MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Game Engineering and Simulation Technology

Comparison of time intervals for high-dimensional industrial process data in Visplore

By: Laurențiu Leahu-Vlăducu, BSc

Student Number: 1810585001

Supervisors: Dipl.-Ing. Dr. Gerd Hesina
Dipl.-Ing. Dr. Harald Piringer

Vienna, September 6, 2020

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, September 6, 2020 Signature

Kurzfassung

In der Produktionsindustrie sind Informationen über den Produktionsablauf für ein Unternehmen sehr wertvoll. Solche Informationen können von Visplore, einer Software für Visual Analytics, geladen und visuell analysiert werden. Im Rahmen dieser Masterarbeit wurde Visplore erweitert, um intervallbasierte Vergleiche zu unterstützen. Dies ist eine juxtapositionierte (Seite an Seite) Vergleichstechnik, die sowohl design- als auch implementierungstechnisch ausführlich erklärt wird. Diese Technik eignet sich am besten für Vergleiche, bei denen die verglichenen Daten nicht ähnlich sind, im Vergleich zu einem superpositionierten (überlagerten) Design, das nur für Daten mit geringen Unterschieden passend ist. Die dadurch entstandenen Benutzbarkeitsprobleme und die Lösungen dafür werden vorgestellt. In Visplore wurde diese neue Vergleichstechnik in den 2D Scatter Plot und in den Horizon Graph View integriert. Kurz gesagt, unsere juxtapositionierte Vergleichstechnik komprimiert die für die BenutzerIn unwichtigen Teile der Grafik und erweitert die interessanten Bereiche, die dann für einen Seite-an-Seite-Vergleich verwendet werden können. Mit der neuen Technik ist es jetzt beispielsweise möglich geworden, die industrielle Produktion von zwei verschiedenen Tagen zu vergleichen, indem sie nebeneinander angezeigt werden, während der Rest der Daten ignoriert wird. Der neue intervallbasierte Ansatz kann jedoch auch hilfreich sein, wenn beispielsweise ein Datensatz große Lücken in den Daten oder viele fehlende Werte aufweist. In diesem Fall werden Intervalle verwendet, um die Teile ohne Daten auszuschneiden, damit möglichst wenig visueller Platz verschwendet wird und damit der Rest der Daten detaillierter angezeigt werden kann. Darüber hinaus wurden Techniken analysiert, die in anderen wissenschaftlichen Arbeiten vorgestellt wurden, und mit unserem Ansatz verglichen.

Schlagworte: Vergleich, Juxtaposition, Zeitintervalle, Industrieprozessdaten, Intervall, Visual Analytics, Visplore

Abstract

In the production industry, information about how the production process performs is very valuable to a company. Such information can be loaded and visually analyzed by Visplore, which is a software for visual analytics. As part of this master's thesis, Visplore has been extended to support interval-based comparisons. The usability problems created by the new technique and the solutions to these problems will be presented. This is a juxtaposed (side-by-side) comparison technique, and will be explained in detail, both design-wise and implementation-wise. This technique works best on comparisons where the compared data is not similar, in comparison to a superposed (overlaid) design, which properly works on data containing only small differences. Within Visplore, this new comparison technique has been integrated in the 2D Scatter Plot and in the Horizon Graph View. In short, our juxtaposed comparison technique works by compressing the parts of the graph which are not important to the user, and expanding the areas of interest, which can then be used for a side-by-side comparison. By using the new technique it is now possible, for example, to compare the industrial production of two different days by displaying them besides each other and ignoring the rest of the data. The new interval-based approach can however also help, for example, when a dataset contains big gaps within the data, or many missing values. In this case, intervals are used to cut out the parts without data, to avoid wasting visual space and to show the rest of the data more detailed. Furthermore, techniques presented in other papers have been analyzed and compared to our approach.

Keywords: Comparison, Juxtaposition, Time intervals, Industrial process data, Interval, Visual Analytics, Visplore

Acknowledgements

I would like to thank everyone who accompanied and motivated me while writing this master's thesis.

