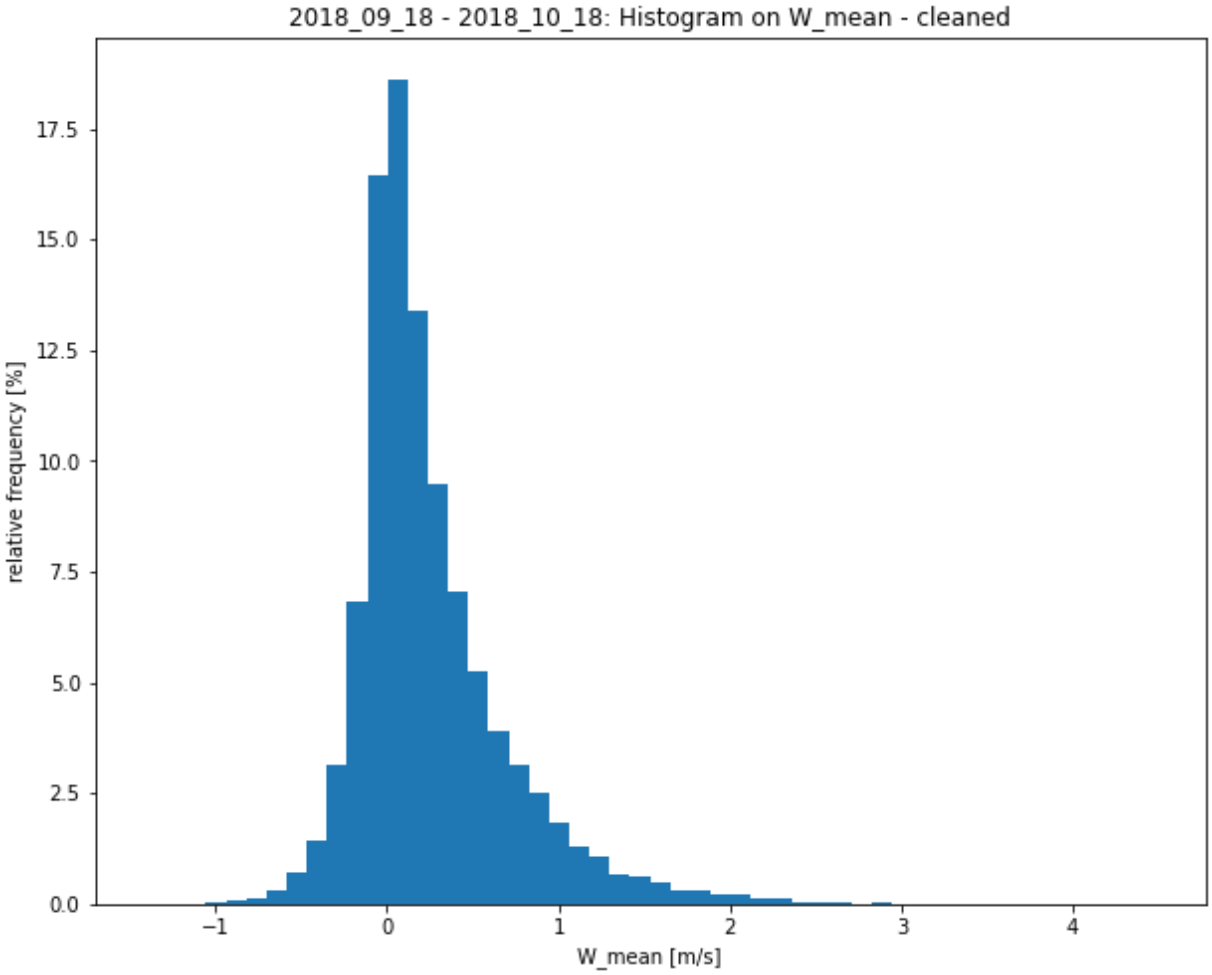


Statistics for H_mean [m/s] - cleaned

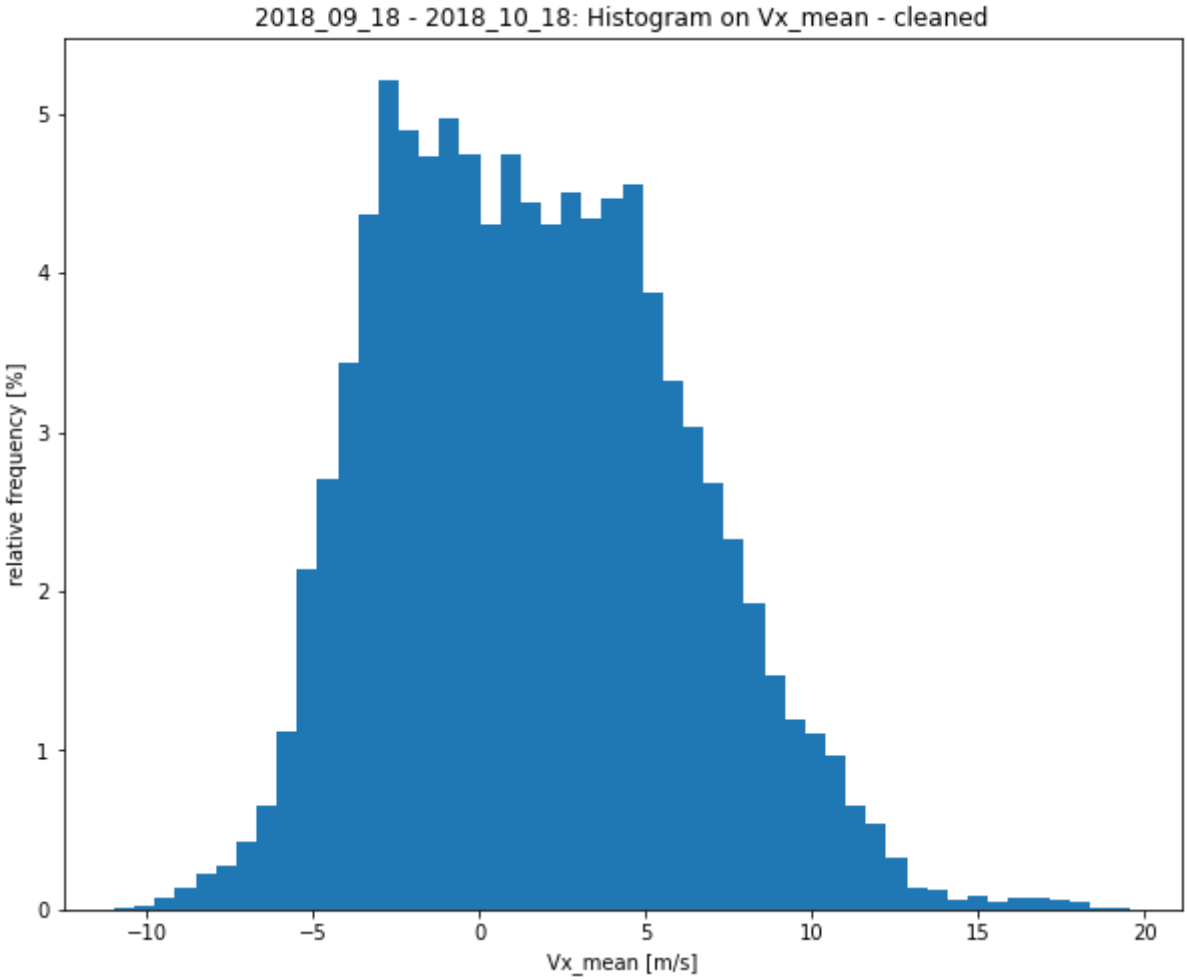
Mean: 4.86
Median: 4.13
Percentile [10, 25, 75, 90]: [1.29 2.45 6.64 9.46]
Minimum: 0.12
Maximum: 21.48
Variance: 10.52
Standard deviation: 3.24
Skewness: 1.07
Kurtosis: 1.19



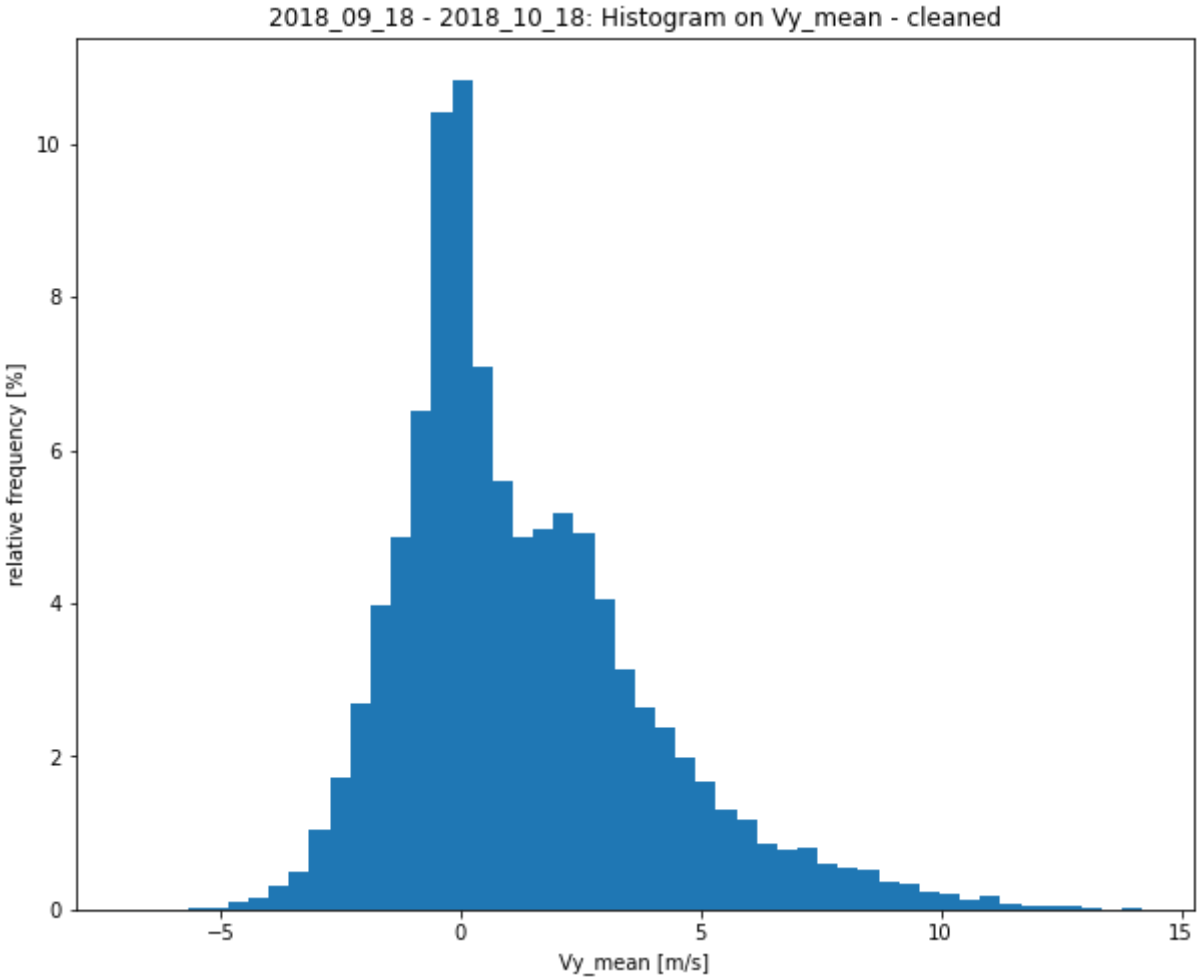

```
Statistics for H_max [m/s]  - cleaned
------------------------------------------------
Mean: 9.17
Median: 8.2
Percentile [10, 25, 75, 90]: [ 2.55  4.78 12.74 17.07]
Minimum: 0.37
Maximum: 35.03
Variance: 31.59
Standard deviation: 5.62
Skewness: 0.79
Kurtosis: 0.33
```



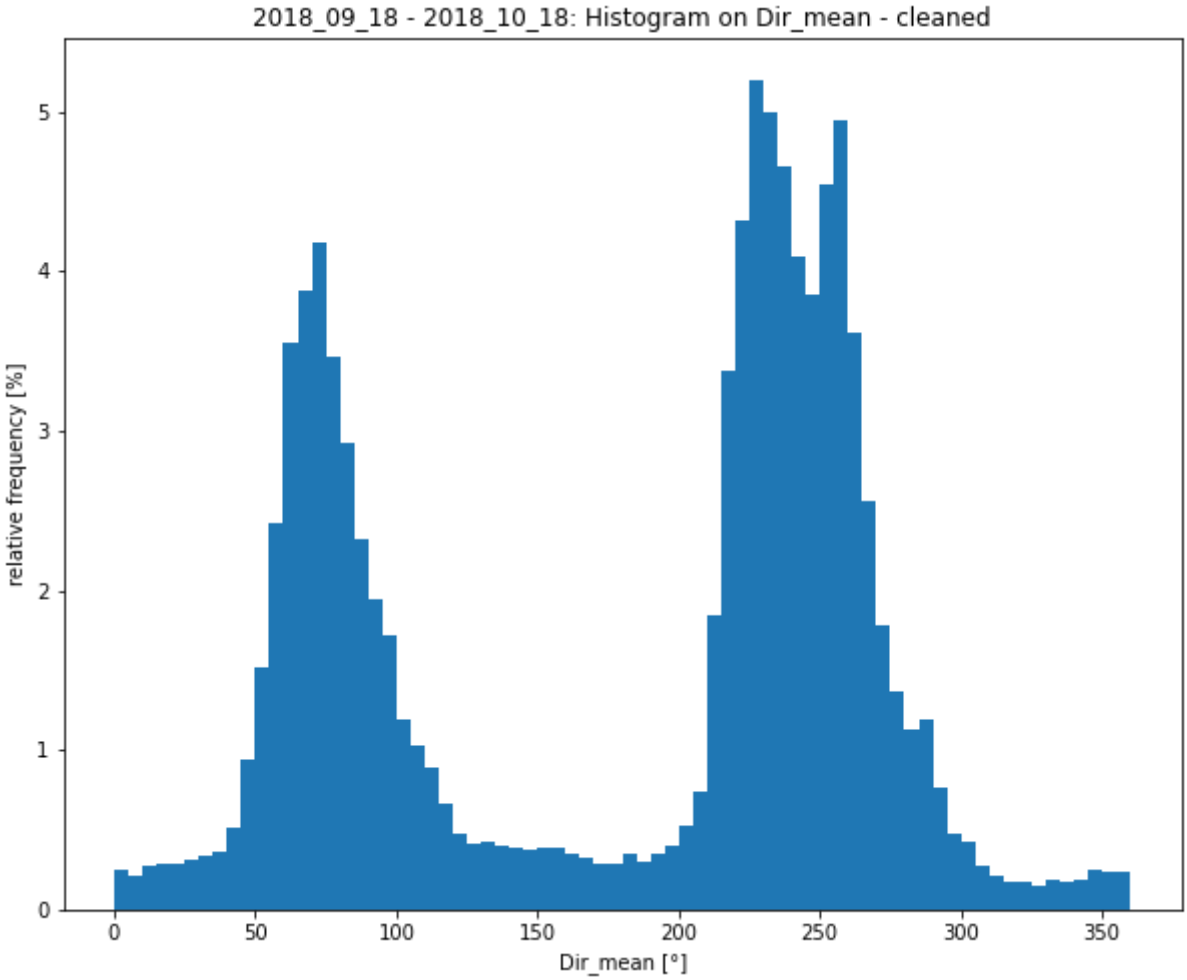

```
Statistics for W_mean [m/s]  - cleaned
------------------------------------------------
Mean: 0.25
Median: 0.14
Percentile [10, 25, 75, 90]: [-0.15 -0.02  0.43  0.83]
Minimum: -1.41
Maximum: 4.48
Variance: 0.2
Standard deviation: 0.44
Skewness: 1.59
Kurtosis: 4.23
```




```
Statistics for Vx_mean [m/s]  - cleaned
------------------------------------------------
Mean: 1.66
Median: 1.35
Percentile [10, 25, 75, 90]: [-3.84 -1.89  4.79  7.68]
Minimum: -10.97
Maximum: 19.58
Variance: 20.14
Standard deviation: 4.49
Skewness: 0.35
Kurtosis: -0.25
```

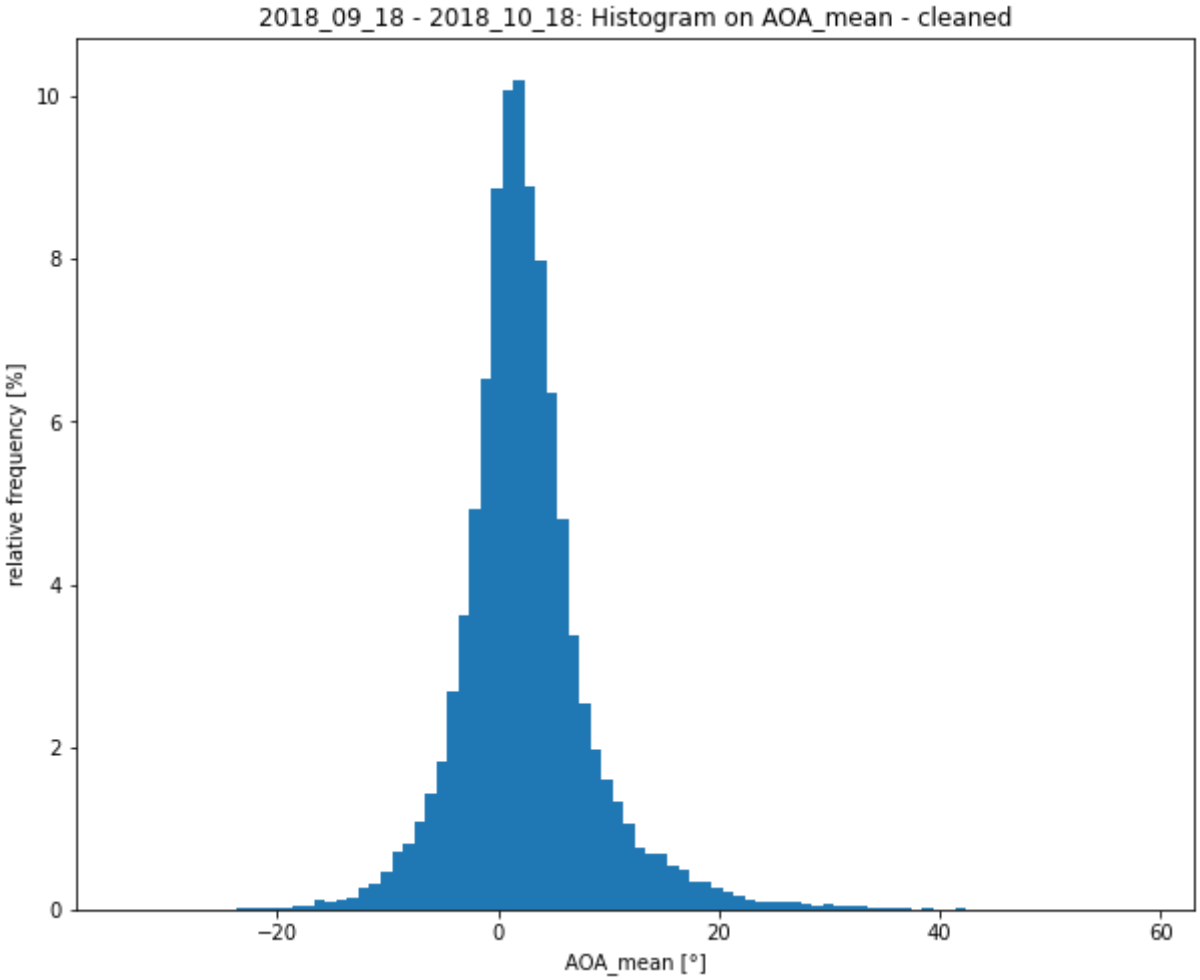



```
Statistics for Vy_mean [m/s]  - cleaned
------------------------------------------------
Mean: 1.3
Median: 0.65
Percentile [10, 25, 75, 90]: [-1.5  -0.47  2.71  4.9 ]
Minimum: -6.95
Maximum: 14.21
Variance: 7.09
Standard deviation: 2.66
Skewness: 1.03
Kurtosis: 1.31
```

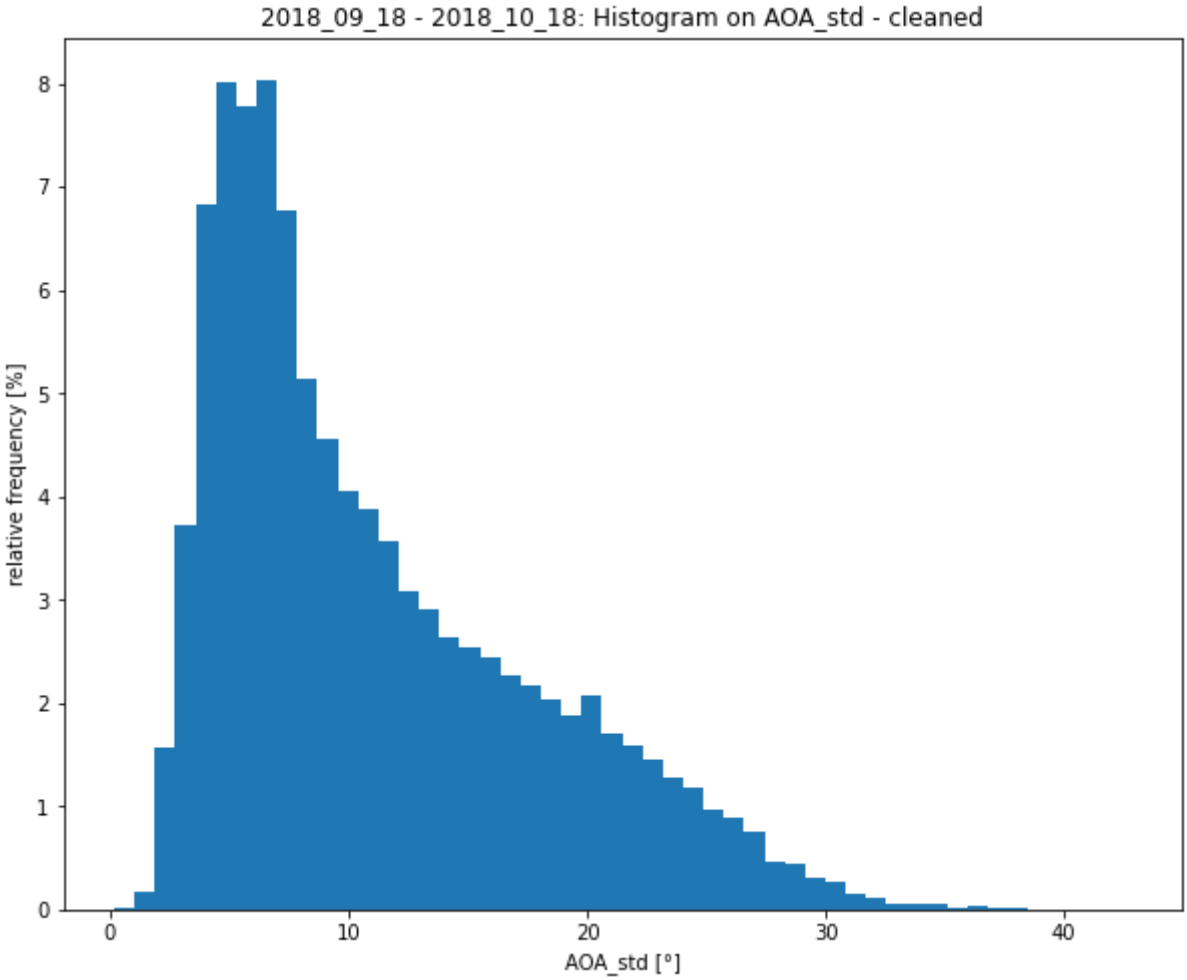


Statistics for Dir_mean [°] - cleaned

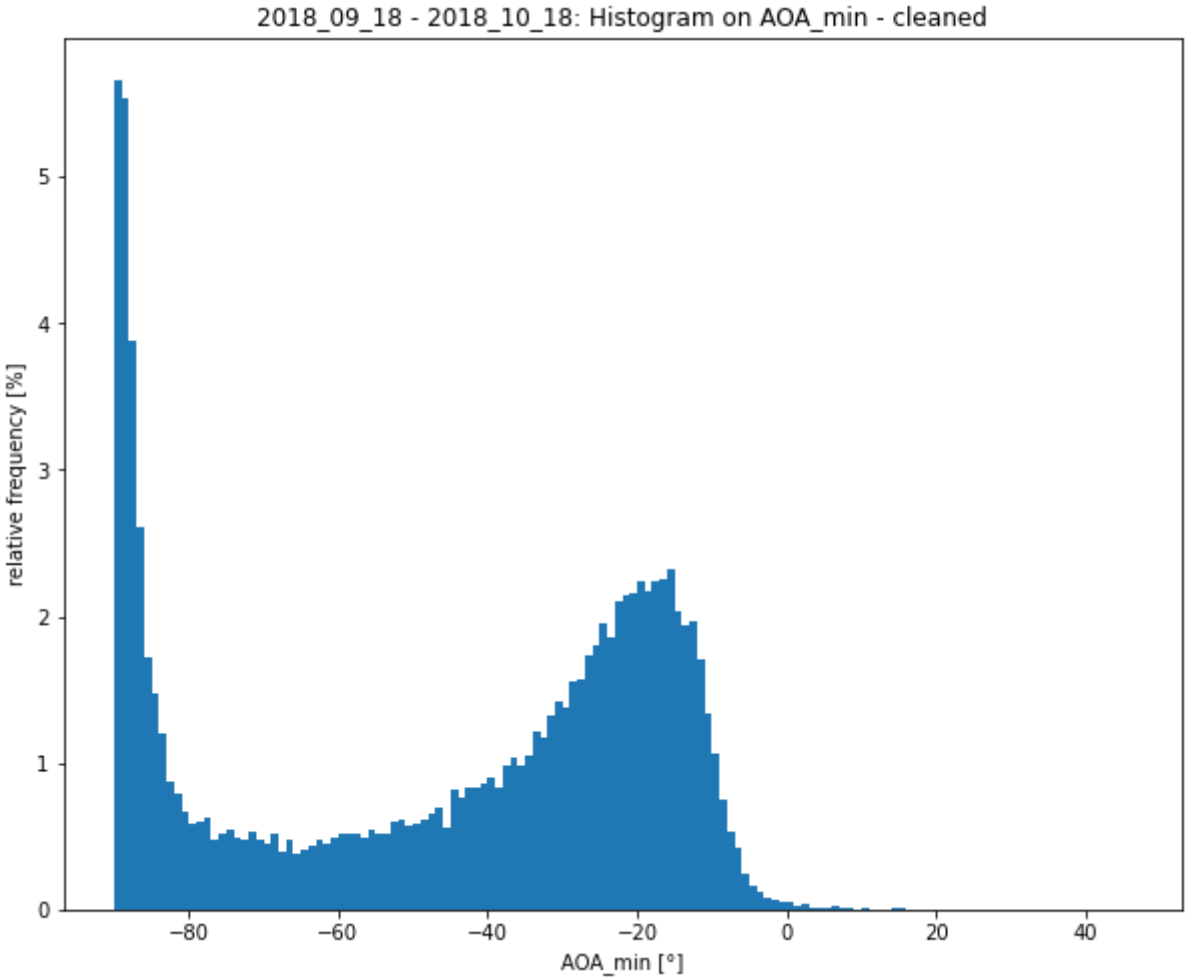
Mean: 180.89
Median: 222.57
Percentile [10, 25, 75, 90]: [63.3   83.71 250.04 268.52]
Minimum: 0.04
Maximum: 359.93
Variance: 7446.38
Standard deviation: 86.29
Skewness: -0.34
Kurtosis: -1.37



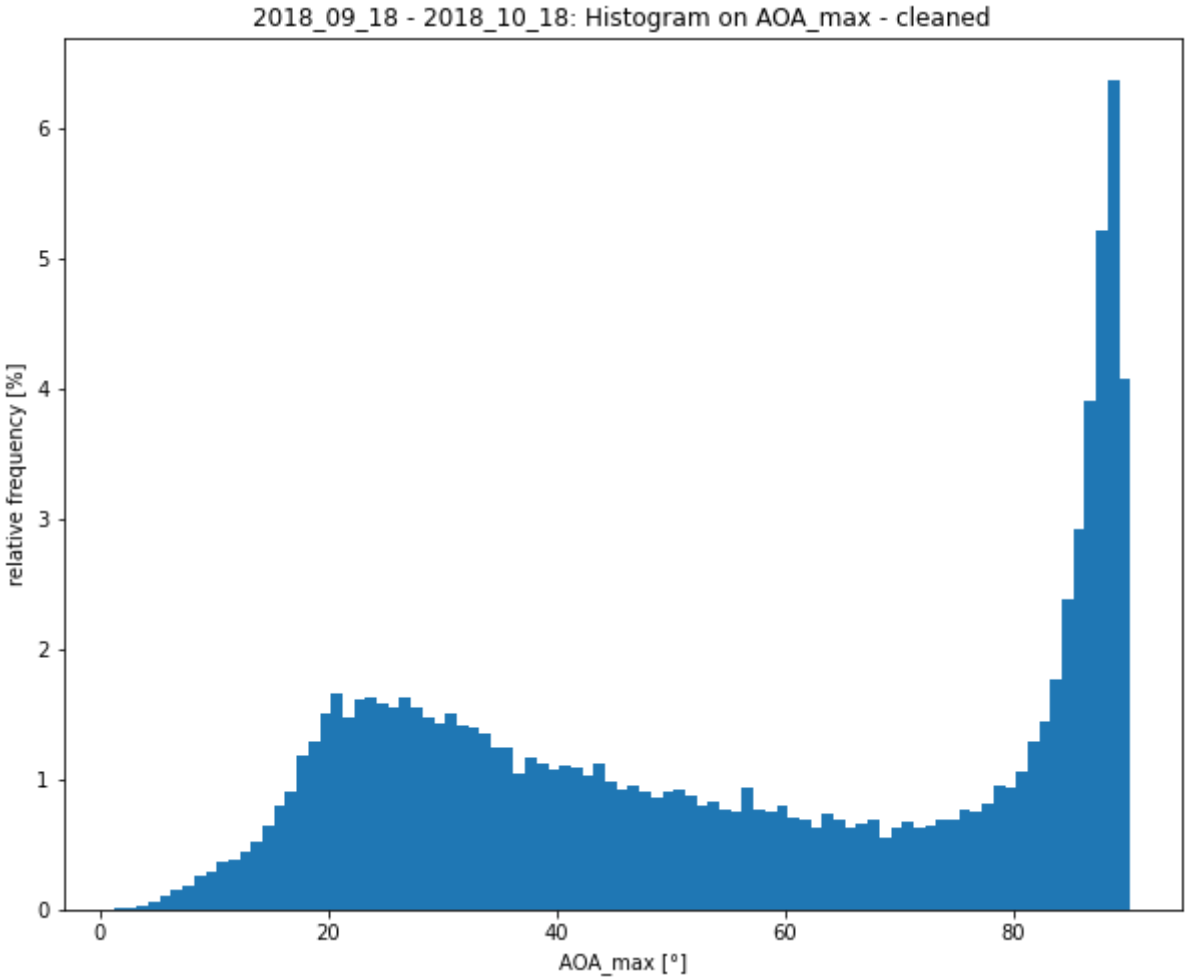

```
Statistics for AOA_mean [°]  - cleaned
------------------------------------------------
Mean: 2.44
Median: 1.91
Percentile [10, 25, 75, 90]: [-3.75 -0.69  4.89  9.06]
Minimum: -33.64
Maximum: 57.53
Variance: 36.94
Standard deviation: 6.08
Skewness: 1.03
Kurtosis: 5.42
```



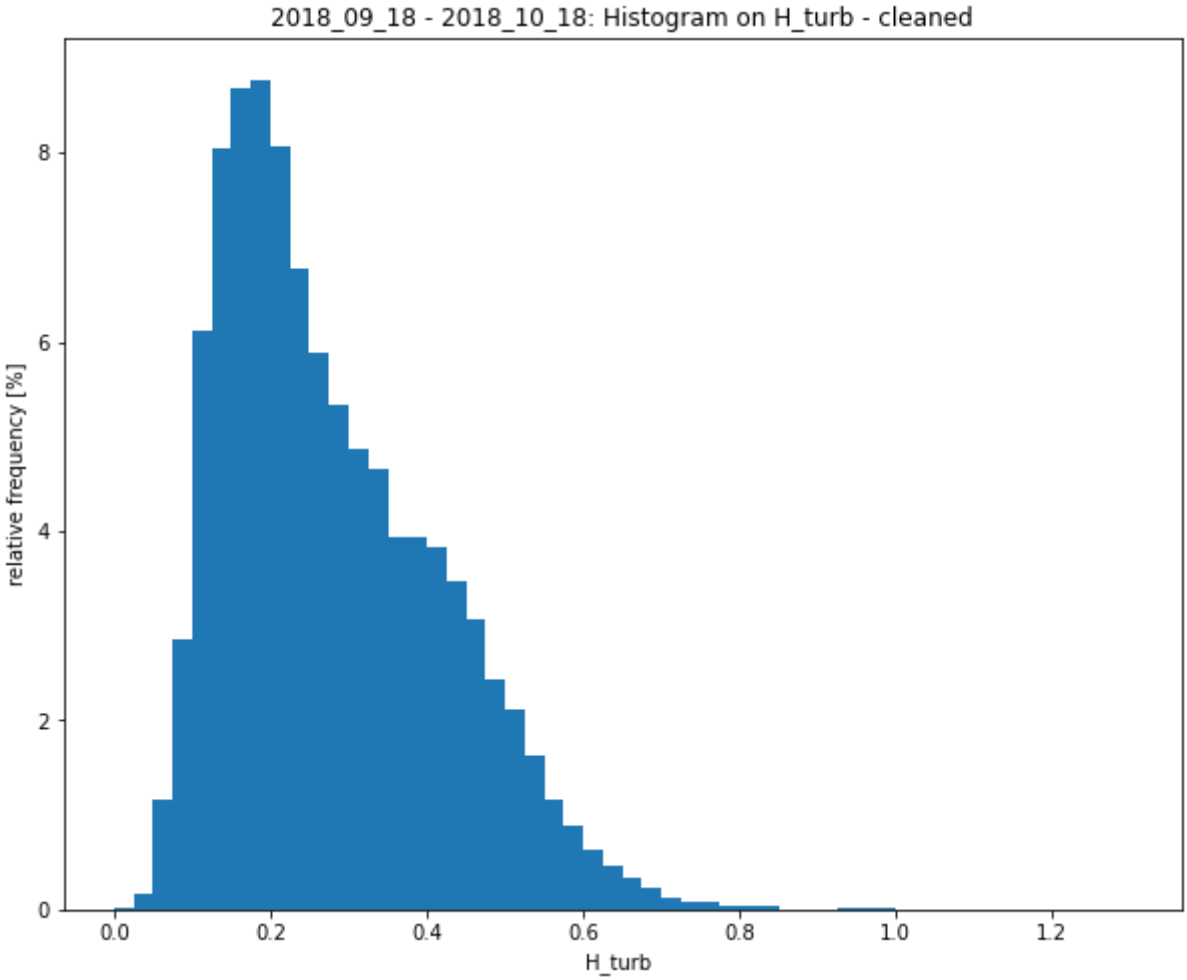

```
Statistics for AOA_std [°]  - cleaned
------------------------------------------------
Mean: 11.13
Median: 9.04
Percentile [10, 25, 75, 90]: [ 4.17  5.78 15.41 21.51]
Minimum: 0.16
Maximum: 42.78
Variance: 45.8
Standard deviation: 6.77
Skewness: 0.94
Kurtosis: 0.14
```




```
Statistics for AOA_min [°]  - cleaned
------------------------------------------------
Mean: -45.97
Median: -36.54
Percentile [10, 25, 75, 90]: [-88.24 -79.03 -20.49 -13.66]
Minimum: -90.0
Maximum: 45.3
Variance: 832.33
Standard deviation: 28.85
Skewness: -0.35
Kurtosis: -1.42
```




```
Statistics for AOA_max [°]  - cleaned
------------------------------------------------
Mean: 55.15
Median: 53.57
Percentile [10, 25, 75, 90]: [20.8  30.44 84.21 88.36]
Minimum: 1.28
Maximum: 90.0
Variance: 695.77
Standard deviation: 26.38
Skewness: -0.05
Kurtosis: -1.5
```



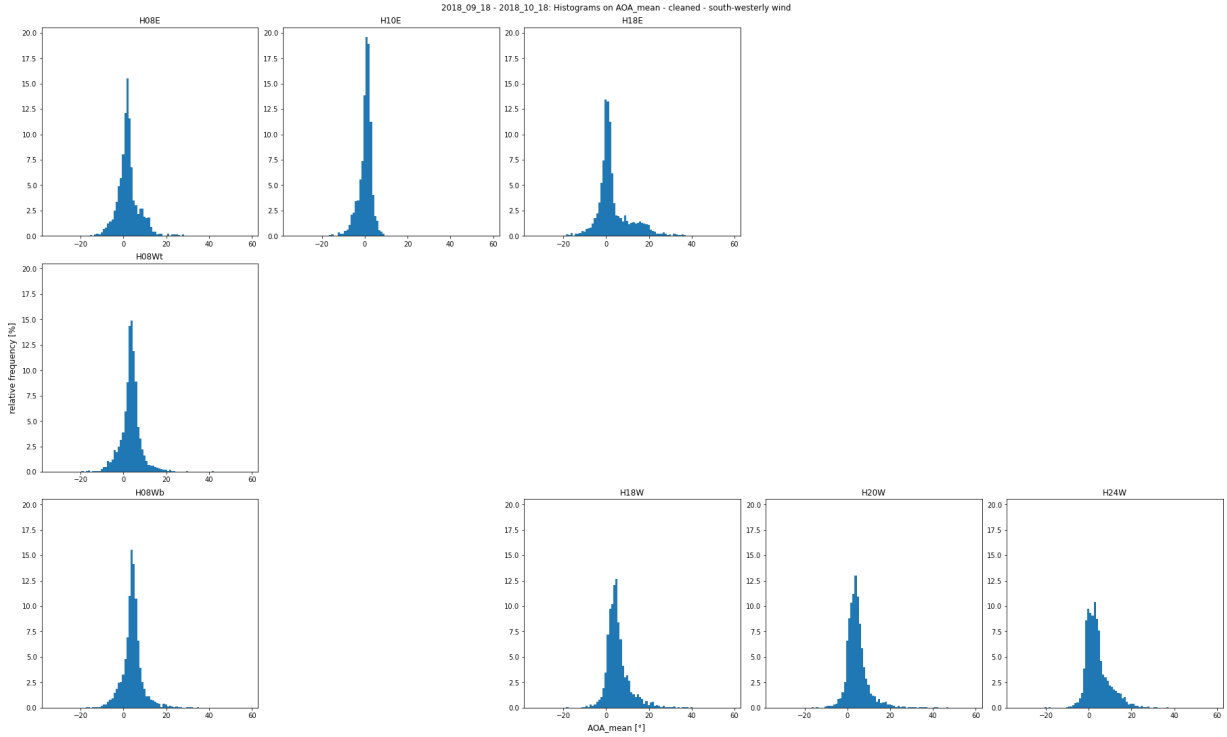

```
Statistics for H_turb   - cleaned
-----------------------------------------------
Mean: 0.28
Median: 0.25
Percentile [10, 25, 75, 90]: [0.12 0.17 0.37 0.48]
Minimum: 0.0
Maximum: 1.29
Variance: 0.02
Standard deviation: 0.14
Skewness: 0.82
Kurtosis: 0.4
```

The code below showcases how direct statistical comparisons between anemometers can be implemented. In this example the mean angle of attack and turbulence intensity are compared between anemometers on the upwind- and downwind-side of the bridge for south-westerly and north-easterly winds.

In [ ]:

```
keys_of_interest = ['AOA_mean','H_turb']
sensor_names_of_interest = ['H08Wb','H08E', 'H18W','H18E']

#south-westerly wind
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs,h
for key in LFB.anemo_cleaned.keys():
    if key in keys_of_interest:
        if key.startswith('Dir_m'): # Adjust the bin width to 5° for directional dat
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('H_m'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.endswith('_turb'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('AOA_m'): # Adjust the bins for consistency.
```

```
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        else:
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        for sensor_name in sensor_names_of_interest:
            if sensor_name in LFB.anemo_names[LFB.anemo_ok_sensor_id[key]]:
                sensor_id = np.where(LFB.anemo_names[LFB.anemo_ok_sensor_id[key]]==s
                print('Statistics for',sensor_name,key,LFB.units[key],title_suffix)
                print('-----------------------------------------------')
                print('Mean:',np.round(np.mean(LFB.anemo_cleaned[key].T[filter_idxs]
                print('Median:',np.round(np.median(LFB.anemo_cleaned[key].T[filter_i
                print('Percentile [10, 25, 75, 90]:',np.round(np.percentile(LFB.anem
                print('Minimum:',np.round(np.min(LFB.anemo_cleaned[key].T[filter_idx
                print('Maximum:',np.round(np.max(LFB.anemo_cleaned[key].T[filter_idx
                print('Variance:',np.round(np.var(LFB.anemo_cleaned[key].T[filter_id
                print('Standard deviation:',np.round(np.std(LFB.anemo_cleaned[key].T
                print('\n')
            else:
                raise KeyError(str(sensor_name)+' not in ok anemometers. Choose a va

#north-easterly wind
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs,l
for key in LFB.anemo_cleaned.keys():
    if key in keys_of_interest:
        if key.startswith('Dir_m'): # Adjust the bin width to 5° for directional dat
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('H_m'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.endswith('_turb'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        elif key.startswith('AOA_m'): # Adjust the bins for consistency.
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        else:
            LFB.hist(LFB.anemo_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.un
        for sensor_name in sensor_names_of_interest:
            if sensor_name in LFB.anemo_names[LFB.anemo_ok_sensor_id[key]]:
                sensor_id = np.where(LFB.anemo_names[LFB.anemo_ok_sensor_id[key]]==s
                print('Statistics for',sensor_name,key,LFB.units[key],title_suffix)
                print('-----------------------------------------------')
                print('Mean:',np.round(np.mean(LFB.anemo_cleaned[key].T[filter_idxs]
                print('Median:',np.round(np.median(LFB.anemo_cleaned[key].T[filter_i
                print('Percentile [10, 25, 75, 90]:',np.round(np.percentile(LFB.anem
                print('Minimum:',np.round(np.min(LFB.anemo_cleaned[key].T[filter_idx
                print('Maximum:',np.round(np.max(LFB.anemo_cleaned[key].T[filter_idx
                print('Variance:',np.round(np.var(LFB.anemo_cleaned[key].T[filter_id
                print('Standard deviation:',np.round(np.std(LFB.anemo_cleaned[key].T
                print('\n')
            else:
                raise KeyError(str(sensor_name)+' not in ok anemometers. Choose a va
```



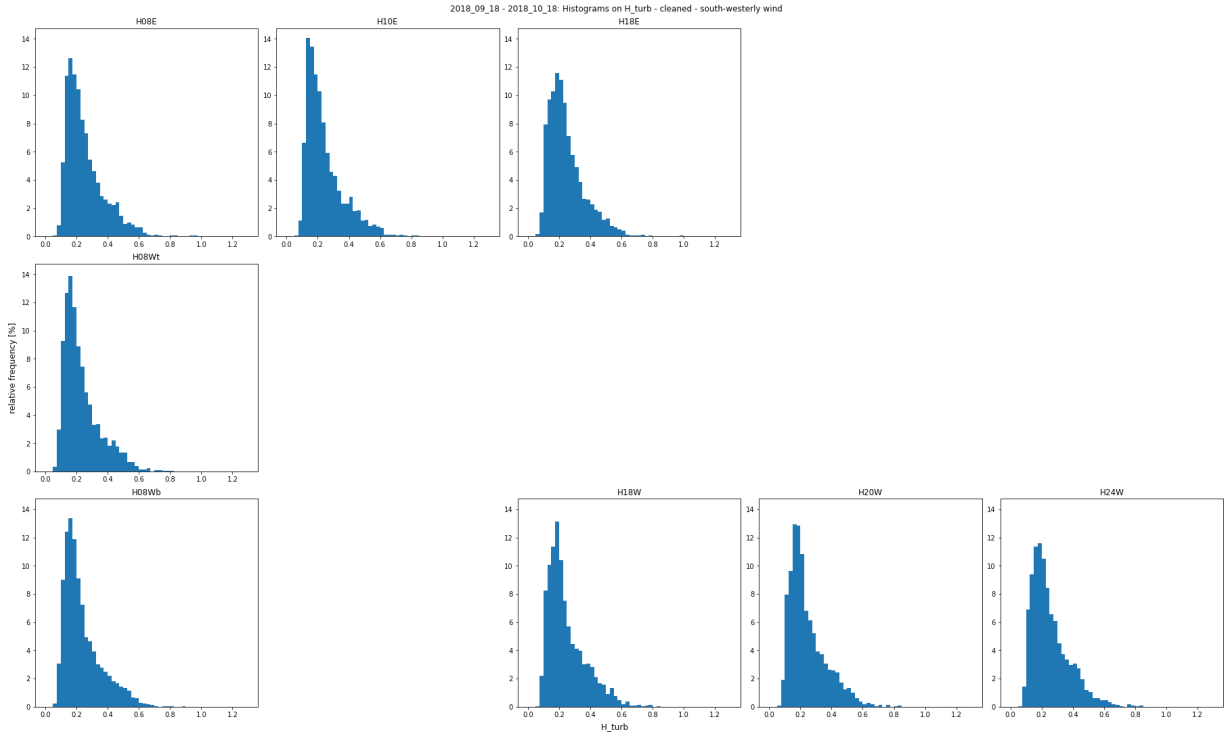
Statistics for H08Wb AOA_mean [°] - cleaned - south-westerly wind
---
Mean: 4.2
Median: 4.13
Percentile [10, 25, 75, 90]: [-1.29 2.15 6.08 8.93]
Minimum: -17.55
Maximum: 35.22
Variance: 23.56
Standard deviation: 4.85

Statistics for H08E AOA_mean [°] - cleaned - south-westerly wind
---
Mean: 2.18
Median: 1.75
Percentile [10, 25, 75, 90]: [-3.35 -0.39 4.02 8.75]
Minimum: -21.04
Maximum: 30.51
Variance: 26.74
Standard deviation: 5.17

Statistics for H18W AOA_mean [°] - cleaned - south-westerly wind
---
Mean: 5.47
Median: 4.5
Percentile [10, 25, 75, 90]: [0.51 2.27 7.21 12.02]
Minimum: -17.74
Maximum: 39.49
Variance: 30.28
Standard deviation: 5.5

Statistics for H18E AOA_mean [°] - cleaned - south-westerly wind
---
Mean: 3.3
Median: 1.18
Percentile [10, 25, 75, 90]: [-3.54 -0.73 5.37 15.46]
Minimum: -21.9
Maximum: 53.22
Variance: 64.0

Standard deviation: 8.0



Statistics for H08Wb H_turb - cleaned - south-westerly wind

Mean: 0.24
Median: 0.2
Percentile [10, 25, 75, 90]: [0.12 0.15 0.3 0.42]
Minimum: 0.06
Maximum: 0.89
Variance: 0.01
Standard deviation: 0.12

