Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Visual analysis and comparison of seismic events

Semester Thesis

Tobias Kuonen
tkuonen@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Matthias Meyer
Andreas Biri
Prof. Dr. Lothar Thiele

December 23, 2019

Acknowledgements

I would like to thank Matthias Meyer and Andreas Biri for their continuous support during this project. I would also like to thank Jan Beutel for the insight into the geophone platform he provided. Furthermore, I would like to thank Prof. Dr. Thiele for supervising this thesis. This work would not have been possible without them.

Abstract

Continuous environmental monitoring using distributed sensor networks produces an enormous amount of data. Getting an understanding of such a big data set can be hard. Having a visualization tool can help a lot in such a case by displaying the relevant data extracted using various filters. Recently, a new seismic sensor platform was deployed in the Swiss Alps, which in contrast to previously deployed seismic sensors records single events instead of continuous signals. Therefore, a new visualization tool is needed, which allows to analyse and compare this new type of seismic data. This thesis presents such a visualization tool. By using user specifiable processing algorithms utilising parameters, which are dynamically changeable in the user interface, we enable the filtering of signals in a flexible manner. We show, that our tool enables clean, flexible visualizations which run normal consumer devices without having to rely in a powerful server.

Contents

Chapter 1

Introduction

1.1 Motivation

The increasing annual mean temperature can destabilize rock slopes [1]. Continuously monitoring such rock slopes helps to better understand the involved processes. This can be done using distributed sensors networks. Such measurements however can produce a large quantity of data. One example of such a dataset is the data collected by the PermaSense project. The PermaSense project collects data using 17 different sensor types at 29 distinct locations in the Swiss Alps. This leads to over 114.5 million data points collected so far in the last 10 years [2] (not considering seismic data). Analysing such a big dataset by hand is not feasible. Therefore, automatic analysis methods are needed. A good understanding of the entire data set is crucial in order to use such methods effectively. A visualization tool can help achieving this by displaying the relevant information extracted by various filters.

1.2 Goals

The goal of this thesis is to develop a flexible visualization tool for the seismic data of the PermaSense project with the following features:

- Ability to compare seismic data from different sensors at different points in time
- Integration of a newly developed seismic sensor platform [3], which records single events instead of continuous seismic signals
- Ability to switch dynamically between different time scales

1.3 Outline

The remainder of this thesis is structured in the following way: The background of this project is briefly presented in section 2. In section 3, the basic concepts and design choices will be elaborated. The actual implementation is discussed in section 4. In section 5, the result is evaluated. Possible future work is presented in section 6. In the end, a conclusion is drawn in section 7.

Chapter 2

Background

2.1 Previous work

This thesis builds upon a lot of previously existing work. This work is briefly presented here.

Since 2008, the PermaSense project collects various measurements on the Matterhorn and other locations in the Alps [2]. Most of these measurement data is publicly accessible at `http://data.permasense.ch/`. The visualisation tool developed in this thesis directly integrates the data collected by this project.

In 2018, a new type of seismic sensor platform has been developed: An embedded system with a single axis geophone sensor. Instead of continuously recording data as traditional systems, the system only samples its sensor after the signal exceeds a configurable threshold in order to reduce its energy consumption [3]. The downside of this measure is that there can be large time spans without any samples which means the signal has no uniform sampling rate. All algorithms relying on a uniform sampling rate can therefore not be used directly with this type of data, here called event-based data.

In a previous semester thesis [4], a visualization tool has already been developed, which enabled the user to analyse continuous seismic data but lacked support for event-based data and for comparing multiple sensors or time frames. Another tool was written by Jan Beutel in R, which is an interpreted programming language designed for statistical computations [5],to analyse the event-based data. It however only contained static plots and had no possibility to integrate user interaction. Therefore, a new tool was needed.

This year, a new data analysis framework called "Stuett"[6] was developed by Matthias Meyer. This framework allows access to the continuous sensor data of the PermaSense project. Furthermore, it enables one to assemble filters for data analysis represented by directed graphs, where each node stands for a processing algorithm and each edge for a data dependency between these algorithms. The terminology "processing graph" is used in the rest of this thesis to refer

to such graphs. Stuett creates these processing graphs using Dask[7] which is a framework exactly designed for this purpose. Xarray[8], a data structure for labelled multi-dimensional arrays [8], is used as a container for the measurement data. Stuett is written in Python[9], a interpreted object oriented programming language which is often used in data science [10, 11].

2.2 Sensor types

The PermaSense project collects data from various sensor types. This section present those integrated into the visualization tool.

Two types of seismic data are present. One is the event-based seismic data presented in the section above and the other one is continuously sampled seismic data. The main focus of the visualization tool lies on analysing these two types of data.

The context is very important when analysing data. Webcam images for example can provide information whether a seismic event was created by a mountaineer or by a rockfall event [1]. For this reason, various additional information sources are displayed in the visualization tool. These consists of weather data, rock temperature data and the already mentioned webcam images.

The data is retrieved from two storage systems. One such system is the GSN server, which provides public access to most of the measurement data. This data can be found at `http://data.permasense.ch` and is grouped by data type into so called "vsensors". The other system is a private network drive called the "permasense vault". This storage system is used for seismic data and the webcam images.

[1] Webcam images were used in conjunction with other data sources to classify seismic events in [12].

Chapter 3

Concepts

The design choices made during the development of the visualization tool are presented here together with their reasons.

3.1 Basic design

Having the possibility to outsource the computation-intensive data processing to a powerful server can be an advantage when analysing big amount of data. Therefore we made the choice to separate the user interface from the data processing by using web technologies. This approach has the additional advantages of supporting the integration of the visualization tool into an existing webpage and of supporting to display the user interface on a broad range of devices, as almost any internet capable device supports these web technologies.

The back-end server, which is performing the actual computations, was created using Python[9]. The main reason for this choice is that the existing processing framework "Stuett"[6] is written in Python.

A framework called Dash[13] is used to connect the graphical user interface (GUI) with the Python server. This allows to easily create interactive web pages using Python and bind events of the user interface to function calls in Python.

The communication between the browser and the Python server works as follows (slightly simplified): Upon changing parameters in the user interface, these parameters are send to the Python server, which assembles and executes the processing graph using these parameters. This processing graph acquires the original data from the configured data storages and processes them in order to produce the relevant data, which is sent back to the browser. The browser then displays this relevant data. This communication process can also be seen in figure 3.1

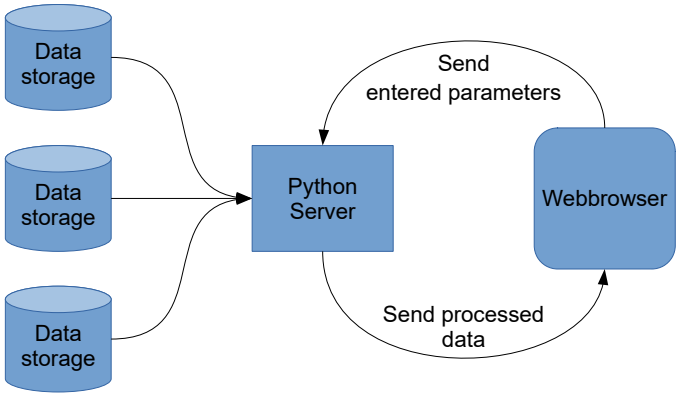


Figure 3.1: The parameters entered in the web browser are sent to the Python server, which fetches the original data from the data storages and then sends back the fully processed data.

3.2 Extensibility

To easily extend the visualization tool with new data sources, processing algorithms, plot types and GUI elements, its Python code is split into modular components. These components are separated into data processing (for example seismic event loader or event downsampling) and user interface logic (for example image viewer or date and time selector). These modules are automatically connected upon initialization using a configuration file. With this subdivision into individual components, adding a new feature is as easy as writing the according module and adding it to the configuration file. This separate configuration also allows a user to modify the layout of the user interface according to his needs.

3.3 Data processing

3.3.1 Processing graph

There are two possibilities to add a parameter to a processing graph. These will be presented here.

The most intuitive way to add a parameter to a processing graph is by adding an additional node, which has no inputs and always returns the constant value of this parameter. There are however cases, where this method does not work. This is the case if the value, which needs to be added to the beginning of the processing graph depends on some state of a node in the middle of the processing graph. If a node for example caches some measurement data, then the parameter specifying

the time frame to load the data from depends on the values already stored by the cache. Stuett solves this by propagating all such parameters backwards through the processing graph. This makes it possible for all nodes to modify or store these parameters as needed.