First of all, I would like to thank all of my work colleagues at VRVis for their support during the development and while writing this master's thesis. Special thanks go to Florian, David and Nadine who helped me with the design and implementation of the time interval comparison feature, answered all my questions and helped me with great advice.

I would also like to thank Mr. Dipl.-Ing. Dr. Hesina for supervising my master's thesis. Thank you for the helpful suggestions and the constructive criticism while writing this thesis, and for the long time needed to read it, in the first place. Additionally, I would like to thank Mr. Dipl.-Ing. Dr. Piringer for being my second supervisor, for coordinating the development outstandingly, and for providing me with great literature needed during the development and while writing of this thesis. Also, thank both of you for the opportunity to write my master's thesis at VRVis.

Special thanks go to my girlfriend, Clarissa, who always supported me and gave me strong emotional support. I would like to thank you for the helpful tips, for your help with proofreading this thesis, and for being very kind and supportive during my bachelor's and master's degrees, and while writing this master's thesis.

I would also like to thank my fellow students for the wonderful and entertaining years together. I will not forget your help and support during the master's degree, and the fun times during the development of our master project and while gaming together.

Finally, I would like to thank my parents Vlăduț and Mihaela, who made my studies possible with their support and always had an open year for my worries, as well as my future in-laws Johanna and Helmut, who always motivated me during this stressful time, and stood by my side with a lot of patience.

Contents

1 Introduction

Nowadays, datasets get increasingly complex and large, which also means that it is increasingly difficult for analysts, scientists and engineers to analyze data (Gleicher et al., 2011). Due to the complexity of the data, comparing specific parts of the data - in other words, comparing data subsets - helps with reducing the complexity and analyzing only the sections of interest.

The comparison of time intervals is an important feature for visual analytics applications. These help with analytical reasoning supported by interactive visual interfaces (Keim et al., 2009). Small amounts of data can be visualized and edited with popular software. However, this may be difficult to do with big amounts of data, on the one hand for the software, which may get very slow or not work at all anymore, and on the other hand for the person who wants to analyze the data, who will see lots of unstructured data.

One of the visual analytics applications is Visplore, which is developed by the Visual Analytics department of VRVis, in Vienna. Some of its features are shown in Figure 1. Visplore is a software for analyzing large, structured data volumes. It helps users to interact and analyze data in real-time, even without having detailed knowledge of statistics (VRVis, n.d.*a*).