Statistics for H08E H_turb - cleaned - south-westerly wind

Mean: 0.25
Median: 0.22
Percentile [10, 25, 75, 90]: [0.13 0.16 0.31 0.43]
Minimum: 0.07
Maximum: 0.95
Variance: 0.01
Standard deviation: 0.12

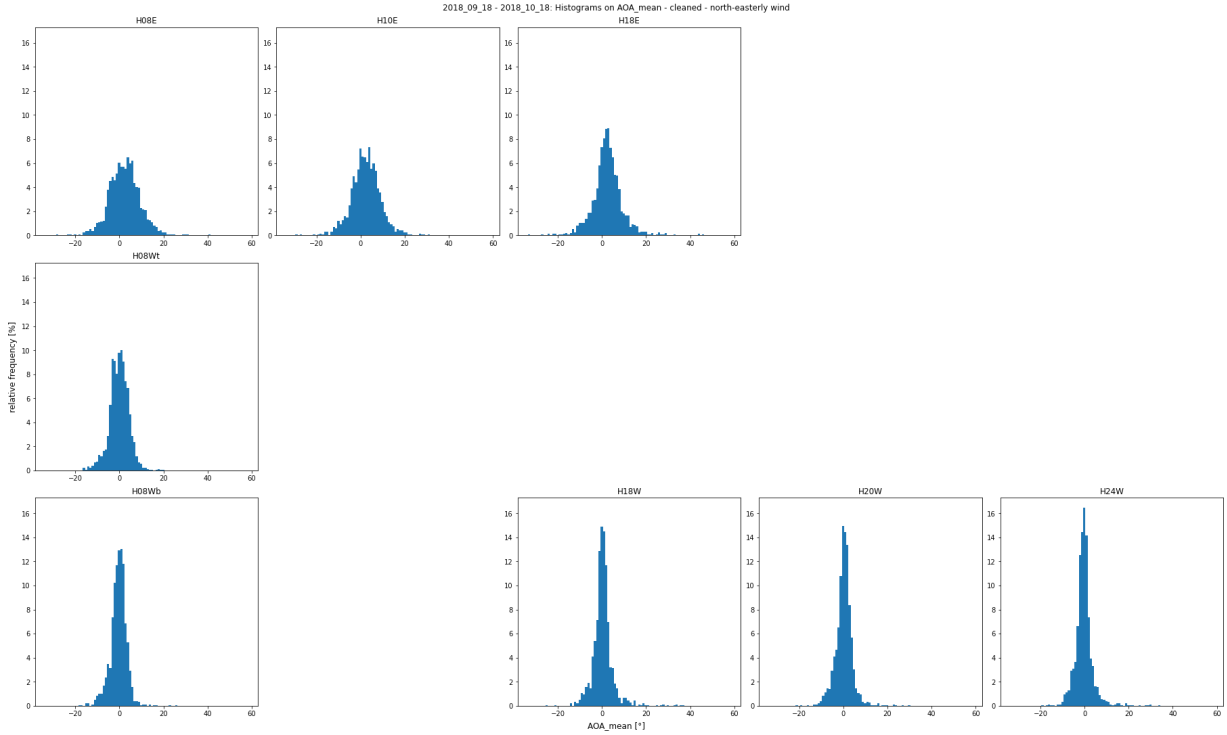
Statistics for H18W H_turb - cleaned - south-westerly wind

Mean: 0.25
Median: 0.21
Percentile [10, 25, 75, 90]: [0.12 0.16 0.31 0.42]
Minimum: 0.07
Maximum: 0.84
Variance: 0.01
Standard deviation: 0.12

Statistics for H18E H_turb - cleaned - south-westerly wind

Mean: 0.25
Median: 0.22
Percentile [10, 25, 75, 90]: [0.13 0.16 0.3 0.41]

Minimum: 0.06
Maximum: 0.98
Variance: 0.01
Standard deviation: 0.11



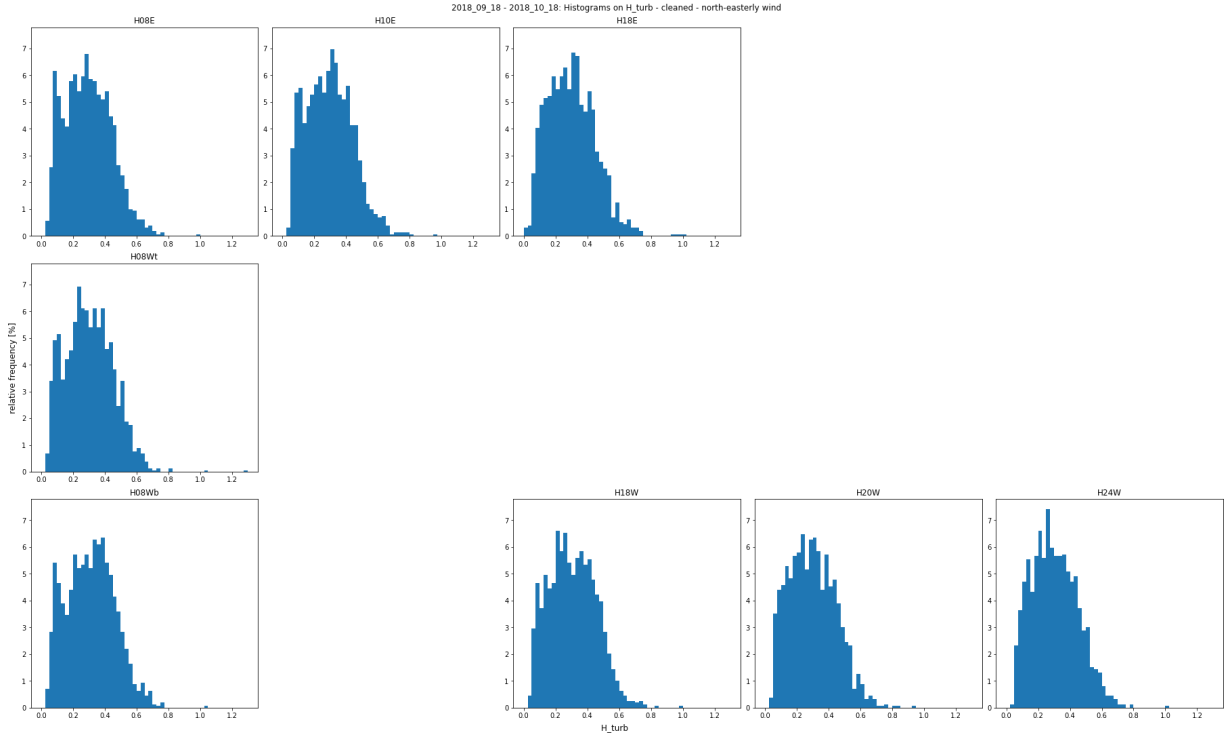
Statistics for H08Wb AOA_mean [°] - cleaned - north-easterly wind
---
Mean: -0.42
Median: -0.09
Percentile [10, 25, 75, 90]: [-4.95 -2.32 1.73 3.62]
Minimum: -17.86
Maximum: 26.1
Variance: 13.94
Standard deviation: 3.73

Statistics for H08E AOA_mean [°] - cleaned - north-easterly wind
---
Mean: 2.41
Median: 2.36
Percentile [10, 25, 75, 90]: [-5.48 -2.15 6.54 10.84]
Minimum: -27.67
Maximum: 40.72
Variance: 47.37
Standard deviation: 6.88

Statistics for H18W AOA_mean [°] - cleaned - north-easterly wind
---
Mean: 0.39
Median: 0.22
Percentile [10, 25, 75, 90]: [-4.11 -1.63 2.04 4.6]
Minimum: -25.3
Maximum: 36.82
Variance: 22.95
Standard deviation: 4.79

Statistics for H18E AOA_mean [°] - cleaned - north-easterly wind
---

Mean: 2.41
Median: 2.3
Percentile [10, 25, 75, 90]: [-5.12 -0.84 5.62 9.67]
Minimum: -33.11
Maximum: 45.93
Variance: 45.89
Standard deviation: 6.77



Statistics for H08Wb H_turb   - cleaned - north-easterly wind
---
Mean: 0.31
Median: 0.31
Percentile [10, 25, 75, 90]: [0.11 0.2 0.42 0.51]
Minimum: 0.04
Maximum: 1.03
Variance: 0.02
Standard deviation: 0.15

Statistics for H08E H_turb   - cleaned - north-easterly wind

Mean: 0.3
Median: 0.29
Percentile [10, 25, 75, 90]: [0.11 0.18 0.4 0.48]
Minimum: 0.04
Maximum: 0.98
Variance: 0.02
Standard deviation: 0.14

Statistics for H18W H_turb   - cleaned - north-easterly wind

Mean: 0.31
Median: 0.3
Percentile [10, 25, 75, 90]: [0.11 0.2 0.41 0.5]
Minimum: 0.03
Maximum: 1.0
Variance: 0.02
Standard deviation: 0.15

```
Statistics for H18E H_turb   - cleaned - north-easterly wind
-----------------------------------------------
Mean: 0.3
Median: 0.29
Percentile [10, 25, 75, 90]: [0.12 0.19 0.4  0.49]
Minimum: 0.0
Maximum: 1.02
Variance: 0.02
Standard deviation: 0.15
```

Weatherstation

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
```

Similarly, data **weather** data from the *weather station* at H10W can be analysed.

In [ ]:
```
for key in LFB.weather_cleaned.keys():
    if key in keys_of_interest:
        LFB.hist(LFB.weather_cleaned[key][filter_idxs],xlabel=key,xunit=LFB.units[ke
        print('Statistics for',key,LFB.units[key])
        print('--------------------------------')
        print('Average:',np.round(np.mean(LFB.weather_cleaned[key].T[filter_idxs].T)
        print('Median:',np.round(np.median(LFB.weather_cleaned[key].T[filter_idxs].T
        print('Percentile 10, 25, 75 and 90:',np.round(np.percentile(LFB.weather_cle
        print('Minimum:',np.round(np.min(LFB.weather_cleaned[key].T[filter_idxs].T),
        print('Maximum:',np.round(np.max(LFB.weather_cleaned[key].T[filter_idxs].T),
        print('Standard deviation:',np.round(np.std(LFB.weather_cleaned[key].T[filte
```

Accelerometers

Similarly, **accelerometer** data can be analysed.

In [ ]:
```
for key in LFB.acc_cleaned.keys():
    if key in keys_of_interest:
        LFB.hist(LFB.acc_cleaned[key].T[filter_idxs].T,xlabel=key,xunit=LFB.units[ke
        print('Statistics for',key,LFB.units[key])
        print('--------------------------------')
        print('Average:',np.round(np.mean(LFB.acc_cleaned[key].T[filter_idxs].T),2))
        print('Median:',np.round(np.median(LFB.acc_cleaned[key].T[filter_idxs].T),2)
        print('Percentile 10, 25, 75 and 90:',np.round(np.percentile(LFB.acc_cleaned
        print('Minimum:',np.round(np.min(LFB.acc_cleaned[key].T[filter_idxs].T),2))
        print('Maximum:',np.round(np.max(LFB.acc_cleaned[key].T[filter_idxs].T),2))
        print('Standard deviation:',np.round(np.std(LFB.acc_cleaned[key].T[filter_id
```

Wind roses

Another usefull visualisation technique are **wind roses**, which can be created using the `windrose` method.

Note that it is possible to provide a specific colormap to the keyword `cmap` and even create your own colormap, for example by using the `ListedColormap` class from the *matplotlib*

library.

In this example the colormap from windfinder.com is recreated.

In [ ]:
```
# Colormap for windroses based on windfinder.com colors
Windfinder_wind_rose_cmap = ListedColormap(['darkblue', 'lightskyblue', 'limegreen',
```

The *filters* work the same way as above and will not be explained anymore further down.

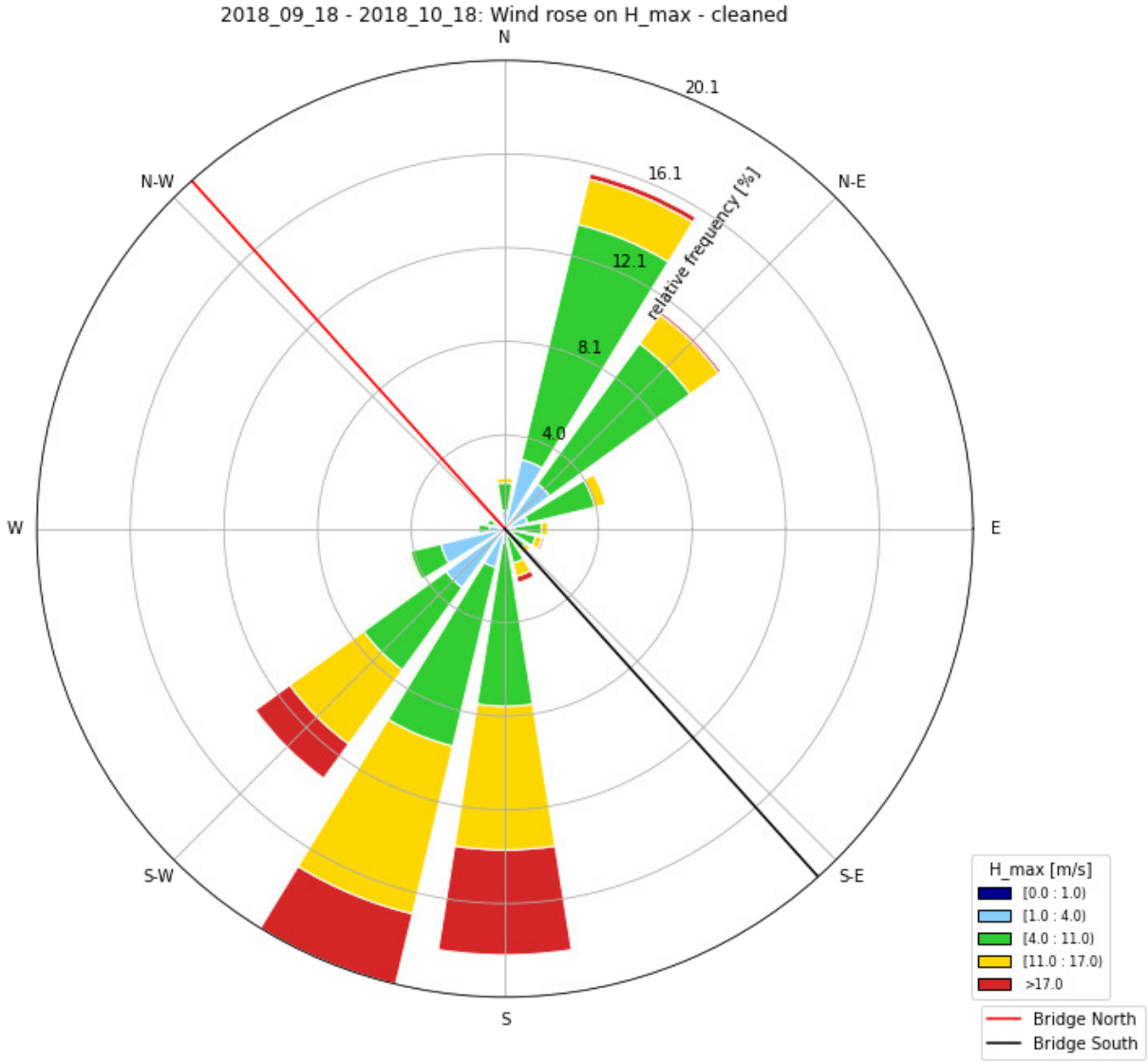
In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
```

The wind roses can for example be created for `'H_max'` or `'H_mean'`. It is also again possible to split it up per sensor using the `split_sensors` keyword.

Note that for *multi-variat* analysis on measurements of different types it is necessary to find the `common_ok_sensors` and respective *sensor indices* first using the `find_common_ok_sensors` method.

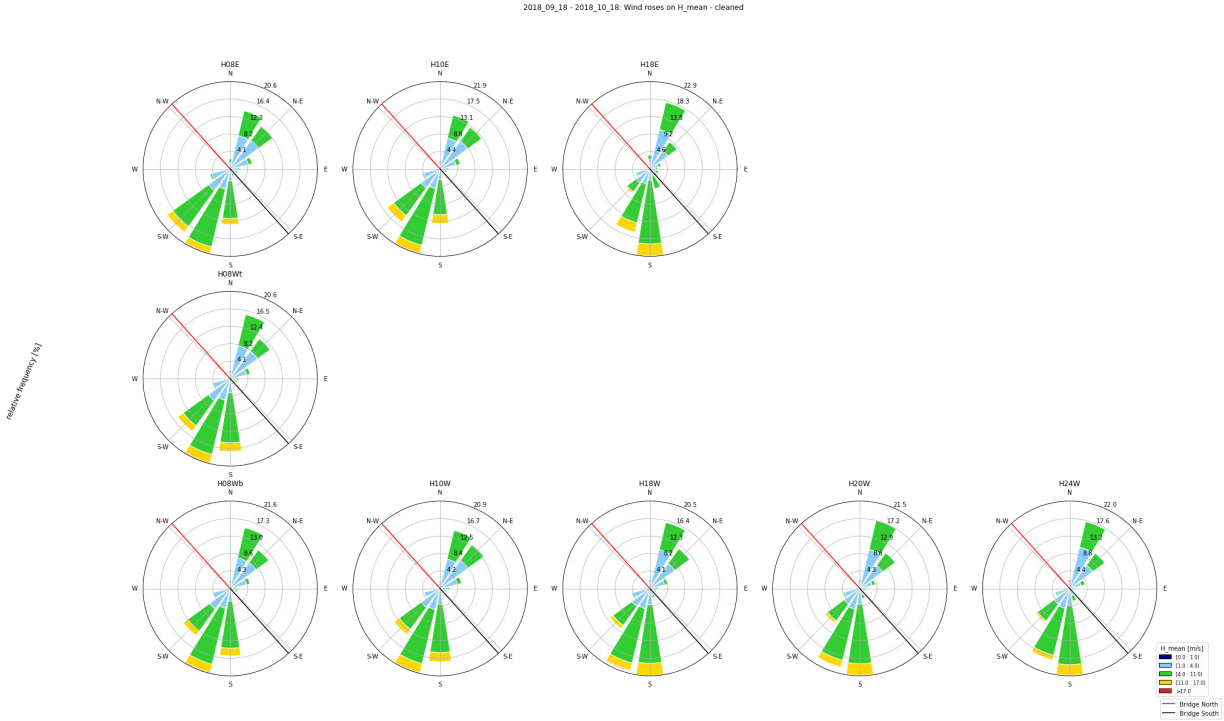
Also note how our custom `Windfinder_wind_rose_cmap` *colormap* is provided to the keyword `cmap` of the `windrose` method for this type of measurement and the respective `bins` are setup.

In [ ]:
```
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
LFB.windrose(LFB.anemo_cleaned['Dir_mean'][s1_ind].T[filter_idxs].T,LFB.anemo_cleane
```

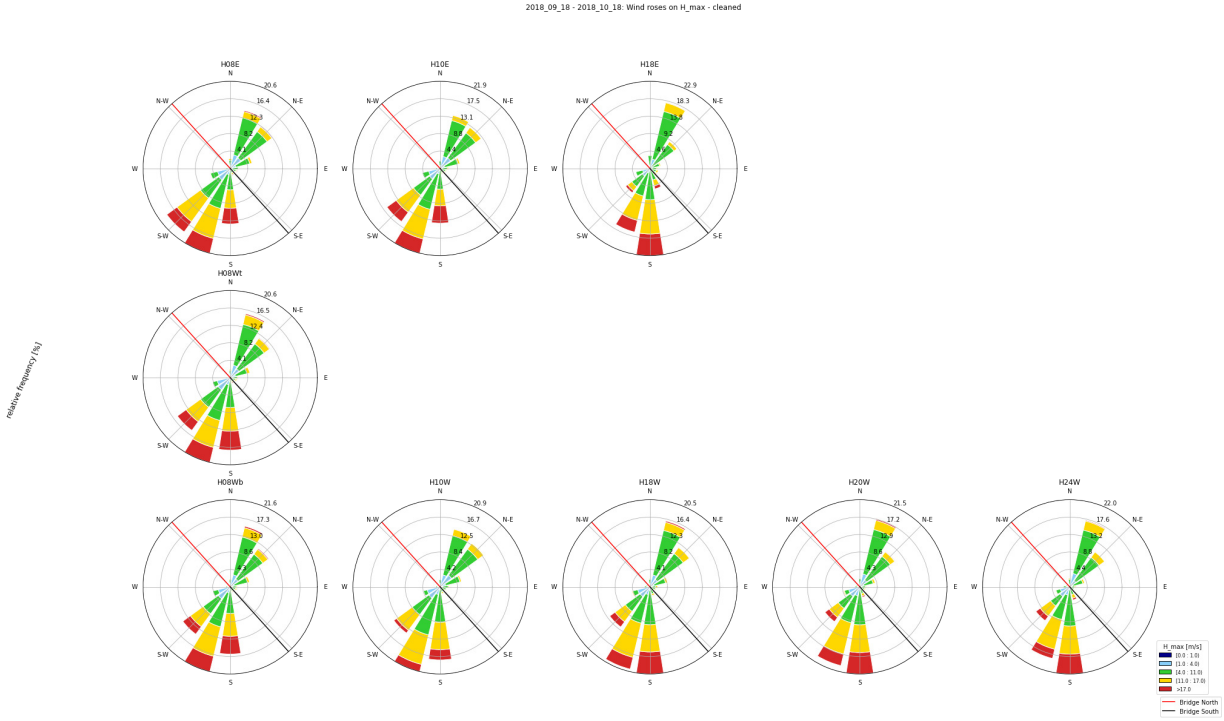


In [ ]:

```
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
LFB.windrose(LFB.anemo_cleaned['Dir_mean'][s1_ind].T[filter_idxs].T,LFB.anemo_cleane
```



In [ ]:
```
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
LFB.windrose(LFB.anemo_cleaned['Dir_mean'][s1_ind].T[filter_idxs].T,LFB.anemo_cleane
```



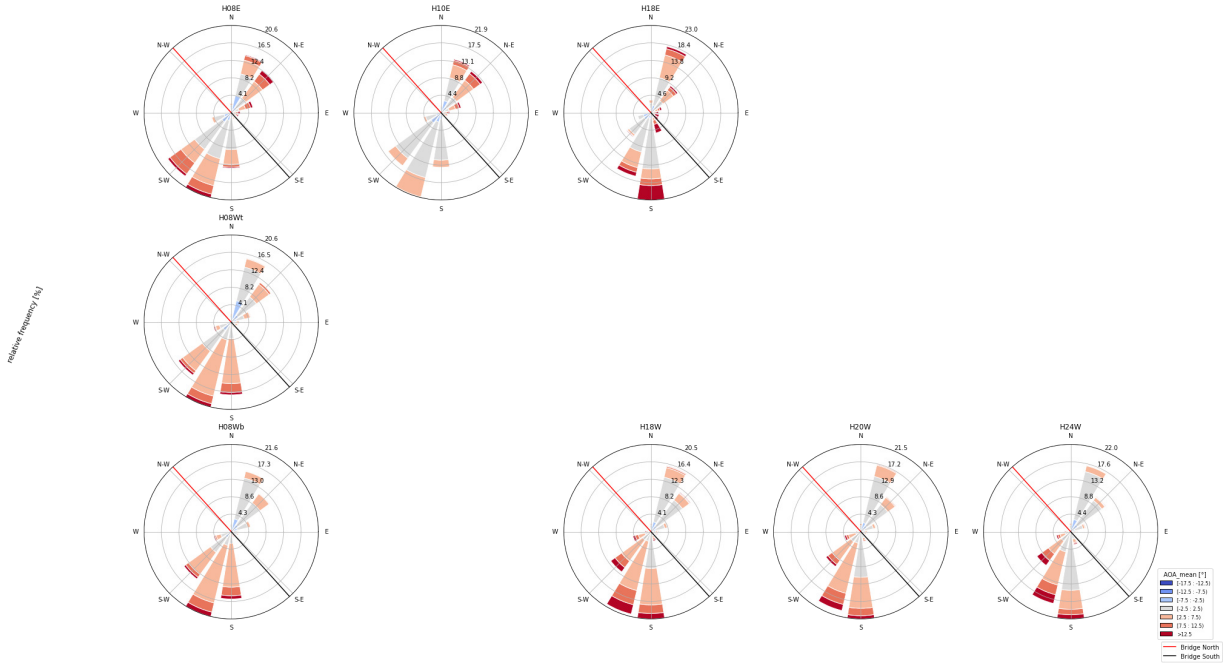
AOA wind roses

It is also possible to create wind roses on other *types of measurements*, such as the **angle of attack (AOA)**, by providing the respective *key* `'AOA_mean'`.

Note how the `'coolwarm'` *colormap* is provided to the keyword `cmap` of the `windrose` method for this type of measurement and the respective `bins` are setup.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
```

In [ ]:
```
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
LFB.windrose(LFB.anemo_cleaned['Dir_mean'][s1_ind].T[filter_idxs].T,LFB.anemo_cleane
```

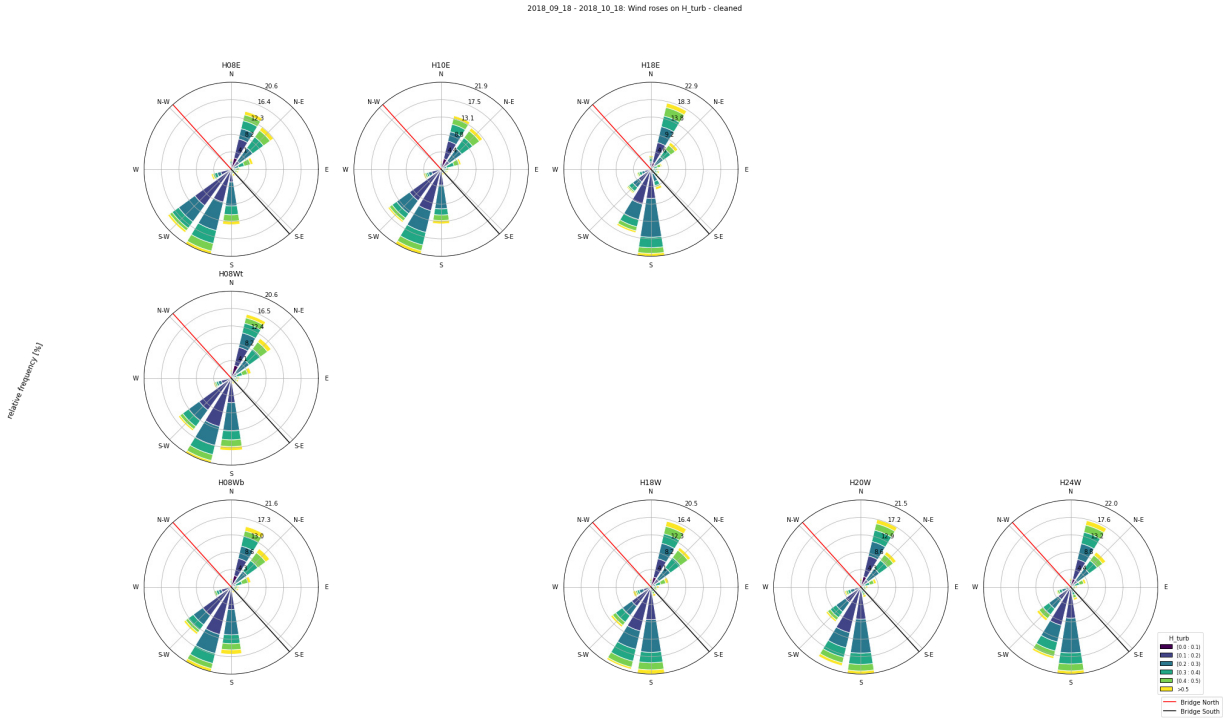


Turbulence wind roses

Similarly, wind roses can be created for the horizontal turbulence intensity `'H_turb'`

In [ ]:

```
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
LFB.windrose(LFB.anemo_cleaned['Dir_mean'][s1_ind].T[filter_idxs].T,LFB.anemo_cleane
```



Polar scatterplots

Another visualisation technique for directional data is creating *polar scatterplots* using the `polar_scatterplot` method.

Again a custom *colormap* `Windfinder_cmap` is set up for wind speed data (`'H_m...'`) using the `ListedColormap` class from the *matplotlib* library to match the colormap from

windfinder.com.

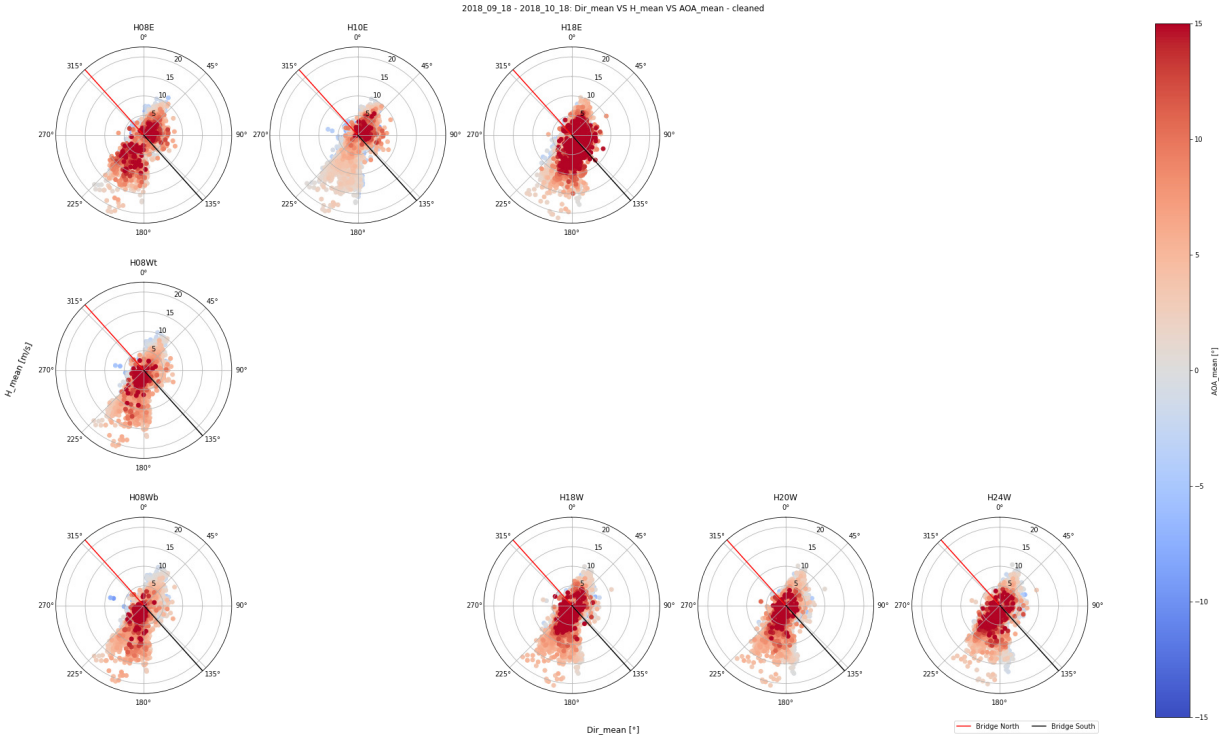
In [ ]:
```
# Colormap for scatterplots based on windfinder.com colors
Windfinder_cmap_list = []
for windspeed in range(36):
    if windspeed <= 1:
        Windfinder_cmap_list.append('darkblue')
    elif windspeed <= 4:
        Windfinder_cmap_list.append('lightskyblue')
    elif windspeed <= 11:
        Windfinder_cmap_list.append('limegreen')
    elif windspeed <= 17:
        Windfinder_cmap_list.append('gold')
    else:
        Windfinder_cmap_list.append('tab:red')
Windfinder_cmap = ListedColormap(Windfinder_cmap_list)
```

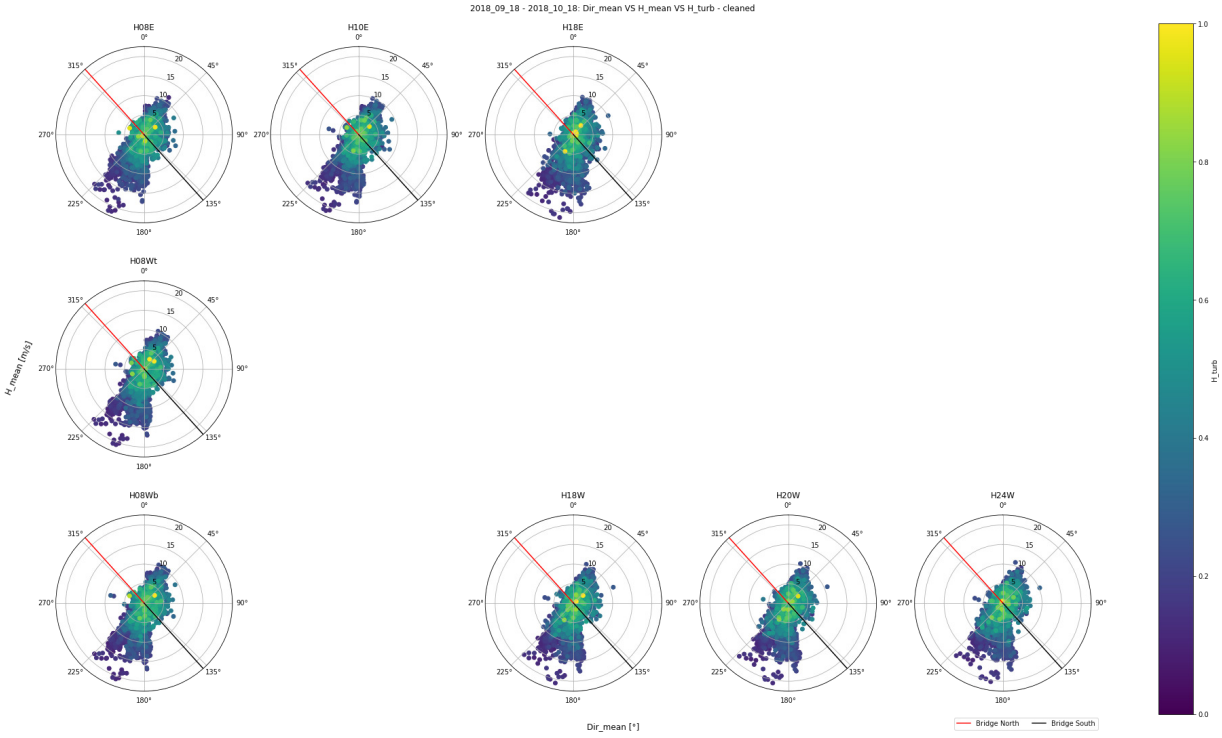
In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'][LFB.get_ok_sensor_ind(
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['AOA_mean'],prior_idxs=filter_idxs
```

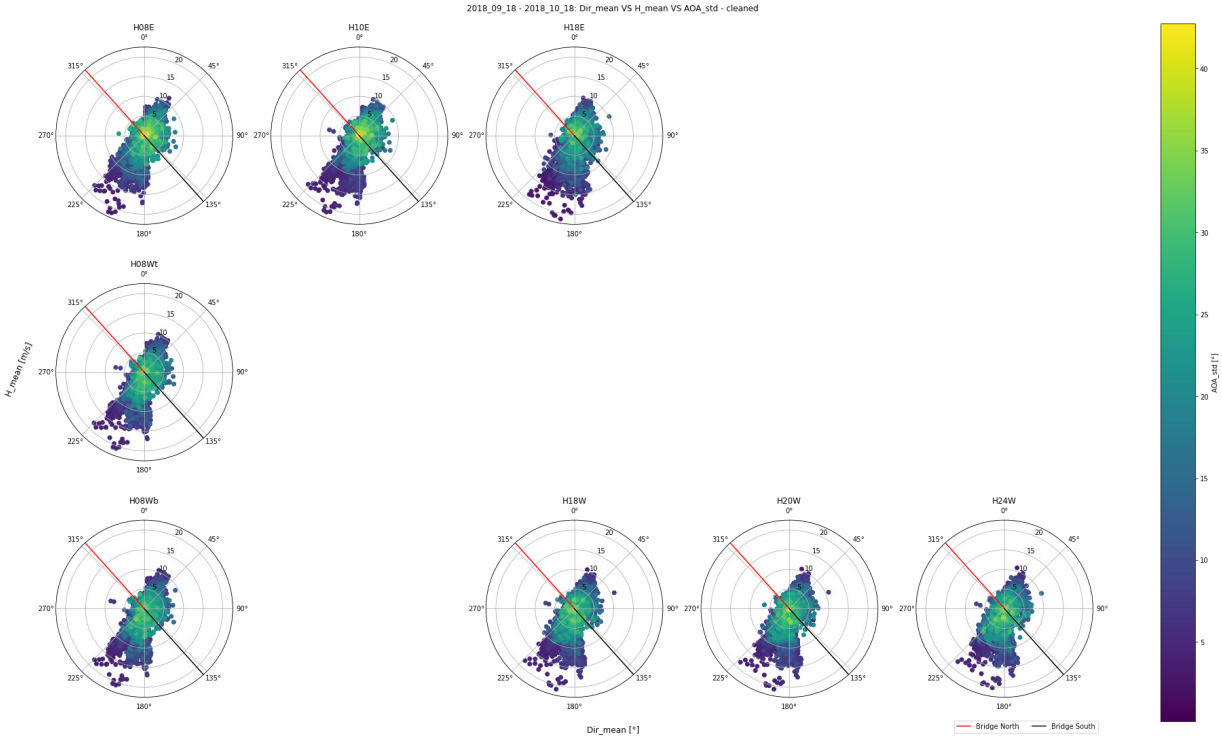
A similar method of *batch-processing* as showcased in the time-series plots is utilised, looping over the `key2s_of_interest` for the *radial axis* and `key3s_of_interest` for the *color-axis*

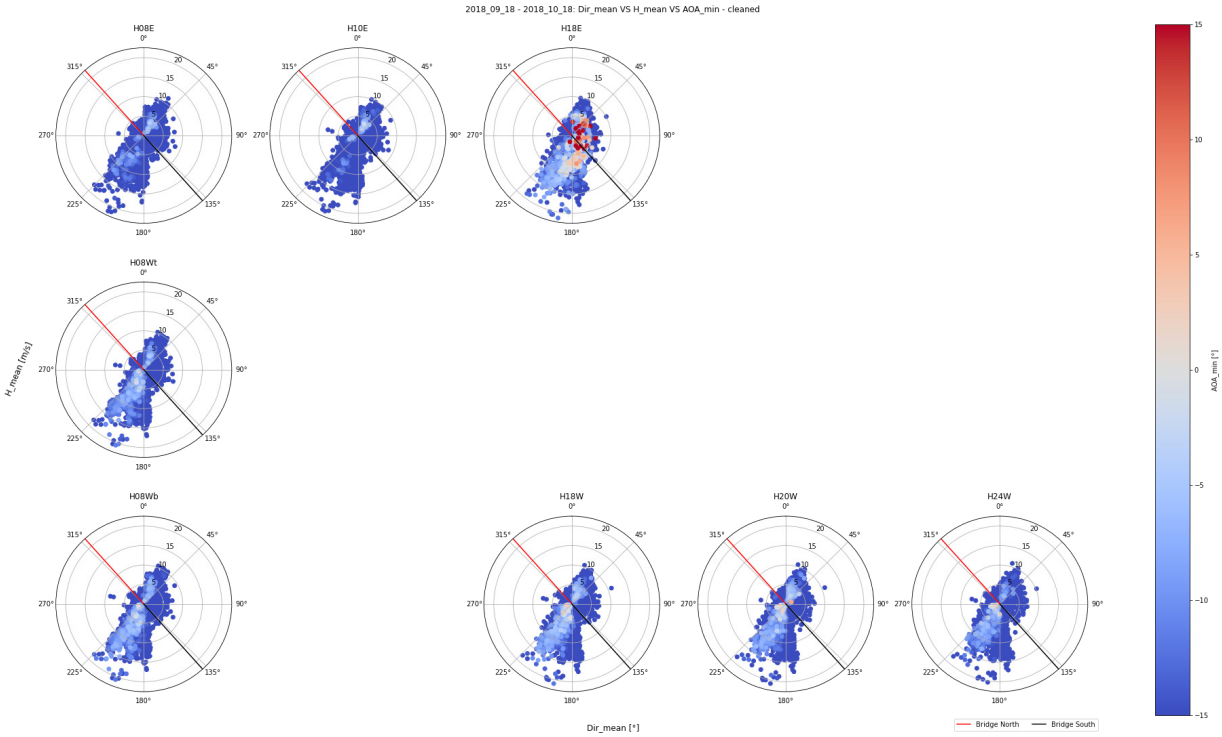
Note how different *colormaps* (`cmap`) and *limits* (`vmin` and `vmax`) are assigned to different *types of measurement* on the *color-axis* (`key3`)

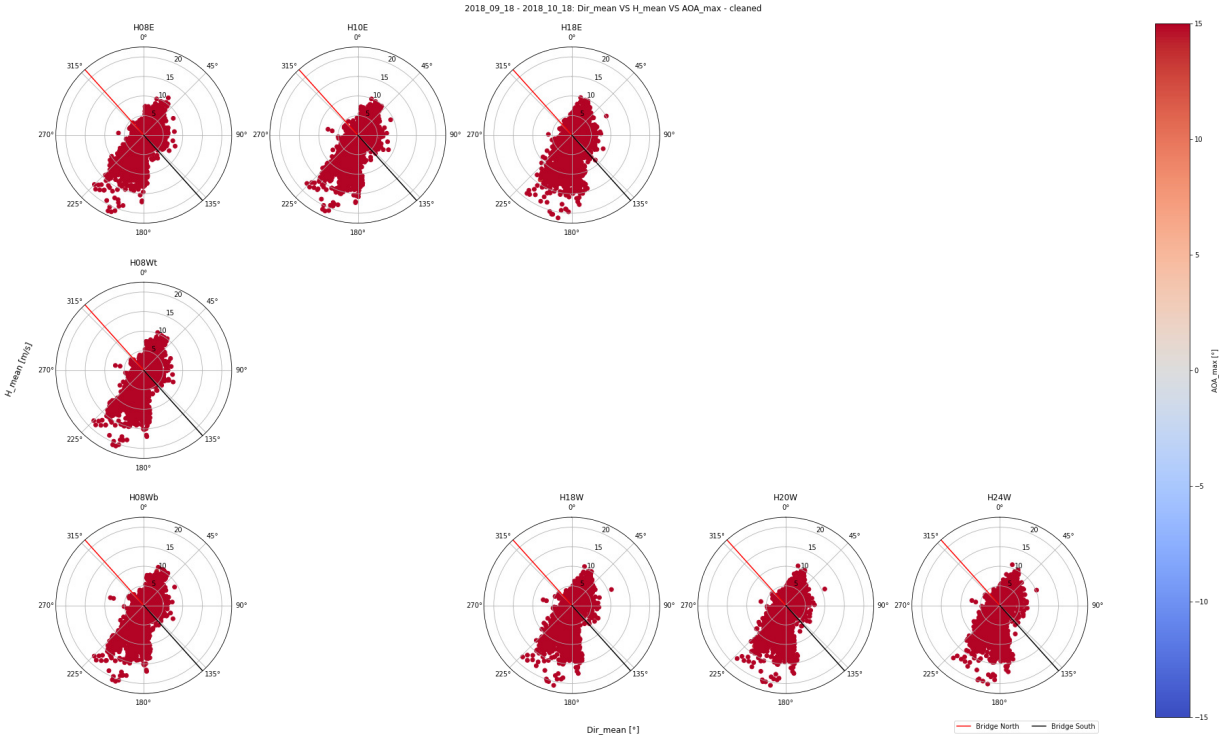
In [ ]:
```
key1 = 'Dir_mean'
key2s_of_interest =['H_mean','H_max','AOA_mean','H_turb']
key3s_of_interest = ['AOA_mean','H_mean','H_turb','AOA_std','AOA_min','AOA_max','W_t

for key2 in key2s_of_interest:
    for key3 in key3s_of_interest:
        if key2 != key3:
            if key3.startswith('AOA_m'):
                cmap = 'coolwarm'
                vmin = -15
                vmax = 15
            elif key3.startswith('H_m'):
                cmap = Windfinder_cmap
                vmin = 0
                vmax = 35
            elif key3.endswith('_turb'):
                cmap = 'viridis'
                vmin = 0
                vmax = 1
            else:
                cmap = 'viridis'
                vmin = None
                vmax = None
            common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_o
            common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_o
            s1_ind = list(np.array(s1_ind)[stemp_ind])
            s2_ind = list(np.array(s2_ind)[stemp_ind])
            LFB.polar_scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,L
```

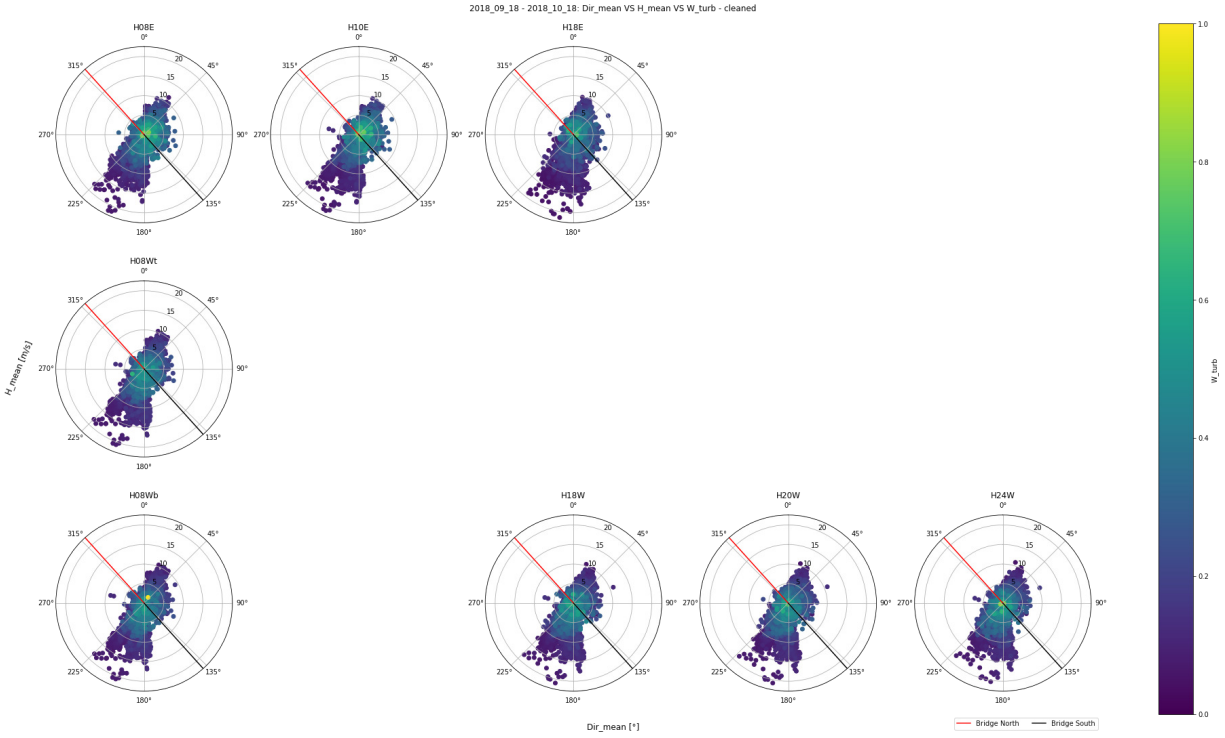

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS AOA_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
AOA_mean [°]
Bridge North
Bridge South