3.3.2 Analysing event-based data

To be able to analyse event-based data using the existing tools developed for continuous signals, this data first needs to be converted into a continuous signal. This can be done by combining all the data from the events and filling the rest with zeroes. Sharp edges in the resulting signal can be prevented by first multiplying every event with a window function. A tukey window [14] is used in order to not loose too much information from the beginning of the events.

Each event has its own metadata. Examples for these metadata are the start time, the end time and the maximum peak value. The presence of these metadata offers additional analysis opportunities. To leverage this potential, a flexible filter based on the metadata, and a statistical plot of the event length are available for the user to display additional information and easily focus on the scope of interest.

3.3.3 Supporting different time scales

The original data from large time frames can have a lot of samples. A sensor continuously recording with 1000Hz for example creates 3.6 million samples per hour. A computer screen however only has a limited resolution. Therefore, displaying all the samples slows down the application without providing any additional information. This visualization tool uses downsampling in order to reduce the amount of data. Depending on the time frame and the type of data, a different downsampling method and rate is used. The selection of these methods and rates is based on [4]. The implementation of the downsampling methods from the Stuett library is used.

When working with event-based data, an additional step is performed after downsampling. Multiple events are merged into a single big event, if they would appear to be too close in the resulting graph. There are two reasons for this choice. One is performance: When analysing a large time frame, the number of events would be rather big. This could considerably slow down the application. Another reason is the usability: When looking at a big time frame, a user is generally not interested in the properties of every single event, but rather in the properties of the combination of those. An example is the event length property. When looking at a short burst of events within a large time frame, it might be more interesting to know the length of the burst rather than the length of every single event. Therefore, it makes sense to combine the events

and their properties. Depending on the screen resolution, the sampling rate, the user preference and other factors, the needed threshold can be different. In order to accommodate for these different needs, the threshold is a configurable parameter. The threshold is specified relative to the sampling order for it to work across different time scales.

3.4 User interface

3.4.1 Basic structure of the plots

The basic structure of the plots in the user interface is presented in this section.

There are multiple plot types in the user interface, created from different data sources. An example of such a plot type is the plot of the weather data or the plot of the seismic data. Contextual information, for example provided by the plot of the weather data, is only useful when it is from the same time frame as the seismic plot currently analysed. Therefore, an instance of each plot type is combined into a group using a shared time axis. There are multiple such groups using the same plot types in order for the user to be able to compare data from different point in time. In the user interface, the plots are ordered by plot type rather than by group to facilitate the comparison of these plots.

3.4.2 Spectrogram

Spectrogram plots visualize both the features of the time space as well as the features of the frequency space. This makes these plots quite useful for analysing seismic data. Therefore, this plot type is used as the main analysis tool of this visualization tool. The effectiveness of spectrograms in seismic data analysis can be seen in [12]. The spectrogram implementation from the Stuett library is used in the visualization tool.

3.4.3 Separation of plots and settings

The developed visualisation tool has an external settings window. This section motivates this choice.

When working with several complex data processing graphs, the number of configurable parameters can be fairly big with each parameter requiring an accompanying GUI element. To not overload the main window with these elements, they were moved into another window. An alternative to this solution would be to leave these elements in the main window and add the option to hide them. This alternative however would have the disadvantage that when having multiple

monitors, one would not have the possibility to show the plots on one monitor and the GUI elements for the parameters in the other one.

Such a setup achieves to goal of providing a clean visualization while preventing the need to constantly switch between the plots and the settings.

Chapter 4

Implementation

The actual implementation can be separated into two parts: GUI logic and data processing. The basic design of the most important modules of both parts will be elaborated in this chapter.

4.1 Configuration

The entire user interface is constructed according to a configuration file in order to have a flexible and customizable visualization tool. This also simplifies adding new data sources, new plot types, new processing algorithms and GUI objects.

The configuration contains five entries. The first one specifies how many instances of each plot are to be added to the user interface. These different instances are used to compare different sensors and points in time as described in 3.4.1.The next two entries specify the start and the end time of the initial time frame. Each of those are specified as a list, to support different initial time frames for the different instances of the plot. The last entry is taken multiple time, in the case that the number of instances is higher than the number of entries in these lists. The third entry is a list of all the configurations for the graph objects. These are further explained in 4.3.2. The last entry is a list of configurations for other GUI objects. Each such configuration contains a reference to the class, which is used to create the object. Any GUI object can be added here, with the limitation, that its initialization function needs to accept the same parameters as the graph class. Currently, the only class that fits this requirement is the images class, described in 4.3.3.

The exact syntax of this configuration file is described in appendix B.

4.2 Data processing

The entire data processing is done using processing graphs in order to allow filtering the signal in a flexible manner. These processing graphs are created using Stuett[6], a framework developed exactly for this purpose. In order to support event-based data, several nodes for this processing graph have been created. These will be presented here.

4.2.1 Loading of seismic events

Stuett has no node to fetch the seismic event data. Therefore, such a node was created for this visualization tool.

Source of the data

The seismic event data consists of the metadata and of the seismic waveforms. The metadata and the waveforms are fetched from the GSN server and the permasense vault.

Most of the event data on the GSN server consists of metadata only. These metadata are wirelessly transmitted to a base station and then uploaded to the GSN Server. Therefore, these metadata are available rather soon after acquiring. However due to transmission errors in the wireless communication, these do not need to be complete. Some events also have the seismic waveforms on the GSN server. This however is only the case, if it was manually queried.

The data on the permasense vault comes from the on-site readouts. Therefore, this data is the most complete one. However, due to the fact that on-site maintenance work does not happen very often, it can take a long time until the data of a newly captured event is added to the permasense vault.

To have the data as up-to-date and as complete as possible, both sources are combined. If an event is present in both sources, the permasense vault is preferred. In the case, that only the metadata of an event is available (when the data has not been read out on-site and not been queried manually), these events are still returned, because he metadata themselves can be interesting to analyse (frequency of the events,event duration, maximum and minimum peak values, ...).

Assumed maximum event length

When requesting event data of a given time frame, it is assumed that the duration of an event is less than one hour. The reason for this assumption is that the metadata of an event only contain the start but not the end time of it. Therefore,

when requesting event data from the GSN server, it is not possible to filter according to the end date, which is necessary when one also wants to include events that started before the given time frame but ended during or after the time frame. Without the assumption of a maximum event duration, one would need to request all event metadata from the beginning of the measurement, until the end of the time frame.

With the assumption of a maximum event duration however, it is sufficient to request the data which started in between one hour before the beginning of the time frame and the end of the time frame. Only these events can possibly intersect with the given time frame.

This is a big reduction in data which needs to be requested from the server and therefore also in delay. This can be seen in 5.2. The correctness of this assumption will be shown in 5.1.

The event data on the permasense vault

The original measurement data on the permasense vault is split into multiple directories. Every deployment has its own base-directory [1].

Inside this base directory, the data is ordered first by readout date, then by device id then by date. The path to the folder, containing the measurement data can be constructed as `<BasePath of deployment>/<Name of readout directory>/<device id>/<YYYY-MM-DD>/`. In some cases however, there are deviations from this template (e.g. `pos_<position id>_<device id>` instead of `<device id>`). Furthermore, the mapping between device id and position id is not static but can change over time. The mapping can be found on `http://data.permasense.ch/topology.html` (tab "Position Mapping"). It can also be retrieved as an xml from the virtual sensor "<deployment>_maping_chart".

Due to these facts, requesting all events which occurred at a specific position during a given time frame using this folder structure would require several intermediate steps [2],which would make it quite complex.

To make the requesting of the data simpler and more efficient, a script was developed, which creates an alternative folder structure. To prevent duplicate data on the filesystem, symlinks to the original data are used. In this new folder structure, all the data are first ordered by deployment, then by position id, then by date. The new path can be constructed as `<constant BasePath>/<Name of deployment>/<position id>/<YYYY-MM-DD>/`. In each of

[1] e.g. data_archive/deployments/matterhorn2007/2018_gpp_data for the deployment on the Matterhorn

[2] First, determine the base-directory for the given deployment. Then iterate over all readout folders. Then determine the (possibly varying) mapping between position id and device id and then iterate over all the necessary device id folders

these date folders there are symlinks to all corresponding folders containing the measurement data of the original folder structure. There might be multiple such folders in case the mapping of position and device id changed in that particular day. The symlinks have integer values as a name, starting from zero up to the number of symlinks in the current folder minus one.