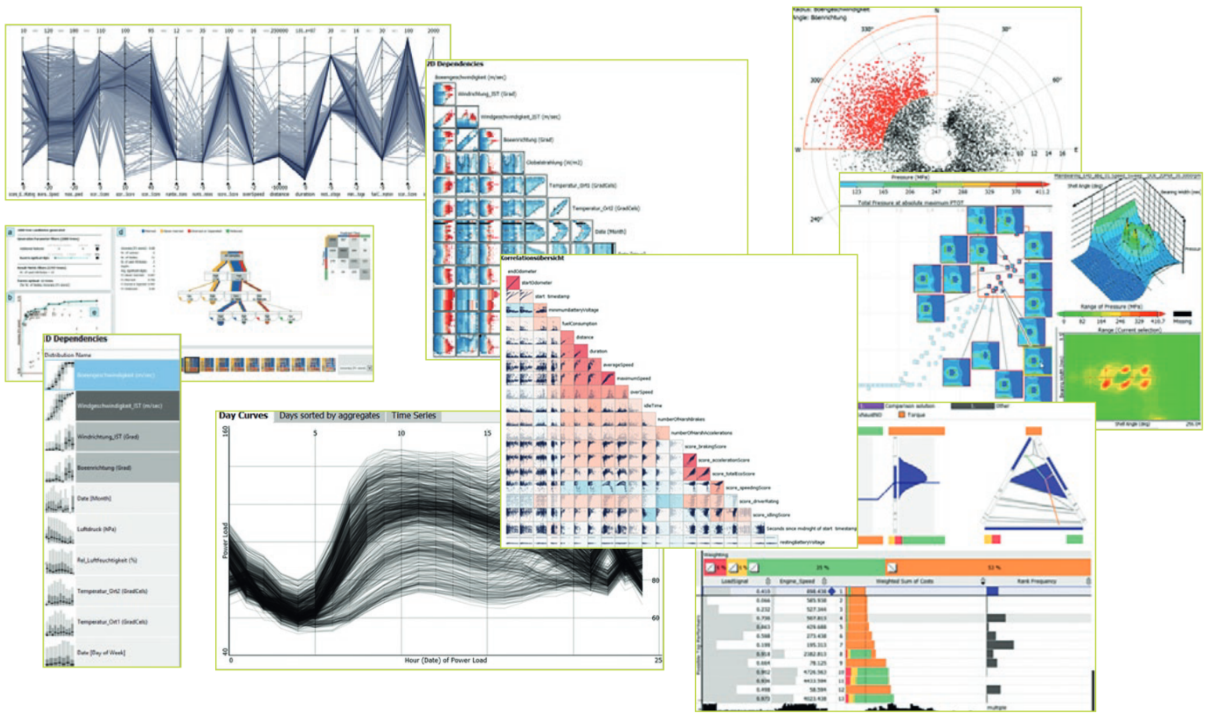


Figure 1: Various possibilities of visual data analysis in Visplore. (VRVis, n.d.*a*)

This master's thesis compares existing comparison strategies presented by other scientific papers and presents the way the comparison of time intervals has been designed and implemented in Visplore. Within Visplore, the new comparison of time intervals has been implemented in the 2D Scatter Plot and in the Horizon Graph View. The new implementation which has been developed as part of this master's thesis does a juxtaposed, or side-by-side, comparison. This was needed for two reasons: on the one hand, the user should have the possibility to analyze in detail and compare multiple subsets of the data with each other. This is especially important for industrial process data, because in the production industry, information about how the production process performs is a very valuable information to a company. Such information is usually measured by sensors and machines that do the work. With Visplore, it is possible to load this data and evaluate it with the help of graphs and different visualization techniques. Previously, filtering out data to only see the areas of interest caused the gaps to appear between these areas, which was a waste of space, but a comparison of data was also difficult because of the distance between the areas which needed to be compared. By developing the new comparison technique for Visplore, it is now possible, for example, to compare the industrial production of two different Mondays, without visually seeing gaps between the two days. On the other hand, many datasets have gaps in their data - in this case, without the user filtering out any data - so previously much space was wasted, instead of only displaying the parts of the graph which actually have data. This problem has also been solved by the new technique.

Simply explained, after gaps in the data are detected and are bigger than the maximum gap size - that is, either when the dataset itself contains gaps, or after the user defines which parts of the graph should be filtered out - what actually happens is that the areas containing no data get compressed and the other areas get expanded, putting emphasis on the areas of interest, which contain data. These areas of interest are called intervals. So, simply explained, the comparison of time intervals is just an expansion of the areas of interest and a compression of all the other parts, resulting in areas of interest which are displayed side by side and allow for an easier comparison, while also saving space.

Figure 2 shows two pictures of the 2D Scatter Plot. At the top, three selections have been made on the X axis, which have been used to create three intervals, as shown at the bottom. The areas between the selections, as well as the areas at the very beginning and at the very end of the graph, have been filtered out. For usability reasons, some space between the intervals is shown, as well as a separation line to easily identify the beginning and the end of an interval. Furthermore, also for usability reasons, some space is added before the beginning of the first interval and after the end of the last interval.

Figure 3 shows again two pictures of the Horizon Graph View, which use the same three selections as in Figure 2. Although intervals are created differently than in the 2D Scatter Plot, the principle is the same. At the top, no intervals have been created yet, but we can observe in the picture at the bottom that the gray areas have been reduced to the separation lines, to save space and show the areas of interest more detailed than before.