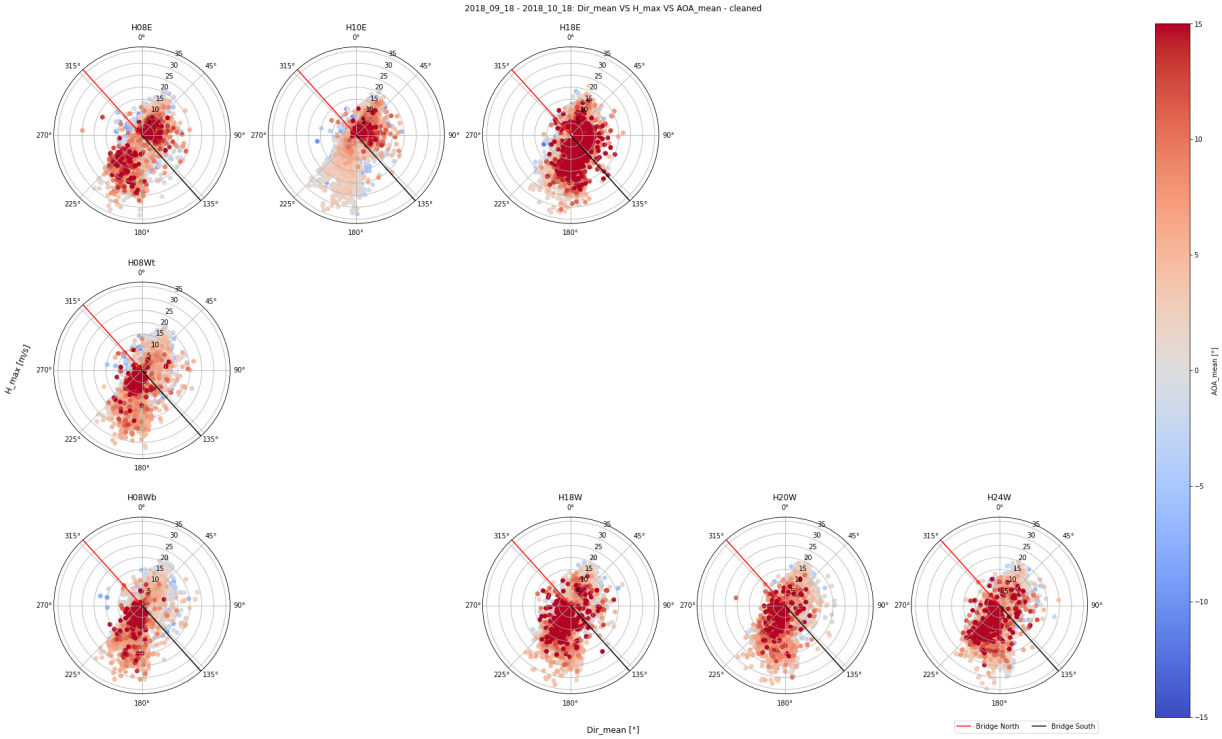

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS H_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
H_turb
Bridge North
Bridge South

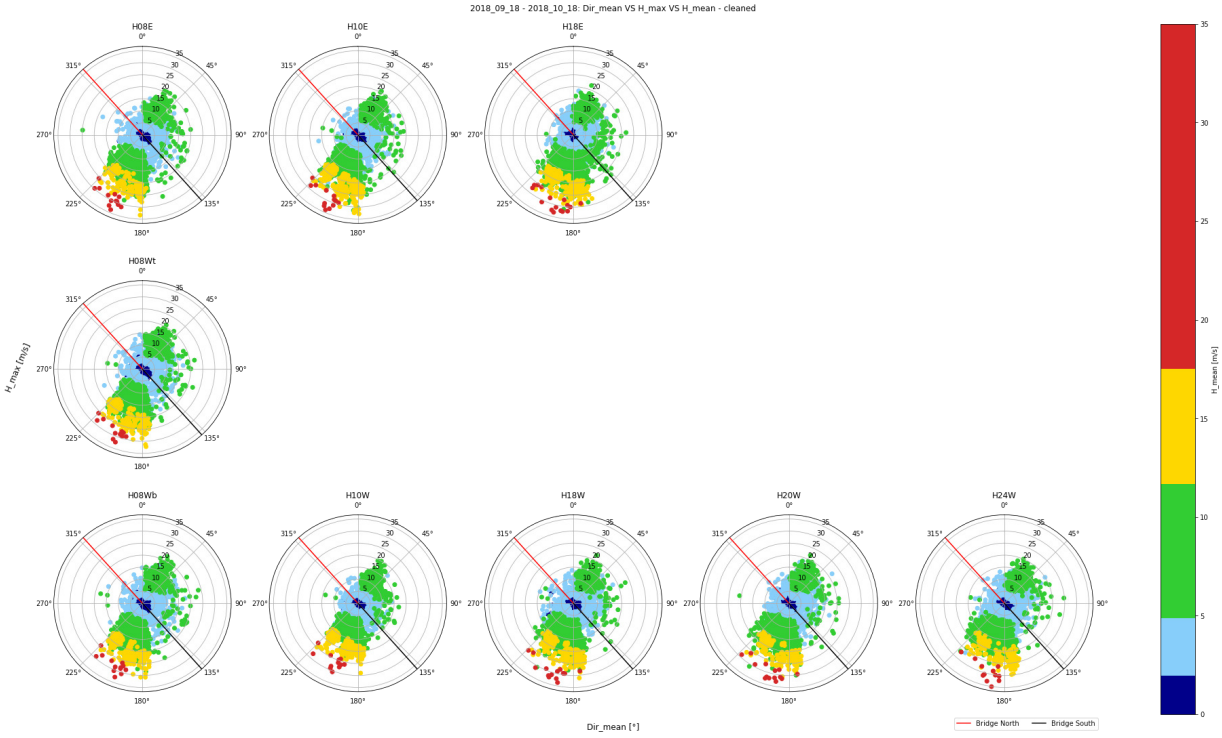

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS AOA_std - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
AOA_std [°]
Bridge North
Bridge South

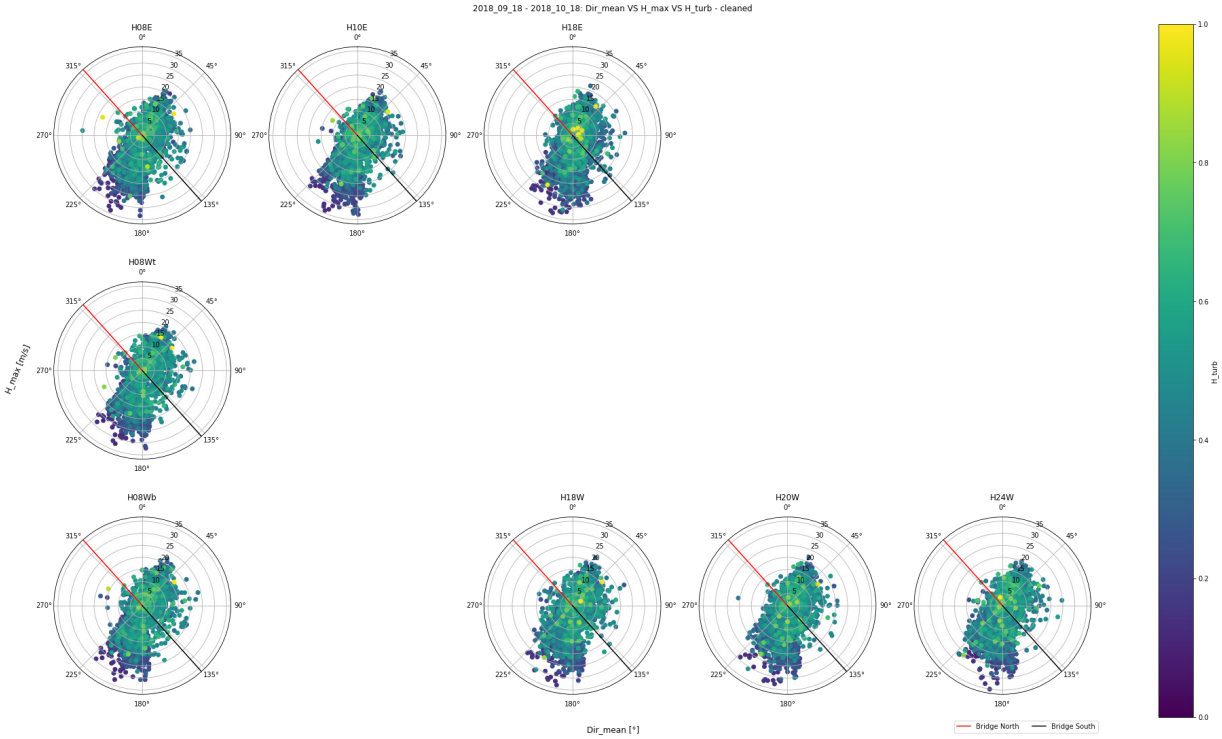

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS AOA_min - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
AOA_min [°]
Bridge North
Bridge South

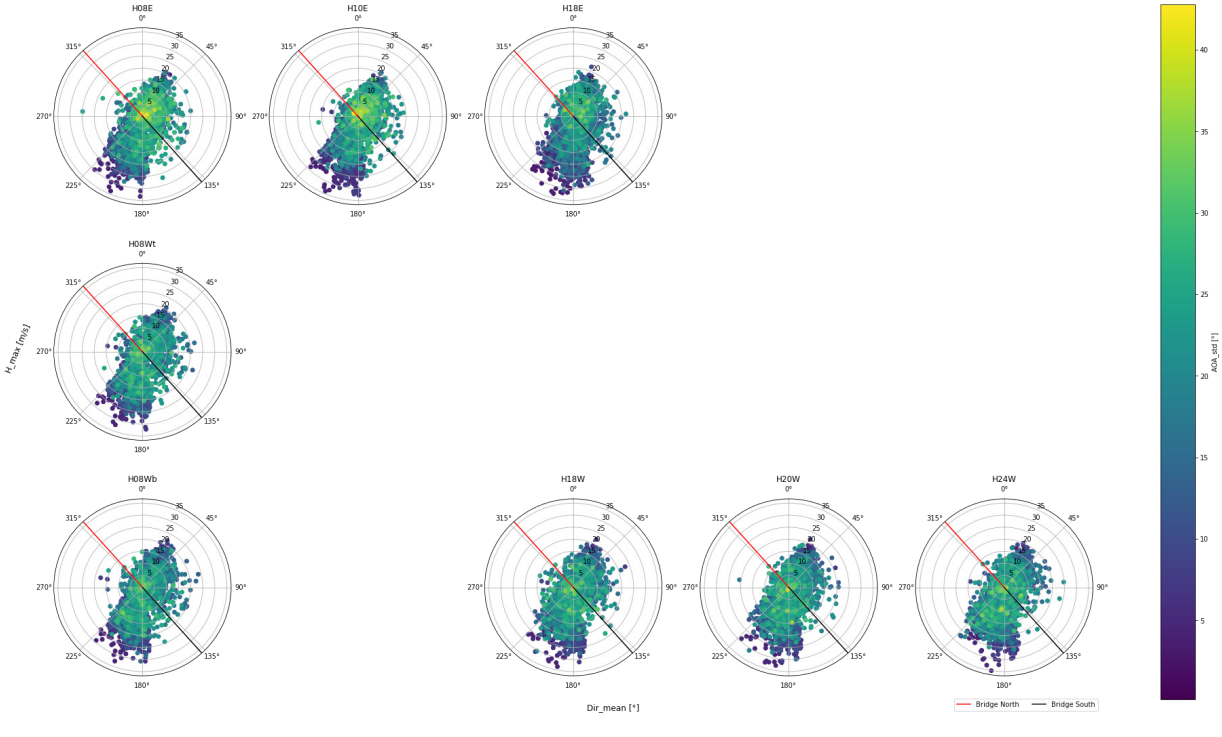

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS AOA_max - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
AOA_max [°]
Bridge North
Bridge South

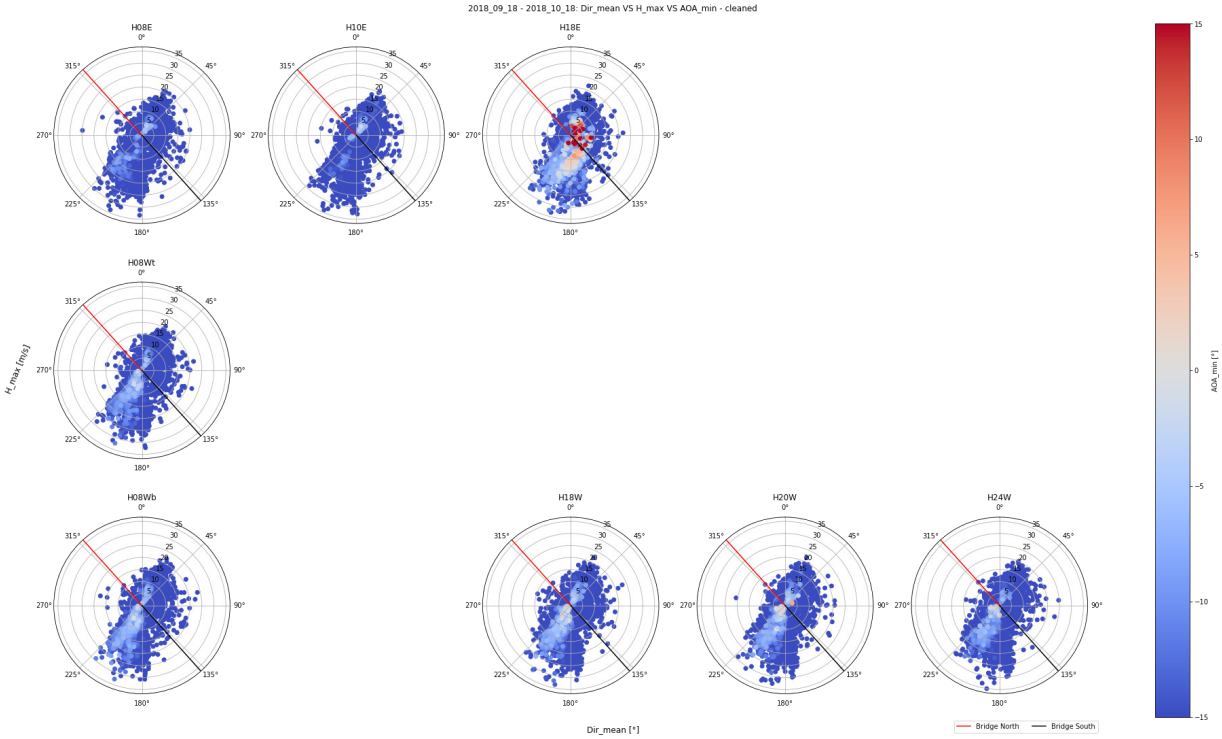

2018_09_18 - 2018_10_18: Dir_mean VS H_mean VS W_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_mean [m/s]
Dir_mean [°]
W_turb
Bridge North
Bridge South

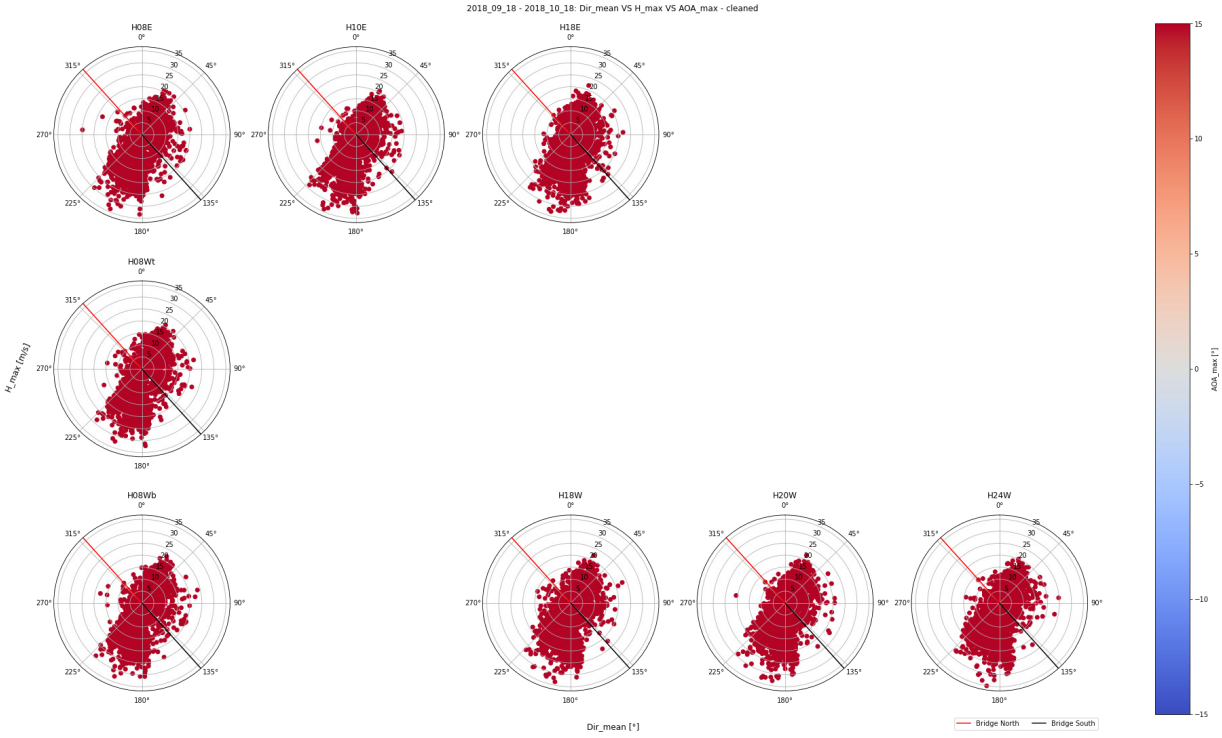

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS AOA_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
AOA_mean [°]
Bridge North
Bridge South

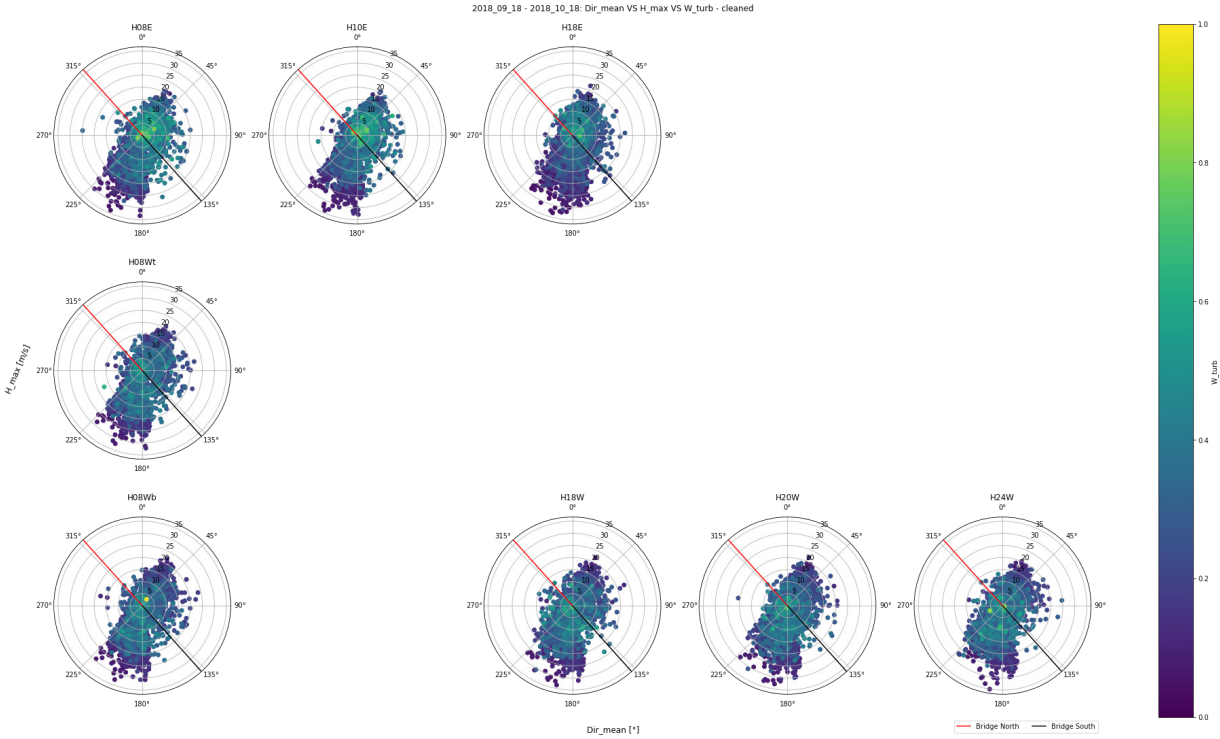

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS H_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H10W
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
H_mean [m/s]
Bridge North
Bridge South

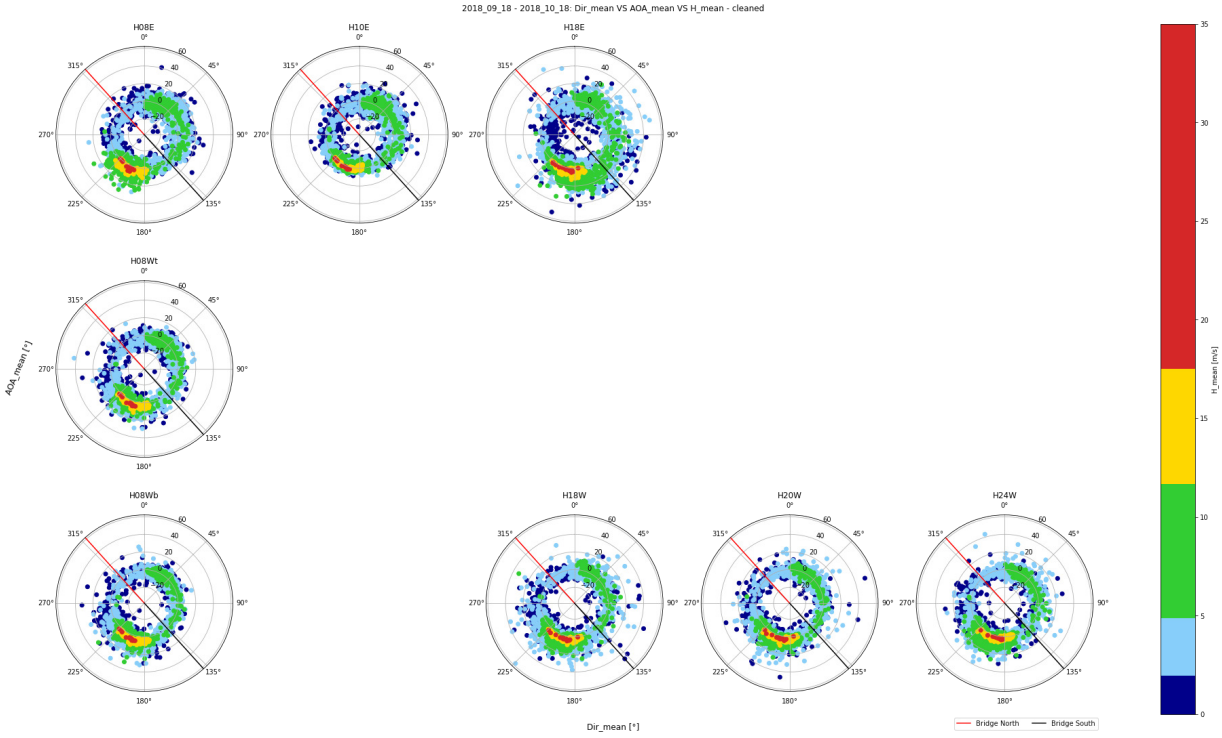

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS H_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
H_turb
Bridge North
Bridge South

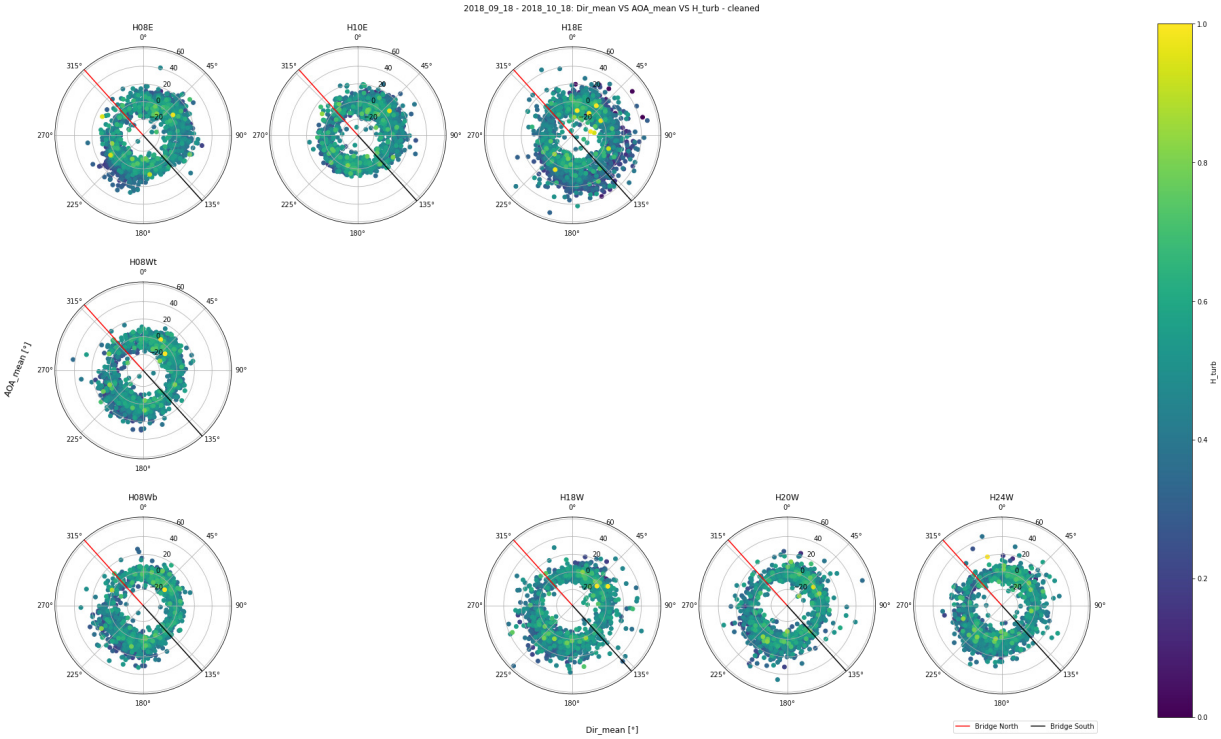

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS AOA_std - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
AOA_std [°]
Bridge North
Bridge South

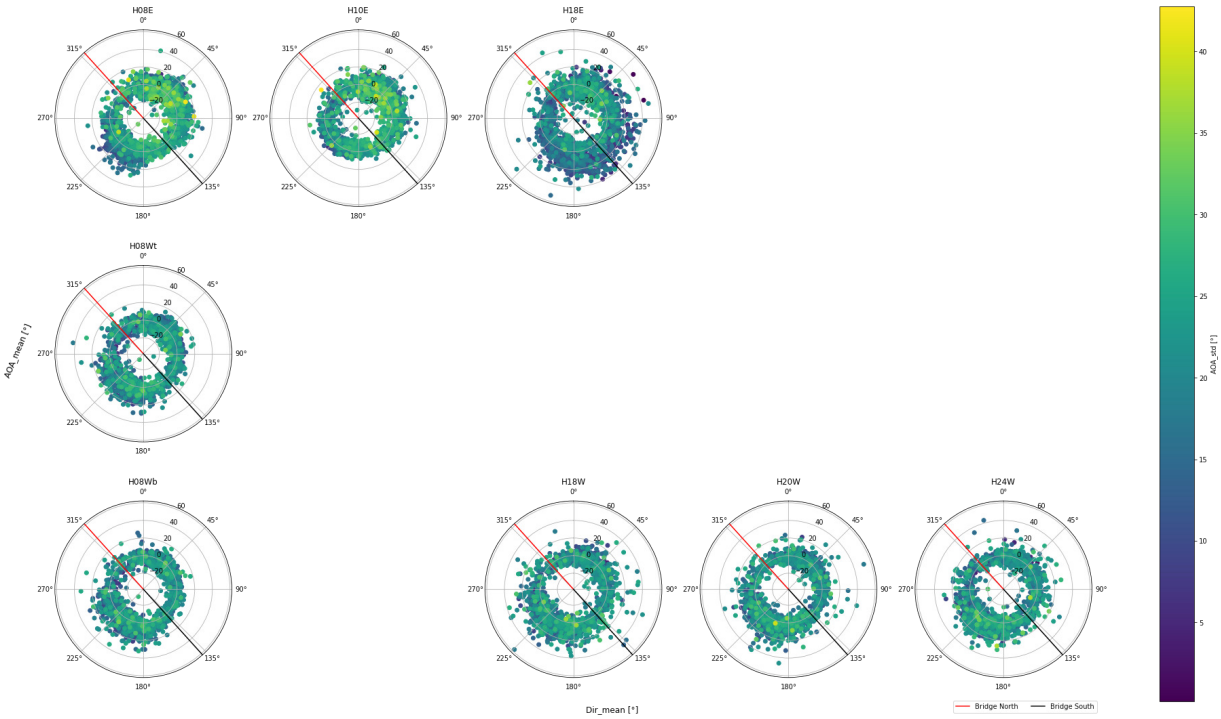

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS AOA_min - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
AOA_min [°]
Bridge North
Bridge South

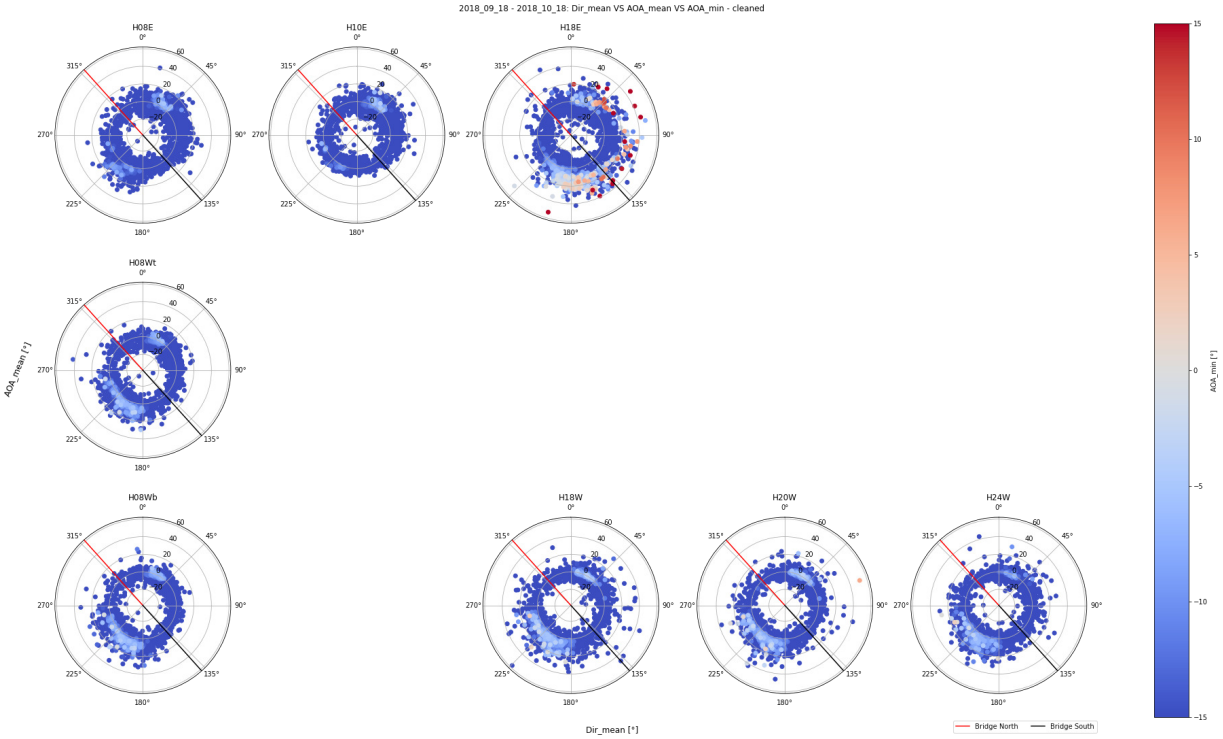

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS AOA_max - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
Dir_mean [°]
AOA_max [°]
Bridge North
Bridge South

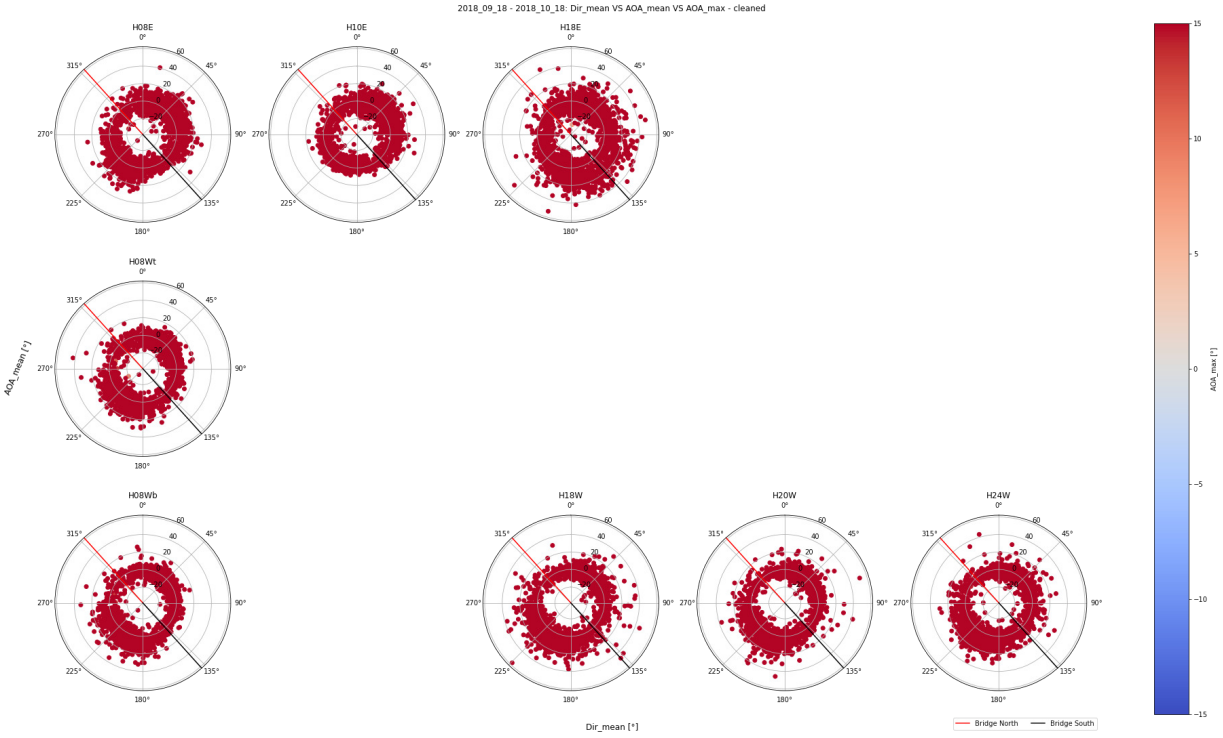

2018_09_18 - 2018_10_18: Dir_mean VS H_max VS W_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_max [m/s]
W_turb
Dir_mean [°]
Bridge North
Bridge South

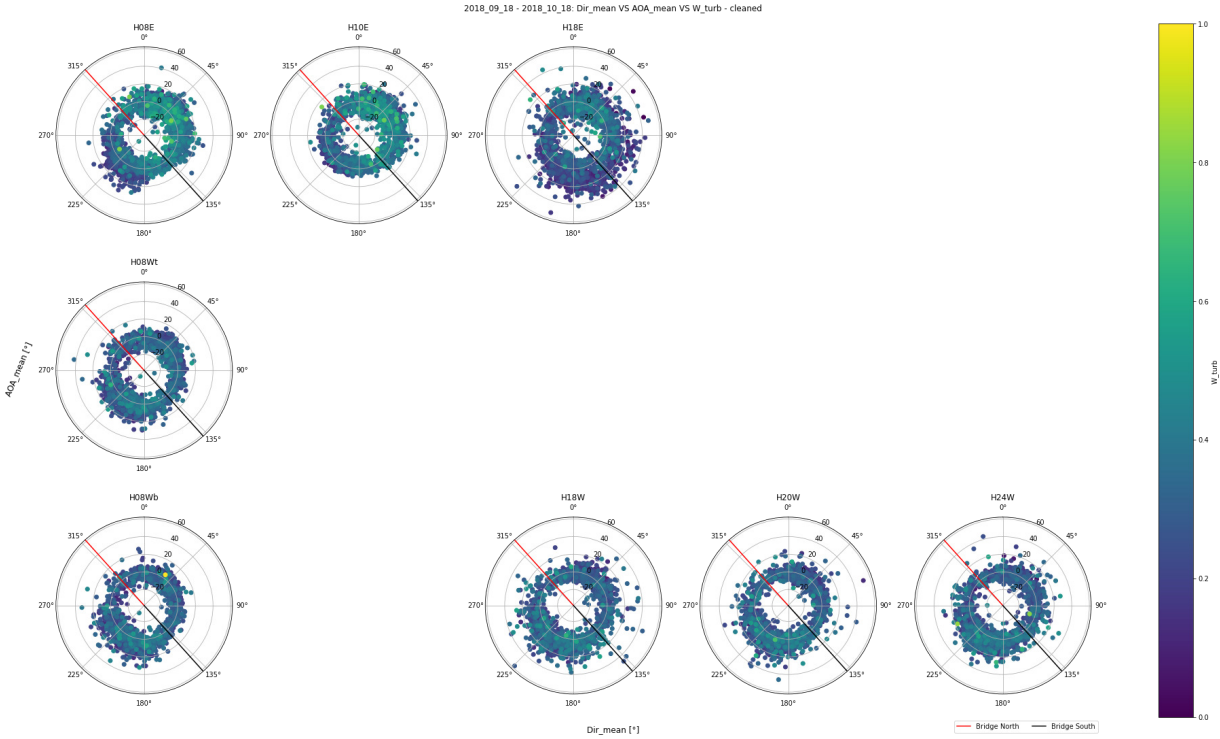

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS H_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
H_mean [m/s]
Dir_mean [°]
Bridge North
Bridge South

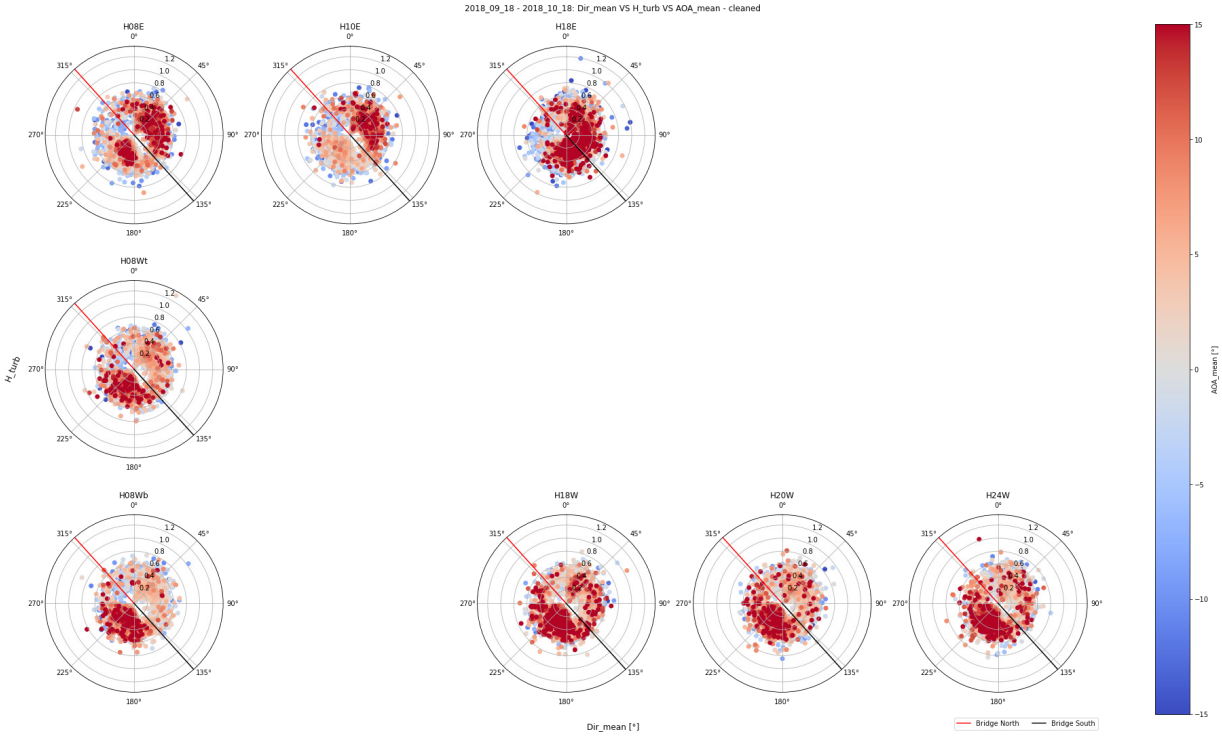

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS H_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
Dir_mean [°]
H_turb
Bridge North
Bridge South

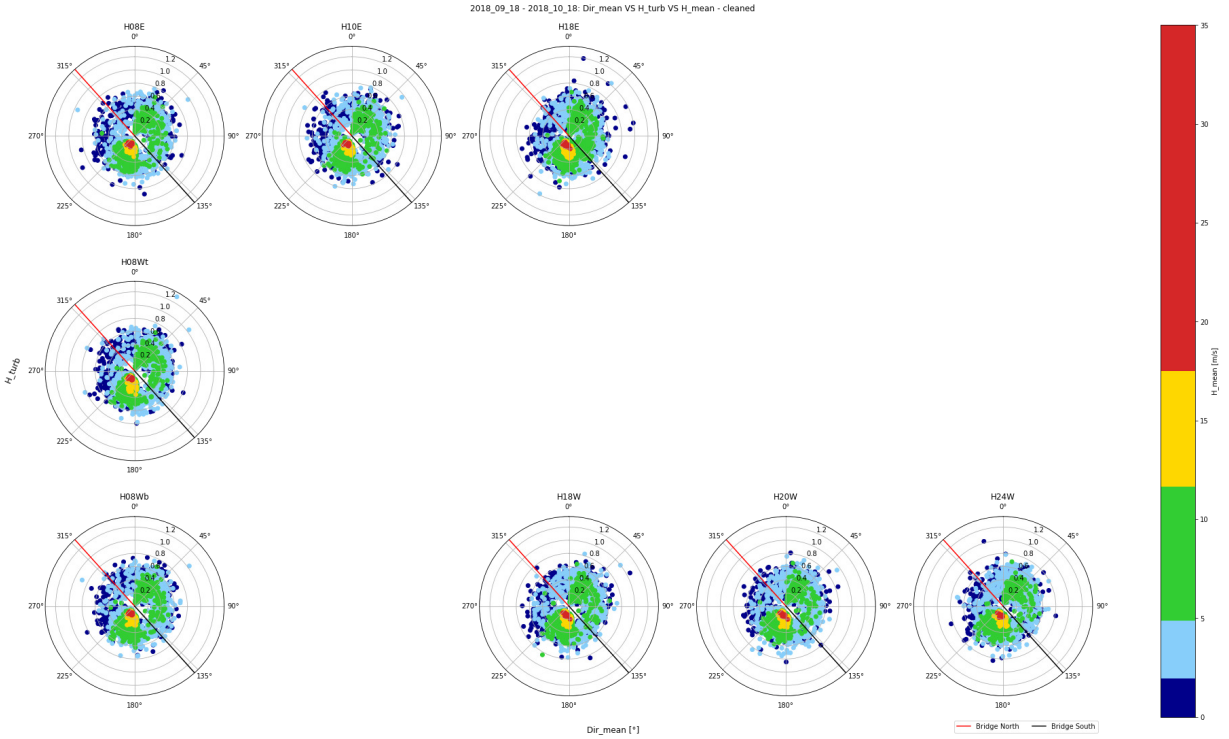

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS AOA_std - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
Dir_mean [°]
AOA_std [°]
Bridge North
Bridge South

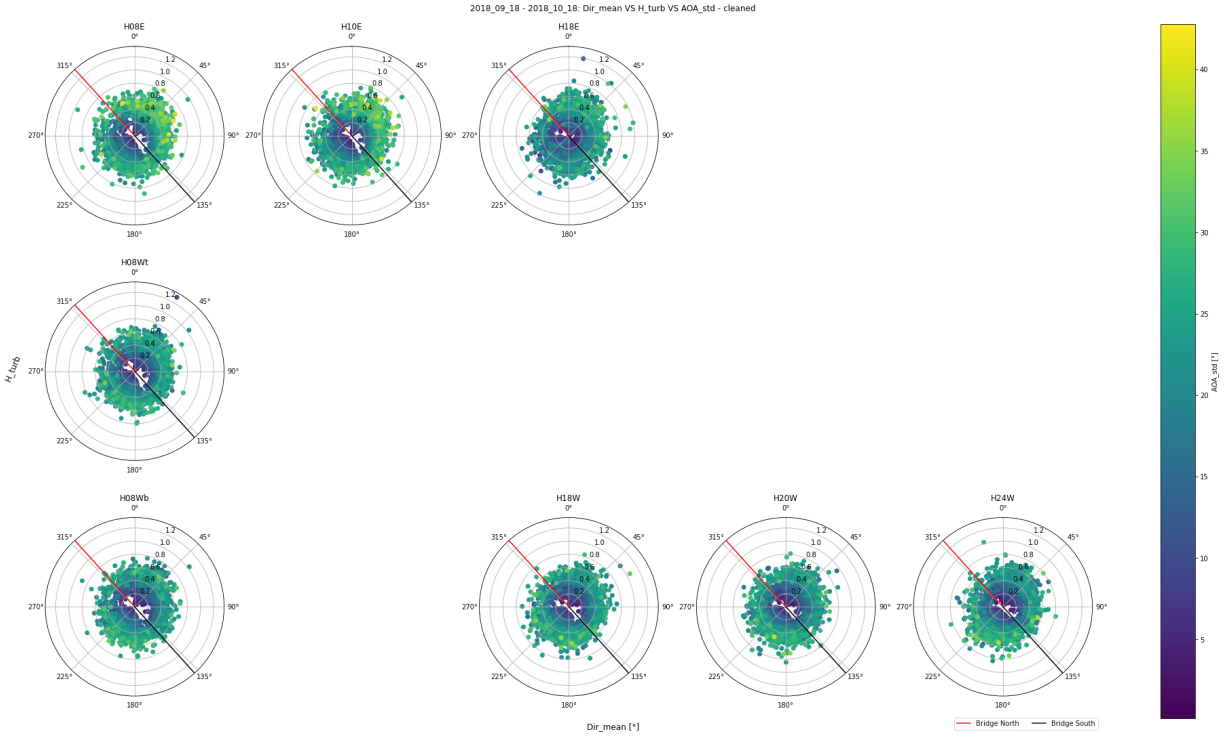

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS AOA_min - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
Dir_mean [°]
AOA_min [°]
Bridge North
Bridge South