Inside the folders containing the measurement data, there is a file which lists all the events contained in that folder together with their metadata. The waveform of each event is stored in binary form in a separate file in the same folder.

The seismic event data on the GSN server

The seismic event data on the GSN server is split across two virtual sensors: "<deployment>_dpp_geophone_acq___conv" and "<deployment>_dpp_geophone_adcdata___conv". The first sensor contains the metadata and the second one the waveforms of the events. Assembling the URL to retrieve this data is explained in [2] and [15].

Converting the seismic data

On the device, the geophone sensor is connected to an ADC. This ADC first amplifies this voltage, and then converts it to unsigned integers. This is then stored in binary form on an sd card (24bit[3] unsigned integers, big endian).

To get back the actual output voltage of the geophone sensor, the binary data is first converted to a list of unsigned integers. The programming code used to to this conversion is based on a script, created by Reto Da Forno. The resulting integers are then mapped to mV using the formula (4.1). The derivation of this formula can be found in appendix D.

$$f(x) = \frac{2500}{g \cdot (2^n - 1)} \left(x - \frac{2^n - 1}{2} \right) \tag{4.1}$$

4.2.2 Event downsampling

Downsampling event-based data works by first applying a given downsampling method to all events. Afterwards, it is tested, if the gap between any two events is smaller than a given threshold. This threshold is given by a constant divided by the new sampling rate, where the constant is a tuneable parameter. If this condition is met, then the waveforms of these events are merged. After merging,

[3]The binary data on the GSN server was sometimes converted to a lower bit depth before transmitting.

the start and the end time of the event is adjusted in the metadata. Furthermore, a counter in the metadata representing the number of merged events gets incremented. The rest of the metadata is taken from the first event. The metadata of the later event gets copied into a list in the new metadata in order to avoid loosing any information.

4.2.3 Converting event-based data to continuous data

To be able to use the existing data analysis tools, a node for the processing graph was created, which converts a list of events to a continuous signal. This conversion is done by first combining the samples of all the events. Then, the data is resampled using the lowest sampling rate of all the events. The value of the closest point in the original data is used as the new value, if it is not further away than 1 divided by the sampling rate. If it is further away, then zero is taken as the new value in order to prevent having a constantly high value in between events.

4.2.4 Multiplying event data with a window function

When converting event-based data to continuous data, sharp edges can occur at the boundaries of events. This can be prevented by first multiplying every event with a window function. A node for the processing graph has been created to do exactly this multiplication. Upon receiving a list of event data, the node generates a window function for every event of the same number of samples as the waveform of the event itself. This window function and the waveform are then multiplied sample by sample using a corresponding function from the Xarray library. The samples of the window function are generated using a Python function, taking the number of samples as an argument and returning the samples of the window function. This Python function can be specified as an argument when initializing the node in order to support arbitrary window function. A function from the SciPy[16] library is used, to generate the samples of the tukey window used in this visualization tool.

4.2.5 Selecting the data columns to visualize

The used data can have several columns. The weather data for example includes among others the wind speed and the relative humidity. Displaying too many columns can overload the GUI. To prevent this, a node for the processing graph was created to filter these columns.

The available columns can differ between the sensors, even if they are of the same type. It would be possible to select the columns to show by name. This however has some disadvantages. To be able to select any column, it would be

necessary to know the names of the available columns. Hard coding these names into the visualization tool would be possible, however, it would be inflexible and would require modifications every time the available columns would change. Another possibility would be to take the names of the columns from the last execution of the processing graph. However, this would only work if there had been a previous execution using the currently selected sensor. All the times, where this would not be the case, no columns could be selected. A solution for this would be to execute the processing graph twice in these cases. This solution however would cause unnecessary computations.

To circumvent these disadvantages, the columns to show are not selected by name but rather by the index they have in an alphabetically sorted list of all currently available columns. In order to still be able to show the names of all the columns which a user can select, the name of all dropped columns are added to the metadata of the data. This way, the sorted list of column names can be reconstructed in the control logic of the GUI. In the case, that there was a previous execution of the processing graph using the same sensor, nothing changes compared to the method mentioned above. However, when changing the selected sensor, the number of selected columns stays the same (if the new sensor has enough columns). This makes it possible to initially select the first n columns without even knowing what columns are available.

In the data, there might be some columns we always want to drop if they are present. One example for this could be the current position id. To support this, a list of such column names can be given to the filter. Not only will these columns always be dropped, but they will also be excluded from the sorted list of columns and from the list of dropped columns added to the metadata. This way, these columns will not show up in the column selection of the GUI.

4.2.6 Filter events using the metadata

Every event has metadata associated with it. Examples for this metadata are the start time, the end time and the maximum peak. To support filtering these events in a flexible manner using these metadata, a node for the processing graph was created which accepts a normal python expression as the filter rule. All those events for which the expression evaluates as false will be discarded.

Having the possibility to enter a python expression as a filter rule gives the opportunity to filter according to arbitrary complex policies. However, allowing the execution of arbitrary expressions from an untrusted source can be a serious security thread. This thread was mitigated by creating a custom parser which only allows the execution of selected operations. Such a parser can easily be created using the ast module of the Python standard library. This module allows one to convert a python expression into a abstract syntax tree. This tree can then be traversed node by node to evaluate the expression using only the selected

operations. The list of the selected operations can be found in C.

The current implementation of the parser does not limit the resources used to evaluate the expression. Therefore, it might be possible to create a malicious expression which exhausts the resources of the server. However, considering the expensive computations that can be started using the web interface, implementing a resource limit should be implemented for the entire processing graph rather than for a single node in it. Furthermore, the number of processing graph executions per user per time would also need to be limited. Otherwise, a malicious user could just overload the server with too many requests.

4.3 GUI logic

The GUI is created using Dash[13], as already mentioned in 3.1. The basic workflow when using Dash is to first assemble the basic HTML layout using simple components, such as text inputs, div tags or a graphical plot. Each such component can have several properties, such as the entered text in a text input. These properties are then connected using so-called "callbacks". These are just Python functions, which take the values of some properties as the input and produce the values of some other properties as the output. Every time, one of the input changes, these functions are executed in order to update the resulting properties [13].

The GUI logic was split into reusable objects in order to support having an arbitrary number of plots in the user interface. These objects have two main functionalities. The first one is to provide the part of the HTML layout needed to display the object. The second one is to provide the needed callbacks, either by creating them directly or by providing the information needed by another object to integrate the properties of the current object into the callbacks of the other one. The rest of this section will present the objects available in this visualization tool.

4.3.1 Date and time selection

In the GUI, in order to be able to compare the data from multiple points in time, we can have several groups of plots. To be able to efficiently select the time frame for each of these groups, an according selector is added to the top of the main window for each of these groups. In order to filter out invalid time frames, for example when the start time is later than the end time, the last valid time frame is stored inside the browser. All callbacks use this stored time frame except the one which updates the stored value. This prevents the execution of the processing graph with invalid inputs.

4.3.2 Graph class

In order to analyse one type of data, multiple plots might be needed. All these plots are based on the same data. If all these plots would be updated individually, this data would need to be computed over and over again. To prevent these unnecessary computations, the graph class was created. It displays multiple plots, which are all updated together. To also be able to compare the data from multiple sources or time frames, multiple instances of this graph class can be created. The HTML layout of these instances are not added one after another to the main interface but rather in an interleaved fashion, in order to display all identical plot types next to each other. This facilitates the comparison of information.

The content of a graph object is defined by an entry in the configuration file in order to be able to visualize any type of data with as many plots as needed. This configuration consists of three parts. The first part specifies the name displayed as a title of the plots. The second part specifies all the sources which can be used to create these plots. Each source is specified by a processing graph, here called "source processing graph", which describes how to retrieve and process the data. Each source also contains a list of parameters used by the source processing graph, which can be set in the settings window. Only one source is used at a time. The currently active one is selected in the settings window. The third part specifies all the plots which can be displayed by this graph object. Each plot is specified by a processing graph, here called "plot processing graph", which describes how the data given by the source processing graph needs to be processed further for the browser to be able to display it. This includes adding layout information such as the plot type or the axis title. Each plot also comes with a list of parameters, which can be set in the settings window. All these plots can individually be hidden in the settings window. The exact syntax of the configuration file can be found in appendix B.