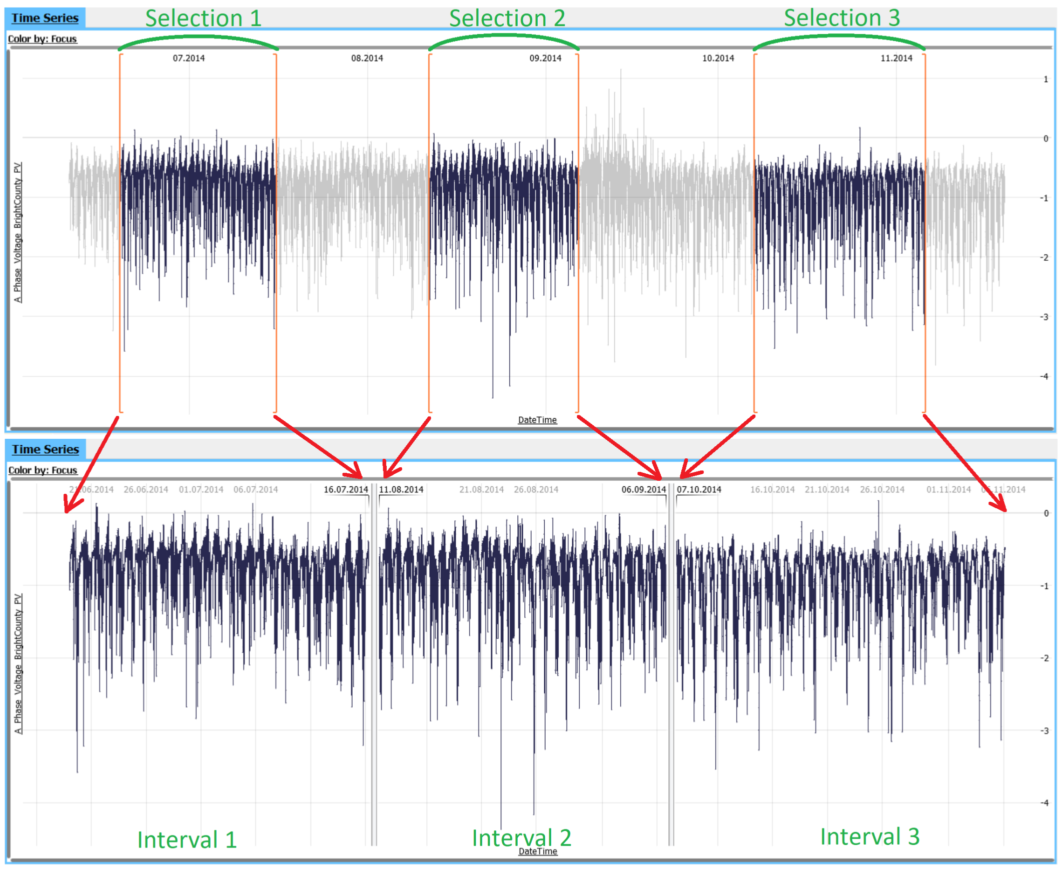
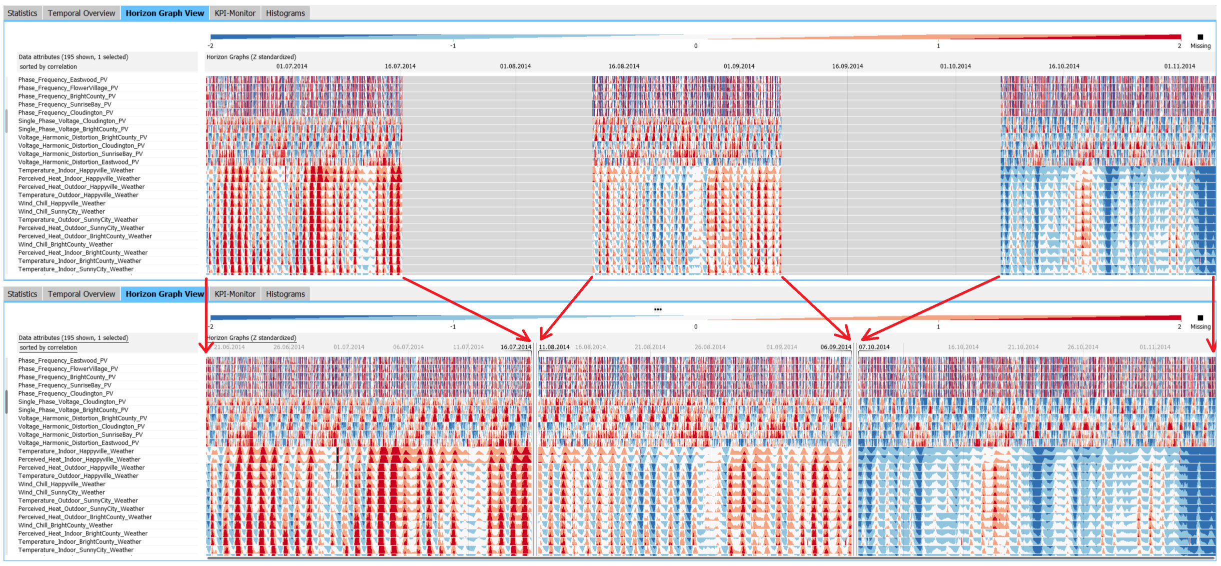