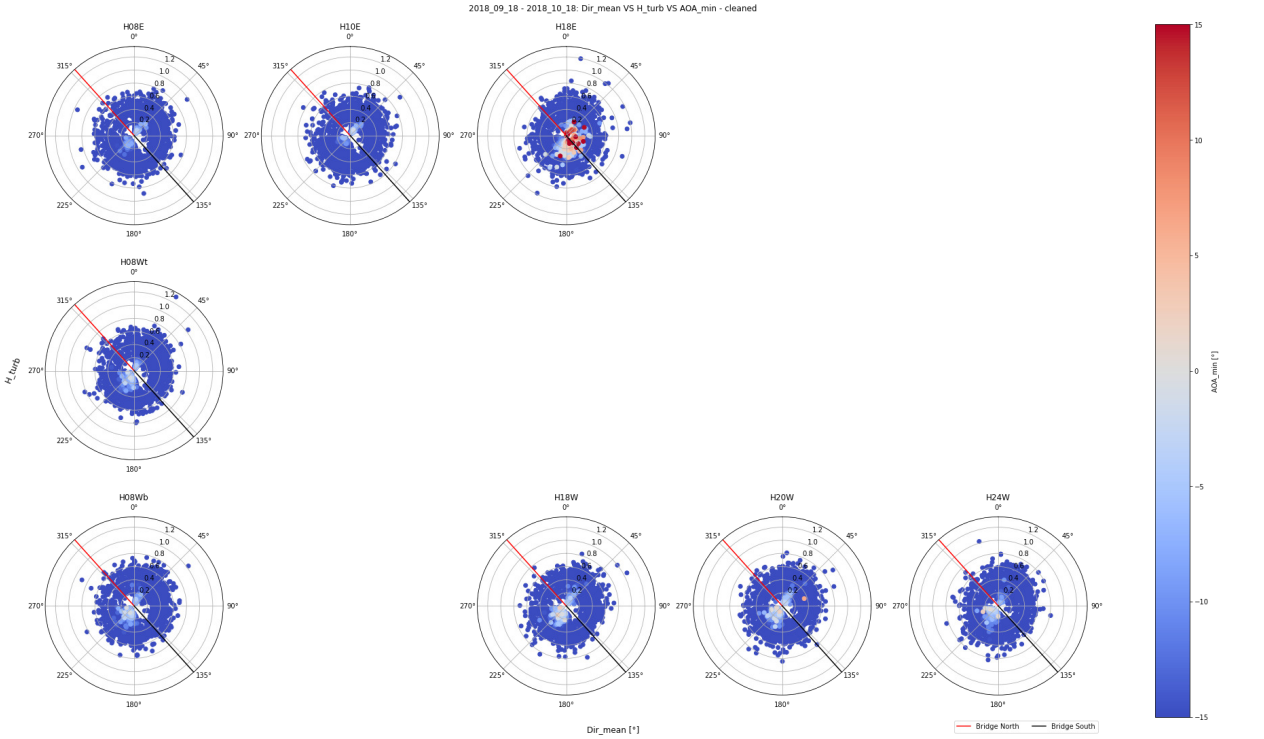

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS AOA_max - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
Dir_mean [°]
AOA_max [°]
Bridge North
Bridge South

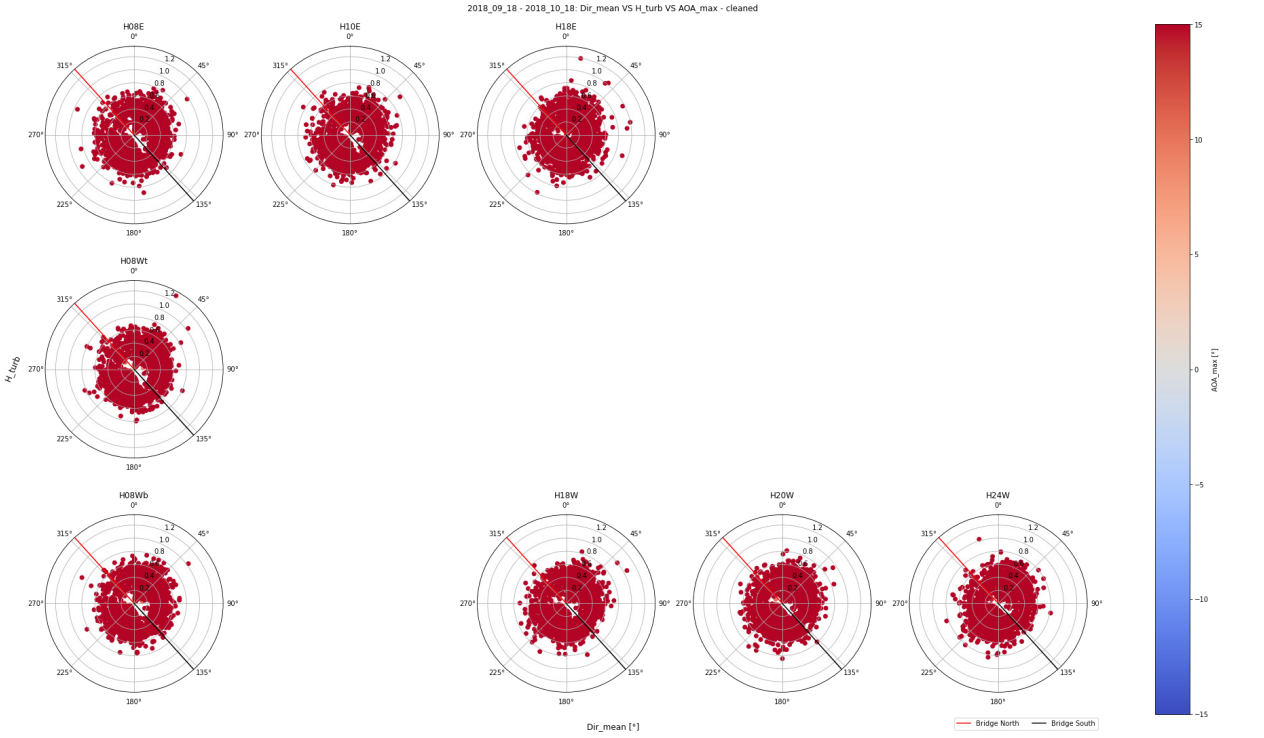

2018_09_18 - 2018_10_18: Dir_mean VS AOA_mean VS W_turb - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
AOA_mean [°]
Dir_mean [°]
W_turb
Bridge North
Bridge South

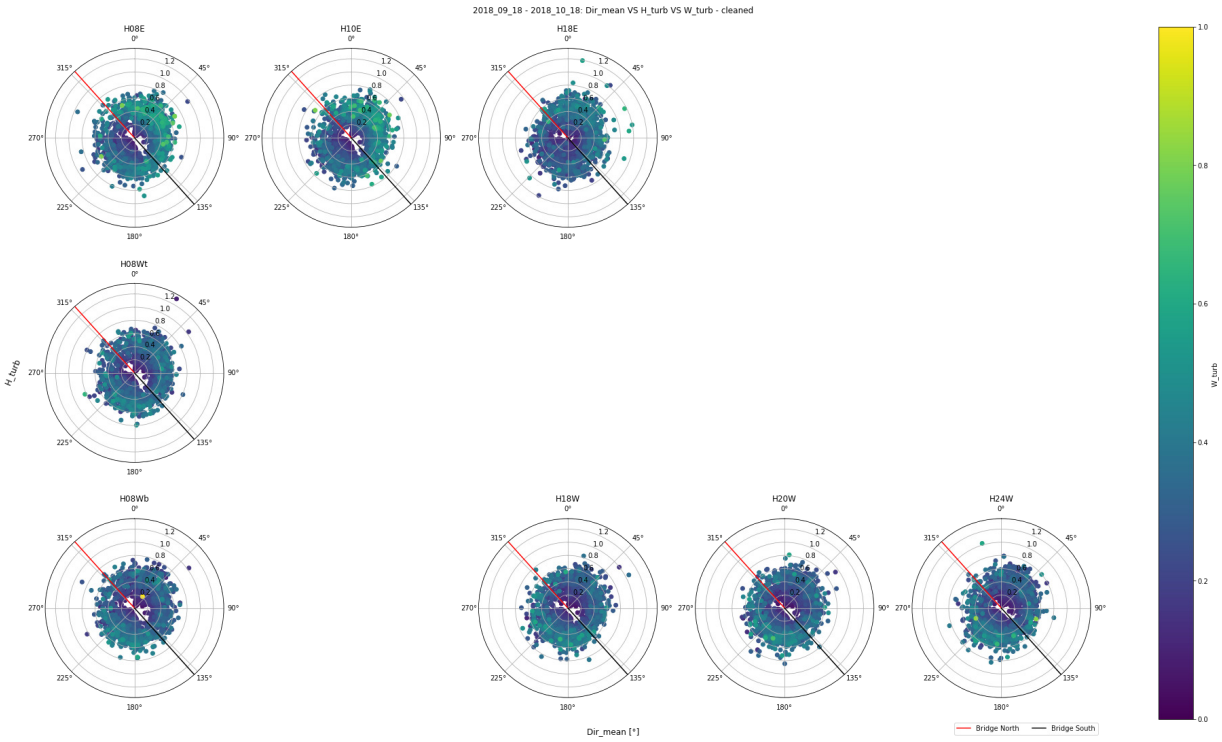

2018_09_18 - 2018_10_18: Dir_mean VS H_turb VS AOA_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_turb
Dir_mean [°]
AOA_mean [°]
Bridge North
Bridge South


2018_09_18 - 2018_10_18: Dir_mean VS H_turb VS H_mean - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_turb
Dir_mean [°]
H_mean [m/s]
Bridge North
Bridge South


2018_09_18 - 2018_10_18: Dir_mean VS H_turb VS AOA_std - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_turb
Dir_mean [°]
AOA_std [°]
Bridge North
Bridge South


2018_09_18 - 2018_10_18: Dir_mean VS H_turb VS AOA_min - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_turb
Dir_mean [°]
AOA_min [°]
Bridge North
Bridge South


2018_09_18 - 2018_10_18: Dir_mean VS H_turb VS AOA_max - cleaned
H08E
H10E
H18E
H08Wt
H08Wb
H18W
H20W
H24W
H_turb
Dir_mean [°]
AOA_max [°]
Bridge North
Bridge South



Boxplots

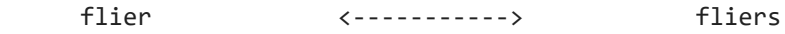
Aside from *splitting the sensors* in the different types of visualisations it is also possible to create *boxplots* to compare data from different sensors using the `boxplot` method.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
```

As the `boxplot` method is based on the function with the same name from the *matplotlib* library, using `help(plt.boxplot)` to refer to the documentation can help to understand this visualisation

In [ ]:
```
help(plt.boxplot)
```

```
Help on function boxplot in module matplotlib.pyplot:

boxplot(x, notch=None, sym=None, vert=None, whis=None, positions=None, widths=None,
patch_artist=None, bootstrap=None, usermedians=None, conf_intervals=None, meanline=N
one, showmeans=None, showcaps=None, showbox=None, showfliers=None, boxprops=None, la
bels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None, whisker
props=None, manage_ticks=True, autorange=False, zorder=None, *, data=None)
    Draw a box and whisker plot.

    The box extends from the first quartile (Q1) to the third
    quartile (Q3) of the data, with a line at the median.  The
    whiskers extend from the box by 1.5x the inter-quartile range
    (IQR).  Flier points are those past the end of the whiskers.
    See https://en.wikipedia.org/wiki/Box_plot for reference.

    .. code-block:: none

                  Q1-1.5IQR   Q1   median  Q3   Q3+1.5IQR
                               |-----:-----|
               o      |--------|     :     |--------|    o  o
                               |-----:-----|
```



Parameters
---
x : Array or a sequence of vectors.
    The input data. If a 2D array, a boxplot is drawn for each column
    in *x*. If a sequence of 1D arrays, a boxplot is drawn for each
    array in *x*.

notch : bool, default: False
    Whether to draw a notched boxplot (`True`), or a rectangular
    boxplot (`False`). The notches represent the confidence interval
    (CI) around the median. The documentation for *bootstrap*
    describes how the locations of the notches are computed by
    default, but their locations may also be overridden by setting the
    conf_intervals parameter.

    .. note::

        In cases where the values of the CI are less than the
        lower quartile or greater than the upper quartile, the
        notches will extend beyond the box, giving it a
        distinctive "flipped" appearance. This is expected
        behavior and consistent with other statistical
        visualization packages.

sym : str, optional
    The default symbol for flier points. An empty string ('') hides
    the fliers. If `None`, then the fliers default to 'b+'. More
    control is provided by the *flierprops* parameter.

vert : bool, default: True
    If `True`, draws vertical boxes.
    If `False`, draw horizontal boxes.

whis : float or (float, float), default: 1.5
    The position of the whiskers.

    If a float, the lower whisker is at the lowest datum above
    ``Q1 - whis*(Q3-Q1)``, and the upper whisker at the highest datum
    below ``Q3 + whis*(Q3-Q1)``, where Q1 and Q3 are the first and
    third quartiles. The default value of ``whis = 1.5`` corresponds
    to Tukey's original definition of boxplots.

    If a pair of floats, they indicate the percentiles at which to
    draw the whiskers (e.g., (5, 95)). In particular, setting this to
    (0, 100) results in whiskers covering the whole range of the data.

    In the edge case where ``Q1 == Q3``, *whis* is automatically set
    to (0, 100) (cover the whole range of the data) if *autorange* is
    True.

    Beyond the whiskers, data are considered outliers and are plotted
    as individual points.

bootstrap : int, optional
    Specifies whether to bootstrap the confidence intervals
    around the median for notched boxplots. If *bootstrap* is
    None, no bootstrapping is performed, and notches are
    calculated using a Gaussian-based asymptotic approximation
    (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and
    Kendall and Stuart, 1967). Otherwise, bootstrap specifies

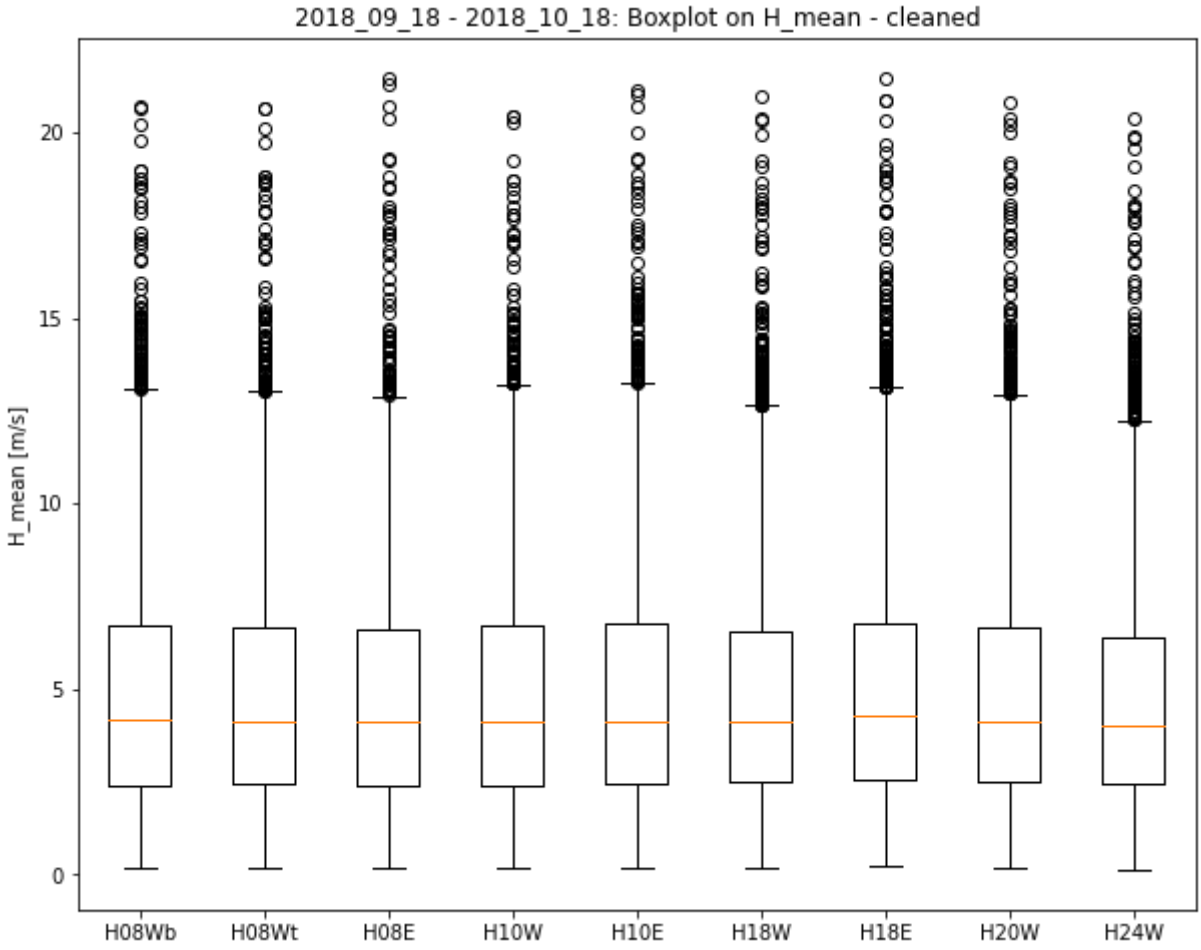
```
    the number of times to bootstrap the median to determine its
    95% confidence intervals. Values between 1000 and 10000 are
    recommended.

usermedians : 1D array-like, optional
    A 1D array-like of length ``len(x)``.  Each entry that is not
    `None` forces the value of the median for the corresponding
    dataset.  For entries that are `None`, the medians are computed
    by Matplotlib as normal.

conf_intervals : array-like, optional
    A 2D array-like of shape ``(len(x), 2)``.  Each entry that is not
    None forces the location of the corresponding notch (which is
    only drawn if *notch* is `True`).  For entries that are `None`,
    the notches are computed by the method specified by the other
    parameters (e.g., *bootstrap*).

positions : array-like, optional
    The positions of the boxes. The ticks and limits are
    automatically set to match the positions. Defaults to
    ``range(1, N+1)`` where N is the number of boxes to be drawn.

widths : float or array-like
    The widths of the boxes.  The default is 0.5, or ``0.15*(distance
    between extreme positions)``, if that is smaller.

patch_artist : bool, default: False
    If `False` produces boxes with the Line2D artist. Otherwise,
    boxes are drawn with Patch artists.

labels : sequence, optional
    Labels for each dataset (one per dataset).

manage_ticks : bool, default: True
    If True, the tick locations and labels will be adjusted to match
    the boxplot positions.

autorange : bool, default: False
    When `True` and the data are distributed such that the 25th and
    75th percentiles are equal, *whis* is set to (0, 100) such
    that the whisker ends are at the minimum and maximum of the data.

meanline : bool, default: False
    If `True` (and *showmeans* is `True`), will try to render the
    mean as a line spanning the full width of the box according to
    *meanprops* (see below).  Not recommended if *shownotches* is also
    True.  Otherwise, means will be shown as points.

zorder : float, default: ``Line2D.zorder = 2``
    The zorder of the boxplot.

Returns
-------
dict
  A dictionary mapping each component of the boxplot to a list
  of the `.Line2D` instances created. That dictionary has the
  following keys (assuming vertical boxplots):

  - ``boxes``: the main body of the boxplot showing the
    quartiles and the median's confidence intervals if
    enabled.

  - ``medians``: horizontal lines at the median of each box.
```

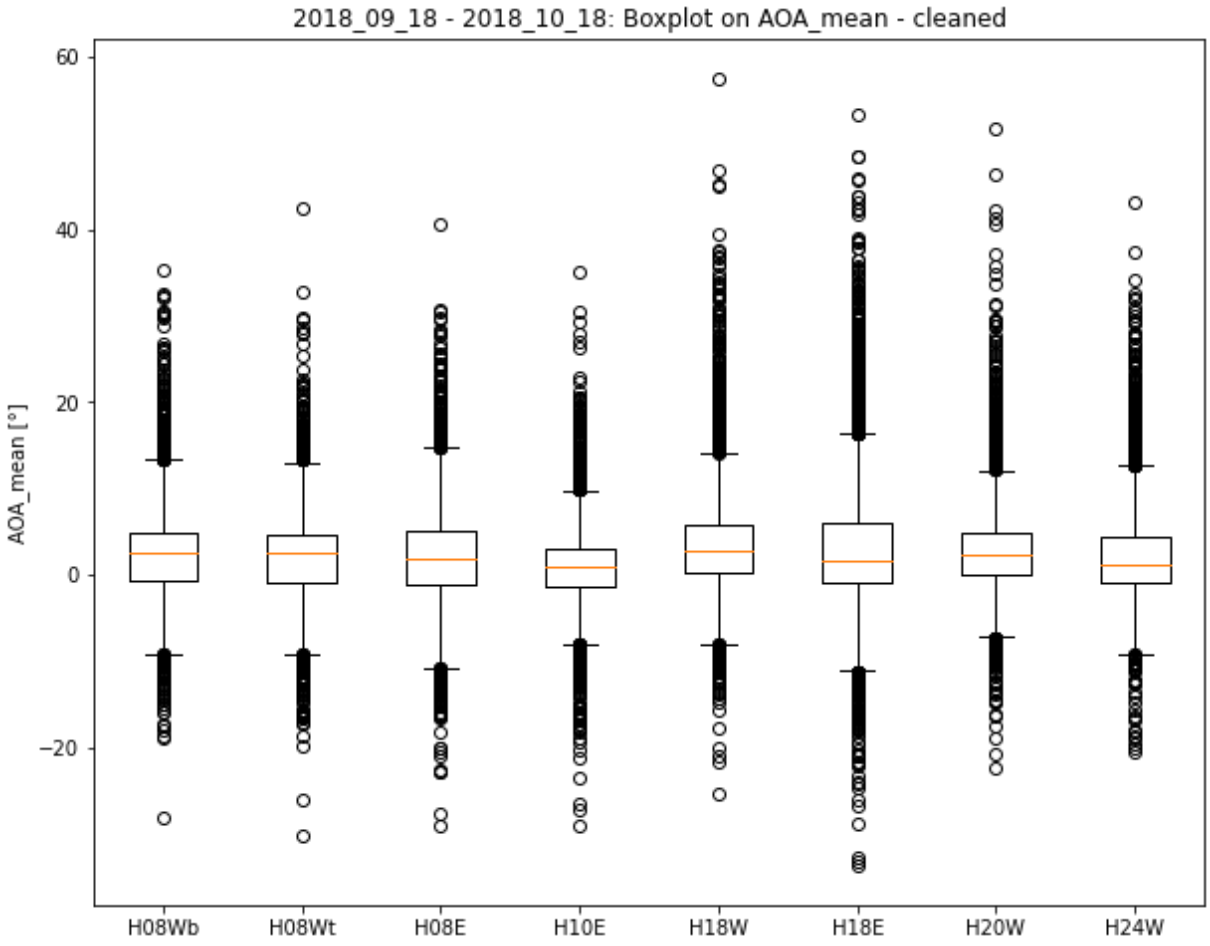
```
      - ``whiskers``: the vertical lines extending to the most
        extreme, non-outlier data points.

      - ``caps``: the horizontal lines at the ends of the
        whiskers.

      - ``fliers``: points representing data that extend beyond
        the whiskers (fliers).

      - ``means``: points or lines representing the means.

    Other Parameters
    ----------------
    showcaps : bool, default: True
        Show the caps on the ends of whiskers.
    showbox : bool, default: True
        Show the central box.
    showfliers : bool, default: True
        Show the outliers beyond the caps.
    showmeans : bool, default: False
        Show the arithmetic means.
    capprops : dict, default: None
        The style of the caps.
    boxprops : dict, default: None
        The style of the box.
    whiskerprops : dict, default: None
        The style of the whiskers.
    flierprops : dict, default: None
        The style of the fliers.
    medianprops : dict, default: None
        The style of the median.
    meanprops : dict, default: None
        The style of the mean.
    data : indexable object, optional
        If given, all parameters also accept a string ``s``, which is
        interpreted as ``data[s]`` (unless this raises an exception).

    See Also
    --------
    violinplot : Draw an estimate of the probability density function.
```

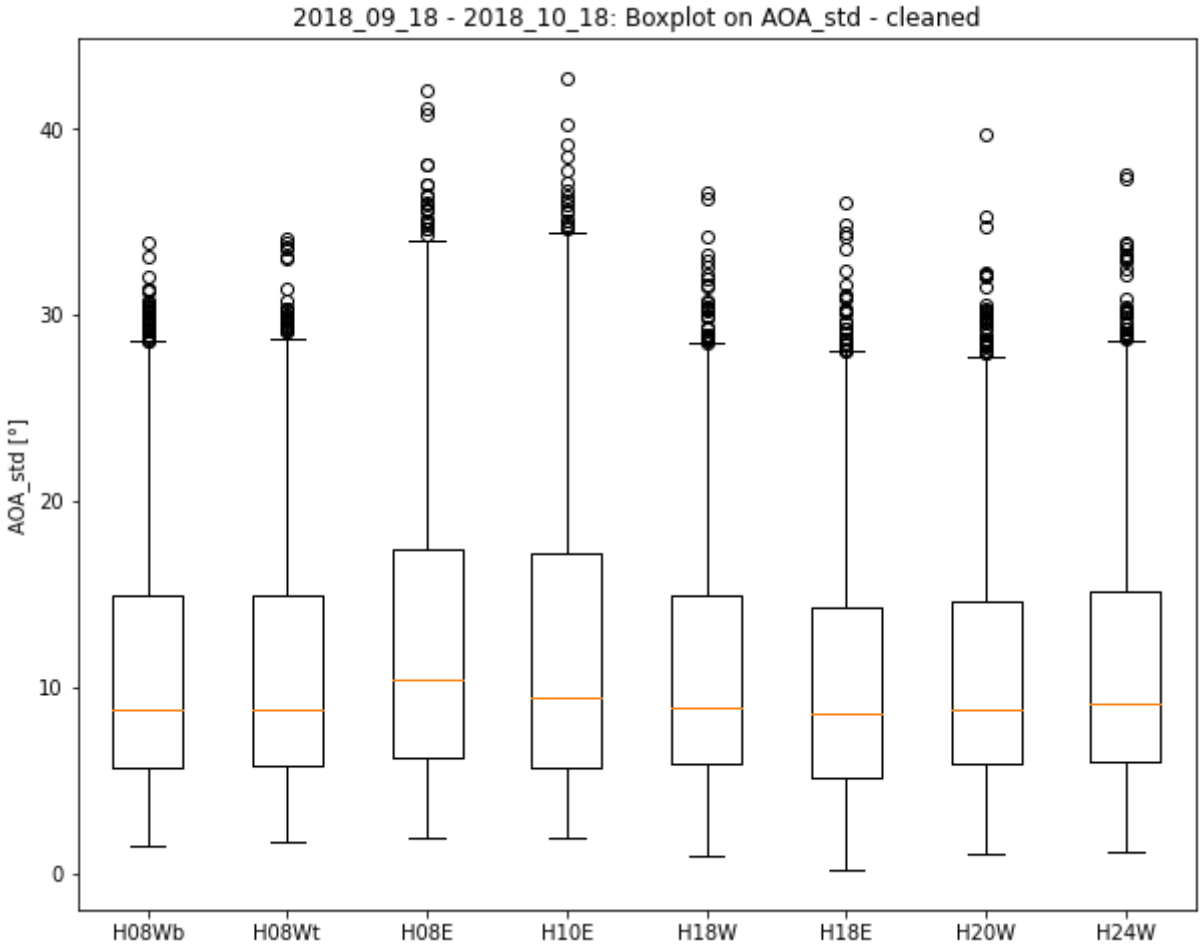
Below are some examples for this type of visualisation, using the same batch-processing technique as before.

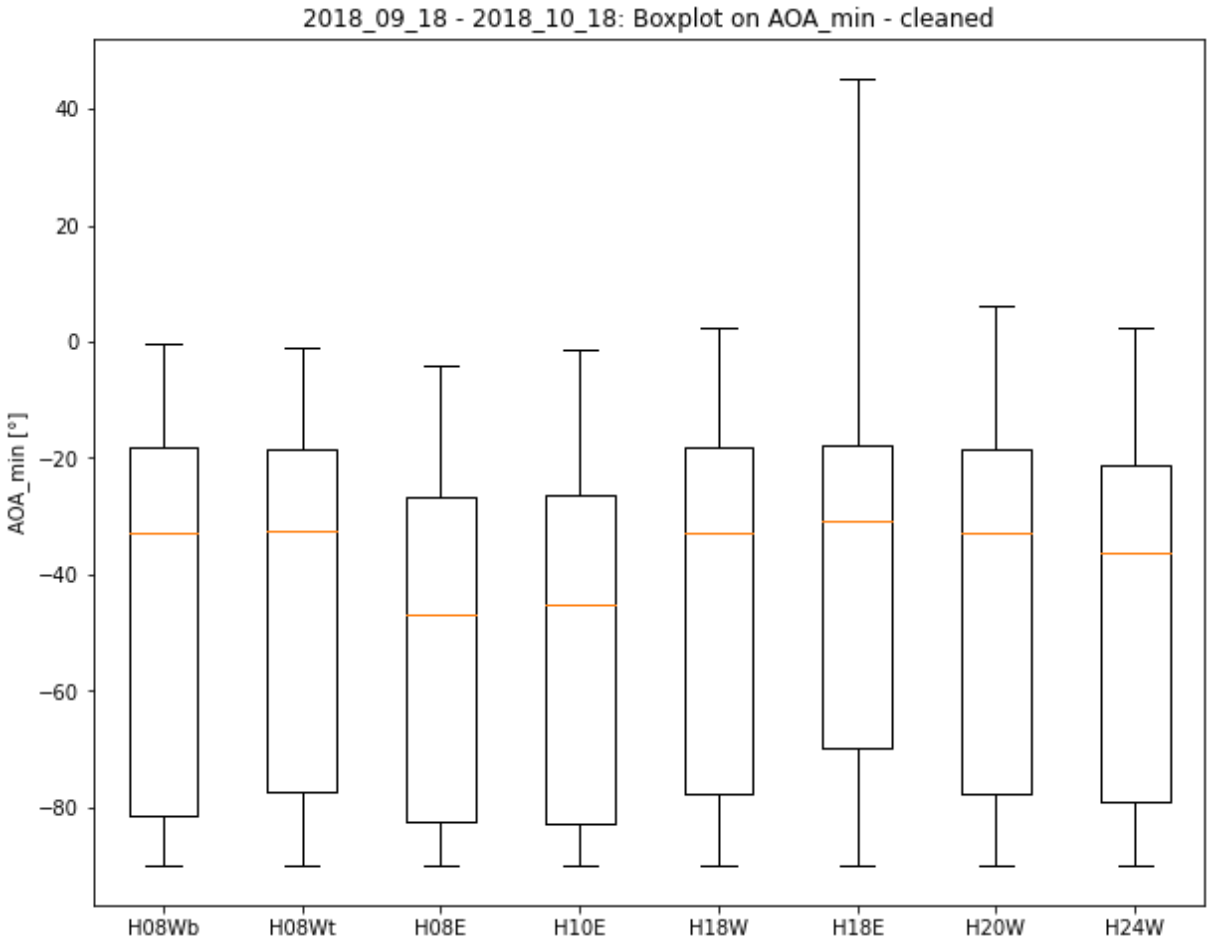
In [ ]:

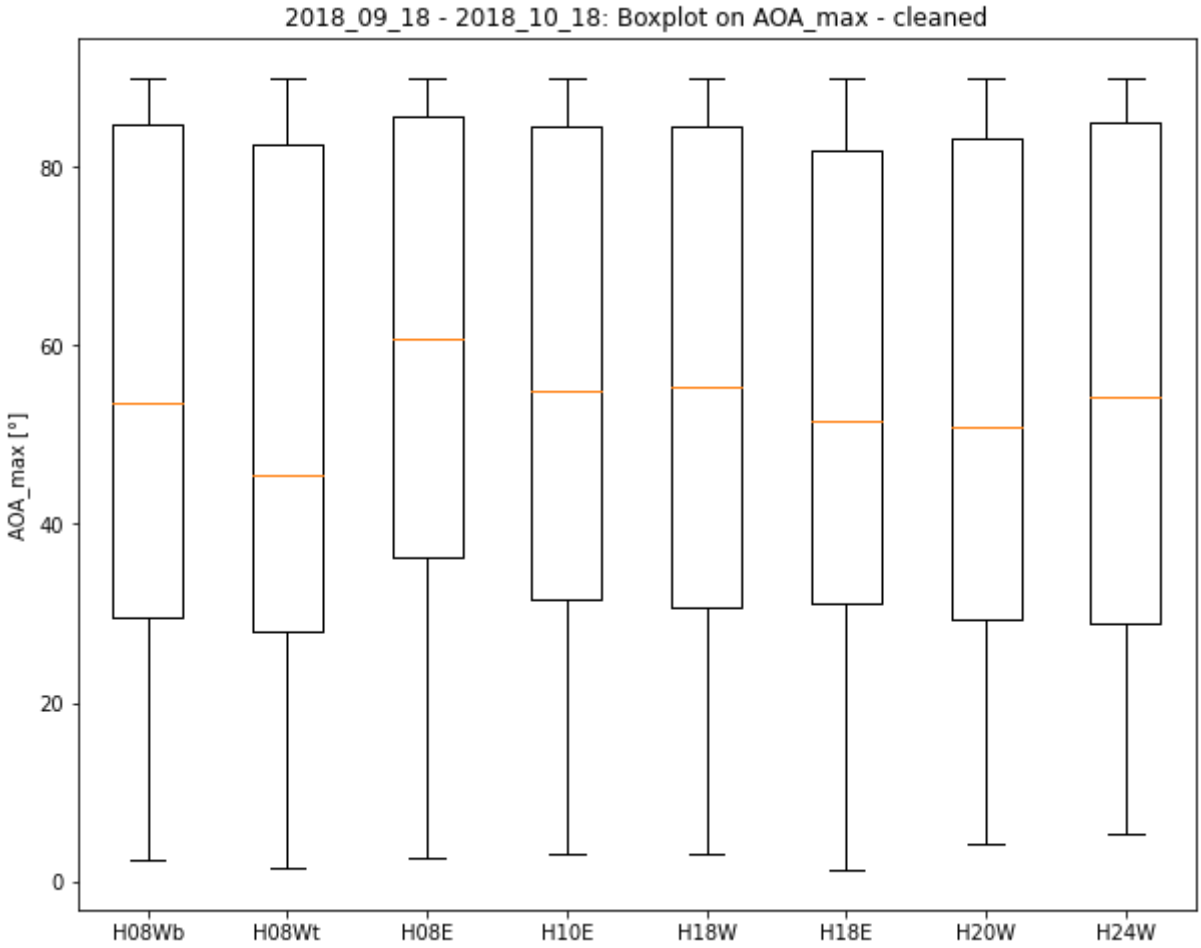
```
keys_of_interest =['H_mean','AOA_mean','AOA_std','AOA_min','AOA_max','H_turb']
for key in keys_of_interest:
    LFB.boxplot(LFB.anemo_cleaned[key].T[filter_idxs].T,labels=LFB.anemo_names[LFB.a
```

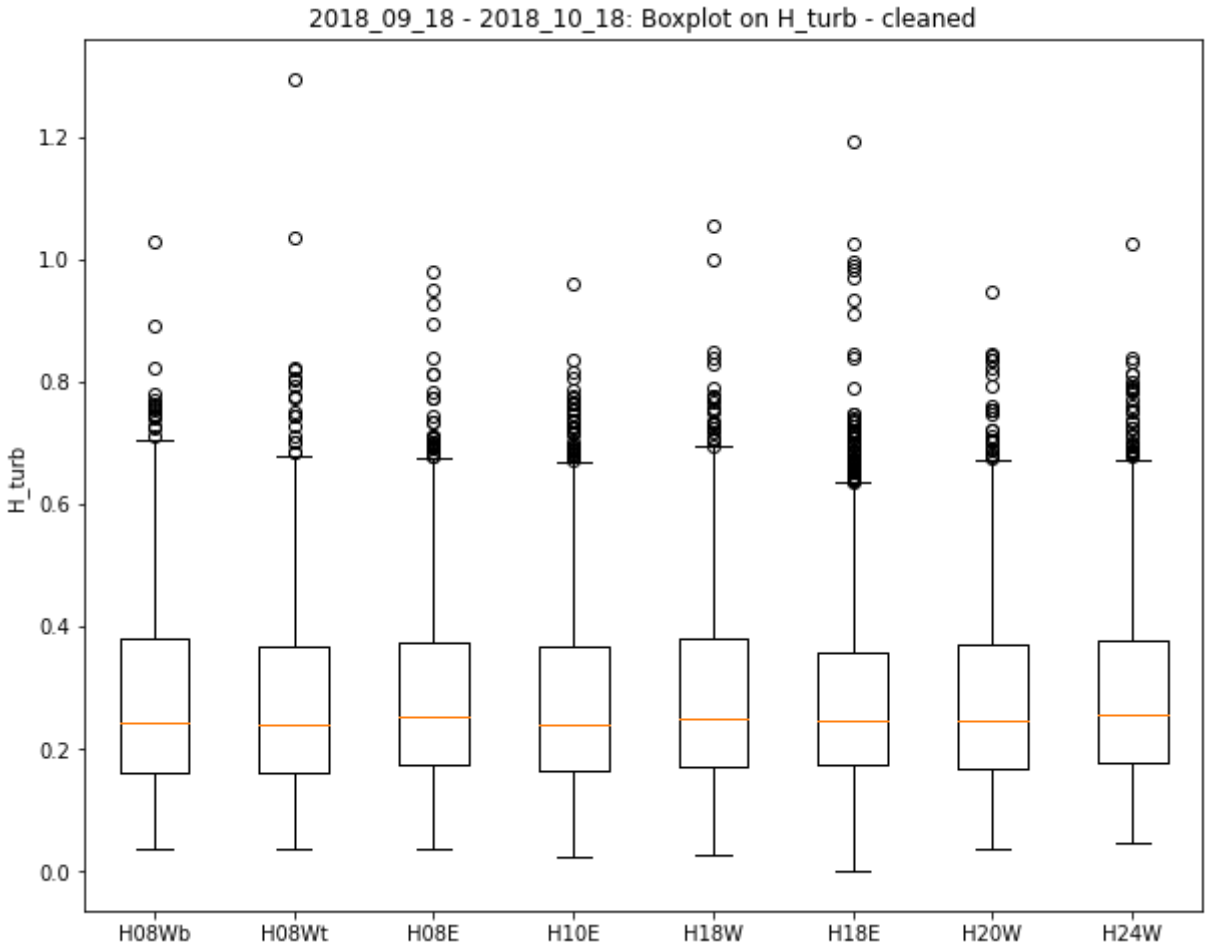

2018_09_18 - 2018_10_18: Boxplot on H_mean - cleaned
H_mean [m/s]
0
5
10
15
20
H08Wb
H08Wt
H08E
H10W
H10E
H18W
H18E
H20W
H24W


2018_09_18 - 2018_10_18: Boxplot on AOA_mean - cleaned
AOA_mean [°]
−20
0
20
40
60
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W


2018_09_18 - 2018_10_18: Boxplot on AOA_std - cleaned
AOA_std [°]
0
10
20
30
40
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W


2018_09_18 - 2018_10_18: Boxplot on AOA_min - cleaned
AOA_min [°]
40
20
0
−20
−40
−60
−80
H08Wb
H08Wt
H08E
H10E
H18W
H18E
H20W
H24W





Scatterplots

Scatterplots can be used to visualize *bi-variat* data or even *tri-variat* data using *colored scatterplots*. These can be created using the `scatterplot` method and setting the keyword `color` to *True*.

Alot of the functionallity of this method is shared with the previously described methods and will not be explained further. For more details refer to the documentation using `help(BridgeData.scatterplot)` .

In [ ]:
```
help(BridgeData.scatterplot)
```

```
Help on function scatterplot in module __main__:

scatterplot(self, data1, data2, data3, label1='data1', label2='data2', label3='data
3', units=['[]', '[]', '[]'], title_suffix='', color=True, plot_in_order_of_color=Tr
ue, cmap='viridis', vmin=None, vmax=None, curve_fit=False, func='quadratic', upper_l
im=inf, lower_lim=-inf, bounds=([-inf, -inf, -inf], [inf, inf, inf]), res=1000, pred
=1.5, curve_fit_return=False, split_sensors=False, sensors=[], ncol=4, bridge_model=
False, Hanger_num=[], West=[], Top=[], save=False)
    Method to create scatterplots on bridge data in a consistent manner.

    Parameters:
    -----------
    data1 : array_like
        Data for x-axis.
    data2 : array_like
        Data for y-axis.
    data3 : array_like
        Data for color-axis.
    label1 : str, default 'data1'
        Label for x-axis.
    label2 : str, default 'data2'
        Label for y-axis.
    label3 : str, default 'data3'
        Label for colorbar.
    units : list of str, default ['[]','[]','[]']
        Units for x-,y- and color-axis.
    title_suffix : str, optional
        Add a suffix to the title.
    color : bool, default True
        The scatterplot is colored based on data3.
    plot_in_order_of_color : bool, default True
        Plot the datapoints of higher color ontop of datapoints of lower color.
    cmap : colormap or str, default 'viridis'
        Colormap to use for the scatterplot.
    vmin : float, optional
        Lower limit for the color-axis.
    vmax : float, optional
        Upper limit for the color-axis.
    curve_fit : bool, default False
        Fit a curve to the scatterplot. Note: Only works for single scatterplot, not
multiple.
    func : {'quadratic', 'linear'}
        Function for curve fitting.
    upper_lim : float, default np.inf
        Upper limit for data3 values.
    lower_lim : float, default -np.inf
        Lower limit for data3 values.
    bounds : tuple, default ([-np.inf,-np.inf,-np.inf],[np.inf,np.inf,np.inf])
        Bounds for curve fitting parameters.
    res : int, default 1000
        Number of points for the curve fitting.
```

```
    pred : float, default 1.5
        Value for prediction of curve fitting.
    curve_fit_return : bool, default False
        Return the parameters of the curve fit.
    split_sensors : bool, default False
        Split the scatterplot into the different sensors.
    sensors : list of str, optional
        List of sensor labels to place above seperate scatterplots if `split_sensors
` is used.
    ncol : int, default 4
        Number of columns defining the grid to place the scatterplots on if `split_s
ensors` is used and `bridge_model` is False.
    bridge_model : bool, default False
        Whether to place the scatterplots on a gird based on their location at the b
ridge if `split_sensors` is used. Uses `Hanger_num`, `West` and `Top`.
    Hanger_num : list of int, optional
        List of the hanger numbers at which each sensor is located.
    West, Top : list of bool, optional
        List of booleans defining whether each sensor is located on the West side, o
r is a top-row sensor respectively.
    save : bool, default False
        Save the scatterplot in ../images/. Note: Requires the prior existence of th
at folder.

    Returns:
    --------
    popt1, popt2, pcov1, pcov2 : array_like, optional
        If `curve_fit_return` is True, returns the parameters of the curve fit.