The selected time frame can not only be set in the "date and time selector" described in 4.3.1 but also in all the plots with a time axis, which use the same "date and time selector". Therefore, to keep everything consistent, the current time frames of all such plots need to be inputs to the callback, which updates these plots. This measure however would lead to unnecessary computations, since a change in the time axis of one plot would cause an update of all the other plots, which would change the time axis of all these plots and therefore again trigger an update of all the plots. This is prevented by saving the time frame used for the last update together with the used parameters from the settings in the browser. An update is then only performed if any of these values have changed. These values are saved in the browser rather than the server in order to be able to support having multiple users.

The graph class has two main callbacks. One takes all the parameters from the settings objects, the time frame from the "date and time selector" and the

time frames from all the plots using the same "date and time selector" as an input and produces the values stored in the browser as the output. The other one takes these values as an input and produces the content of all the plots as an output. To obtain this output, first, the processing graph is assembled from the selected source processing graph, the plot processing graphs of the not hidden plots and all the stored values. This processing graph is then executed. The values returned at the end of each plot processing graph is then taken as the content of the corresponding plot. Every plot has one additional callback. This callback hides the plot when the according parameter is set in the settings window.

4.3.3 Images class

In order to provide some contextual information in the form of (webcam) images to the measurement data, a image viewer class was created. The basic idea and design choices of this image viewer are based on [4]

The image class displays three images from the current time frame. One image each is taken from the beginning and the end of the current time frame. Another image is taken from somewhere in between, selected by a slider.

The images are loaded from with a processing graph, specified in the configuration file, in order to support arbitrary image sources. These processing graphs are expected to return a base64 encoded image together with the image type such as "jpg". These images are then converted into a data-URL, as described in [17] and then used as the "src" attribute of the image. The reason for this choice is, that Dash[13] only accepts URLs as the content of images, not images directly. An alternative solution would be to create a second server for these images. This would make the transmission of an image much more efficient. However, the implementation would get much more complex in order to still support an arbitrary number of image objects each with a possibly different processing graph to load the images. The overhead caused by the data-URL was preferred over the complexity of the image loader, due to the fact that only three images need to be loaded per graph object per update. Therefore, the overhead should not be too high.

It is not always necessary to update all the images. For example, the first image does not need to be updated when only the end the time frame changes. Such unnecessary updates are prevented by saving the time used to load each image. This allows to load the image only when necessary.

4.3.4 Settings objects

Each parameter of the processing graphs needs an according object in the settings window in order for a user to be able to set it. Several classes to create such

objects were developed in order to support different types of parameters. These include a boolean switch, a text input, a class to enable the log scale on several axis and multiple classes to select values from a list.

Three values need to be specified in order to add such an object to the configuration file. These values are the name, which the object should have in the settings window, the class, which is used to create the object, and the parameters needed to instantiate the class.

4.3.5 External settings window

Creating callbacks between Dash components from different windows is not supported directly in Dash. To still be able to have the settings objects in an external window, a custom Dash component called "MessageBox" was created, which can be used to copy a parameter from one window to the other.

The instances of this MessageBox always come in pairs. One MessageBox is in the main window, the other one in the settings window. If the property "send" is updated in one MessageBox, then the value is transmitted to the "receive" property in the other MessageBox. Transmitting the value between the different windows works by using the `postMessage` function of the browser, which is described in [18, 19]. This allows a window to send a message to another window. To support having multiple pairs of MessageBoxes, even though only one communication channel is present, an id identifying the recipient is sent along every message. To avoid inconsistencies upon reloading one window, a special message is send when initially loading a MessageBox. This causes the MessageBox in the other window to repeat the last send value. This guarantees the consistency in one communication direction. The other direction does not need a special measure since all callbacks are initially triggered (as described in [20]). This causes the sent property of the MessageBox in the reloaded window to be updated, which triggers the normal transmission process.

In order to integrate the settings objects into the callbacks of the main window using this MessageBox component, the callbacks need to be adjusted to use the parameters of these MessageBoxes instead of the parameters of the original settings object. Furthermore, a callback needs to be added, which connects the parameters of the settings object with the sent parameters of the MessageBoxes. A class was created to automate this process. This class also automatically adds the needed MessageBoxes to the HTML layout.

Chapter 5

Evaluation

This chapter evaluates the results based on performance and the comparison capabilities. Furthermore, some statistical analysis of the event data is presented to support the assumption made in 4.2.1.

5.1 Length of events

When loading the events, it is assumed, that all events are shorter than one hour. This section will show that this assumption is reasonable.

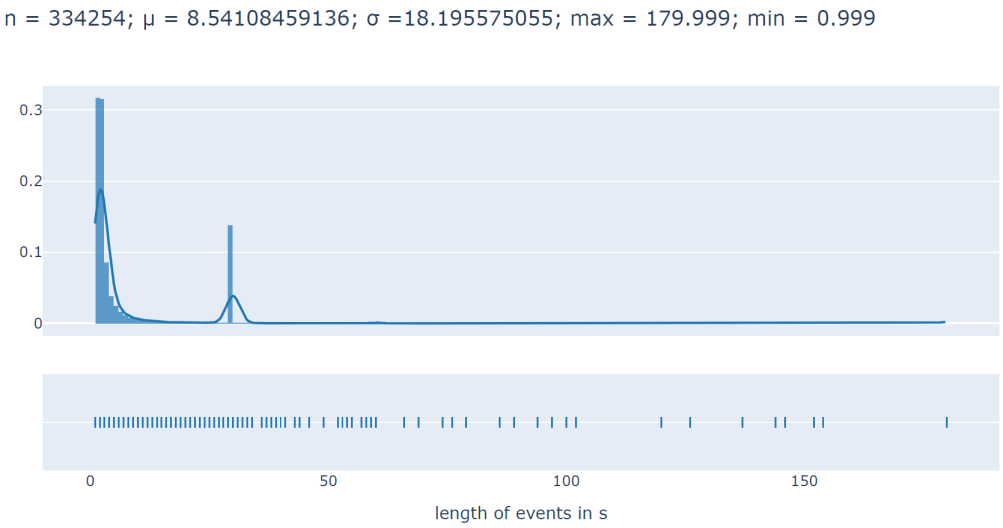


Figure 5.1: Distribution of the event length of all the events until noon, December 9, 2019; All events have a duration much shorter than the assumed maximum of one hour.

Figure 5.1 shows the distribution of the length of all events (until noon, December 9, 2019). In this figure, one can see that the average length is about

8.5 seconds with a standard deviation of about 18.2 seconds. The maximum observed event length is about 3 minutes. This shows, that the assumption of all events being shorter than one hour is valid in the current dataset.

Under the assumption, that the expected value of the event length corresponds to the measured average event length (μ) and that the measured standard deviation (σ) corresponds to the expected standard deviation, an upper bound to the probability, that an event will have a length larger than one hour, can be given using Chebyshev's inequality. This inequality state that for any (integrable) random variable X, with an expected value of μ, a non zero variance σ^2 and $k > 0$, inequality (5.1) holds [21].

$$\Pr(|X - \mu| > k\sigma) < \frac{1}{k^2} \tag{5.1}$$

By defining X as the length of the events (in seconds) and by setting k to $\frac{3600-\mu}{\sigma}$, we obtain:

$$\Pr(|X - \mu| > 3600 - \mu) < \frac{\sigma^2}{(3600 - \mu)^2} \tag{5.2}$$

By using $X \geq 0$ (events cannot have a negative length) and $\mu - x < 3600 - \mu \forall 0 \leq x \leq \mu$ (holds for $\mu < 1600$), this can further be simplified to:

$$\Pr(X > 3600) < \frac{\sigma^2}{(3600 - \mu)^2} \approx 2.6 \cdot 10^{-5} \tag{5.3}$$

This shows that the probability is pretty high that this assumption will also hold in the future, as long as the mean and the standard deviation of the event length will not change significantly.

5.2 Performance of event loading

As explained in 4.2.1, assuming a maximum event length reduces the amount of data which needs to be requested from the server. To illustrate the difference in delay this causes, the time it takes to load all events from an entire month was measured with and without the assumption.

During the test, only the metadata was loaded, since loading the actual waveforms is not affected by the assumption [1]. Furthermore, only the GSN

[1] Before loading the waveforms, it is tested whether the event intersects with the requested time frame. Therefore, the same number of waveforms would be fetched in both cases

server was used, because it would be possible to load the data efficiently from the permasense vault without the assumption using the following idea: On the permasense vault, the metadata are stored in normal text files, one event per line. This makes it possible to access the start time of the previous and the next event. If this start time is earlier than the beginning of the requested time frame, then one can be certain that all prior events start and end before the requested time frame, due to the fact that a geophone sensor can only record one event at a time and that these events are sorted by their start time. By iterating backwards through the events from the actual start time until this condition is met, one can be certain to have found all prior events intersecting with the given time frame without the need to iterate through all events.