Figure 2: The time intervals comparison in the 2D Scatter Plot in Visplore, simply explained. At the top, 3 selections have been made, out of which intervals have been created, as seen at the bottom.



Statistics
Temporal Overview
Horizon Graph View
KPI-Monitor
Histograms
Data attributes (195 shown, 1 selected)
sorted by correlation
Horizon Graphs (Z standardized)
Missing
Phase_Frequency_Eastwood_PV
Phase_Frequency_FlowerVillage_PV
Phase_Frequency_BrightCounty_PV
Phase_Frequency_SunriseBay_PV
Phase_Frequency_Cloudington_PV
Single_Phase_Voltage_Cloudington_PV
Single_Phase_Voltage_BrightCounty_PV
Voltage_Harmonic_Distortion_BrightCounty_PV
Voltage_Harmonic_Distortion_Cloudington_PV
Voltage_Harmonic_Distortion_SunriseBay_PV
Voltage_Harmonic_Distortion_Eastwood_PV
Temperature_Indoor_Happyville_Weather
Perceived_Heat_Indoor_Happyville_Weather
Perceived_Heat_Outdoor_Happyville_Weather
Temperature_Outdoor_Happyville_Weather
Wind_Chill_Happyville_Weather
Wind_Chill_SunnyCity_Weather
Temperature_Outdoor_SunnyCity_Weather
Perceived_Heat_Outdoor_SunnyCity_Weather
Perceived_Heat_Outdoor_BrightCounty_Weather
Wind_Chill_BrightCounty_Weather
Perceived_Heat_Indoor_BrightCounty_Weather
Temperature_Indoor_BrightCounty_Weather
Temperature_Indoor_SunnyCity_Weather

Figure 3: The time intervals comparison in the Horizon Graph View in Visplore, simply explained. At the top, the same three selections from Figure 2 have been used, out of which intervals have been created, as seen at the bottom.

1.1 The dataset

Our dataset is available as a CSV file. CSV, or Comma-Separated Values, is a simple format where data is separated by commas or semicolons. The first line contains the names of each data column. Each following line contains data corresponding to one data record. When this file is loaded by Visplore, a data table is being created, containing rows (also called entries) and columns (also called data attributes). Normally, each data record should contain as much data as the number of columns, but this may not always be the case. If data does not exist, it is marked as missing. After importing the data, the result looks at first sight similar to popular spreadsheet software, but visual analytics software products allow for much more flexibility and interactivity with the data.

For most of the figures used in this master's thesis, the Photovoltaic (PV) Solar Panel Energy Generation dataset (UK Power Networks, 2011) has been used. Its license, as well as the modifications made to it, are mentioned in Appendix A. This dataset contains measurements made over half a year at a distance of 10 minutes of each other. This dataset has a total of 23422 entries and contains 39 data attributes, like the indoor and outdoor temperature, the indoor and outdoor dew point, solar radiation, wind speed and phase frequency, as well as many further attributes. One possible use case for this dataset is to find correlations between different environmental factors.