    See Also
    --------
    `polar_scatterplot`
```

Note that it is also possible to analyse data of different *measurement types* from **anemometers** in relation to data of different *measurement types* from **anccelerometers** in different ways to examine the **wind response** of the bridge.

Anemo VS Anemo VS Anemo - Wind characteristics

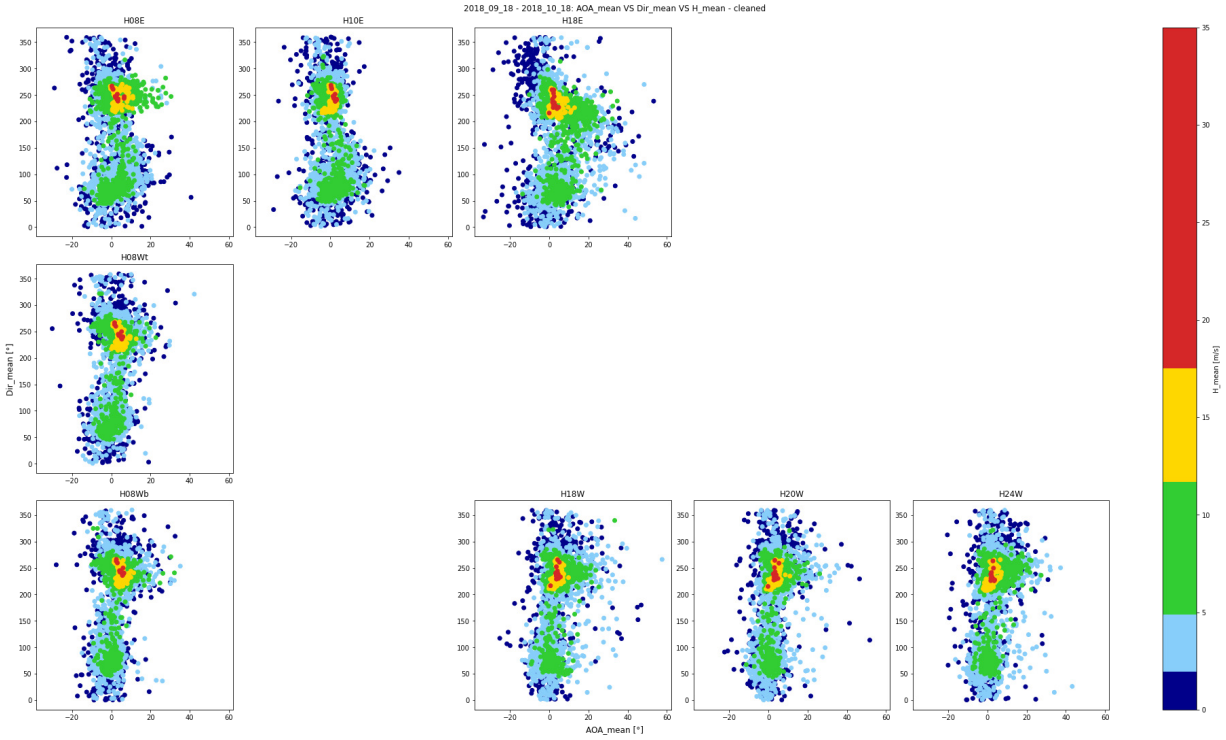
In [ ]:

```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
```

In [ ]:

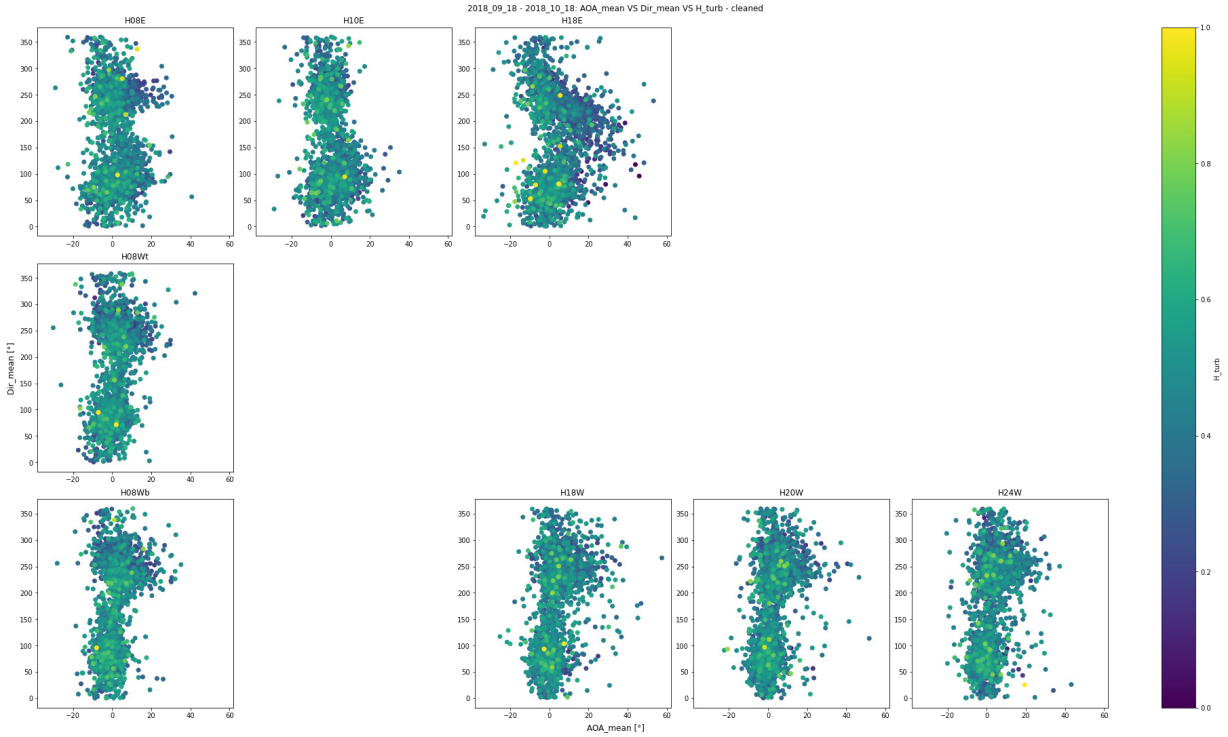
```
key1 = 'AOA_mean'
key2 = 'Dir_mean'
key3 = 'H_mean'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
```

```
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



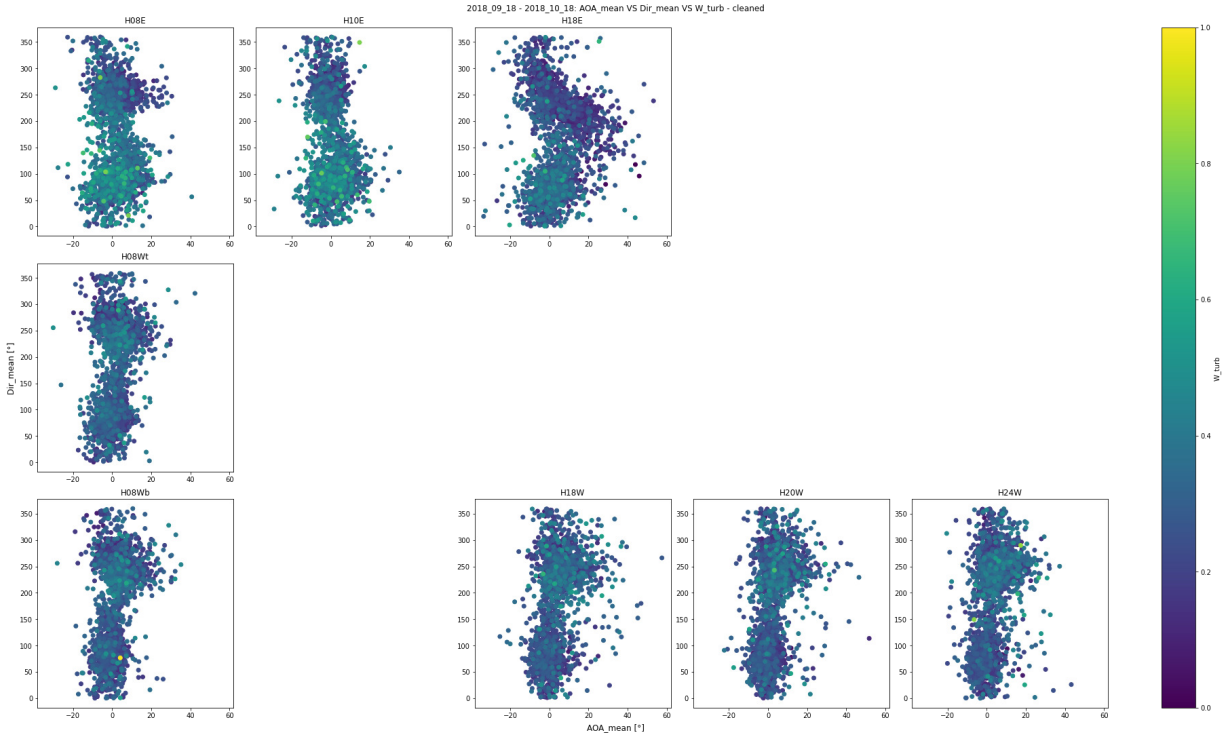
In [ ]:

```
key1 = 'AOA_mean'
key2 = 'Dir_mean'
key3 = 'H_turb'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



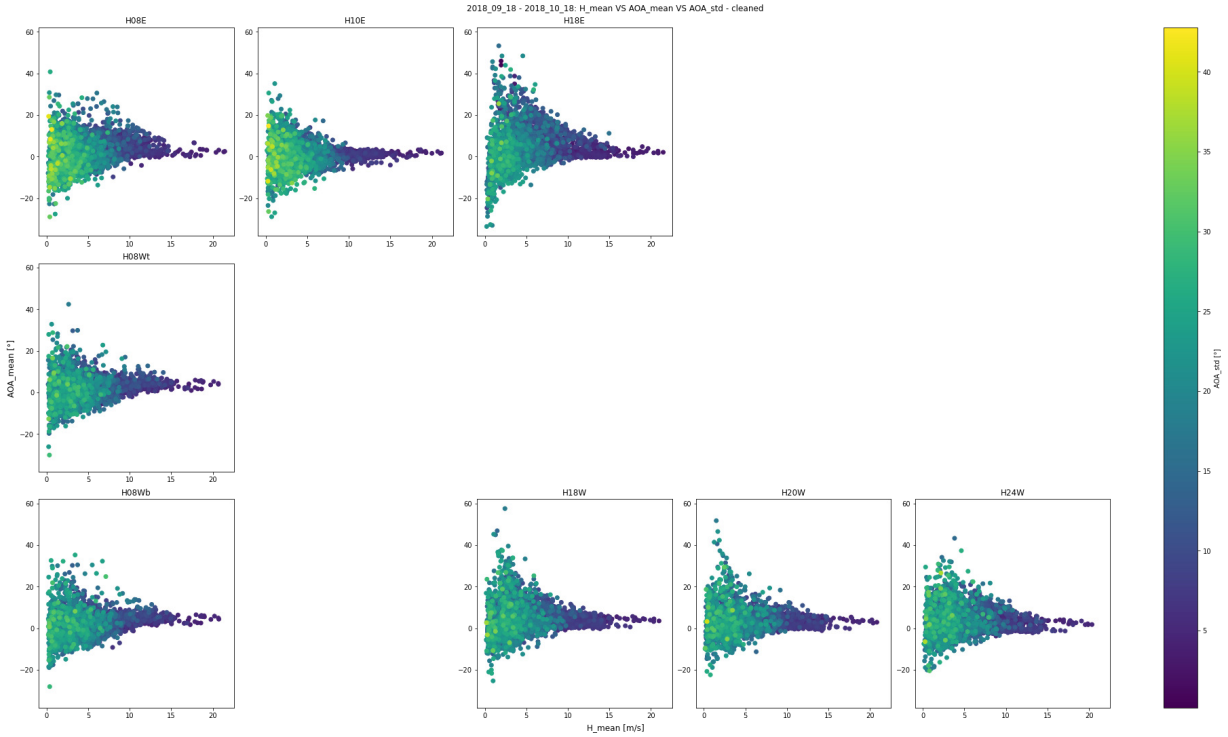
In [ ]:

```
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['W_turb'],prior_idxs=filter_idxs,l
key1 = 'AOA_mean'
key2 = 'Dir_mean'
key3 = 'W_turb'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



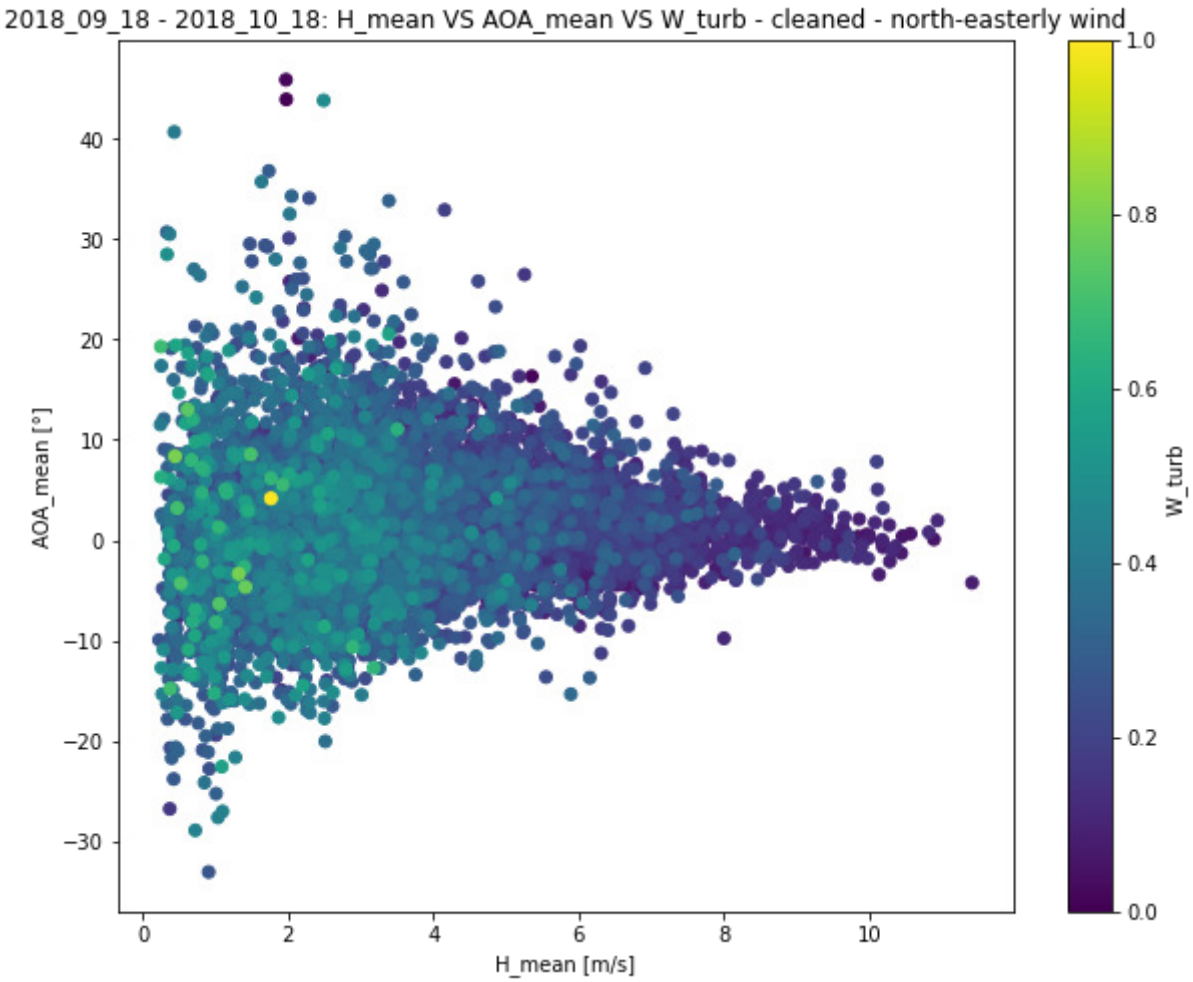
In [ ]:

```
key1 = 'H_mean'
key2 = 'AOA_mean'
key3 = 'AOA_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



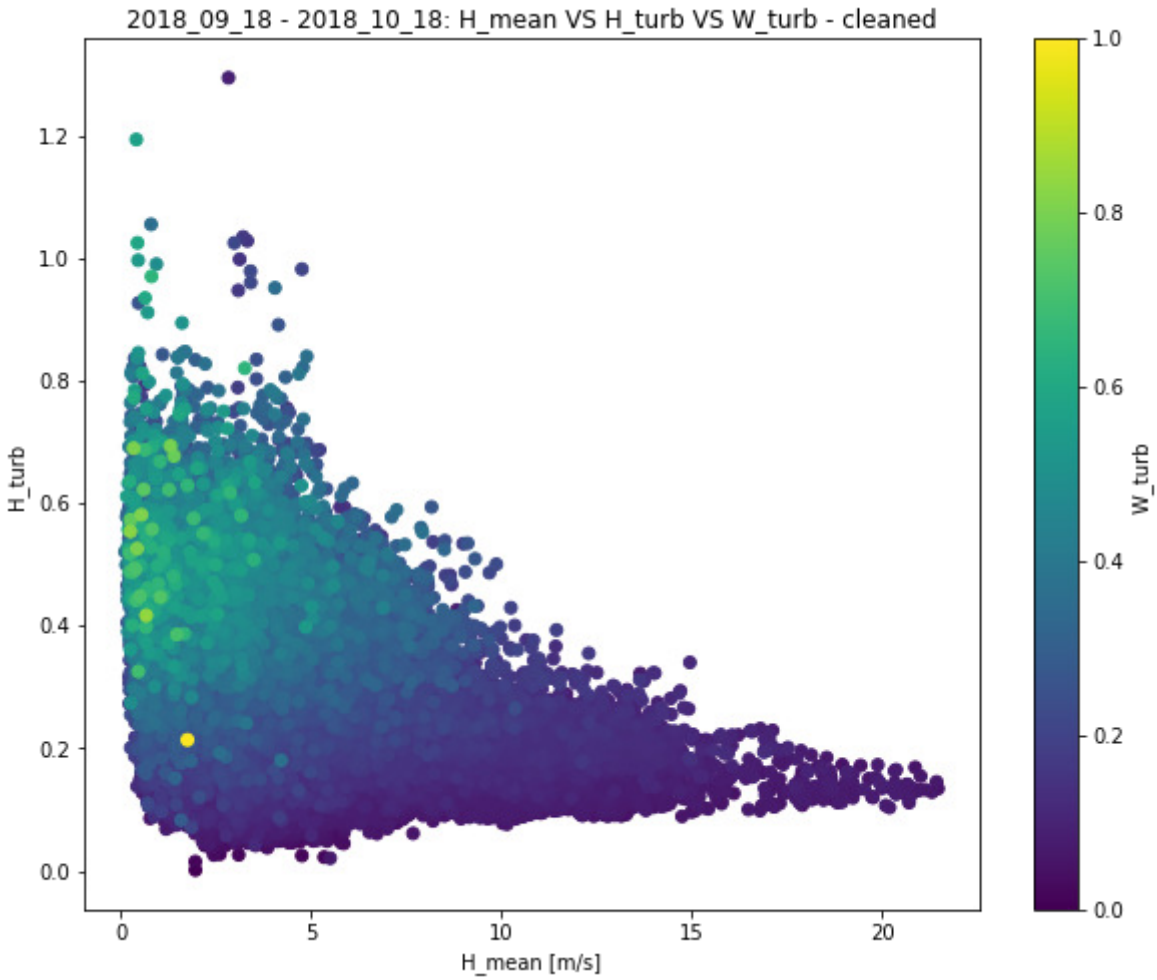
In [ ]:

```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs
filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs,l
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['W_turb'],prior_idxs=filter_idxs,l
key1 = 'H_mean'
key2 = 'AOA_mean'
key3 = 'W_turb'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



In [ ]:

```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['W_turb'],prior_idxs=filter_idxs,l
key1 = 'H_mean'
key2 = 'H_turb'
key3 = 'W_turb'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.anemo_ok_sensor_id[
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```



Anemo VS Acc VS Anemo - Wind response

Note that in the following scatterplots the data is filtered such that traffic dominated responses are ignored.

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=True,
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
```

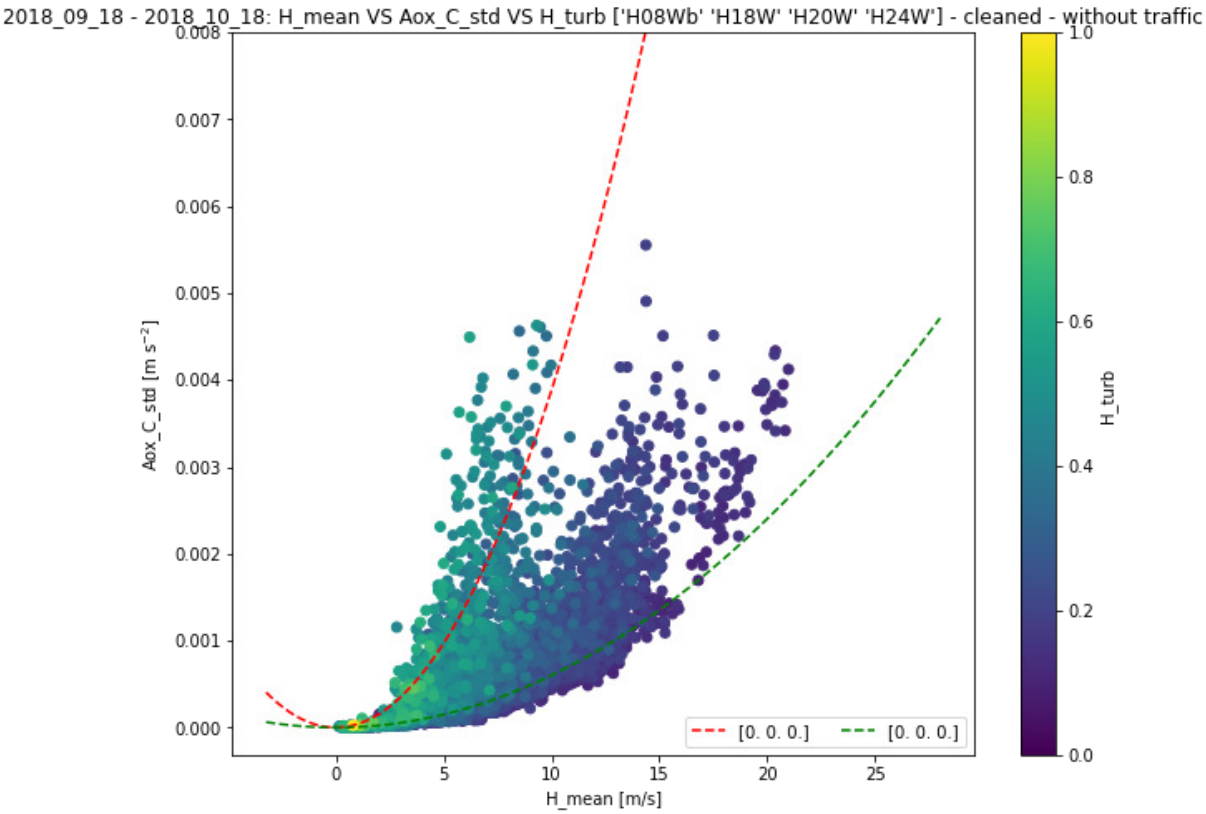
Through *batch-processing* the same plots are generated for the lateral bridge response `'Aox_C_std'`, vertical bridge response `'Aoz_C_std'` and torsional bridge response `'theta_std'`

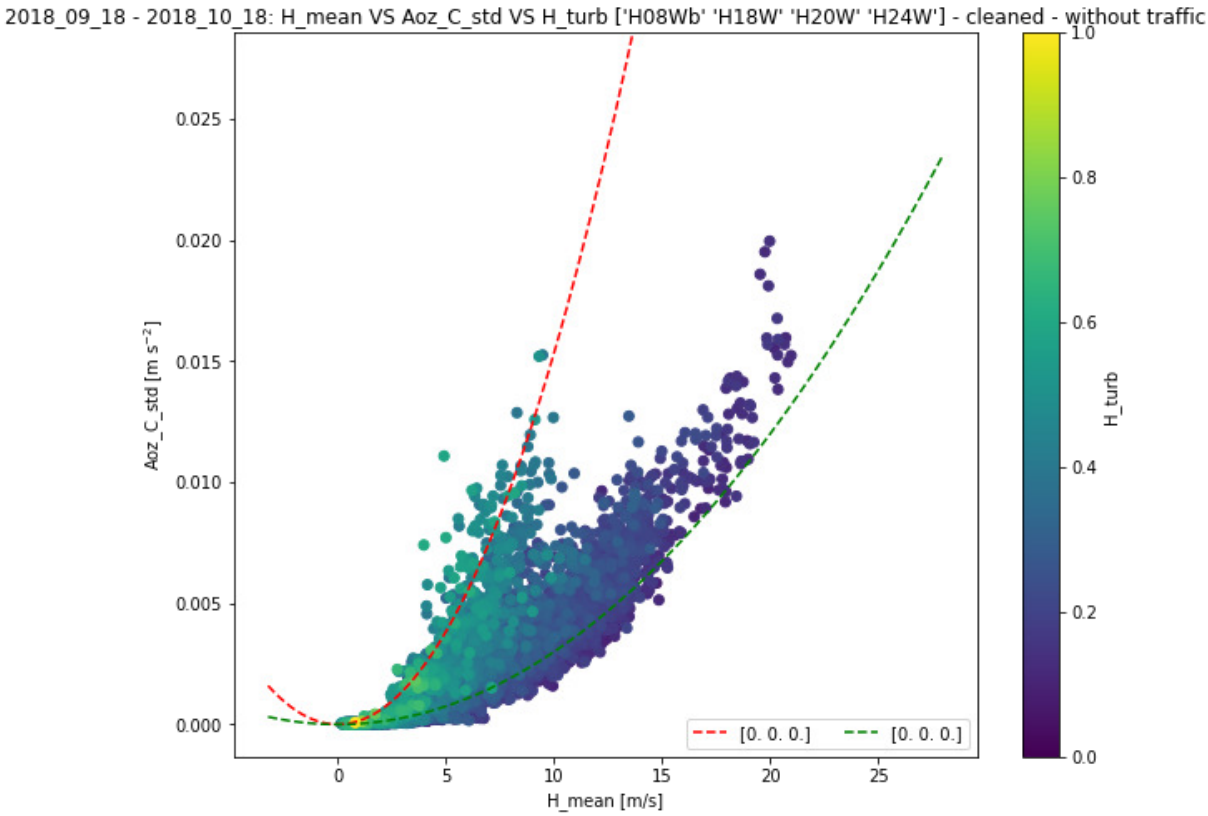
The scatterplots below show the bridge response at different mean horizontal wind velocities `'H_mean'`. The color-axis displays the horizontal turbulence intensity `'H_turb'`.

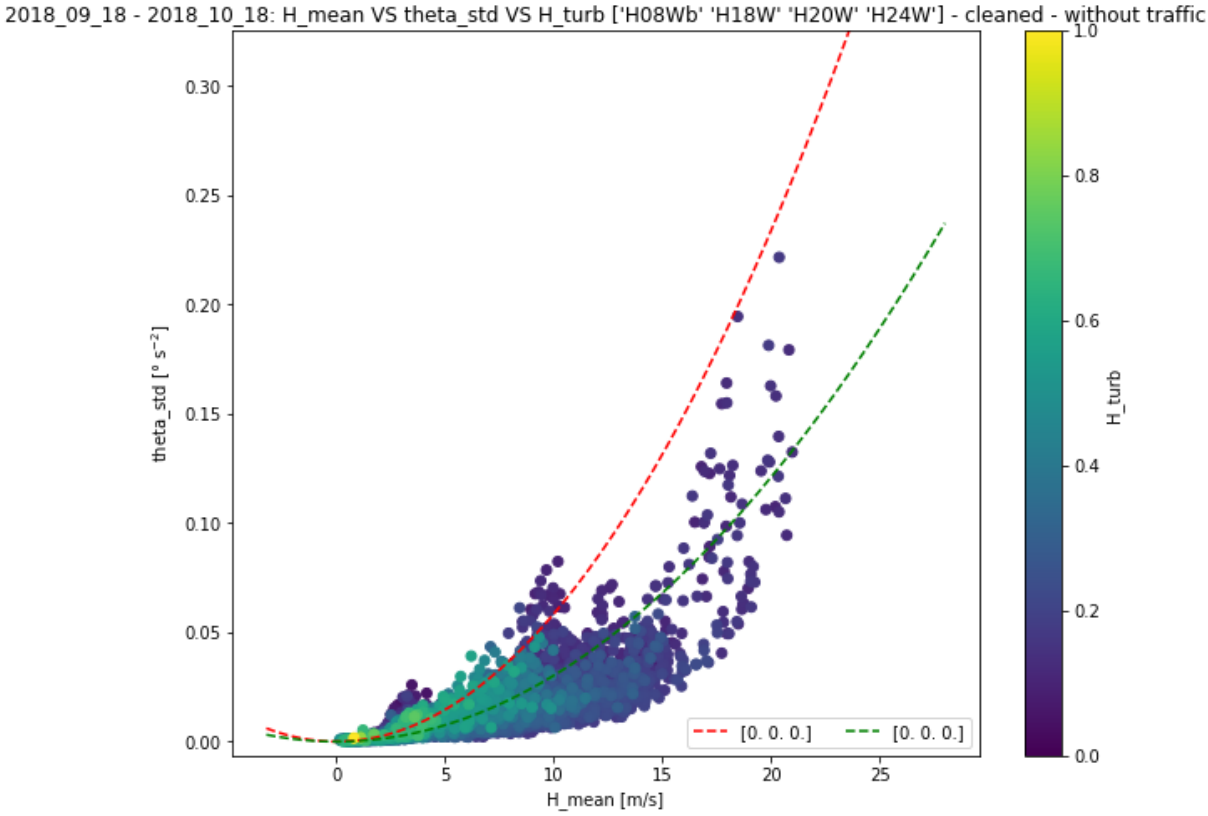
A red and green parabola are fitted to the data, limited to datapoints with turbulence intensities above 0.6 and below 0.1 respectively.

In [ ]:
```
key1 = 'H_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'H_turb'
for key2 in key2s_of_interest:
    selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
    selected_sensors_key2=['H09','H18','H24','H30']
    selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
    s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
    s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
```

```
    s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
    diff = np.abs(len(s1_ind)-len(s2_ind))
    if key3.startswith('AOA_m'):
        cmap = 'coolwarm'
        vmin = -15
        vmax = 15
    elif key3.startswith('H_m'):
        cmap = Windfinder_cmap
        vmin = 0
        vmax = 35
    elif key3.endswith('_turb'):
        cmap = 'viridis'
        vmin = 0
        vmax = 1
    else:
        cmap = 'viridis'
        vmin = None
        vmax = None
    LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned
```





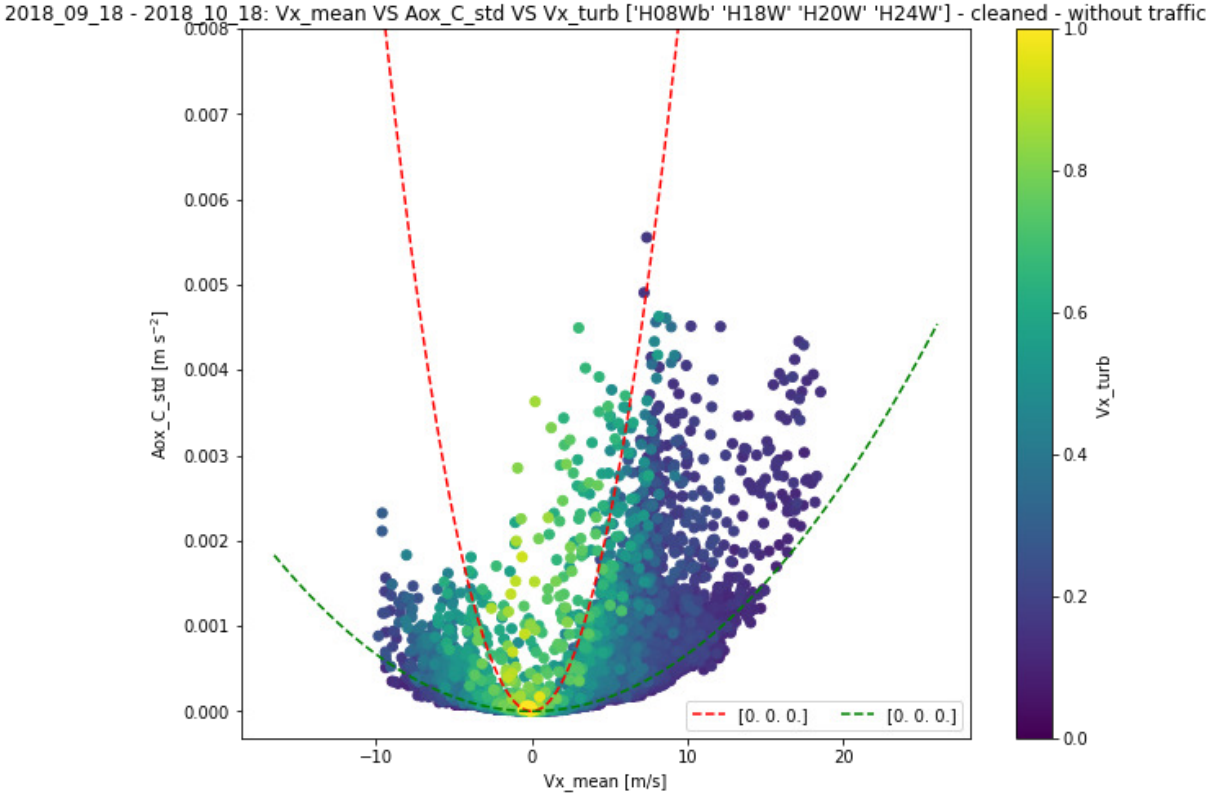


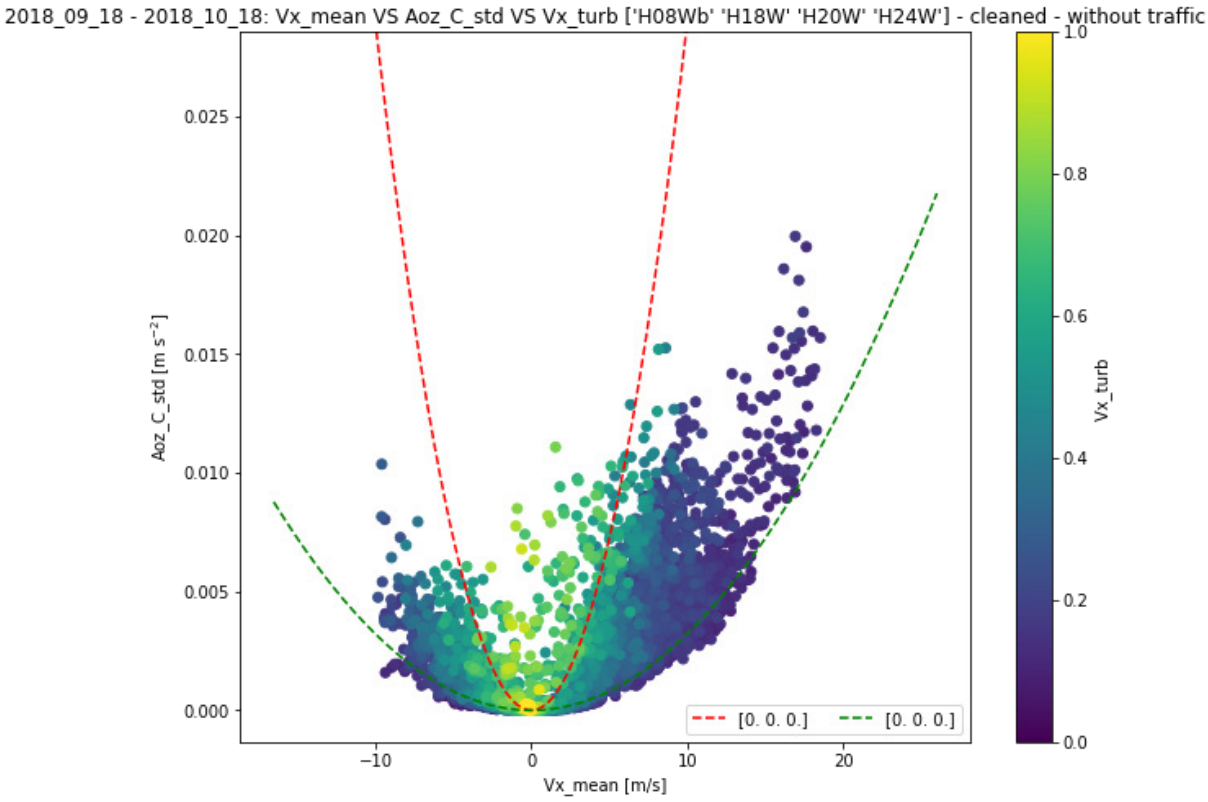
The scatterplots below only take the mean velocity `'Vx_mean'` and turbulence intensity `'Vx_turb'` in x-direction, which is perpendicular to the bridge, instead of the full horizontal component. Note that also splits the data in westerly and easterly wind directions, as the x-component is signed.

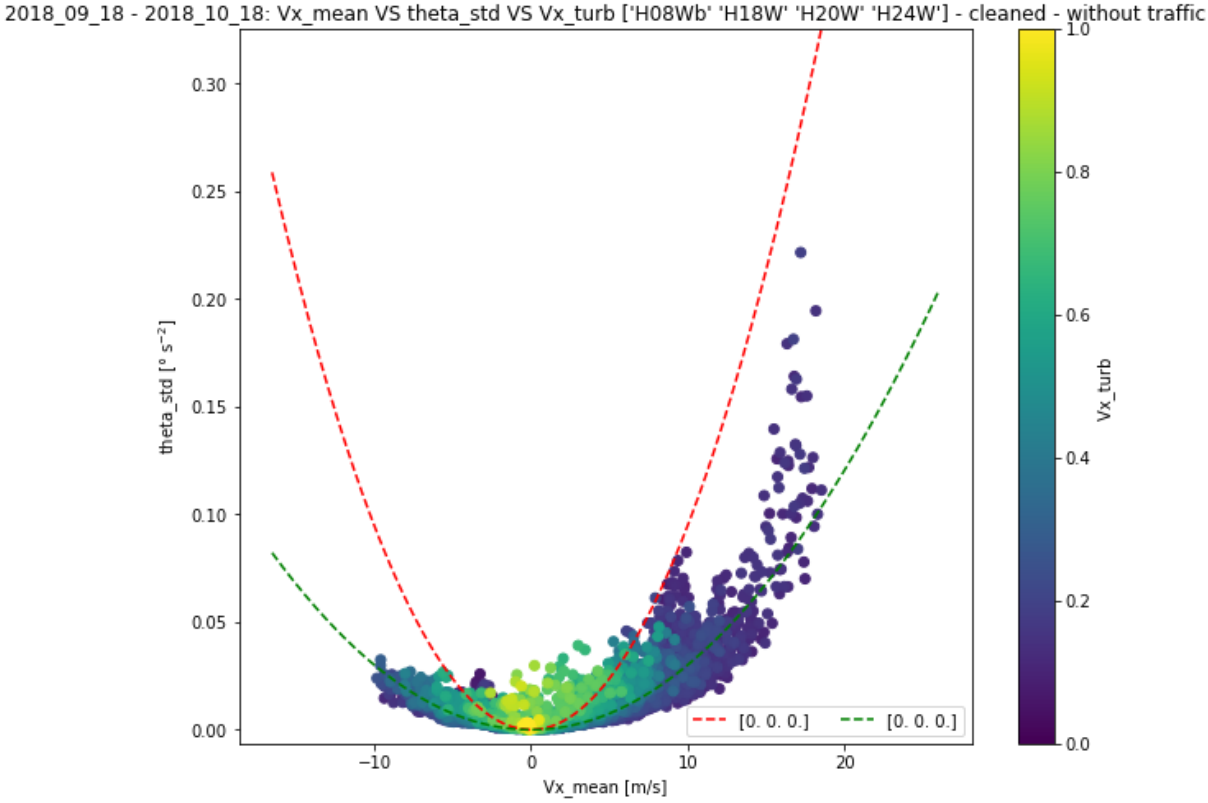
In [ ]:

```
key1 = 'Vx_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'Vx_turb'
for key2 in key2s_of_interest:
    selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
```

```
selected_sensors_key2=['H09','H18','H24','H30']
selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
diff = np.abs(len(s1_ind)-len(s2_ind))
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned
```





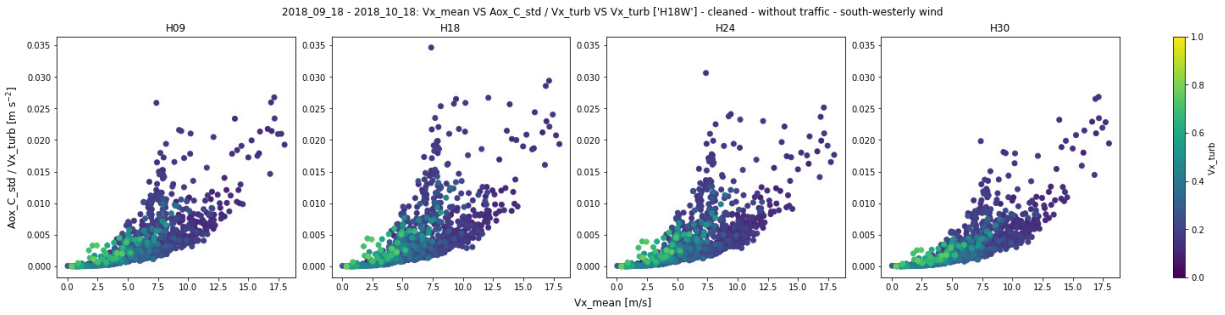


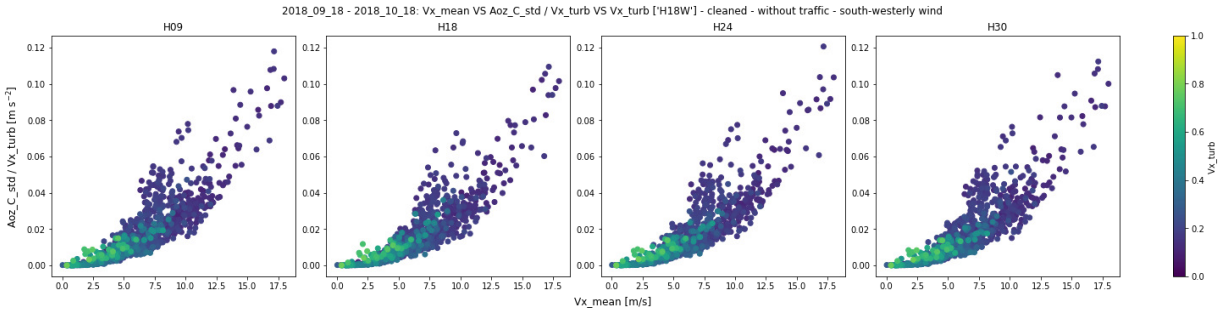
In the scatterplots below the bridge response divided by the turbulence intensity `'Vx_turb'` at different mean wind speeds in x-direction `'Vx_mean'`, measured at midspan from the downwind anemometer `'H18W'` or `'H18E'`, depending on the filtered wind direction. The color-axis is displays `'Vx_turb'`. The scatterplots are split across the accelerometer pairs `'H09'`, `'H18'`, `'H24'` and `'H30'`.

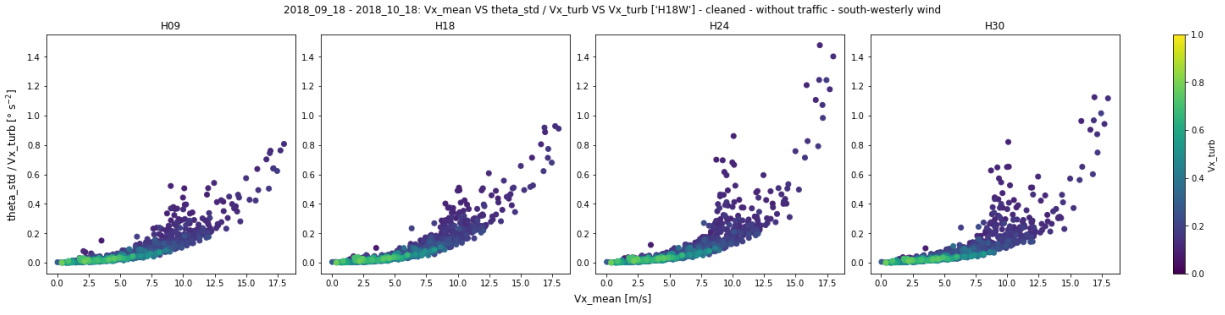
In [ ]:

```
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=None,zeros=True,mode='a
filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs,h
key1 = 'Vx_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'Vx_turb'
```

```
for key2 in key2s_of_interest:
    selected_sensors_key1=['H18W']
    selected_sensors_key2=['H09','H18','H24','H30']
    selected_sensors_key3=selected_sensors_key1
    s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
    s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
    s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
    diff = np.abs(len(s1_ind)-len(s2_ind))
    if key3.startswith('AOA_m'):
        cmap = 'coolwarm'
        vmin = -15
        vmax = 15
    elif key3.startswith('H_m'):
        cmap = Windfinder_cmap
        vmin = 0
        vmax = 35
    elif key3.endswith('_turb'):
        cmap = 'viridis'
        vmin = 0
        vmax = 1
    else:
        cmap = 'viridis'
        vmin = None
        vmax = None
    LFB.scatterplot(np.repeat(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,diff+
```



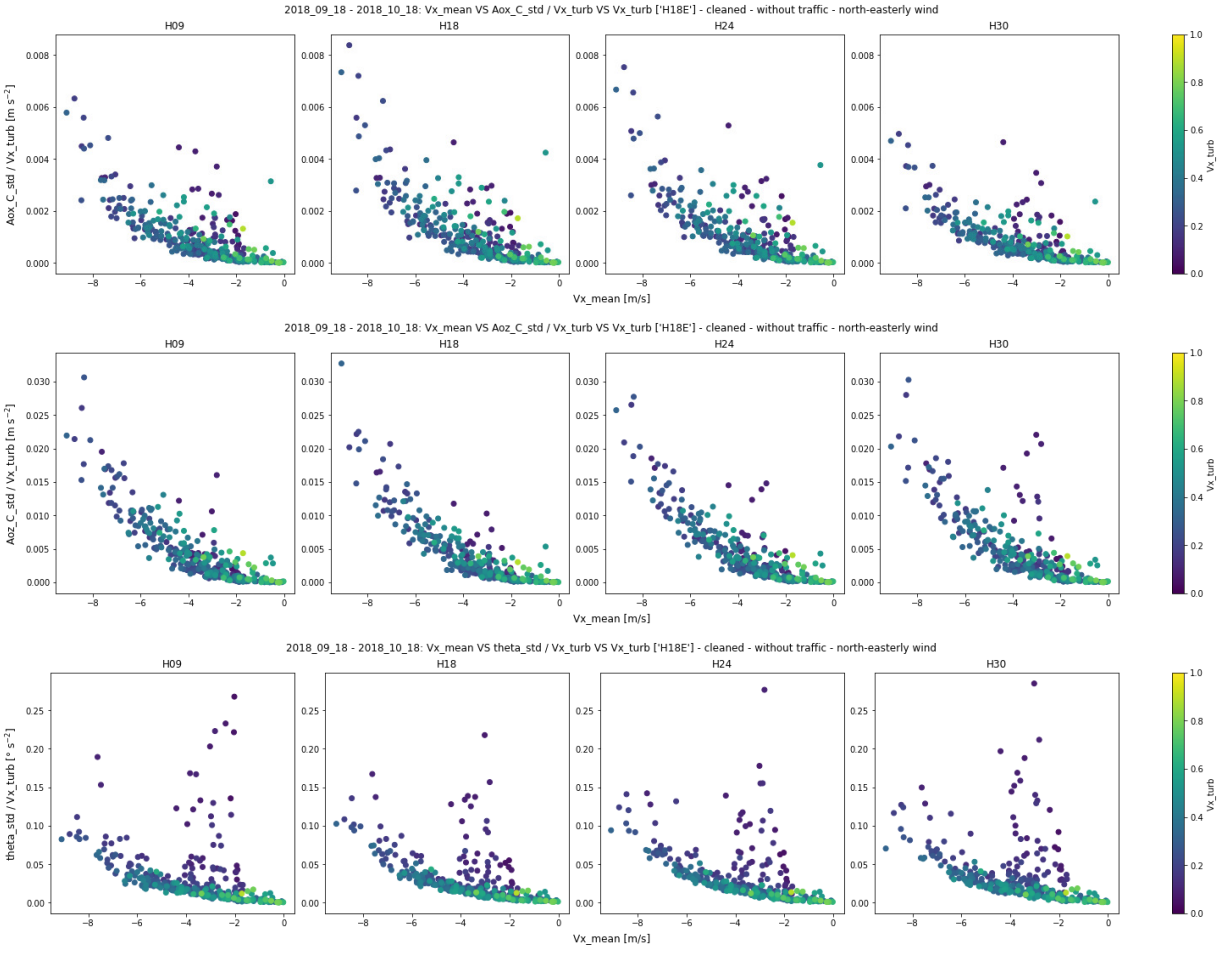




In [ ]:

```
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=None,zeros=True,mode='a
filter_idxs = LFB.filter_data(LFB.anemo_cleaned['Dir_mean'],prior_idxs=filter_idxs,l
key1 = 'Vx_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'Vx_turb'
for key2 in key2s_of_interest:
    selected_sensors_key1=['H18E']
```

```
    selected_sensors_key2=['H09','H18','H24','H30']
    selected_sensors_key3=selected_sensors_key1
    s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
    s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
    s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
    diff = np.abs(len(s1_ind)-len(s2_ind))
    if key3.startswith('AOA_m'):
        cmap = 'coolwarm'
        vmin = -15
        vmax = 15
    elif key3.startswith('H_m'):
        cmap = Windfinder_cmap
        vmin = 0
        vmax = 35
    elif key3.endswith('_turb'):
        cmap = 'viridis'
        vmin = 0
        vmax = 1
    else:
        cmap = 'viridis'
        vmin = None
        vmax = None
    LFB.scatterplot(np.repeat(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,diff+
```