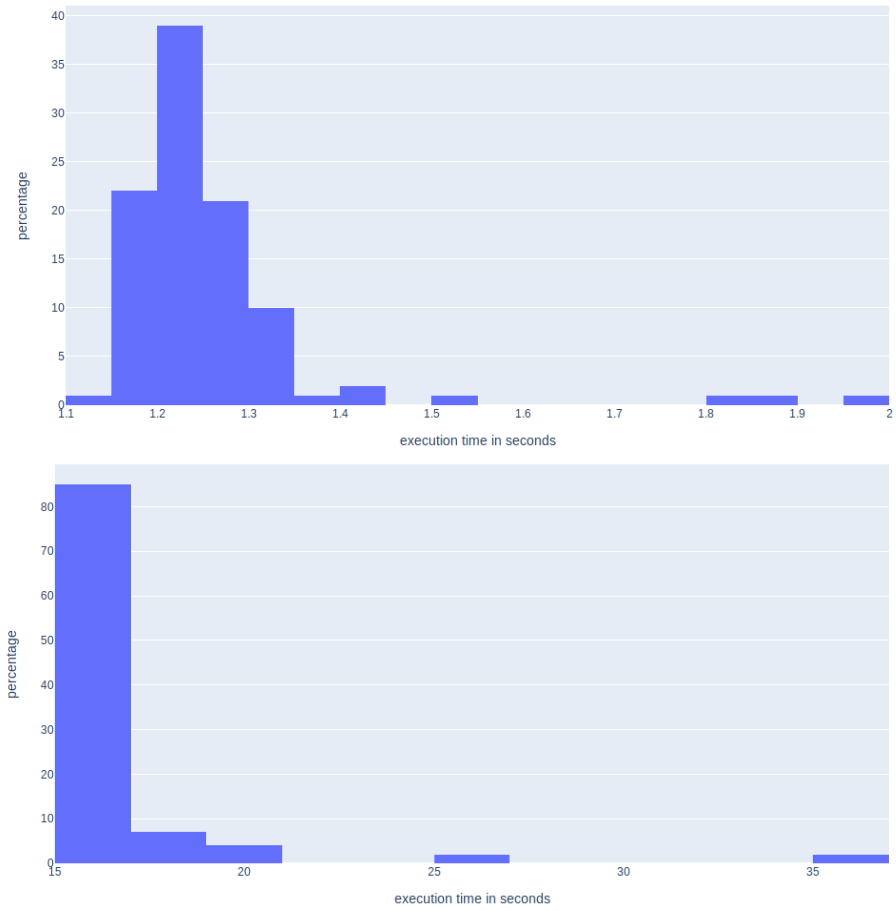


Figure 5.2: Distribution of the loading times, when using the assumption of a maximum event length of one hour (top) and when not (bottom)

Figure 5.2 shows the distribution of execution times observed by loading the data 100 times with and without the assumption of a maximum event length of one hour. The observed average loading time with the assumption was about 1.3

seconds with a standard deviation of about 0.1 seconds. Without the assumption, the observed average loading time was 16.6 seconds with a standard deviation of about 3.3 seconds. This shows, that this assumption indeed leads to a big performance gain.

5.3 Performance of the visualization tool

The performance of the final version depends on the used operating system and the amount of requested data. During this test, a device running Ubuntu 18.10 with 7.7GiB ram and an Intel® Core™ i5-8250U CPU is used. The analysed time frame was chosen to be rather short (one hour). The reason for this decision was that when performing the evaluation, the implementation of the local cache in the Stuett repository was not finished yet and hence caching not yet available. Therefore, loading a bigger time frame would have taken considerably longer than it will, once the cache is integrated.

To test the performance, the loading time of the webpage and the time it took to start the server were measured in four cases. All of these used slightly modified versions of the default configuration file. Two of the cases used the event-based seismic data, the other two the continuous seismic data. For both types of data, the measurement was performed when having every plot only once (denoted with $n = 1$) and when having it twice (denoted with $n = 2$), which would be needed in order to compare the data from two sensors or points in time. The loading time of the webpage can be seen in table 5.1.

	event-based	continuous
$n = 1$	7s	14s
$n = 2$	17s	22s

Table 5.1: The observed loading times of the webpage

All these measurements have been performed only once. Therefore, these numbers should be taken with a grain of salt. However, it is enough to get a rough idea of the loading times, which can currently be achieved. The transfer rate observed when copying data from the permasense vault to the local hard drive without using the visualization tool indicates, that the current bottleneck is the transfer rate of the storage server. Therefore, a significant speed up is expected, once the local cache is integrated.

In all the cases it took about 3 seconds to start the server. With Windows 10, start-up times of about 23 seconds were observed. However, this was only the case for the first start up after the last reboot of the system. All subsequent start-ups took between 4 and 7 seconds. On access scans of antivirus software might be cause of this.

Next, we investigate the amount of data sent and received. During the case with the biggest amount of data (when using continuous seismic data and having every plot twice), 24.9 MB data is downloaded from the storage servers. To request this data, 0.7MB is send. While loading the page (with disabled cache), a total of 9.9MB is downloaded by the browser. The amount of data send by the browser sums up to 4.7MB. Considering that the average mean internet in Switzerland is 38.5Mb/s [22], this amount of data is almost nothing.

5.4 Comparison of seismic events

This section presents, how this visualization tool can be used to analyse and compare data.

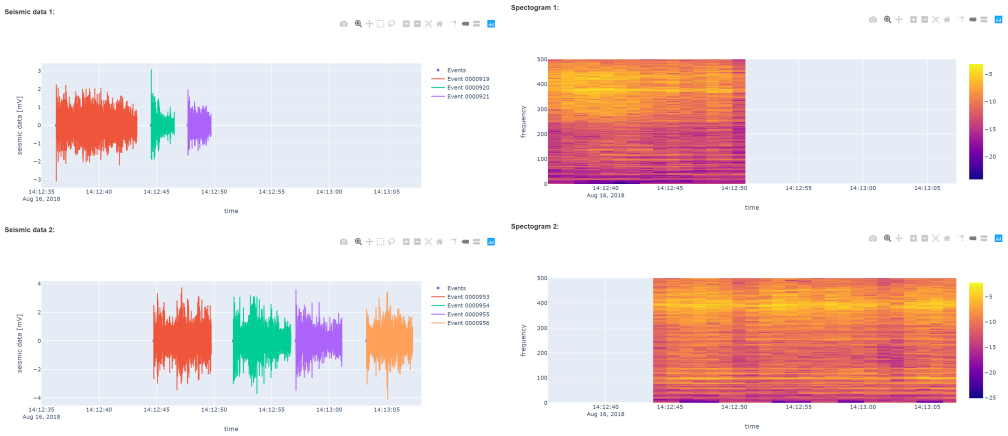


Figure 5.3: Analysing and comparing seismic events from different sensors

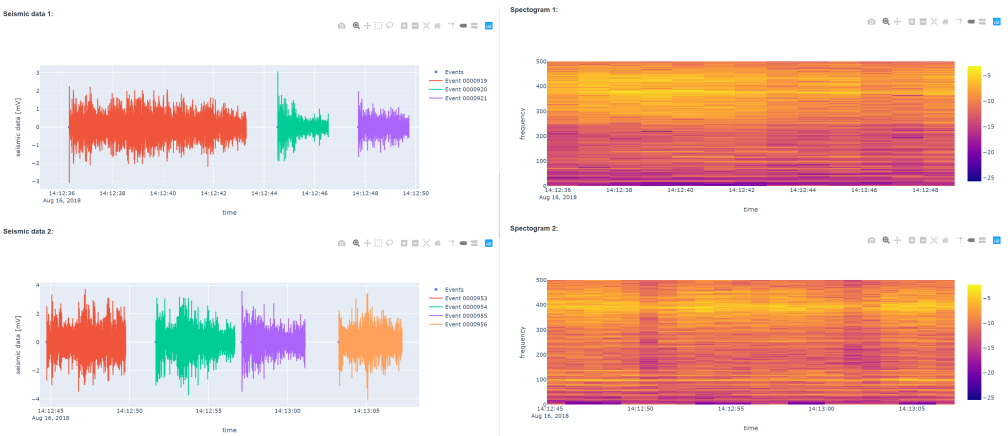


Figure 5.4: Analysing and comparing seismic events from different points in time

Figure 5.5: Working with different time scales

Figure 5.3 shows, how seismic events from different sensor positions can be analysed and compared. Figure 5.4 shows, how events from different points in time can be compared. Figure 5.5 shows, that the visualization tool supports different time scales.

Chapter 6

Future work

This work could be extended in several ways in the future. One possibility would be the integration of a local cache in the Python sever. Having a local cache would remove the need to always download the raw data and downsample it. This would greatly improve the performance.

Another possibility would be to improve the current implementation of the spectrogram. Its current implementation handles downsampled data rather poorly. The used downsampling method only preserves the features of the time domain, but not the ones of the frequency domain. This produces a lot of distortions. By using a local cache, it would be possible to implement a more sophisticated version of the spectrogram with integrated downsampling, as proposed in [4]. This was not yet implemented, as it was out of the scope of this thesis.

Other analysis methods could also be integrated. This could be for example a plot of the Fourier transform, the possibility to apply classical filter like a low pass or signal classifiers.

When classifying a seismic signal using a convolutional neural network as in [12], one needs to have a labelled dataset. Integrating the already existing labelling tool into the visualization tool could simplify the workflow.

Another possible future work is to port this visualization tool to other datasets. Having the possibility to reuse an existing user interface to analyse a dataset reduces the amount of needed duplicated work. Since the data processing is separate from the GUI logic, doing this would only require the creation of a processing graph which loads the data of this dataset and converts it into the data type used by this visualization tool.

Another interesting topic would be to investigate the possibility to execute the processing graph on a cluster of machines. This would allow the usage of computation intensive algorithms running on several servers. This is supported by Dask[7], which is used to create processing graphs with Stuett[6].

Chapter 7

Conclusion

In this thesis, we were able to create a dynamic visualization tool for the data of the PermaSense project, which has support for the event-based seismic data. Using this visualization tool, it is possible to compare data from multiple sensors at differing points in time as shown in 5.4. Using the integrated downsampling methods, it is also possible to operate on different time scales in a user-friendly way. The modular design of this visualization tools makes it possible to easily extend it with new features such as data sources, processing algorithms and plot types as described in 3.2.

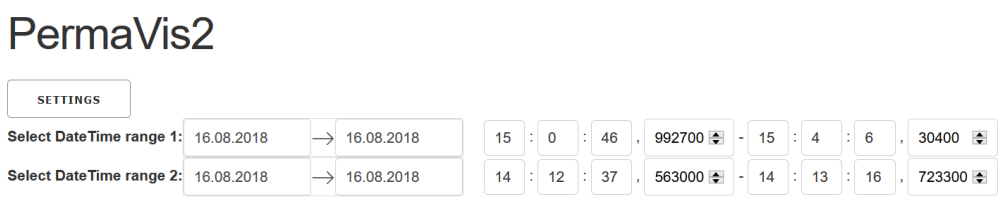
Having such a visualization tool can help getting an understanding of the entire dataset. The knowledge gained with this can help improving the on-site measurements and can be used as an additional tool in research yet to come.

Appendix A

Screenshots

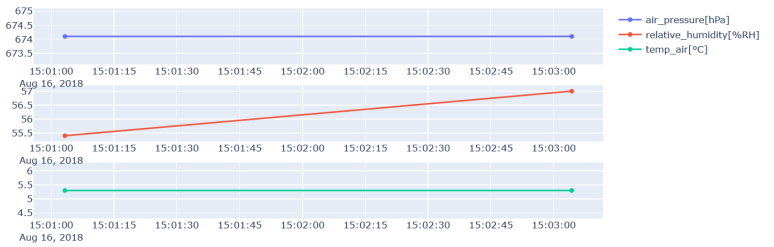
This chapter shows screenshots of the entire user interface. All the screenshots use the same parameters, set in the settings window.

A.1 The main window

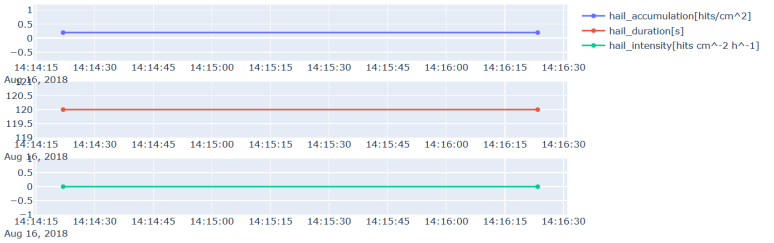


Weather data

Weather data 1:

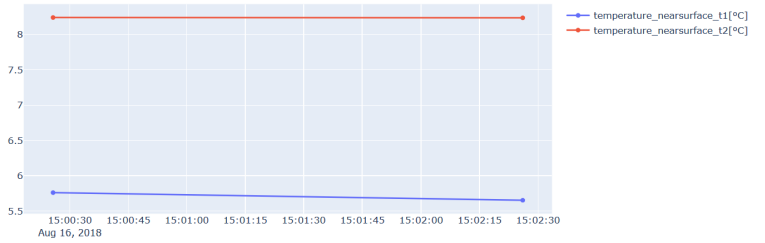


Weather data 2:

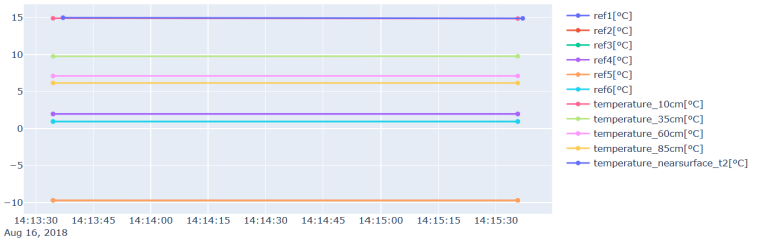


Rock temperature data

Rock temperature 1:

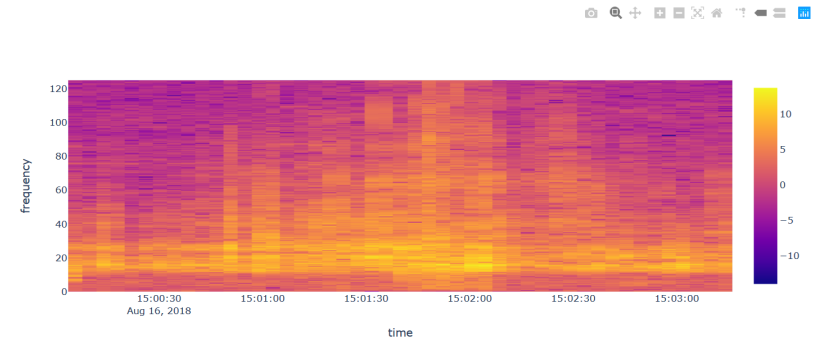


Rock temperature 2:

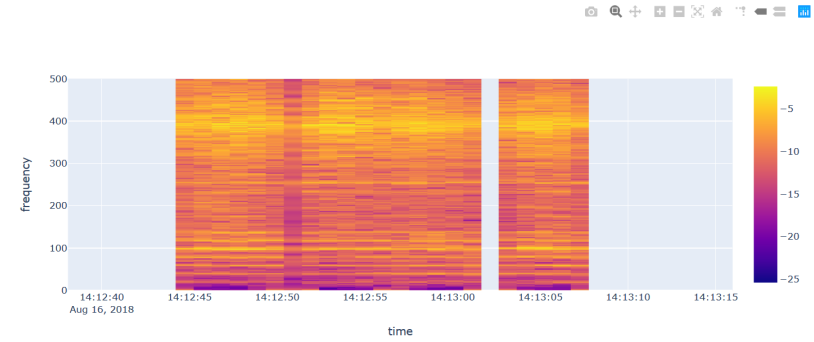



Seismic data
Seismic data 1:
4D.MH36.A.EHE
4D.MH36.A.EHN
4D.MH36.A.EHZ
seismic data
15k
10k
5k
0
−5k
−10k
15:00:30
Aug 16, 2018
15:01:00
15:01:30
15:02:00
15:02:30
15:03:00
time
Seismic data 2:
Events
Event 0000953
Event 0000954
Event 0000955
Event 0000956
seismic data [mV]
4
2
0
−2
−4
14:12:40
Aug 16, 2018
14:12:45
14:12:50
14:12:55
14:13:00
14:13:05
14:13:10
14:13:15
time
Event length statistics 1:
0
Event length statistics 2:
n = 4; μ = 4.499; σ =0.5; max = 4.999; min = 3.999
1
0.8
0.6
0.4
0.2
0
4
4.2
4.4
4.6
4.8
5
length of events in s