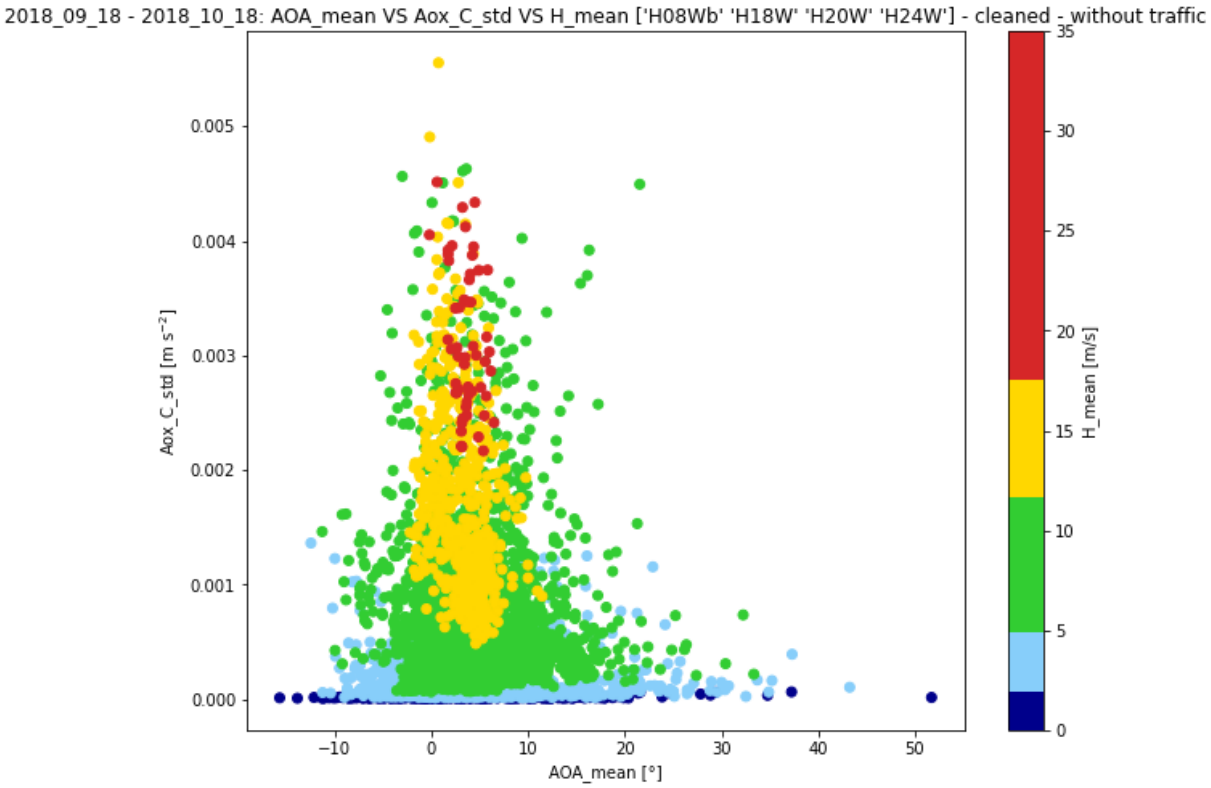
In [ ]:

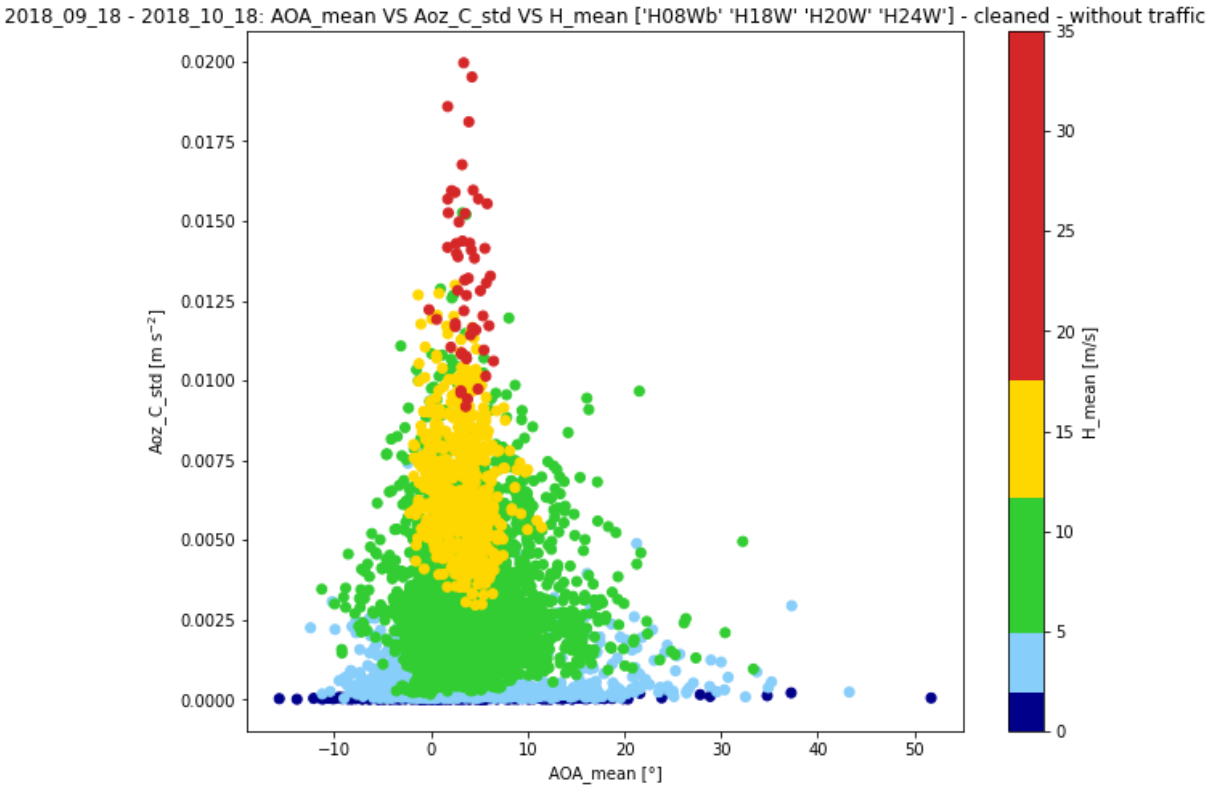
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=True,
# filter_idxs = LFB.filter_data(LFB.anemo_cleaned['H_mean'],prior_idxs=filter_idxs,h
```

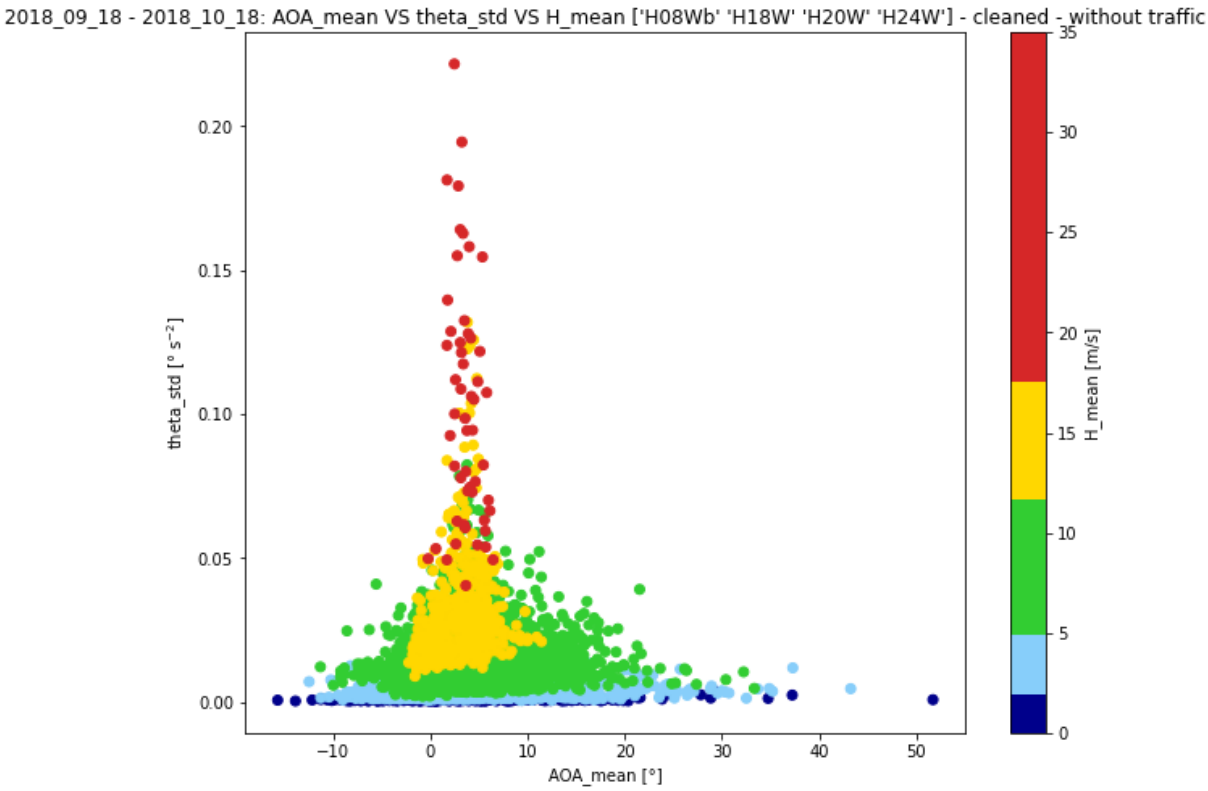
In [ ]:

```
key1 = 'AOA_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'H_mean'
for key2 in key2s_of_interest:
```

```
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H09','H18','H24','H30']
selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned
```



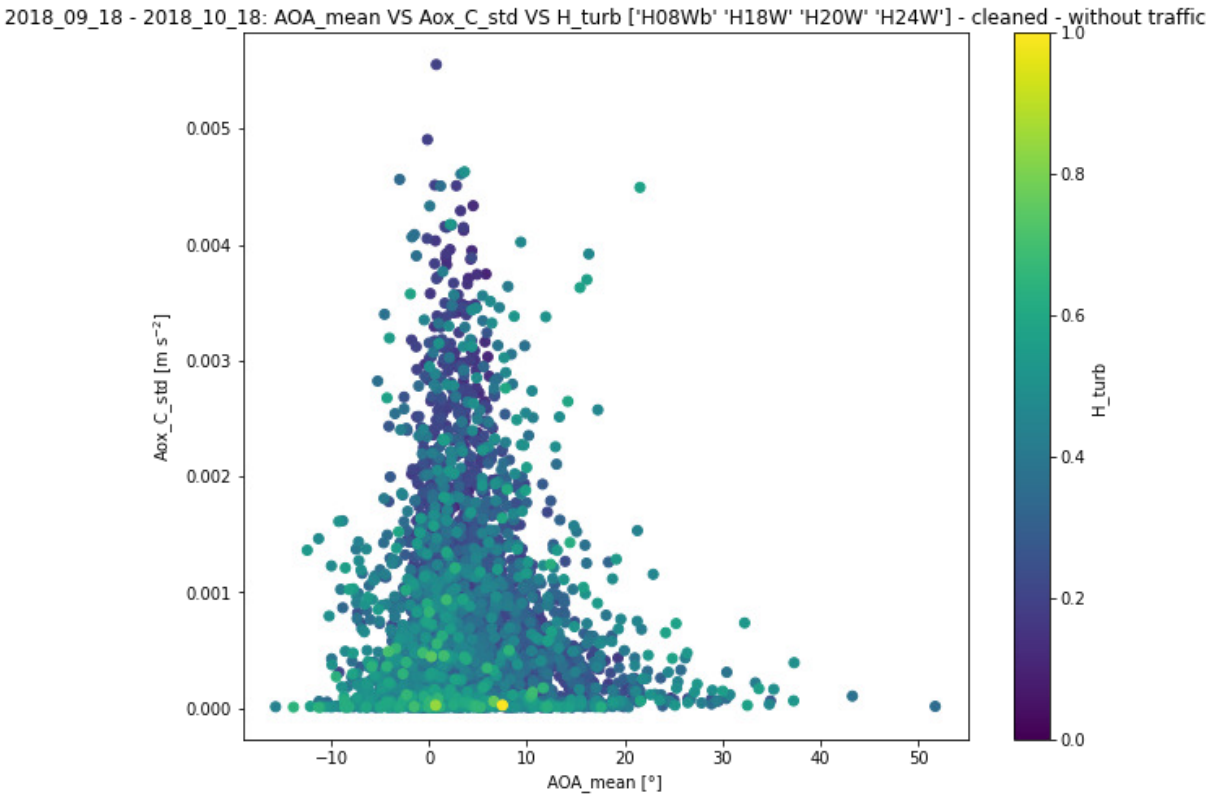


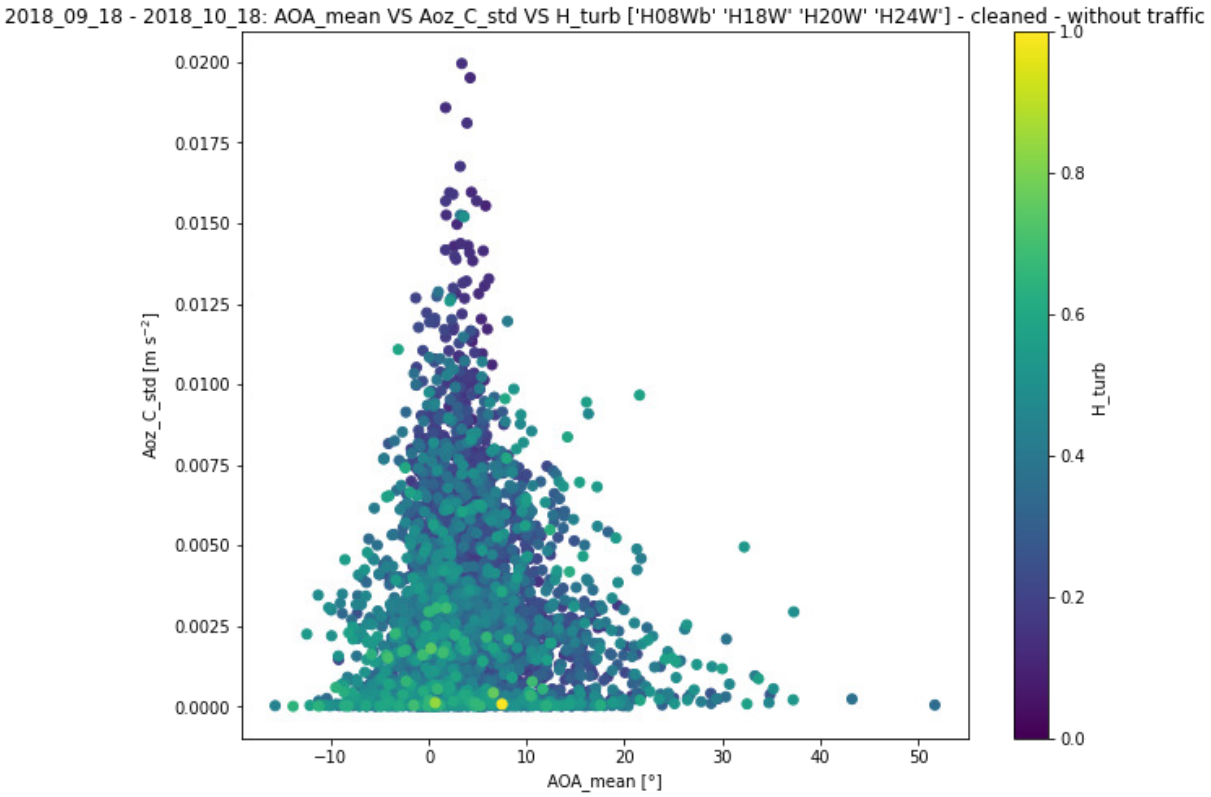


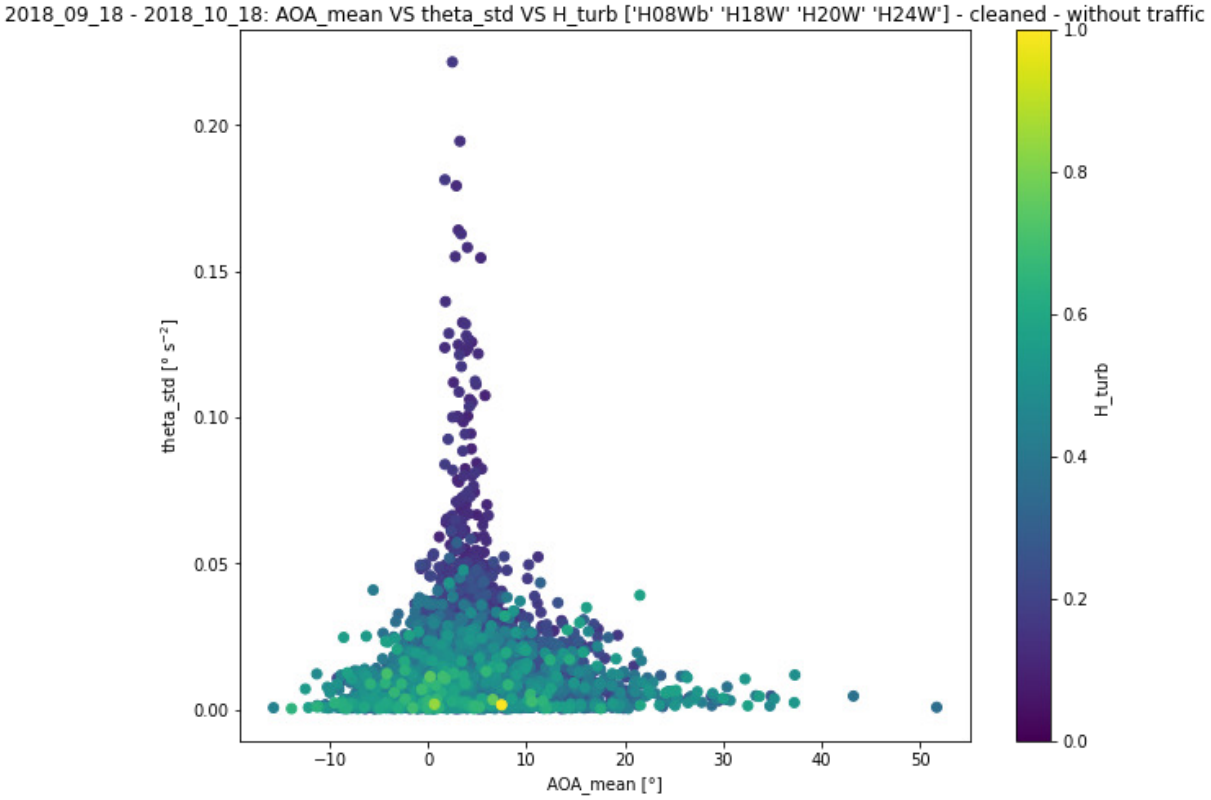
In [ ]:

```
key1 = 'AOA_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'H_turb'
for key2 in key2s_of_interest:
    selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
    selected_sensors_key2=['H09','H18','H24','H30']
    selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
    s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
    s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
    s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
    if key3.startswith('AOA_m'):
        cmap = 'coolwarm'
        vmin = -15
```

```
        vmax = 15
    elif key3.startswith('H_m'):
        cmap = Windfinder_cmap
        vmin = 0
        vmax = 35
    elif key3.endswith('_turb'):
        cmap = 'viridis'
        vmin = 0
        vmax = 1
    else:
        cmap = 'viridis'
        vmin = None
        vmax = None
    LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned
```



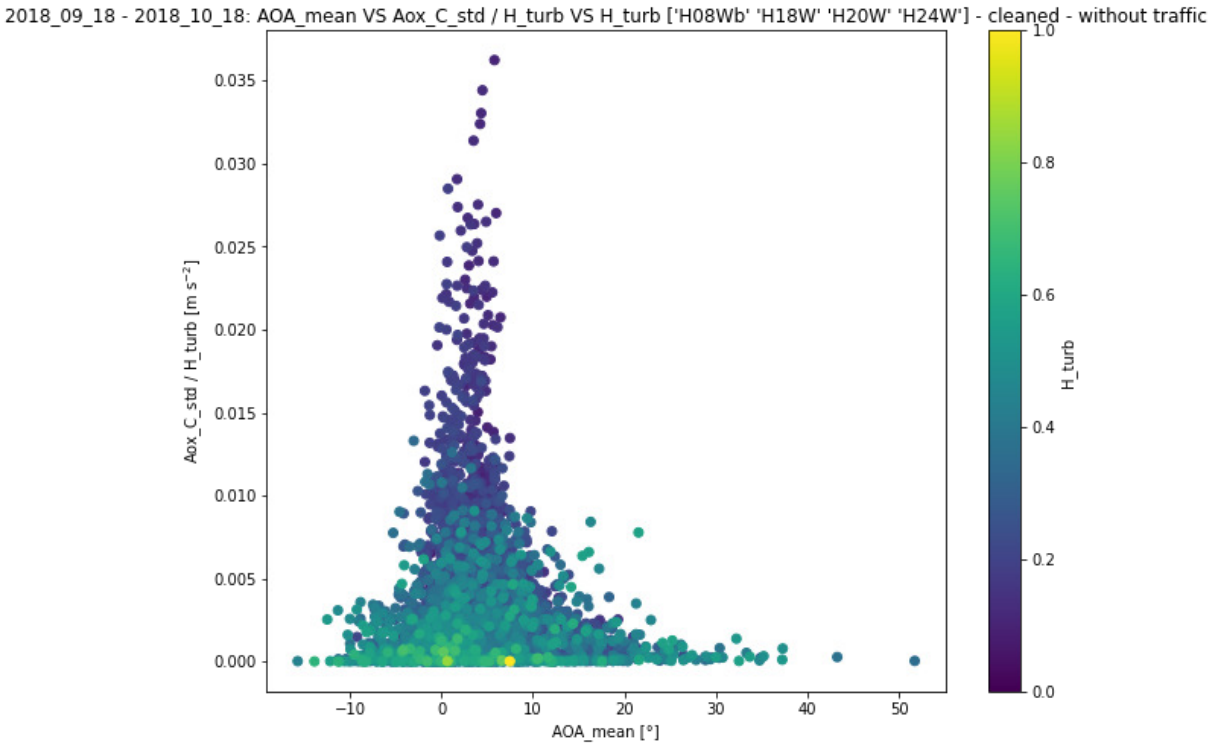


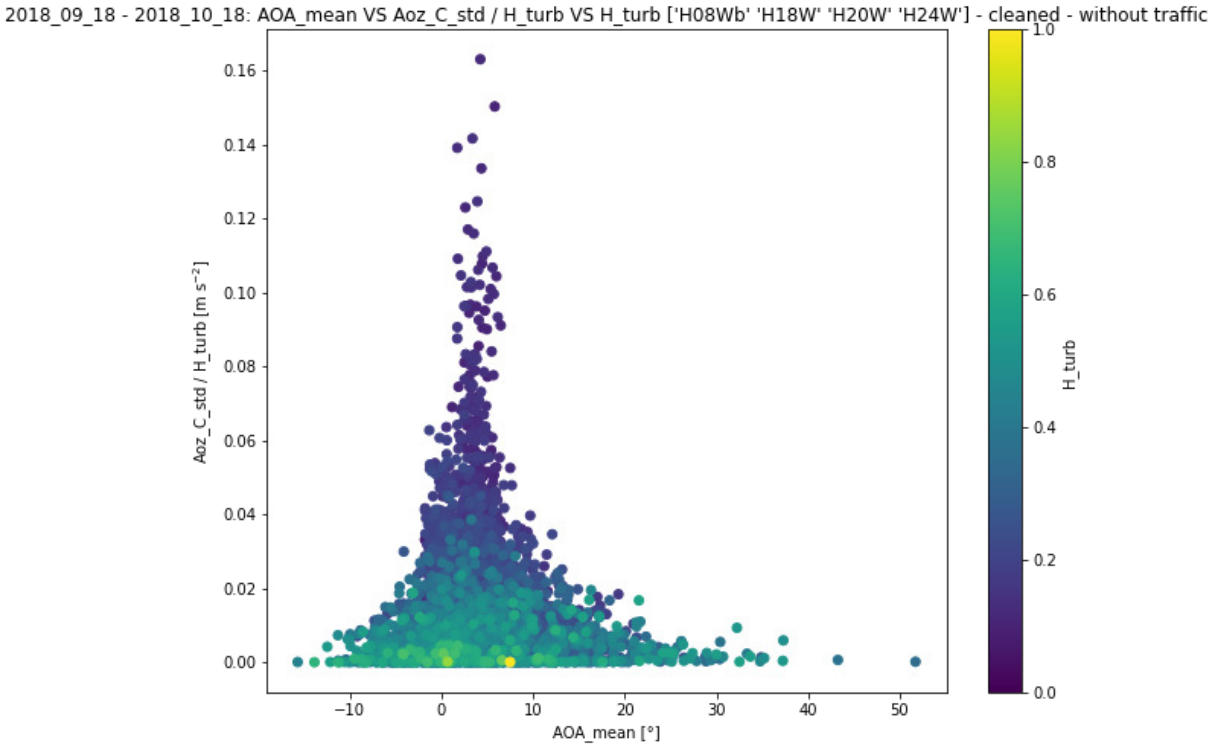


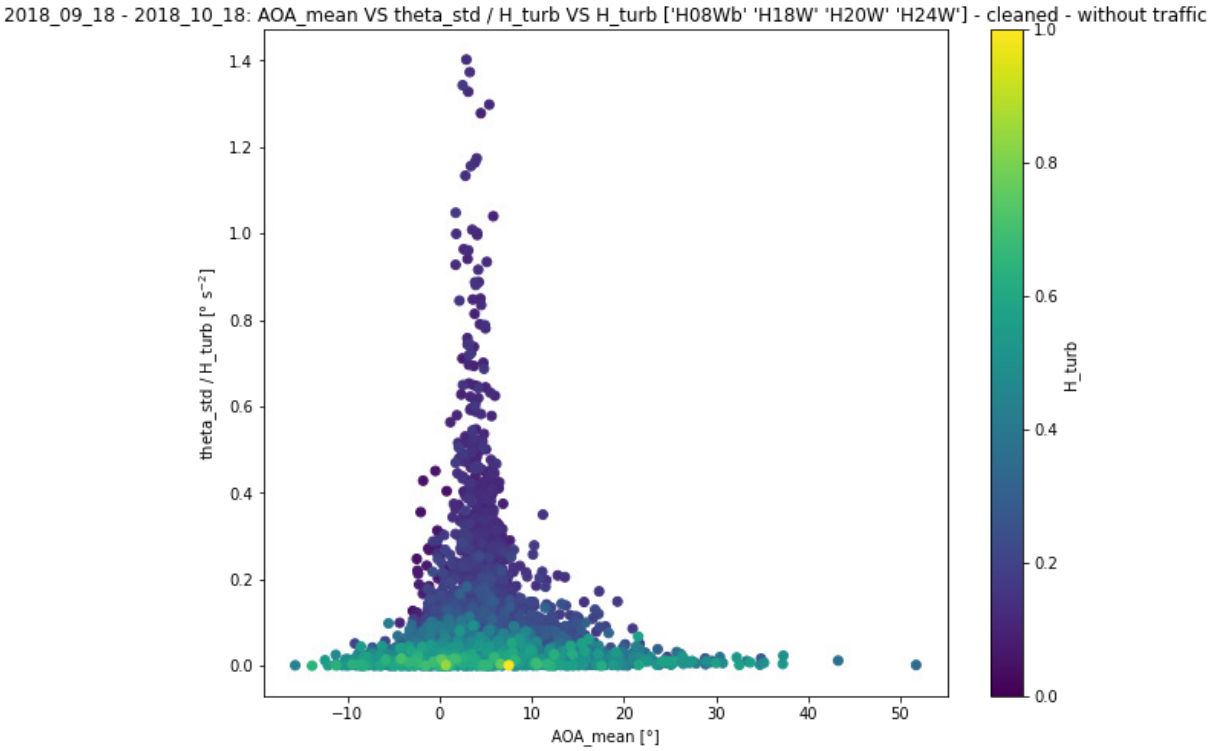
In [ ]:

```
key1 = 'AOA_mean'
key2s_of_interest = ['Aox_C_std','Aoz_C_std','theta_std']
key3 = 'H_turb'
for key2 in key2s_of_interest:
    selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
    selected_sensors_key2=['H09','H18','H24','H30']
    selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
    s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='a
    s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='a
    s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='a
    if key3.startswith('AOA_m'):
        cmap = 'coolwarm'
```

```
        vmin = -15
        vmax = 15
    elif key3.startswith('H_m'):
        cmap = Windfinder_cmap
        vmin = 0
        vmax = 35
    elif key3.endswith('_turb'):
        cmap = 'viridis'
        vmin = 0
        vmax = 1
    else:
        cmap = 'viridis'
        vmin = None
        vmax = None
    LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned
```







Below are more examples for different combinations of measurements in a scatterplot.

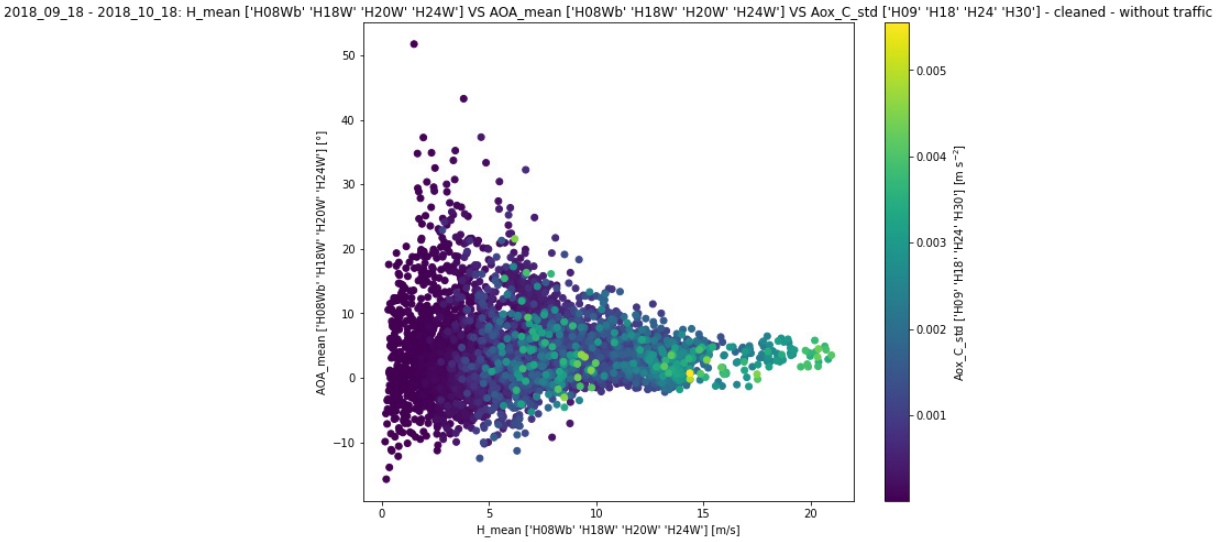
Anemo VS Anemo VS Acc - Wind response

Lateral

In [ ]:

```
key1 = 'H_mean'
key2 = 'AOA_mean'
key3 = 'Aox_C_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

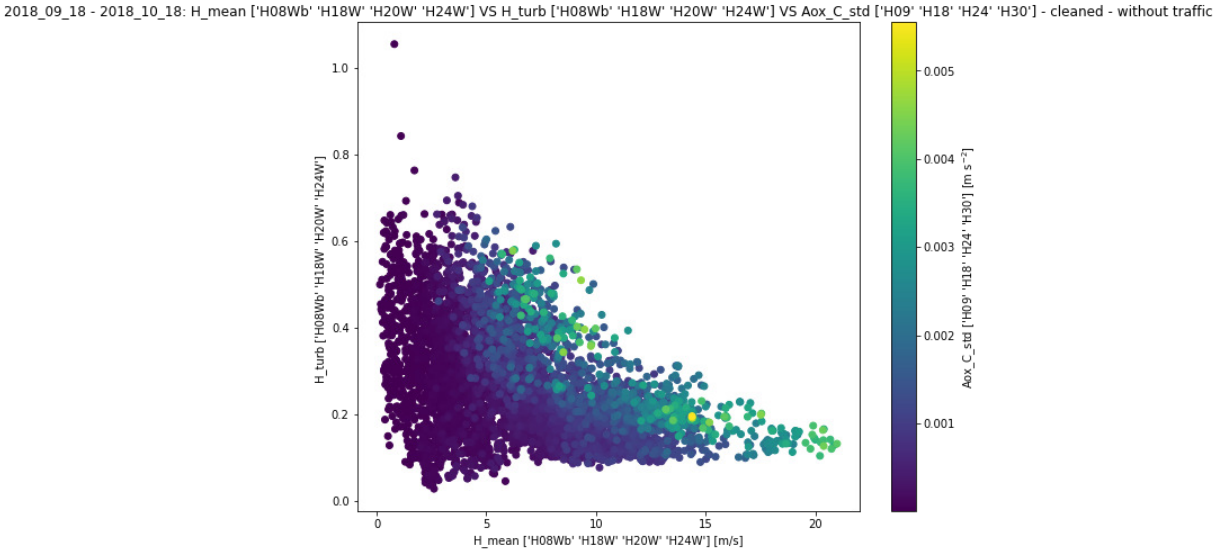
```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  AOA_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W'
'H24W']
ok accelerometers for  Aox_C_std : ['H09' 'H18' 'H24' 'H30']
```



In [ ]:

```
key1 = 'H_mean'
key2 = 'H_turb'
key3 = 'Aox_C_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  H_turb : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H2
4W']
ok accelerometers for  Aox_C_std : ['H09' 'H18' 'H24' 'H30']
```



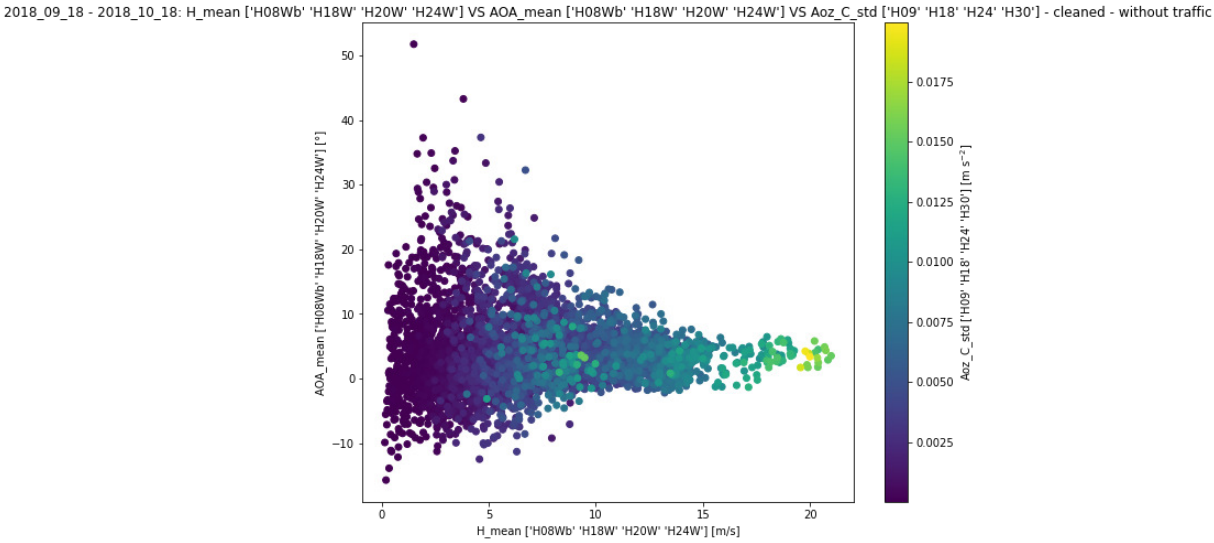
Vertical

In [ ]:

```
key1 = 'H_mean'
key2 = 'AOA_mean'
key3 = 'Aoz_C_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
```

```
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

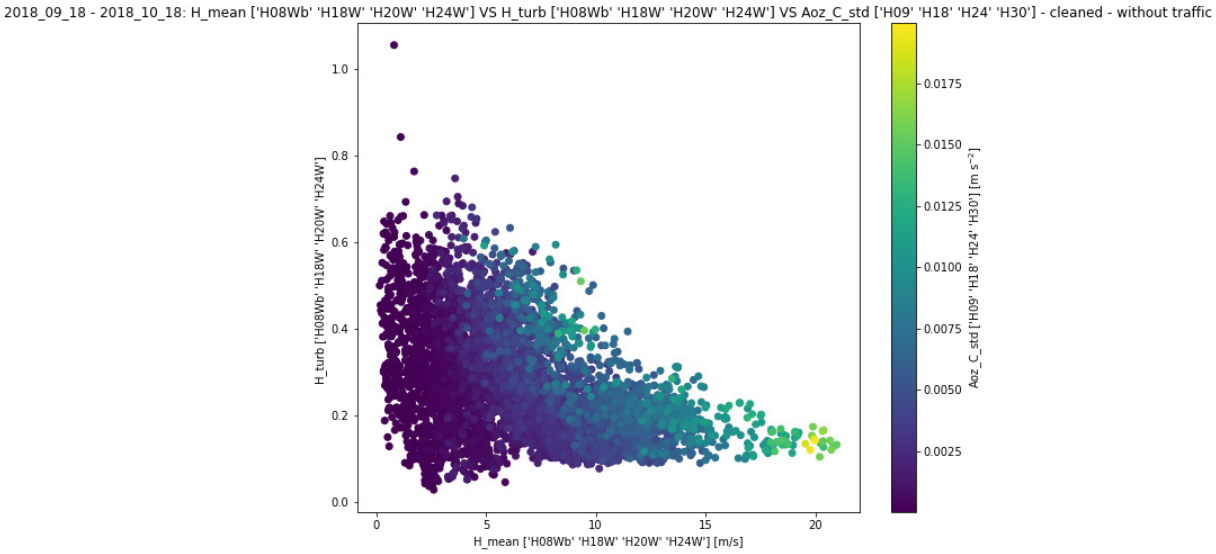
```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  AOA_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W'
'H24W']
ok accelerometers for  Aoz_C_std : ['H09' 'H18' 'H24' 'H30']
```



In [ ]:

```
key1 = 'H_mean'
key2 = 'H_turb'
key3 = 'Aoz_C_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  H_turb : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H2
4W']
ok accelerometers for  Aoz_C_std : ['H09' 'H18' 'H24' 'H30']
```

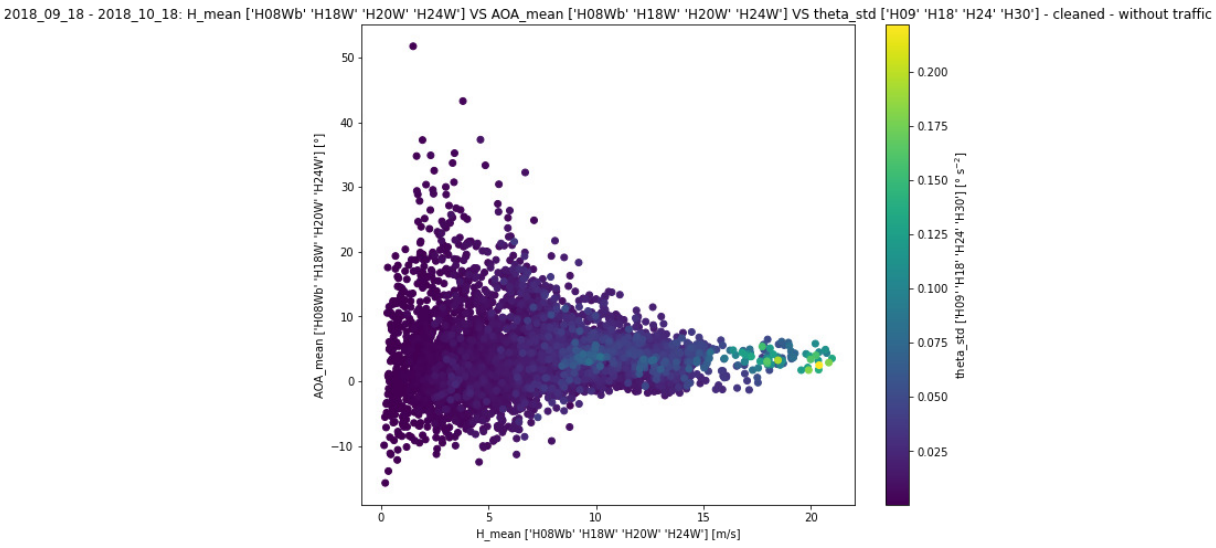


Torsional

In [ ]:

```
key1 = 'H_mean'
key2 = 'AOA_mean'
key3 = 'theta_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  AOA_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W'
'H24W']
ok accelerometers for  theta_std : ['H09' 'H18' 'H24' 'H30']
```

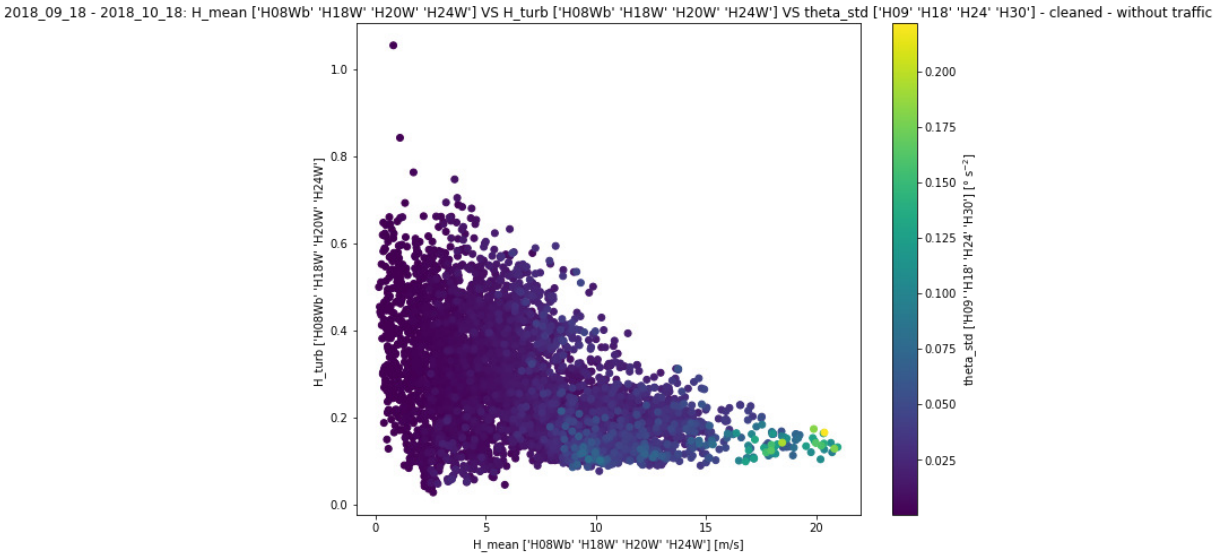


In [ ]:

```
key1 = 'H_mean'
key2 = 'H_turb'
key3 = 'theta_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
```

```
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',

LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[k
```

```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok anemometers for  H_turb : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H2
4W']
ok accelerometers for  theta_std : ['H09' 'H18' 'H24' 'H30']
```

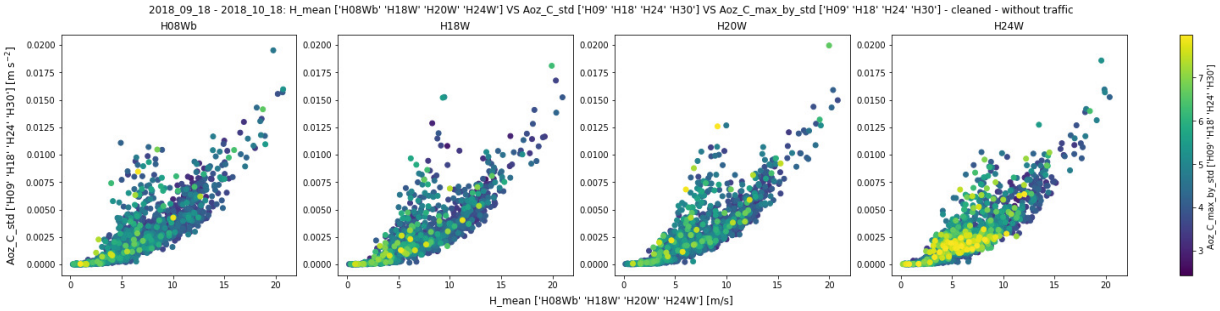


Anemo VS Acc VS Acc - Wind response

In [ ]:

```
key1 = 'H_mean'
key2 = 'Aoz_C_std'
key3 = 'Aoz_C_max_by_std'
print('ok anemometers for ',key1,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key1]])
print('ok accelerometers for ',key2,':',LFB.acc_names[LFB.acc_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H08Wb','H18W','H20W','H24W']
selected_sensors_key2=['H09','H18','H24','H30']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='anemo
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='acc',
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',


LFB.scatterplot(LFB.anemo_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key
```

```
ok anemometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E' 'H2
0W' 'H24W']
ok accelerometers for  Aoz_C_std : ['H09' 'H18' 'H24' 'H30']
ok accelerometers for  Aoz_C_max_by_std : ['H09' 'H18' 'H24' 'H30']
```

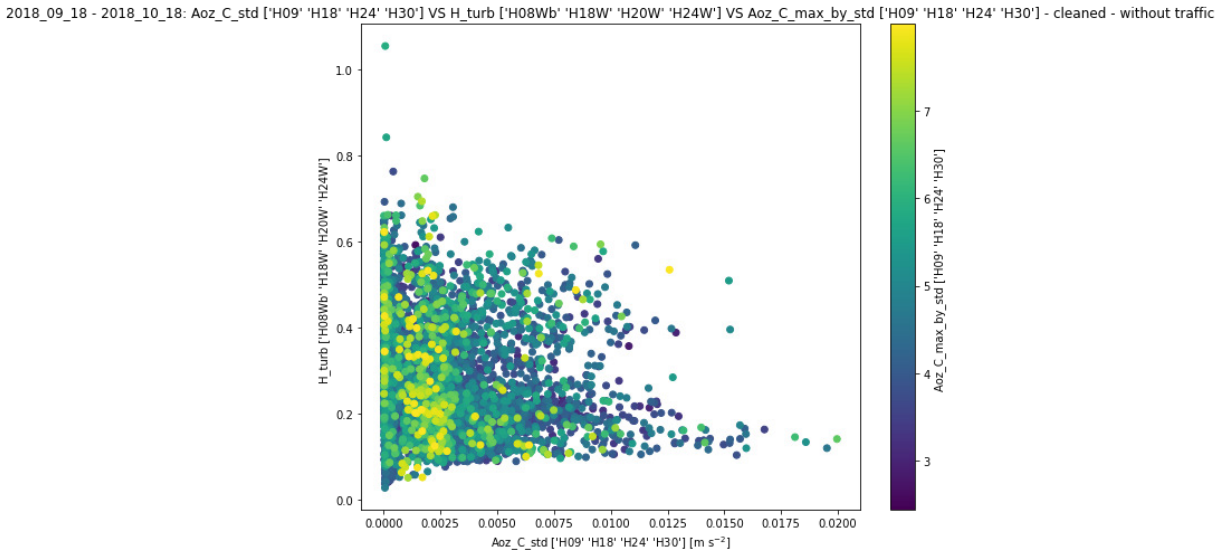


Acc VS Anemo VS Acc - Wind response

In [ ]:

```
key1 = 'Aoz_C_std'
key2 = 'H_turb'
key3 = 'Aoz_C_max_by_std'
print('ok anemometers for ',key1,':',LFB.acc_names[LFB.acc_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.acc_names[LFB.acc_ok_sensor_id[key3]])
selected_sensors_key1=['H09','H18','H24','H30']
selected_sensors_key2=['H08Wb','H18W','H20W','H24W']
selected_sensors_key3=['H09','H18','H24','H30']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='acc',
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='anemo
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='acc',


LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.anemo_cleaned[key
```

```
ok anemometers for  Aoz_C_std : ['H09' 'H18' 'H24' 'H30']
ok anemometers for  H_turb : ['H08Wb' 'H08Wt' 'H08E' 'H10E' 'H18W' 'H18E' 'H20W' 'H2
4W']
ok accelerometers for  Aoz_C_max_by_std : ['H09' 'H18' 'H24' 'H30']
```



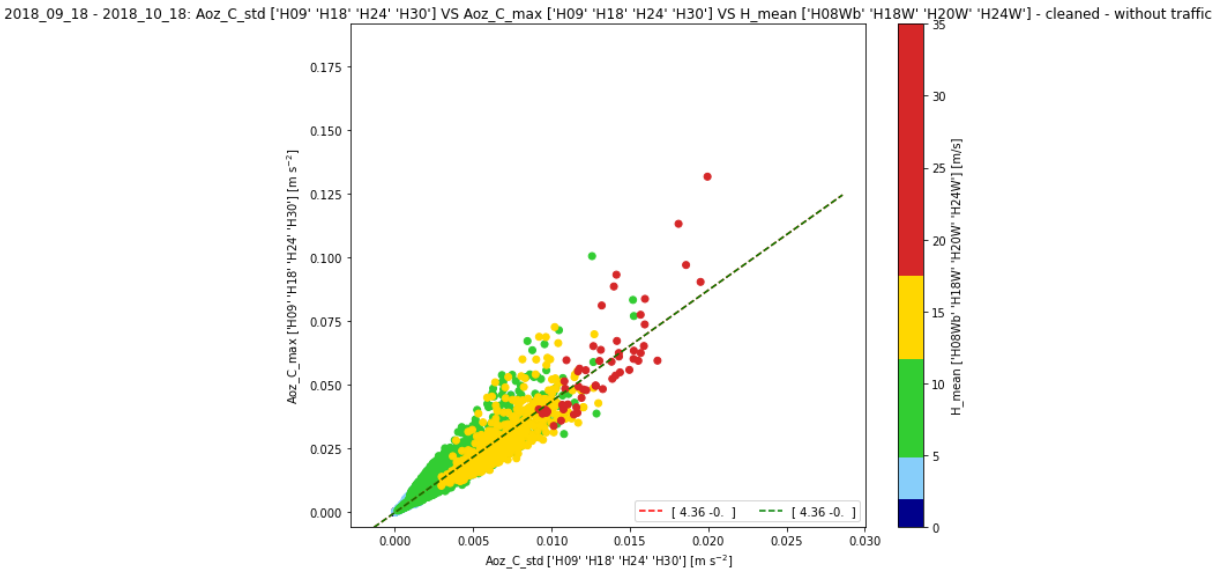
Acc VS Acc VS Anemo - Wind response

In [ ]:

```
key1 = 'Aoz_C_std'
key2 = 'Aoz_C_max'
key3 = 'H_mean'
print('ok anemometers for ',key1,':',LFB.acc_names[LFB.acc_ok_sensor_id[key1]])
print('ok anemometers for ',key2,':',LFB.acc_names[LFB.acc_ok_sensor_id[key2]])
print('ok accelerometers for ',key3,':',LFB.anemo_names[LFB.anemo_ok_sensor_id[key3]
selected_sensors_key1=['H09','H18','H24','H30']
selected_sensors_key2=['H09','H18','H24','H30']
```

```
selected_sensors_key3=['H08Wb','H18W','H20W','H24W']
s1_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key1,sensor_type='acc',
s2_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key2,sensor_type='acc',
s3_ind = LFB.get_ok_sensor_ind(sensor_names=selected_sensors_key3,sensor_type='anemo
if key3.startswith('AOA_m'):
    cmap = 'coolwarm'
    vmin = -15
    vmax = 15
elif key3.startswith('H_m'):
    cmap = Windfinder_cmap
    vmin = 0
    vmax = 35
elif key3.endswith('_turb'):
    cmap = 'viridis'
    vmin = 0
    vmax = 1
else:
    cmap = 'viridis'
    vmin = None
    vmax = None
LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```

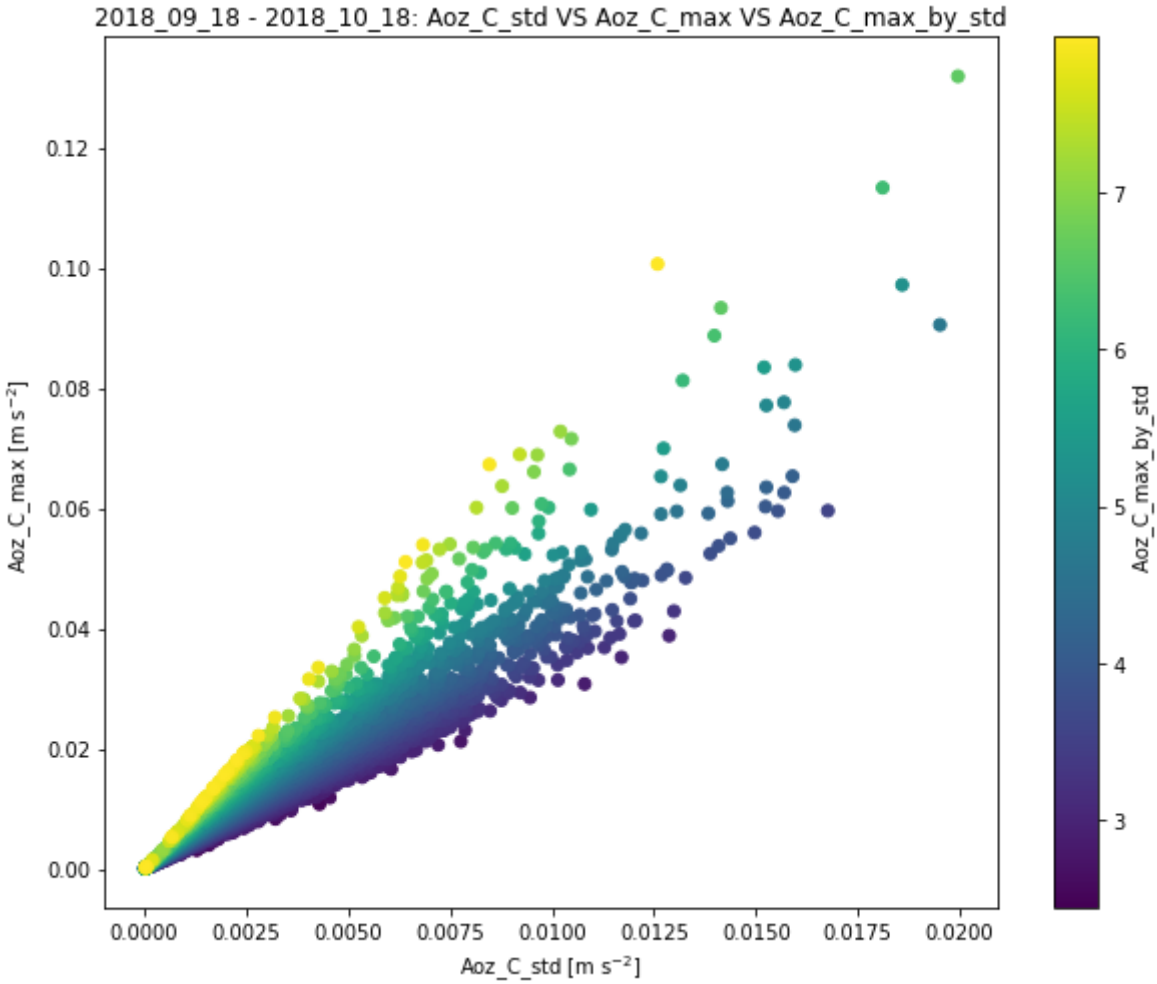
```
ok anemometers for  Aoz_C_std : ['H09' 'H18' 'H24' 'H30']
ok anemometers for  Aoz_C_max : ['H09' 'H18' 'H24' 'H30']
ok accelerometers for  H_mean : ['H08Wb' 'H08Wt' 'H08E' 'H10W' 'H10E' 'H18W' 'H18E'
'H20W' 'H24W']
```



Acc VS Acc vs Acc - Acceleration characteristics

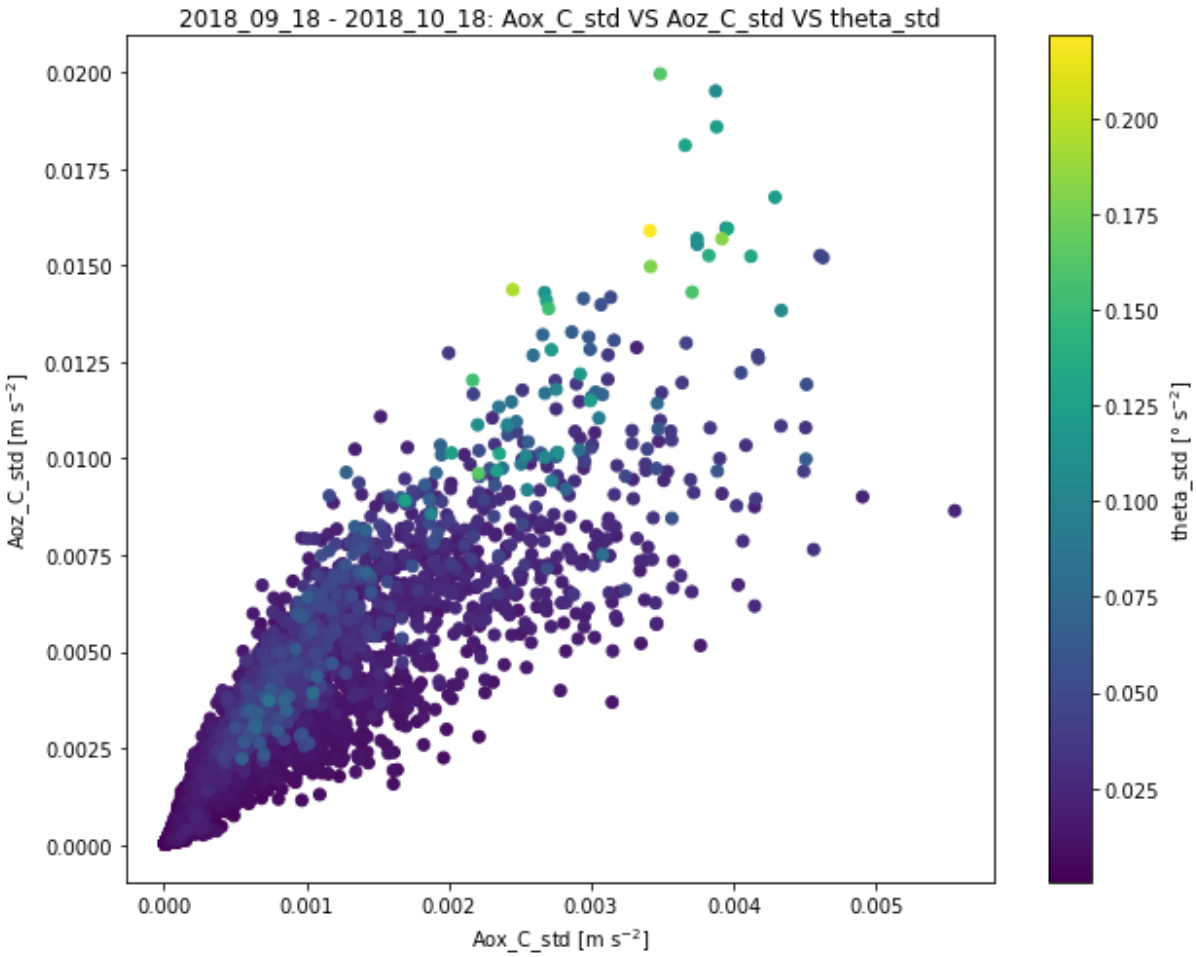
In [ ]:

```
key1 = 'Aoz_C_std'
key2 = 'Aoz_C_max'
key3 = 'Aoz_C_max_by_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



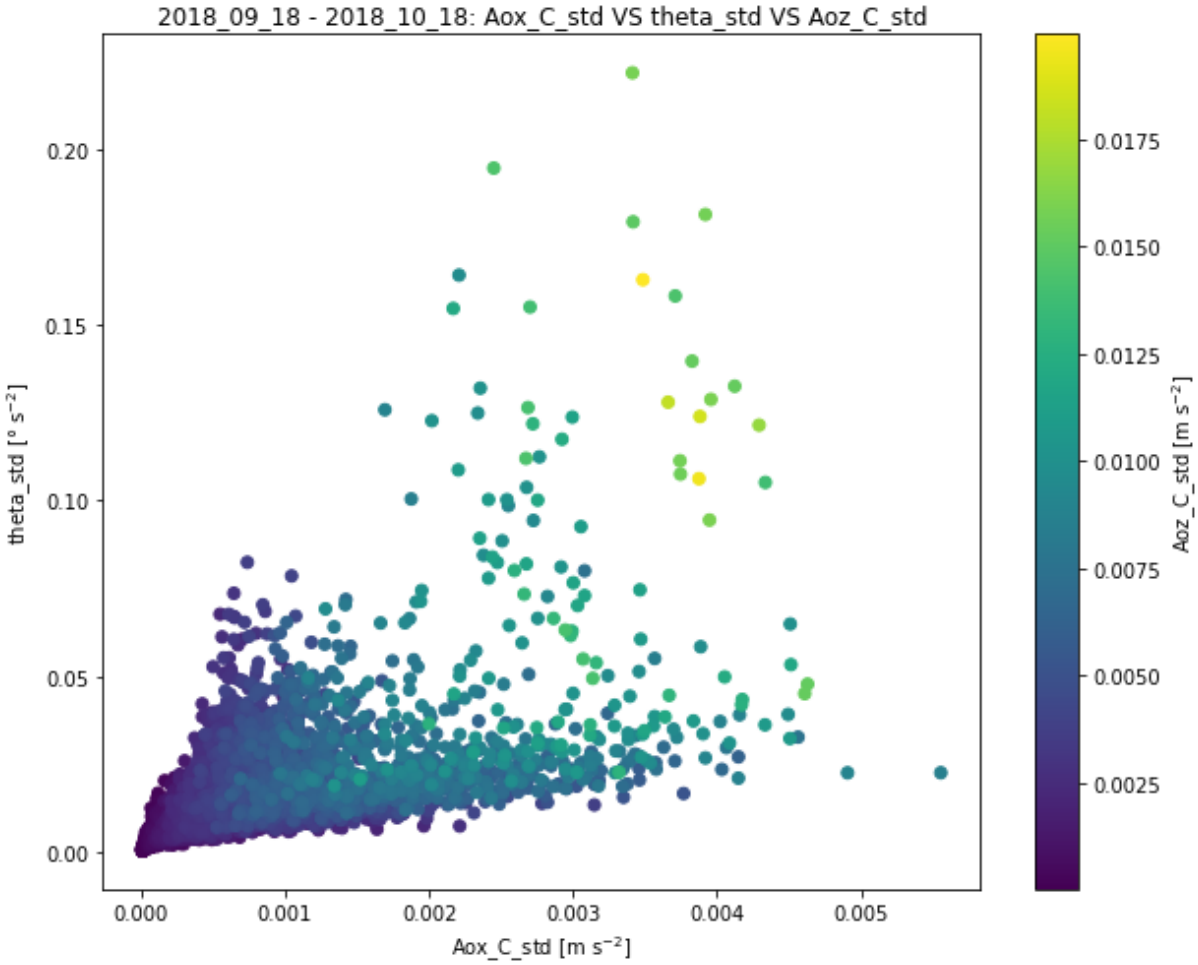
In [ ]:

```
key1 = 'Aox_C_std'
key2 = 'Aoz_C_std'
key3 = 'theta_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



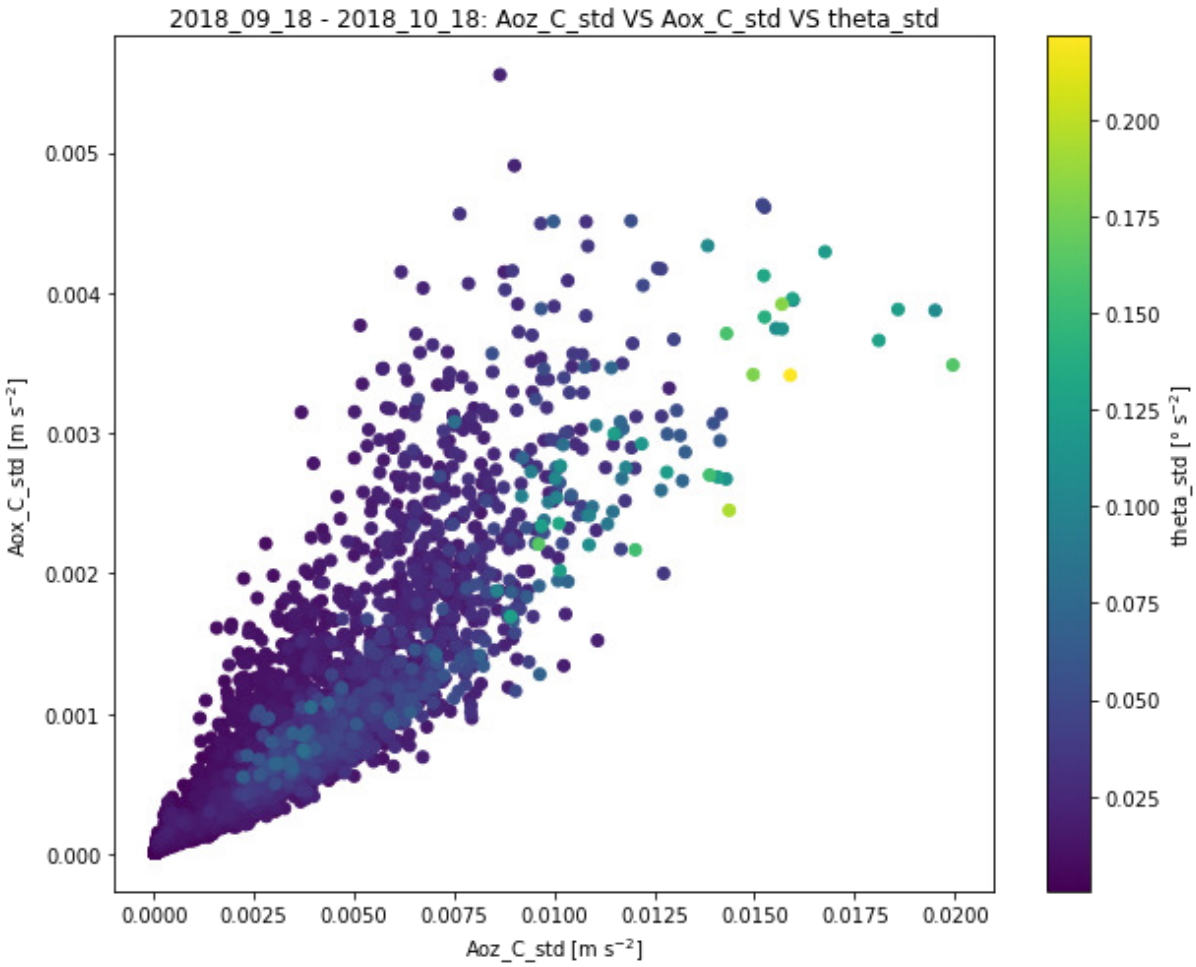
In [ ]:

```
key1 = 'Aox_C_std'
key2 = 'theta_std'
key3 = 'Aoz_C_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



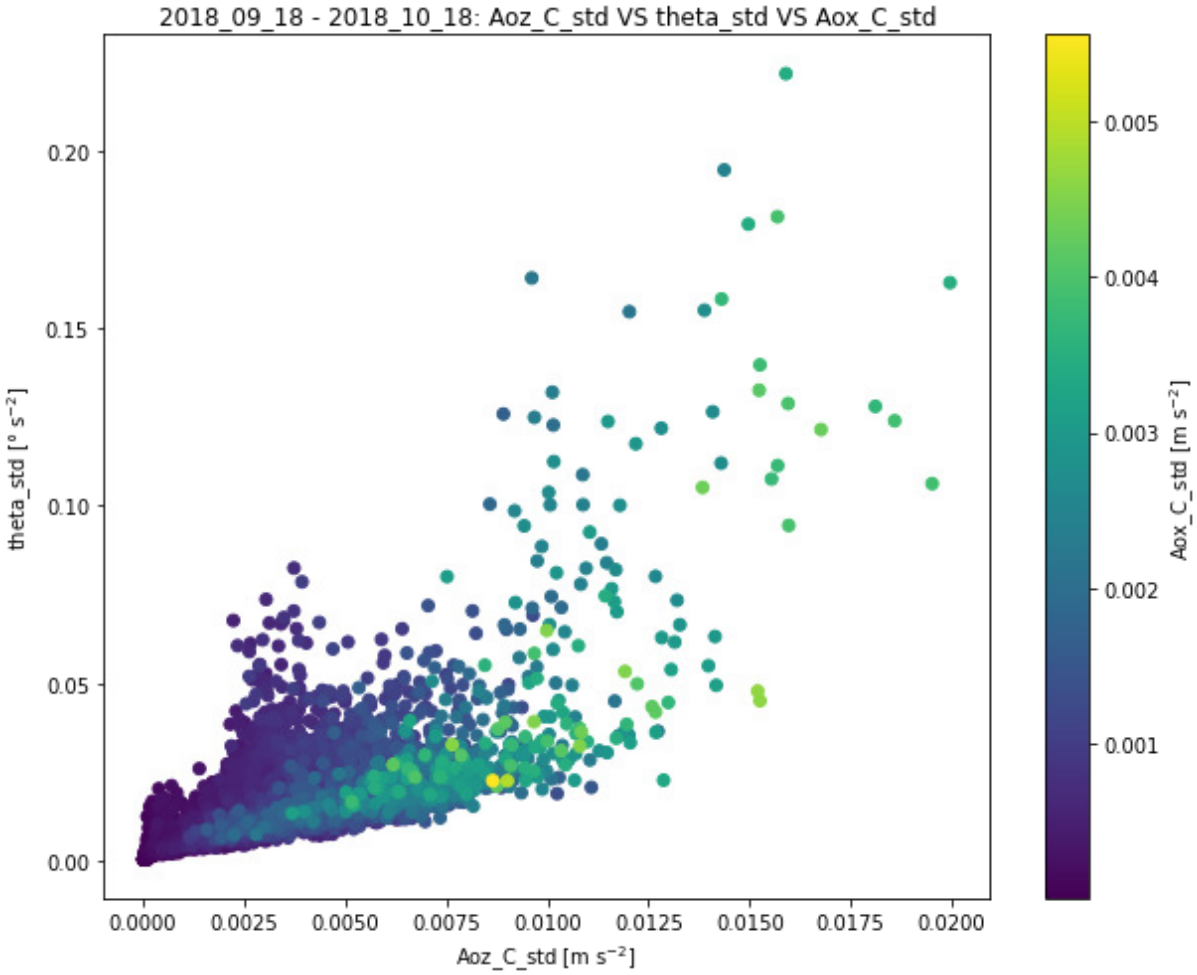
In [ ]:

```
key1 = 'Aoz_C_std'
key2 = 'Aox_C_std'
key3 = 'theta_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



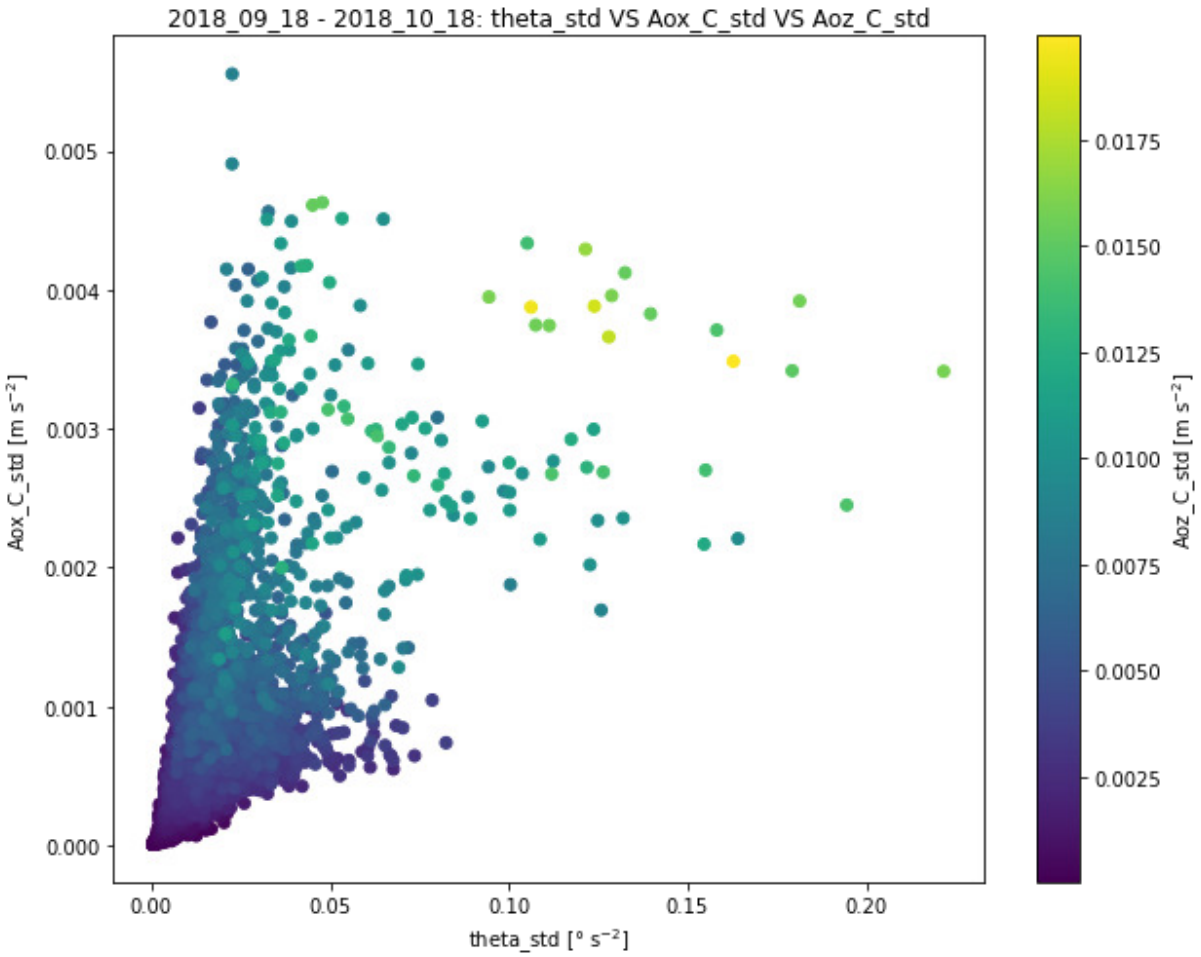
In [ ]:

```
key1 = 'Aoz_C_std'
key2 = 'theta_std'
key3 = 'Aox_C_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



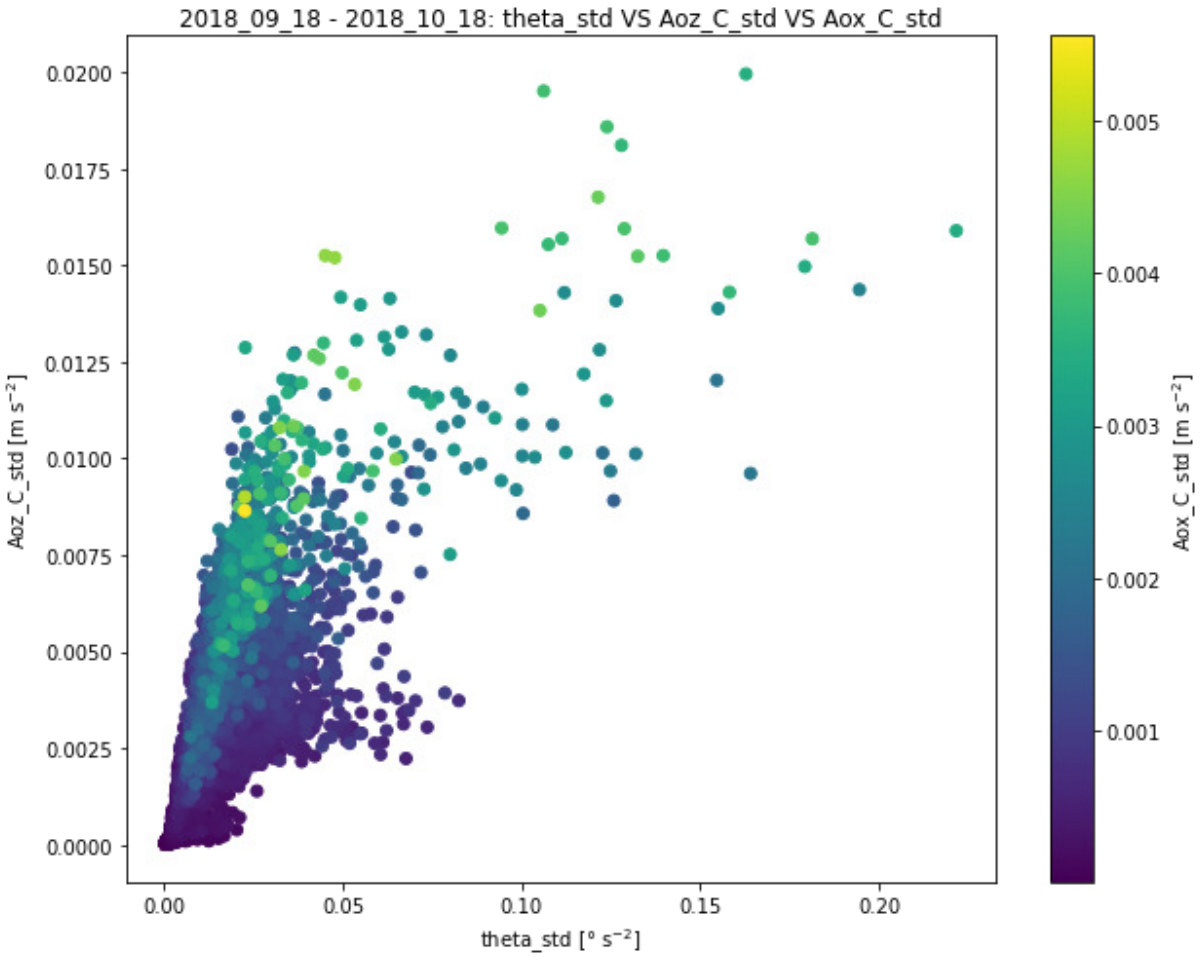
In [ ]:

```
key1 = 'theta_std'
key2 = 'Aox_C_std'
key3 = 'Aoz_C_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



In [ ]:

```
key1 = 'theta_std'
key2 = 'Aoz_C_std'
key3 = 'Aox_C_std'
common_ok_sensors,s1_ind,s2_ind = LFB.find_common_ok_sensors(LFB.acc_ok_sensor_id[ke
common_ok_sensors,stemp_ind,s3_ind = LFB.find_common_ok_sensors(common_ok_sensors,LF
s1_ind = list(np.array(s1_ind)[stemp_ind])
s2_ind = list(np.array(s2_ind)[stemp_ind])

LFB.scatterplot(LFB.acc_cleaned[key1][s1_ind].T[filter_idxs].T,LFB.acc_cleaned[key2]
```



Correlation matrices

Below are some correlation matrices for relevant measurements, selected using `keys_of_interst` .

In [ ]:
```
# keys_of_interest = ['H_mean','H_turb','W_mean','W_turb','Vx_mean','Vx_turb','Vy_me
keys_of_interest = ['H_mean','H_turb','AOA_mean','Aox_C_std','Aoz_C_std','theta_std'
```

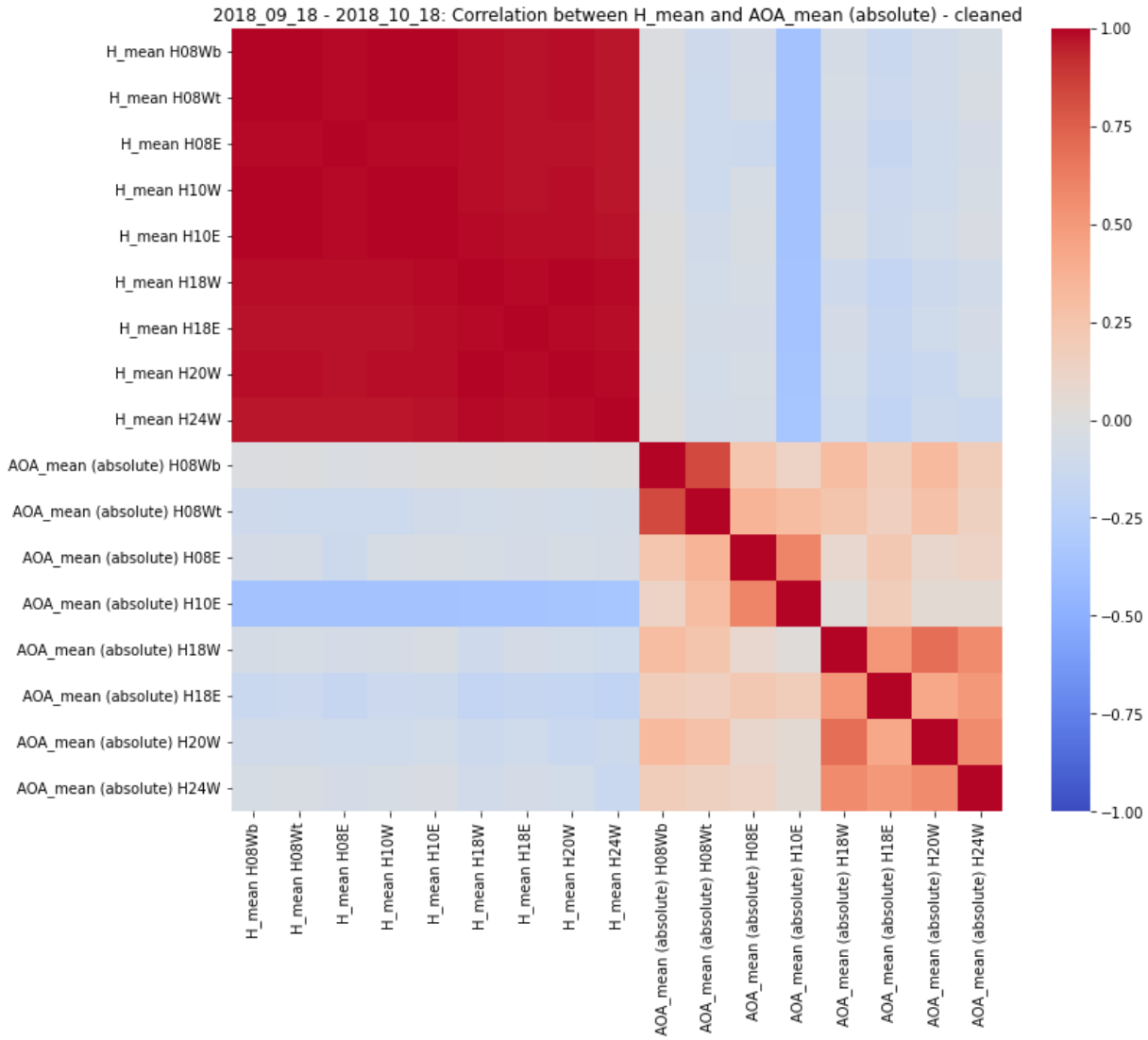
Anemo & Anemo - Wind characteristics

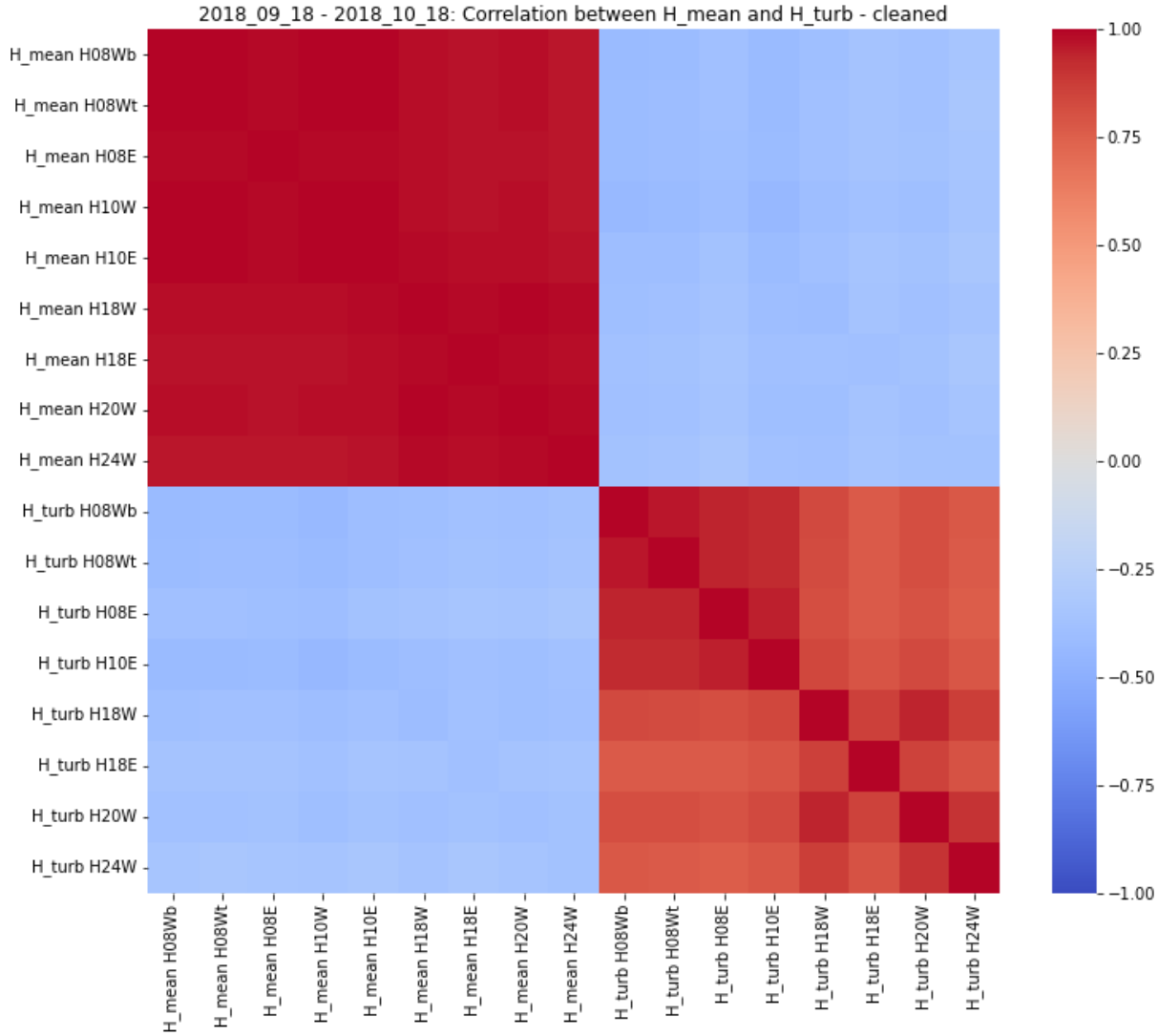
In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
# filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=Tru
```

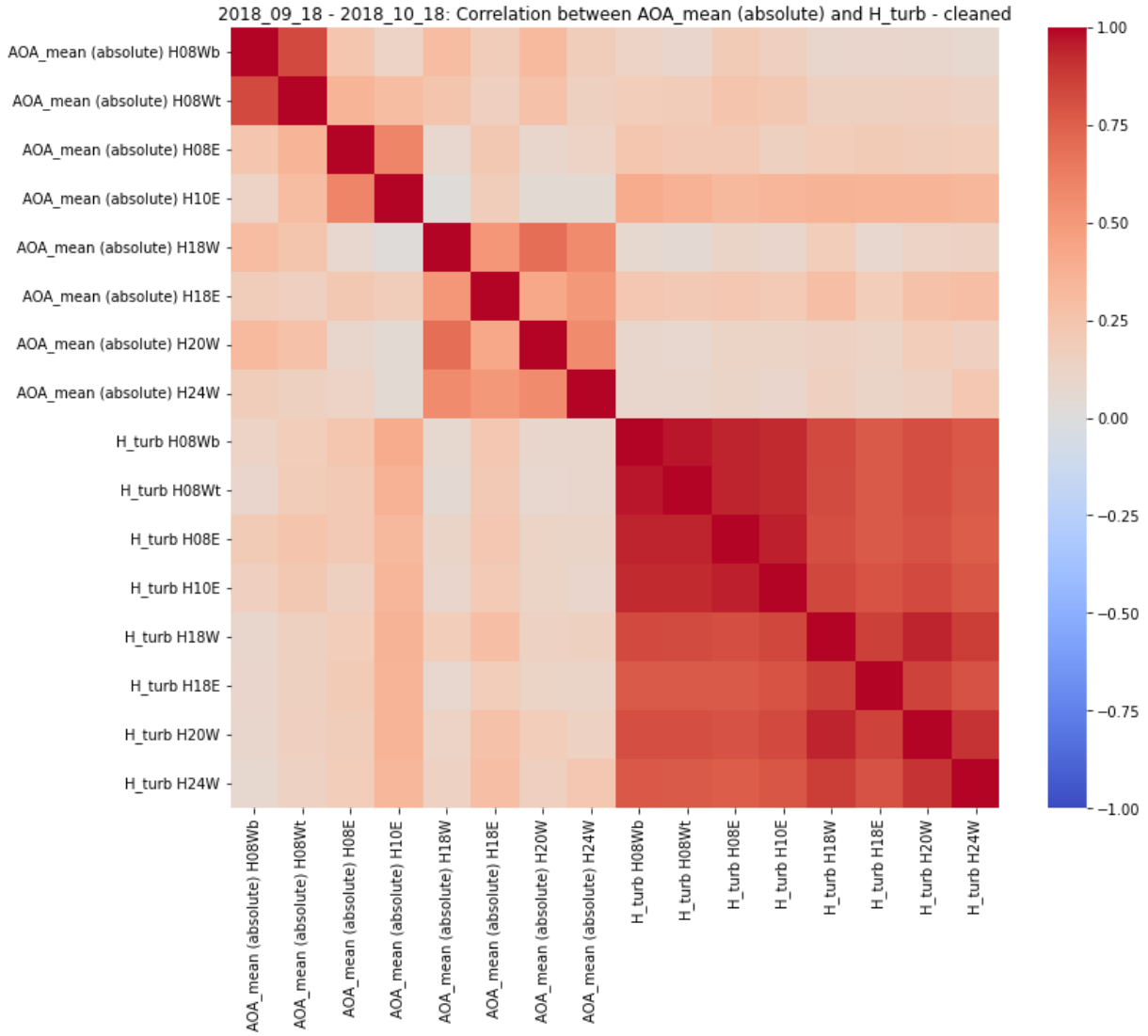
Batch-processing using two nested *for-loops* over `key1` and `key2` is used. *if-statements* are used to selectively process *keys* from `keys_of_interest` and process each key pairing only once. Another *if-statement* is used to apply the absolute opperation `np.abs` to data of keys starting with `'AOA_m'` only.

In [ ]:
```
for key1_num, key1 in enumerate(LFB.anemo_cleaned.keys()):
    if key1 in keys_of_interest:
        for key2_num, key2 in enumerate(LFB.anemo_cleaned.keys()):
            if key2 in keys_of_interest:
                if key1_num < key2_num:
                    if key1.startswith('AOA_m'):
                        data1 = np.abs(LFB.anemo_cleaned[key1].T[filter_idxs].T)
                        title1_suffix = ' (absolute)'
                    else:
```