Spectogram 1:



Spectogram 2:



Webcam images 1:

2018-08-16T15:00:08.000000000 2018-08-16T15:00:08.000000000 2018-08-16T15:04:10.000000000

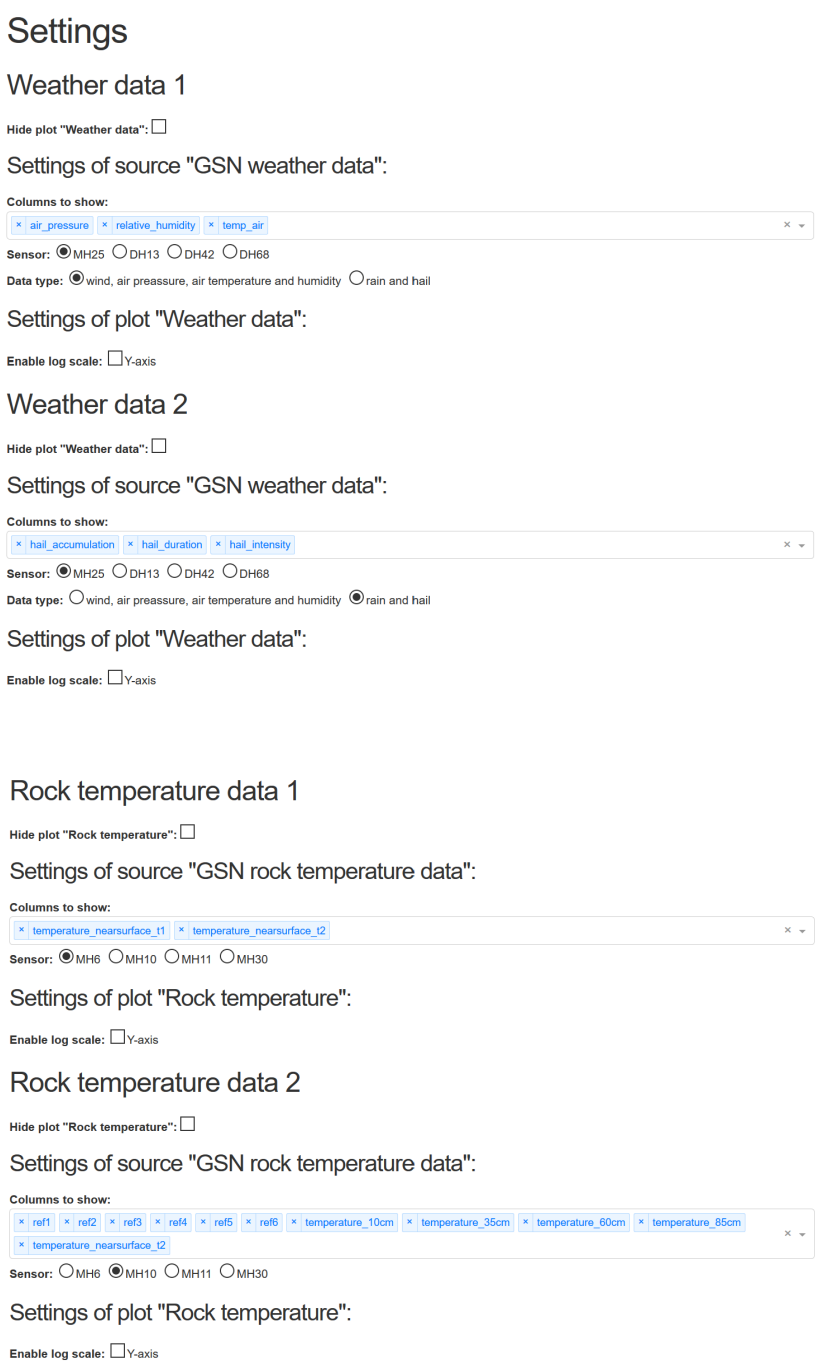
Choose the time of the middle image:

Webcam images 2:

2018-08-16T14:12:07.000000000 2018-08-16T14:12:07.000000000 2018-08-16T14:12:07.000000000

Choose the time of the middle image:

A.2 The settings window



Seismic data 1

Select Source: ○ Event-based seismic data ◉ Continuous seismic data
Hide plot "Seismic data": ☐
Hide plot "Event length statistics": ☐
Hide plot "Spectogram": ☐

Settings of source "Event-based seismic data":

Sensor:
○ MH100 ○ MH101 ○ MH102 ○ MH103 ○ MH104 ○ MH105 ○ MH106 ◉ MH107 ○ MH108 ○ DH100 ○ DH101 ○ DH102
○ DH103 ○ DH104 ○ DH105 ○ DH106 ○ DH107 ○ DH108 ○ DH109 ○ DH110 ○ DH111 ○ DH112 ○ DH113 ○ DH114 ○ DH115
○ DH116 ○ DH117 ○ DH118 ○ DH119
Enable fetching of data: ☑
Filter events:
HELP
Enter a python expression
APPLY

Settings of source "Continuous seismic data":

Sensor: ◉ MH36 ○ MH38 ○ MH44 ○ MH48 ○ MH52 ○ MH54
Channels: ☑ EHE ☑ EHN ☑ EHZ

Settings of plot "Seismic data":

Enable log scale: ☐ Y-axis

Settings of plot "Spectogram":

Enable log scale: ☑ Colour-axis ☐ Y-axis

Seismic data 2

Select Source: ◉ Event-based seismic data ○ Continuous seismic data
Hide plot "Seismic data": ☐
Hide plot "Event length statistics": ☐
Hide plot "Spectogram": ☐

Settings of source "Event-based seismic data":

Sensor:
○ MH100 ◉ MH101 ○ MH102 ○ MH103 ○ MH104 ○ MH105 ○ MH106 ○ MH107 ○ MH108 ○ DH100 ○ DH101 ○ DH102
○ DH103 ○ DH104 ○ DH105 ○ DH106 ○ DH107 ○ DH108 ○ DH109 ○ DH110 ○ DH111 ○ DH112 ○ DH113 ○ DH114 ○ DH115
○ DH116 ○ DH117 ○ DH118 ○ DH119
Enable fetching of data: ☑
Filter events:
HELP
Enter a python expression
APPLY

Settings of source "Continuous seismic data":

Sensor: ○ MH36 ◉ MH38 ○ MH44 ○ MH48 ○ MH52 ○ MH54
Channels: ☑ EHE ☑ EHN ☑ EHZ

Settings of plot "Seismic data":

Enable log scale: ☐ Y-axis

Settings of plot "Spectogram":

Enable log scale: ☑ Colour-axis ☐ Y-axis

Appendix B

Configuration file

The configuration is stored inside the variable `dui_config` in the file `config.py`. This chapter shows the basic structure of this configuration. Always replace <...> with the appropriate content.

Main config:

```
dui_config = {
        "n_parallel":<n>,
        "start_time":[<time1>,<time2>],
        "end_time":[<time3>,<time4>],
        "graphs":{
                "<unique id>":<Graph config>,
                "<unique id>":<Graph config>
        },
        "others":{
                "<unique id>":<Other GUI element config>
        }
}
```

In this configuration, `<n>` needs to be replaced with the number of signals one wants to compare. All `<timeX>` need to be replaced with python datetime objects. All `<unique id>` need to be replaced with unique simple strings. These will be used as ids in the resulting HTML. The entry `graphs` contains all the configurations, used to create the graph objects. Other types of objects can be added to the entry `others`. `graphs` as well `others` can contain an arbitrary number of objects. `start_time` and `end_time` can contain one or more datetime objects.

Graph config:

```
{
        "friendly_name":<name>,
        "sources":{
                "<uniqueId>":<Source config>,
```

```
                "<uniqueId>":<Source config>
        },
        "plots":{
                "<uniqueId>":<Plot config>,
                "<uniqueId>":<Plot config>
        }
}
```

`<name>` is the name displayed in the GUI as a title for these plots. The number of entries in `sources` and `plots` need to be at least one. More entries are possible.

Source config:

```
{
        "dask_graph":<graph function source>,
        "stuett_config":<config function>,
        "settings":{
                "<unique id>":<Settings config>,
                "<unique id>":<Settings config>
        }
}
```

`<graph function source>` is a function, which takes a python dictionary containing the current values of the `settings` parameters of this source as an argument and returns a processing graph which loads the requested measurement data. `<config function>` is a function, getting the same argument and returning the object, which gets propagated backwards through the processing graph. `settings` can contain an arbitrary number of entries.

Plot config:

```
{
        "friendly_name":"<name>",
        "sync_xasis":<sync xasis boolean>,
        "dask_graph":<graph function plot>,
        "settings":{
                "<unique id>":<Settings config>,
                "<unique id>":<Settings config>
        }
}
```

`<sync xasis boolean>` specifies, weather the x axis of this plot should be interpreted as a time axis or not. `<graph function plot>` is a function, taking the processing graph from the selected source and a python dictionary containing the current values of the `settings` of this plot as an argument and returns a

processing graph, which produces the data to plot. `settings` can contain an arbitrary number of entries.

Settings config:

```
{
        "friendly_name":"<name>",
        "class":<reference to class>,
        "init_arguments":{
                "<argument name 1>":<value of argument 1>
        }
}
```

`<reference to class>` is used to create the settings object, which is added to the settings window. `friendly_name` is the title the setting has in the settings window. The entries of `init_arguments` are used as parameters to initialize the object.

Other GUI element config:

```
{
        "class":<reference to class>,
        <rest of configuration depends on class type>
}
```

`reference to class` is used to create the object. The rest of the configuration file depends on the chosen class. One example of such a config file is the image viewer config.

Image viewer config:

```
{
        "class":<reference to image viewer class>,
        "friendly_name":"<name>",
        "dask_graph":<graph function image>,
        "stuett_config":<config function>,
        "slider_text":"<slider text>",
        "transform":"<css transform rules>"
}
```

`<name>` is the title used for the image viewer. `<slider text>` is the text displayed as a title for the slider, which selects the middle image. `<transform>` is the value, which gets set as the CSS transform property of the images. This can be used to rotate the images. `<graph function image>` is a function taking the argument `{"start_time":<time of image>}` and returning a dask graph to fetch the requested image. `<config function>` is a function, getting the same argument and returning the object, which gets propagated backwards through the processing graph.

Appendix C

Syntax of the filter rule

The filter rule is an ordinary python expression, supporting the following functions:

- `datetime(year,month,day,hour,minute,second,microsecond)`: Create a corresponding datetime object
- `date(datetimeObject)`: Create a date object from a datetime object
- `date(year,month,day)`: Create a corresponding date object
- `time(datetimeObject)`: Create a time object from a datetime object
- `time(hour,minute,second,microsecond)`: Create a corresponding time object
- `timedelta(**KeywordArguments)`: Create a timedelta object
  Supported keyword arguments: weeks, days, hours, minutes, seconds, milliseconds, microseconds
- `abs(value)`: returns |value|

Following attributes are supported:

- datetime objects: year, month, day, hour, minute, second, microsecond
- date objects: year, month, day
- time objects: hour, minute, second, microsecond

Appendix D

Derivation of the conversion formula for the event data

The following section derives the conversion formula used in 4.2.1.

The ADC output zero corresponds to an amplified input voltage of about -1.25V. The value $2^{24} - 1$ corresponds to about 1.25V. The voltage levels of the integer values in between are distributed equidistant between those two values. Therefore, the mapping is linear.[1] With this, a conversion formula can be derived:

Maximum output voltage (in mV) of the geophone sensor the ADC correctly represent: u_{max}
Minimum output voltage (in mV) of the geophone sensor the ADC correctly represent: u_{min}
Gain of the ADC: g
Number of bits of the output of the ADC: n
Maximum output of the ADC: $x_{max} = 2^n - 1$
Minimum output of the ADC: $x_{min} = 0$
Function, that maps the integer values returned by the ADC to the estimated geophone voltages (in mV): f

$$u = f(x) = a \cdot x + b \tag{D.1}$$

$$u_{min} = f(x_{min}) = f(0) \rightarrow b = u_{min} \tag{D.2}$$

$$u_{max} = f(x_{max}) = f(2^n - 1) \rightarrow a = \frac{u_{max} - u_{min}}{2^n - 1} \tag{D.3}$$

$$\tag{D.4}$$

[1] These informations were kindly provided by Jan Beutel.

$$\rightarrow f(x) = a \cdot x + b = \frac{u_{max} - u_{min}}{2^n - 1} \cdot x + u_{min} \tag{D.5}$$

$$= \frac{u_{max} - u_{min}}{2^n - 1} \left(x + \frac{u_{min}}{u_{max} - u_{min}} (2^n - 1) \right) \tag{D.6}$$

With $u_{max} = 1250/g; u_{min} = -1250/g$:

$$f(x) = \frac{2500}{g \cdot (2^n - 1)} \left(x - \frac{2^n - 1}{2} \right) \tag{D.7}$$

Bibliography

[1] Michael C. R. Davies, Omar Hamza, and Charles Harris. The effect of rise in mean annual temperature on the stability of rock slopes containing ice-filled discontinuities. *Permafrost and Periglacial Processes*, 12(1):137–144, 2001. doi: 10.1002/ppp.378.

[2] S. Weber, J. Beutel, R. Da Forno, A. Geiger, S. Gruber, T. Gsell, A. Hasler, M. Keller, R. Lim, P. Limpach, M. Meyer, I. Talzi, L. Thiele, C. Tschudin, A. Vieli, D. Vonder Mühll, and M. Yücel. A decade of detailed observations (2008–2018) in steep bedrock permafrost at the matterhorn hörnligrat (zermatt, ch). *Earth System Science Data*, 11(3):1203–1237, 2019. doi: 10.5194/essd-11-1203-2019.

[3] Akos Pasztor. Event-based geophone platform with co-detection, 2018. Computer Engineering Group ETH Zürich, Master Thesis.

[4] Feiyu Jia. Dynamic visualization of geophysical data, 2019. Computer Engineering Group ETH Zürich, Semester Thesis.

[5] R FAQ. `https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-is-R_003f`, Accessed on 21.12.19.

[6] Stuett repository. `https://gitlab.ethz.ch/tec/public/employees/matthias-meyer/stuett`, Accessed on 06.12.2019.

[7] Dask — dask 2.8.1 documentation. `https://docs.dask.org/en/latest/`, Accessed on 06.12.2019.

[8] xarray: N-d labeled arrays and datasets in python. `http://xarray.pydata.org/en/stable/`, Accessed on 06.12.2019.

[9] Python 3.7.5 documentation. `https://docs.python.org/3.7/`, Accessed on 06.12.2019.

[10] General Python FAQ. `https://docs.python.org/3/faq/general.html#what-is-python`, Acessed on 13.12.2019.

[11] Applications for Python. `https://www.python.org/about/apps/`, Accessed on 06.12.2019.

[12] M. Meyer, S. Weber, J. Beutel, and L. Thiele. Systematic identification of external influences in multi-year microseismic recordings using convolutional neural networks. *Earth Surface Dynamics*, 7(1):171–190, 2019. doi: 10.5194/esurf-7-171-2019.

[13] Dash user guide. `https://dash.plot.ly/`, Accessed on 06.12.2019.

[14] Wikipedia contributors. Window function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Window_function&oldid=931321486`, 2019. Accessed on 22.12.2019.

[15] EPFL Distributed Information Systems Laboratory. Gsn wiki. `https://github.com/LSIR/gsn/wiki/Web-Interface`, Accessed on 22.12.2019.

[16] Scipy — scipy v1.4.1 reference guide. `https://docs.scipy.org/doc/scipy/reference/`, Accessed on 22.12.2019.

[17] Larry M Masinter. The "data" URL scheme. RFC 2397, August 1998.

[18] Mdn web docs. `https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`, Accessed on 20.12.2019.

[19] Html standard. `https://html.spec.whatwg.org/multipage/web-messaging.html#dom-window-postmessage`, Accessed on 20.12.2019.

[20] Basic dash callbacks. `BasicDashCallbacks`, Accessed on 20.12.2019.

[21] Wikipedia contributors. Chebyshev's inequality — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Chebyshev%27s_inequality&oldid=930349002`, 2019. Accessed on 22.12.2019.

[22] Worldwide broadband speed league 2019. `https://www.cable.co.uk/broadband/speed/worldwide-speed-league/`, Accessed on 18.12.2019.