```
        data1 = LFB.anemo_cleaned[key1].T[filter_idxs].T
        title1_suffix = ''
    if key2.startswith('AOA_m'):
        data2 = np.abs(LFB.anemo_cleaned[key2].T[filter_idxs].T)
        title2_suffix = ' (absolute)'
    else:
        data2 = LFB.anemo_cleaned[key2].T[filter_idxs].T
        title2_suffix = ''
    LFB.correlation_matrix(data1,data2,title1=key1+title1_suffix,tit
```




2018_09_18 - 2018_10_18: Correlation between H_mean and H_turb - cleaned
H_mean H08Wb
H_mean H08Wt
H_mean H08E
H_mean H10W
H_mean H10E
H_mean H18W
H_mean H18E
H_mean H20W
H_mean H24W
H_turb H08Wb
H_turb H08Wt
H_turb H08E
H_turb H10E
H_turb H18W
H_turb H18E
H_turb H20W
H_turb H24W
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



Anemo & Acc - Wind response

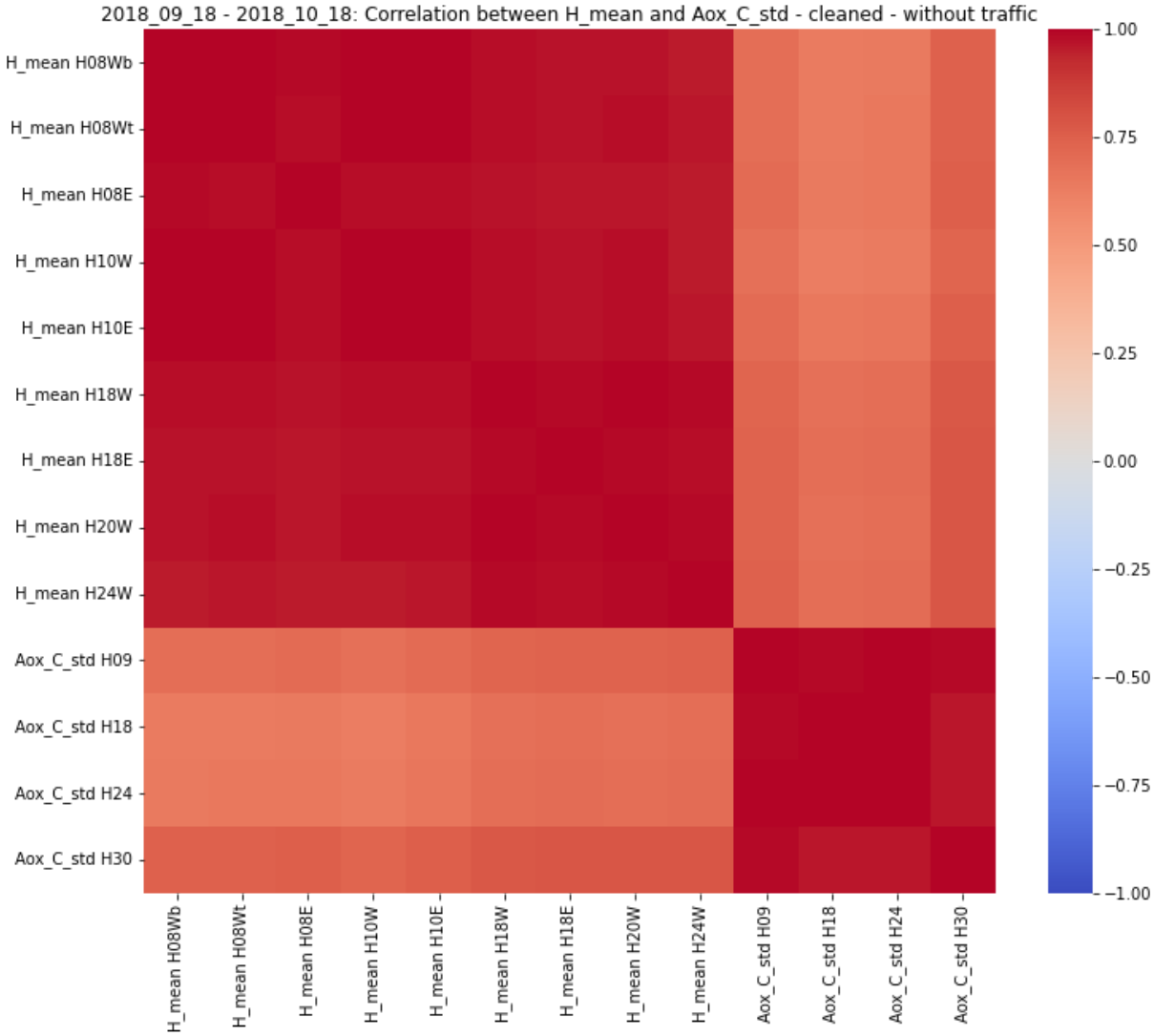
For correlations with accelerometer data the data is filtered for traffic, as only the wind response is of interest.

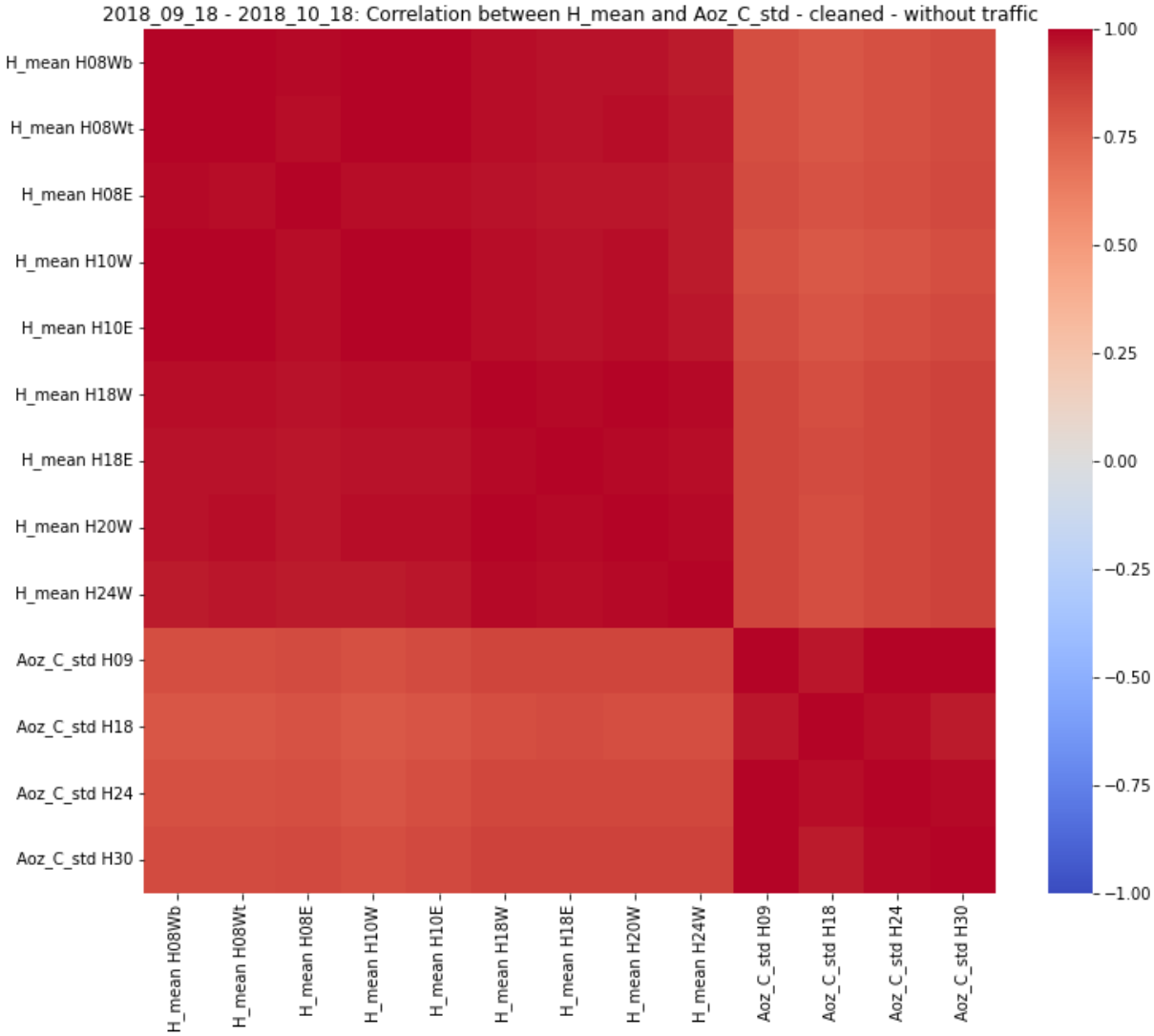
In [ ]:

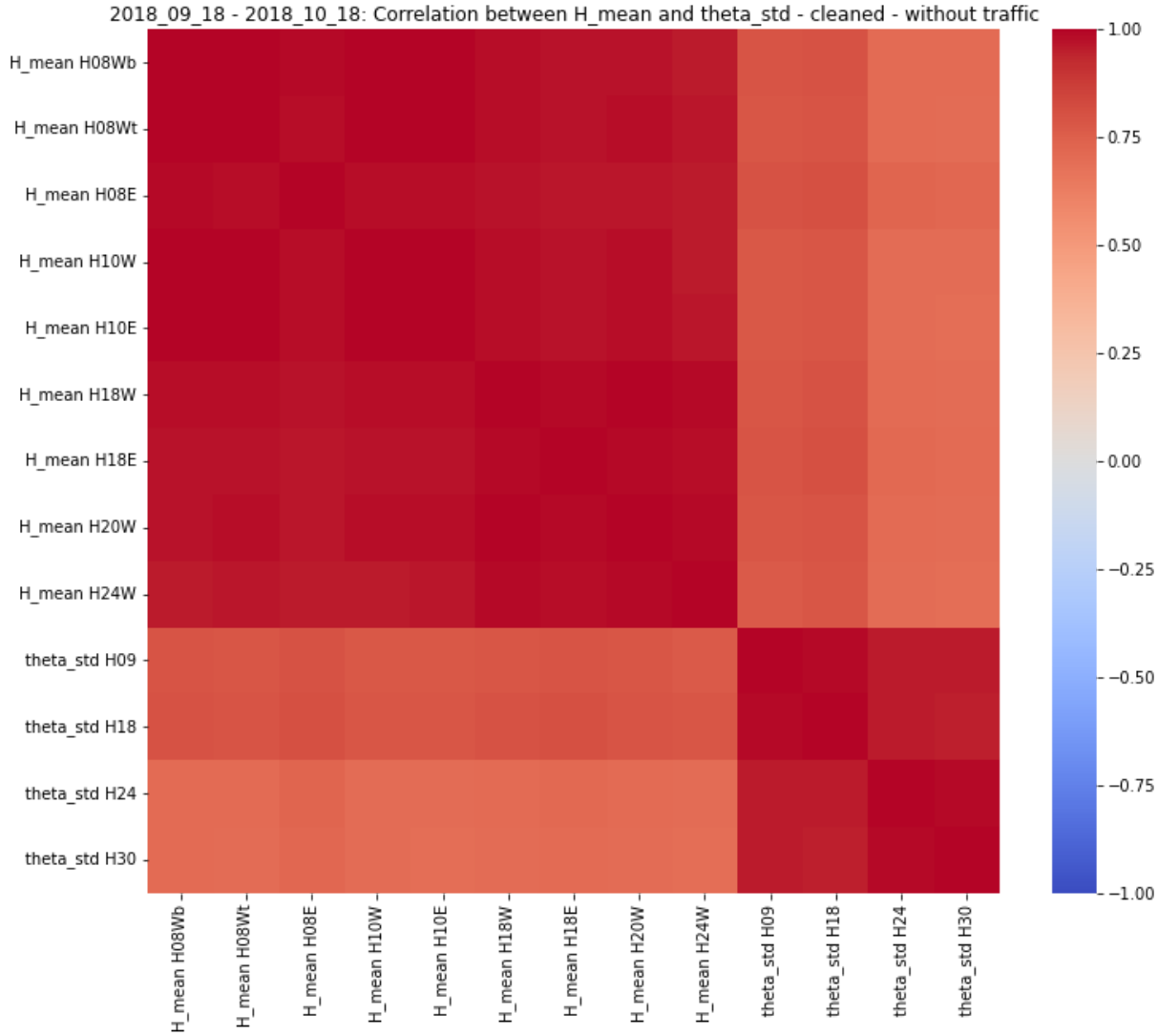
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=True,
```

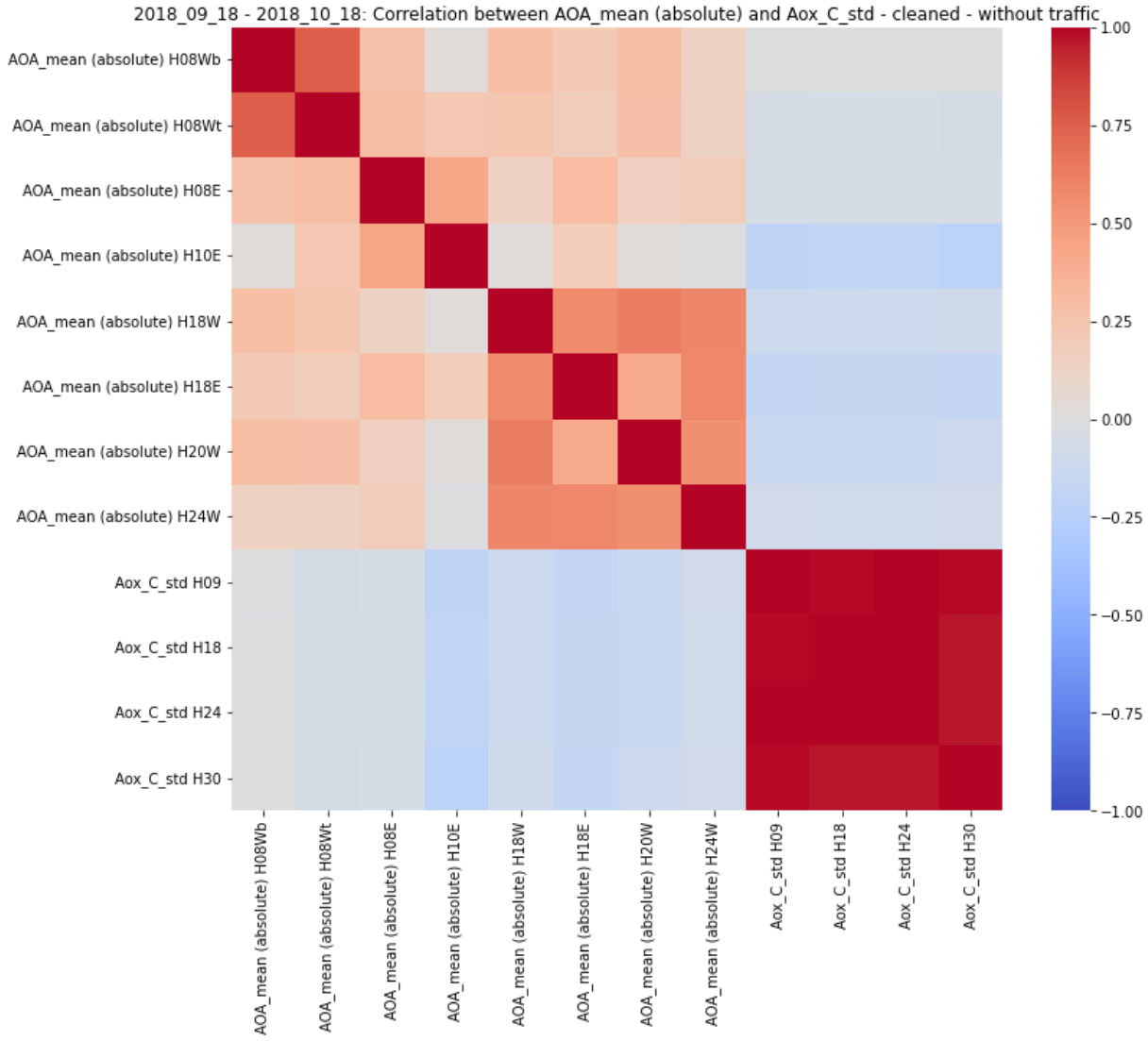
In [ ]:

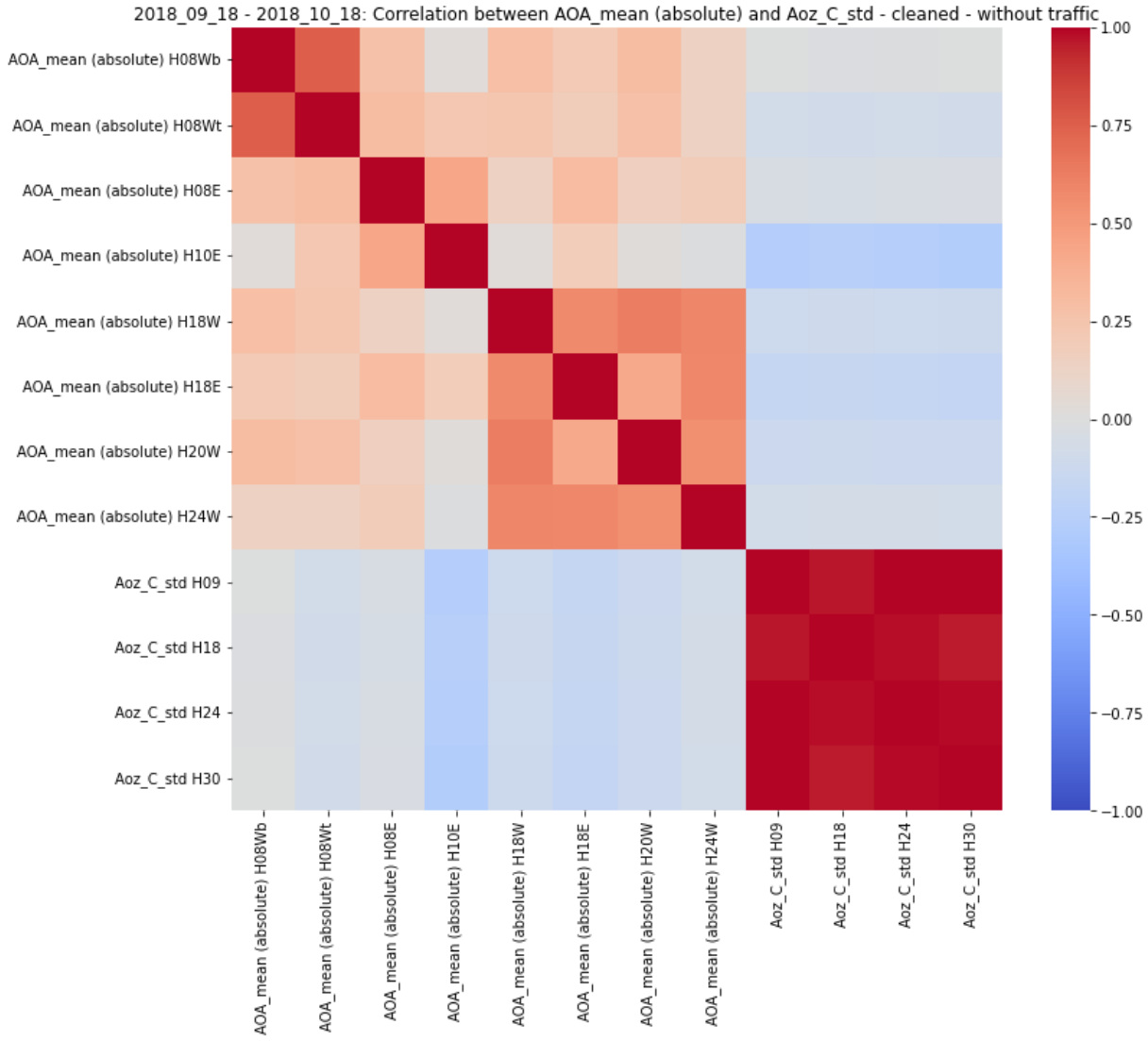
```
for key1_num, key1 in enumerate(LFB.anemo_cleaned.keys()):
    if key1 in keys_of_interest:
        for key2_num, key2 in enumerate(LFB.acc_cleaned.keys()):
            if key2 in keys_of_interest:
                if key1.startswith('AOA_m'):
                    data1 = np.abs(LFB.anemo_cleaned[key1].T[filter_idxs].T)
                    title1_suffix = ' (absolute)'
                else:
                    data1 = LFB.anemo_cleaned[key1].T[filter_idxs].T
                    title1_suffix = ''
                data2 = LFB.acc_cleaned[key2].T[filter_idxs].T
                title2_suffix = ''
                LFB.correlation_matrix(data1,data2,title1=key1+title1_suffix,title2=
```

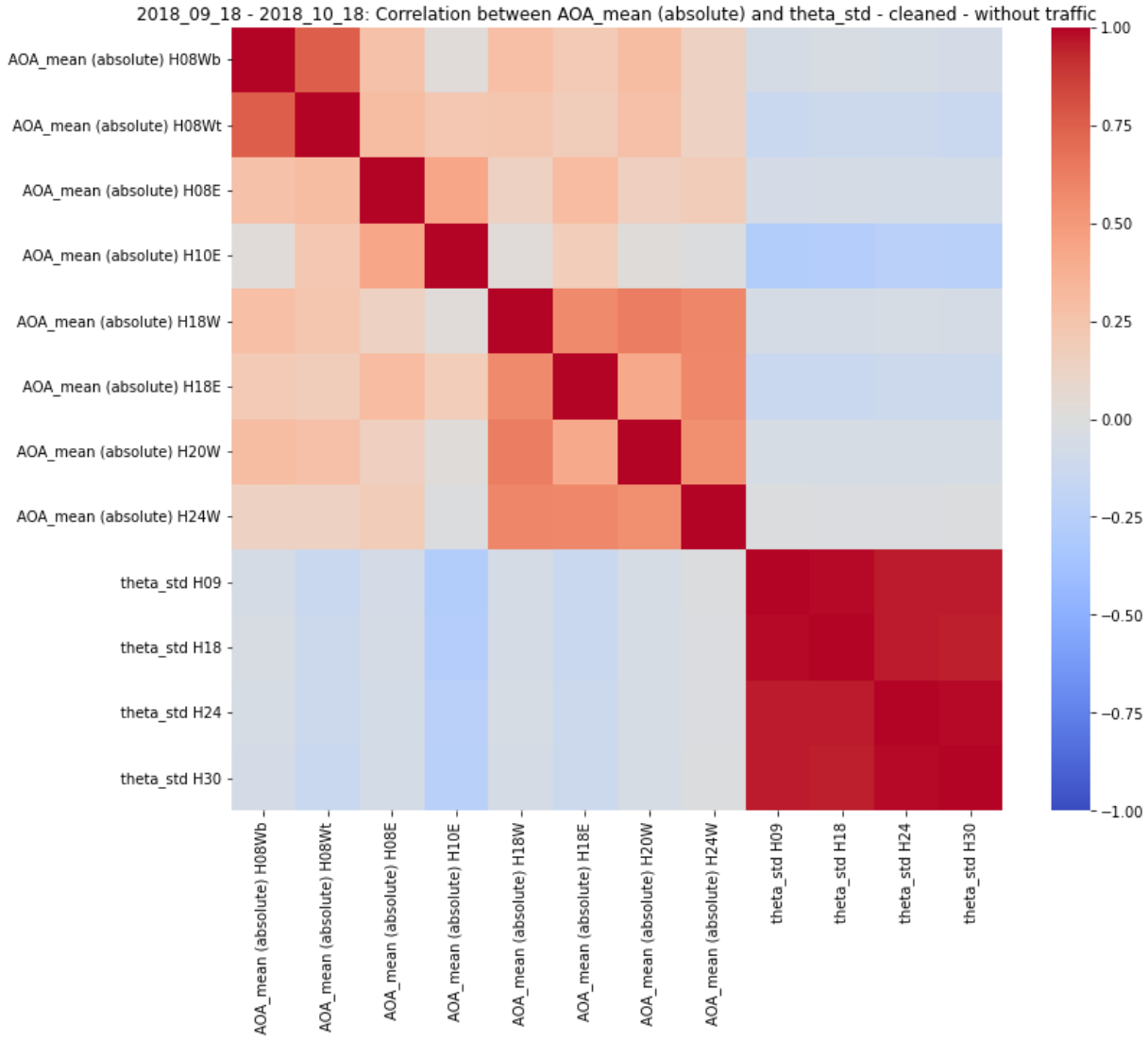

2018_09_18 - 2018_10_18: Correlation between H_mean and Aox_C_std - cleaned - without traffic
H_mean H08Wb
H_mean H08Wt
H_mean H08E
H_mean H10W
H_mean H10E
H_mean H18W
H_mean H18E
H_mean H20W
H_mean H24W
Aox_C_std H09
Aox_C_std H18
Aox_C_std H24
Aox_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

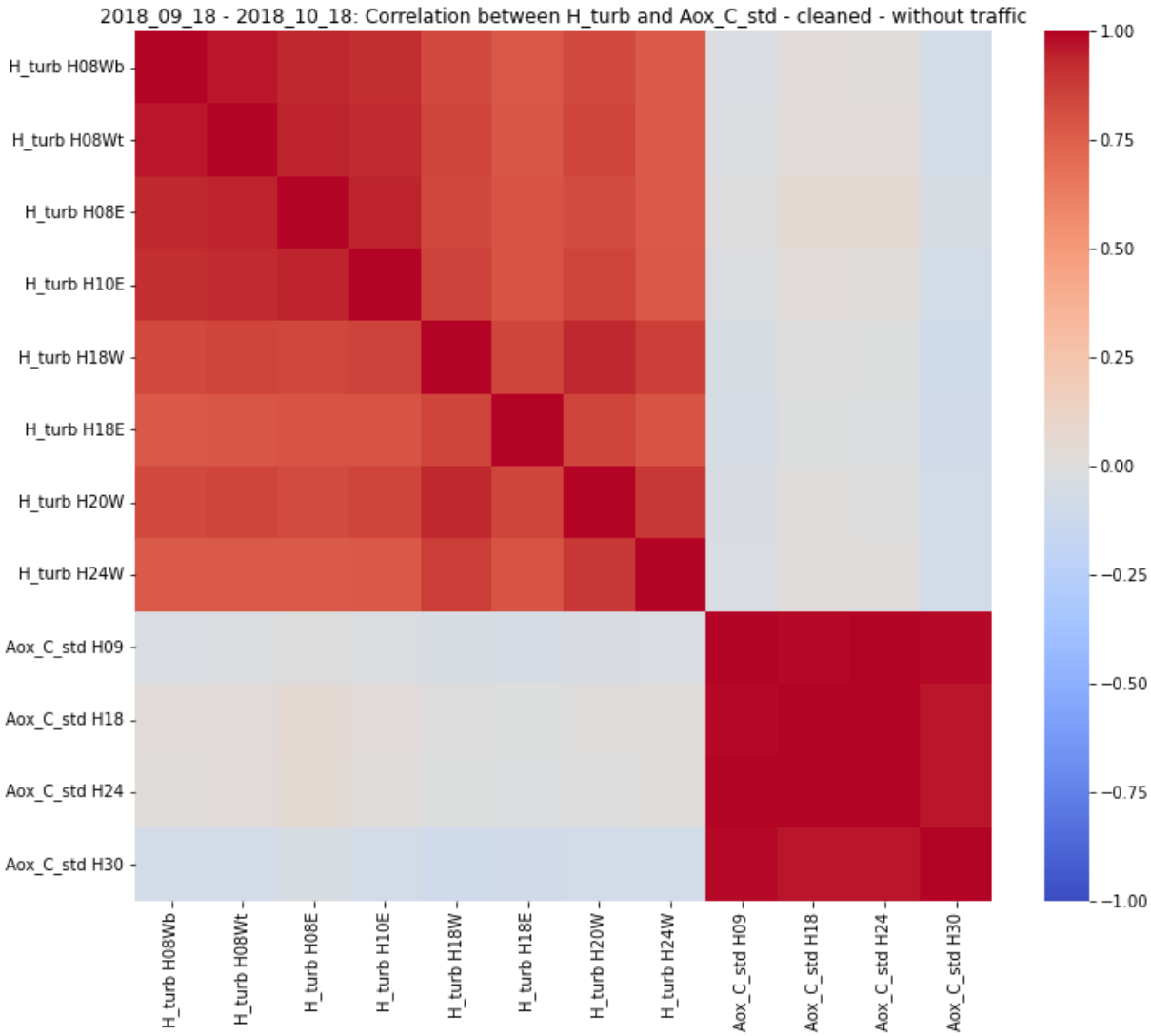

2018_09_18 - 2018_10_18: Correlation between H_mean and Aoz_C_std - cleaned - without traffic
H_mean H08Wb
H_mean H08Wt
H_mean H08E
H_mean H10W
H_mean H10E
H_mean H18W
H_mean H18E
H_mean H20W
H_mean H24W
Aoz_C_std H09
Aoz_C_std H18
Aoz_C_std H24
Aoz_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

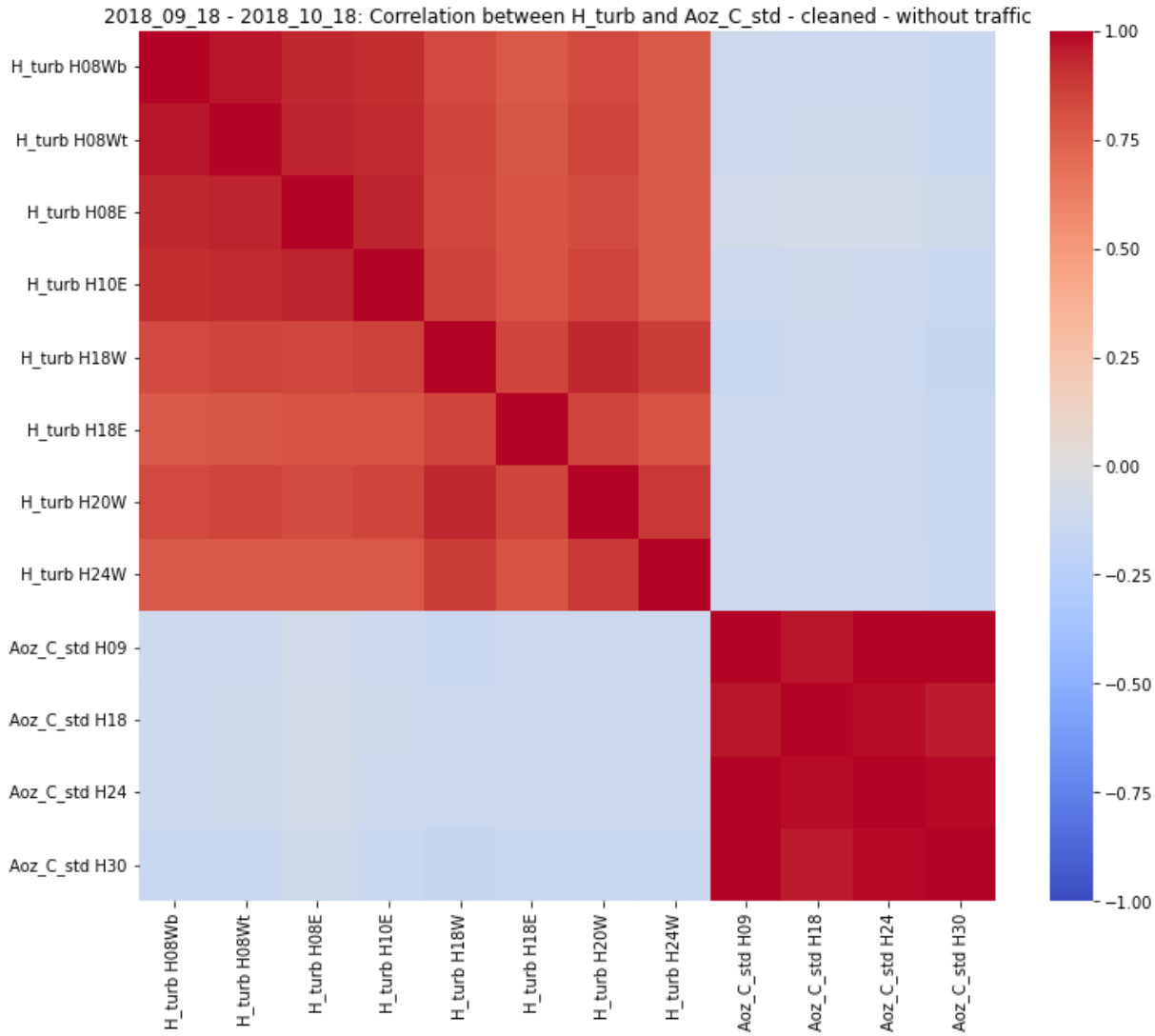

2018_09_18 - 2018_10_18: Correlation between H_mean and theta_std - cleaned - without traffic
H_mean H08Wb
H_mean H08Wt
H_mean H08E
H_mean H10W
H_mean H10E
H_mean H18W
H_mean H18E
H_mean H20W
H_mean H24W
theta_std H09
theta_std H18
theta_std H24
theta_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

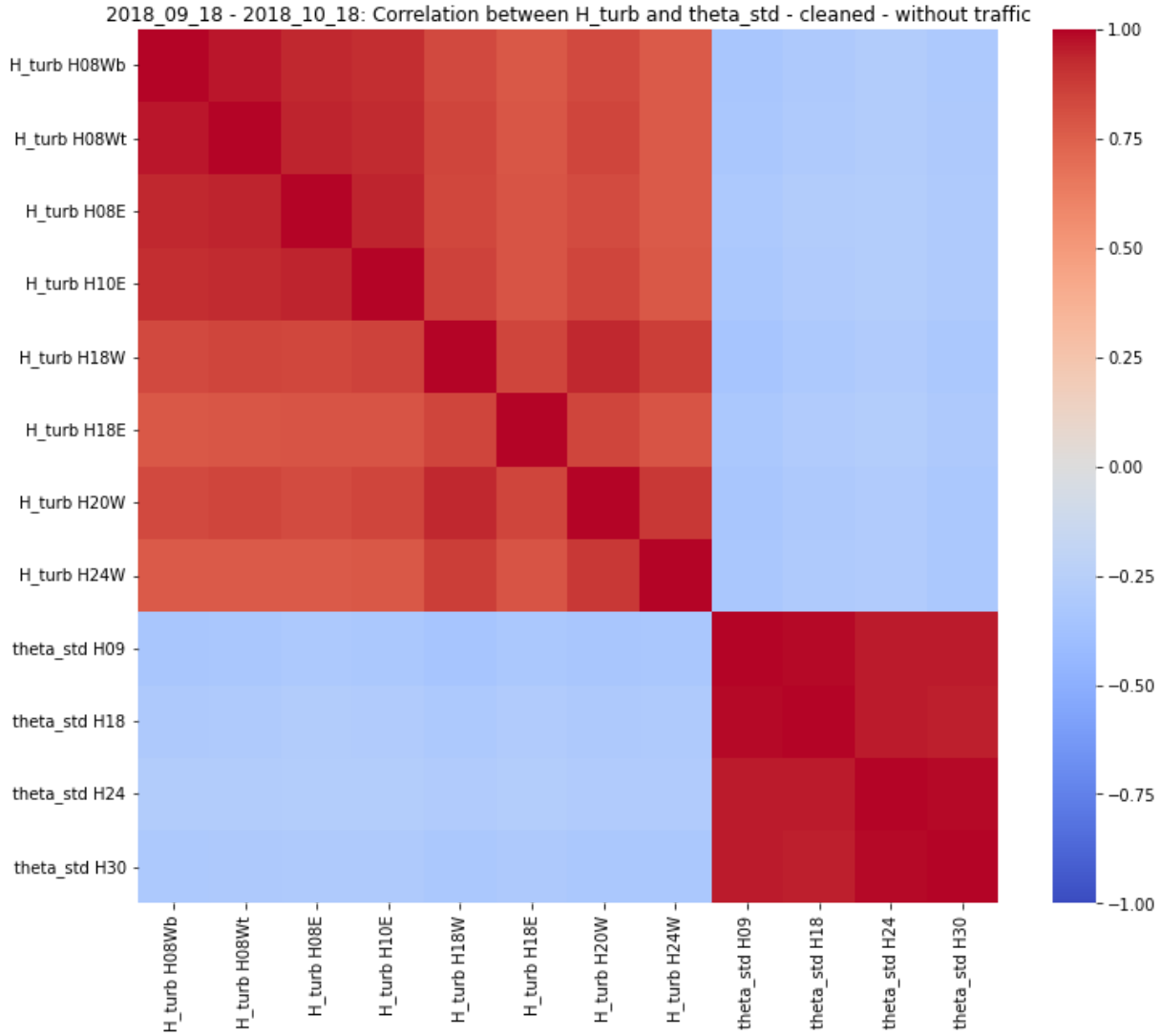

2018_09_18 - 2018_10_18: Correlation between AOA_mean (absolute) and Aox_C_std - cleaned - without traffic
AOA_mean (absolute) H08Wb
AOA_mean (absolute) H08Wt
AOA_mean (absolute) H08E
AOA_mean (absolute) H10E
AOA_mean (absolute) H18W
AOA_mean (absolute) H18E
AOA_mean (absolute) H20W
AOA_mean (absolute) H24W
Aox_C_std H09
Aox_C_std H18
Aox_C_std H24
Aox_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


2018_09_18 - 2018_10_18: Correlation between AOA_mean (absolute) and Aoz_C_std - cleaned - without traffic
AOA_mean (absolute) H08Wb
AOA_mean (absolute) H08Wt
AOA_mean (absolute) H08E
AOA_mean (absolute) H10E
AOA_mean (absolute) H18W
AOA_mean (absolute) H18E
AOA_mean (absolute) H20W
AOA_mean (absolute) H24W
Aoz_C_std H09
Aoz_C_std H18
Aoz_C_std H24
Aoz_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


2018_09_18 - 2018_10_18: Correlation between AOA_mean (absolute) and theta_std - cleaned - without traffic
AOA_mean (absolute) H08Wb
AOA_mean (absolute) H08Wt
AOA_mean (absolute) H08E
AOA_mean (absolute) H10E
AOA_mean (absolute) H18W
AOA_mean (absolute) H18E
AOA_mean (absolute) H20W
AOA_mean (absolute) H24W
theta_std H09
theta_std H18
theta_std H24
theta_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


2018_09_18 - 2018_10_18: Correlation between H_turb and Aox_C_std - cleaned - without traffic
H_turb H08Wb
H_turb H08Wt
H_turb H08E
H_turb H10E
H_turb H18W
H_turb H18E
H_turb H20W
H_turb H24W
Aox_C_std H09
Aox_C_std H18
Aox_C_std H24
Aox_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

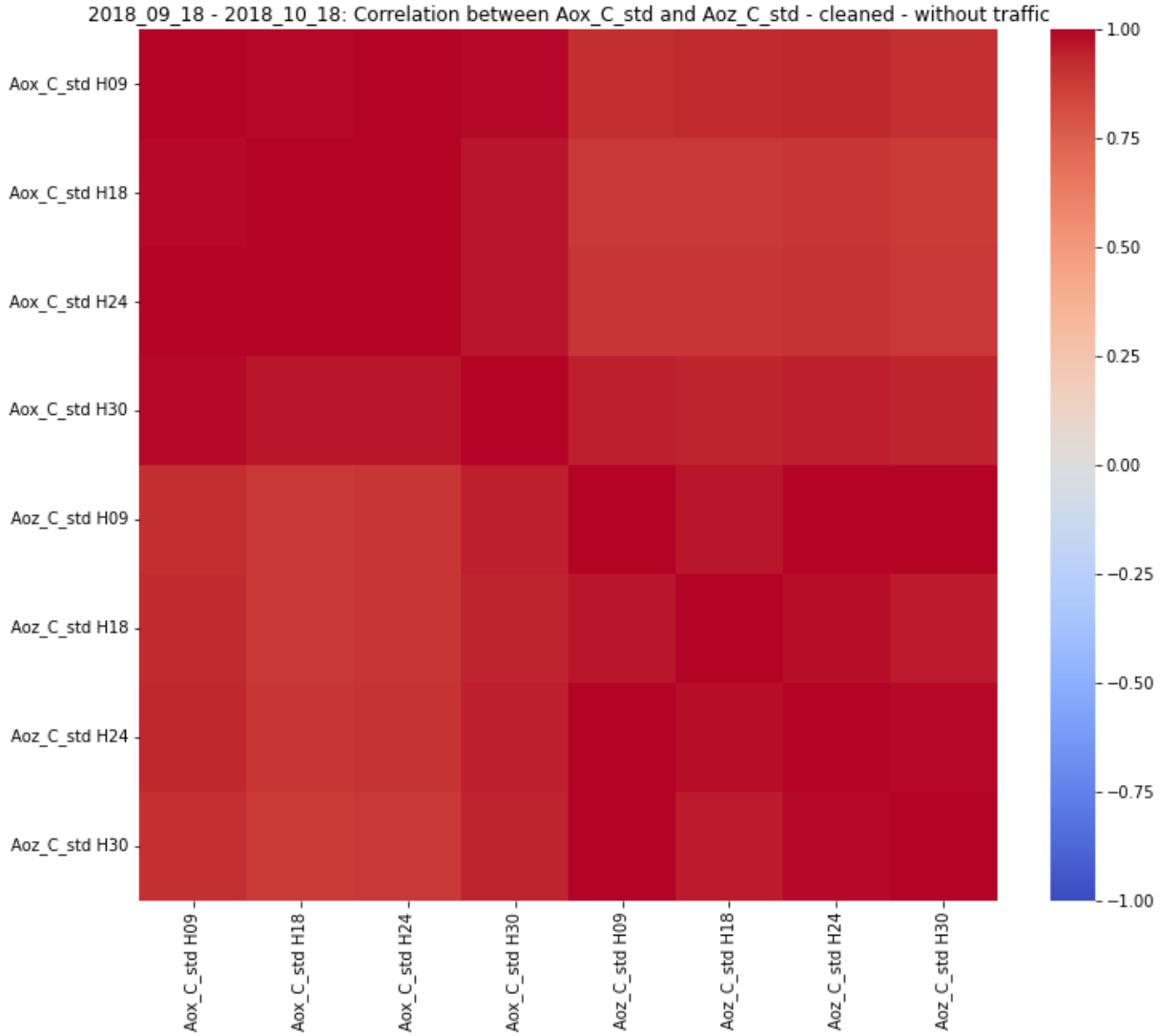

2018_09_18 - 2018_10_18: Correlation between H_turb and Aoz_C_std - cleaned - without traffic
H_turb H08Wb
H_turb H08Wt
H_turb H08E
H_turb H10E
H_turb H18W
H_turb H18E
H_turb H20W
H_turb H24W
Aoz_C_std H09
Aoz_C_std H18
Aoz_C_std H24
Aoz_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

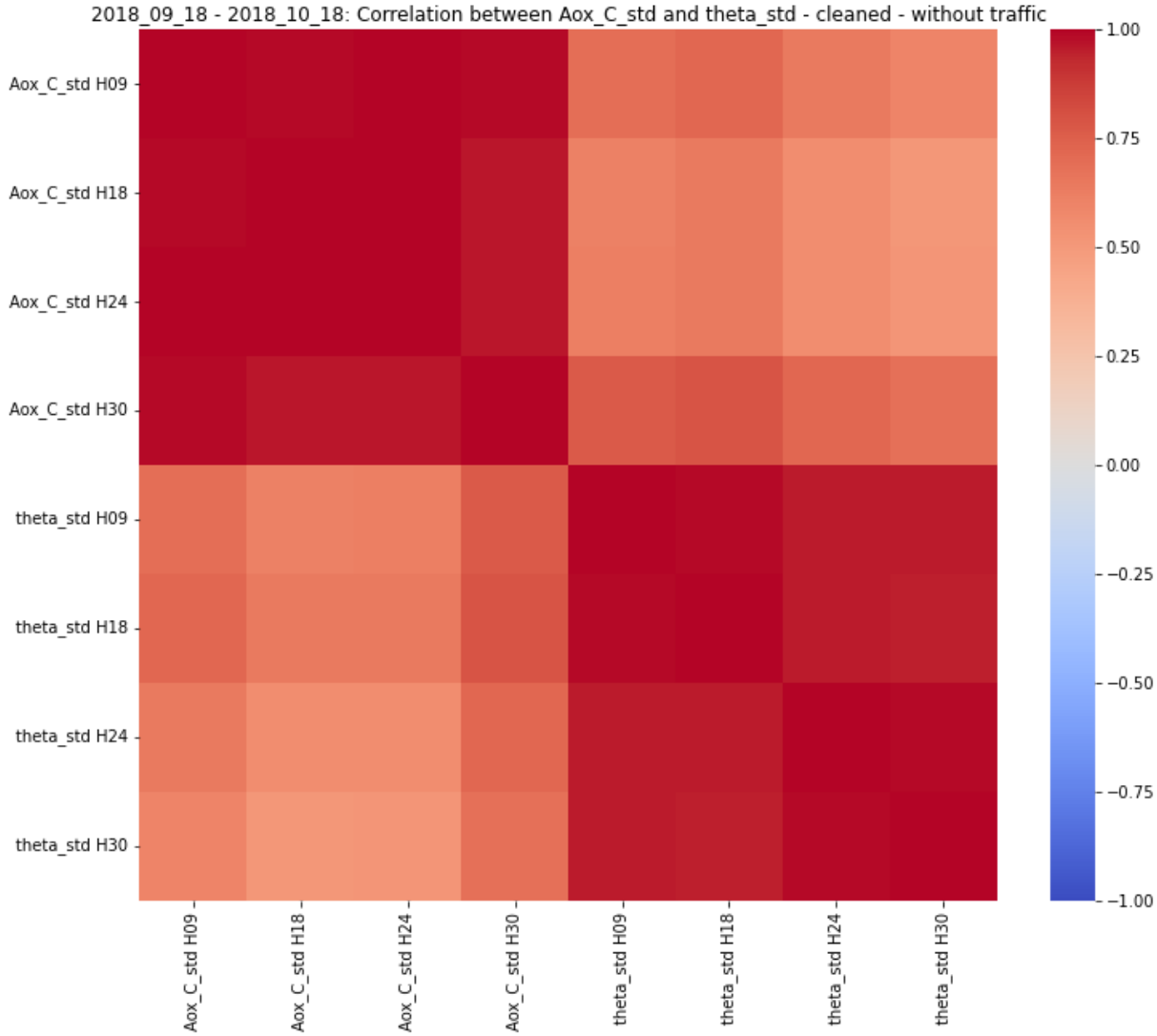


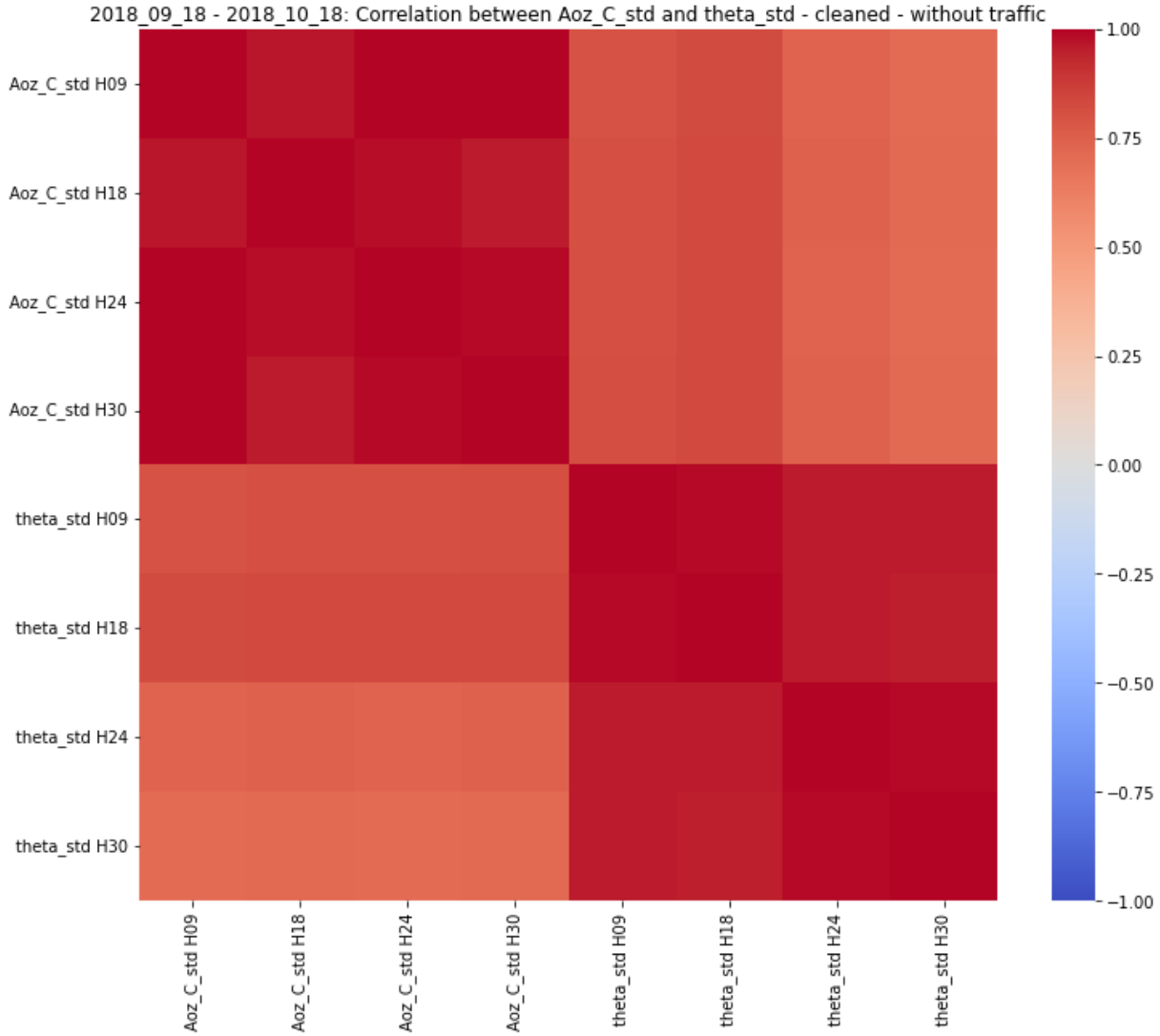
Acc & Acc - Acceleration characteristics

In [ ]:
```
filter_idxs = np.arange(len(LFB.time_array_cleaned)); title_suffix = ' - cleaned'
filter_idxs = LFB.filter_data(LFB.traffic_cleaned,prior_idxs=filter_idxs,zeros=True,
```

In [ ]:
```
for key1_num, key1 in enumerate(LFB.acc_cleaned.keys()):
    if key1 in keys_of_interest:
        for key2_num, key2 in enumerate(LFB.acc_cleaned.keys()):
            if key2 in keys_of_interest:
                if key1_num < key2_num:
                    LFB.correlation_matrix(LFB.acc_cleaned[key1].T[filter_idxs].T,LF
```


2018_09_18 - 2018_10_18: Correlation between Aox_C_std and Aoz_C_std - cleaned - without traffic
Aox_C_std H09
Aox_C_std H18
Aox_C_std H24
Aox_C_std H30
Aoz_C_std H09
Aoz_C_std H18
Aoz_C_std H24
Aoz_C_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


2018_09_18 - 2018_10_18: Correlation between Aox_C_std and theta_std - cleaned - without traffic
Aox_C_std H09
Aox_C_std H18
Aox_C_std H24
Aox_C_std H30
theta_std H09
theta_std H18
theta_std H24
theta_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


2018_09_18 - 2018_10_18: Correlation between Aoz_C_std and theta_std - cleaned - without traffic
Aoz_C_std H09
Aoz_C_std H18
Aoz_C_std H24
Aoz_C_std H30
theta_std H09
theta_std H18
theta_std H24
theta_std H30
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00