One important difference between data attributes is whether the attribute contains continuous or discrete data. The data within a data attribute is considered discrete when it can only take certain values, and nothing else in between. Examples for that are integer numbers or a number of categories - for example, it is not possible to have 1.5 categories. Otherwise the data is considered being continuous. Continuous data have an infinite number of values between two values, e.g. there is an infinite number of possible values between 1.0 and 2.0. Both the values and the timestamp of the entry can be either continuous or discrete. The timestamp column is discrete when the data records have only been taken at certain times - otherwise the column is continuous. This information is saved as metadata for each column.

However, it may sometimes make sense to interpret continuous data as discrete, or the other way around. For example, time is continuous because even a second or smaller time units can be infinitely divided, at least theoretically. For practical reasons, depending on the use case, it may be easier and/or more practical to consider the time as being discrete, e.g. to only consider every second, minute or even hour, and nothing in between.

Having said that, the difference between continuous and discrete data is only theoretical. For example, for practical reasons, datasets can only have a finite number of entries, so according to our definition above, a data attribute within a dataset can never have continuous data. In practice, when an attribute contains a large enough number of different values, it can be considered as being continuous.

1.2 The structure of Visplore

To understand some of the challenges and implementation details regarding the comparison feature in Visplore, it is important to understand its internal structure.

Visplore is a software which has been written to a great extent in C++, with some parts which are not performance-critical written in Python. C++ allows a very efficient programming with zero-cost abstractions, which is very important to visual analytics software like Visplore, that often have to deal with millions of data entries. At the same time, C++ has a high level of abstraction, so it is relatively easy to program with, and at the same time allows software to be compiled with ease for multiple platforms and architectures. One advantage of C++ is its compatibility with the C programming language, which allows using libraries written in C also with C++.

Newer programming languages have an automatic memory management, which is usually implemented by using a garbage collector that deletes the allocated memory when no references point to that memory anymore. C++ does not have a garbage collector, so memory management needs to be done either manually, or by using more modern language concepts such as smart pointers. Visplore is a software which evolved over many years, so some parts of it only use older C++ features, while newer code also uses smart pointers.

For the user interface, GTK 2 is used. GTK (GIMP Toolkit) is a free, open source and platform independent tool for creating graphical user interfaces (GUI). GTK has been written in C, also works with C++, but also provides bindings for other programming languages.

For efficiency reasons, the drawing of the graphs is done using the 3D graphics library OpenGL (Open Graphics Library). The advantage of using it is that the rendering is hardware accelerated by the graphics card to be able to show millions of data on the screen, while maintaining the interactivity, so that the program can still remain responsive to user interaction. OpenGL is a platform-independent API (Application Programming Interface) for developing computer graphics software, which is used for the communication between Visplore and the graphics card. OpenGL has been written in the C programming language, also works with C++, but also has bindings for other programming languages.

Visplore is a modular program: there is a main program (Core), which contains code relevant to all parts of Visplore, as well as plugins, which only contains code relevant to certain parts of Visplore. For example, both the 2D Scatter Plot and the Horizon Graph View are plugins, but other parts like the data mapping or the new multi-interval data mapping are part of the core, because they are used in multiple plugins. Some code used in the Horizon Graph View is also part of the core, to allow it to be used somewhere else in the future.

2 Comparison of data

2.1 Time series

First, in order to understand why the comparison of time series plays an important role in visual data analytics, it is important to understand what exactly time series data is. A time series is a collection of data points taken over time, which means that every data point has a time stamp. Such data describes a change over time. Imagine a graph where the X axis represents the time, and the Y axis contains values takes from a sensor. Showing both together in a graph would display the change in values over time. The same can be done by using weather data, electricity or gas consumption, or any other kind of data taken over time.

In practice, time series can be used to make predictions about the development of future trends, for example when analyzing stock prices, or to visualize and control or correct the operation of production machines used for manufacturing.

2.2 Comparison strategies

The following section is based on work done by (Gleicher et al., 2011). Many software programs already support data comparison, but are made for comparison of specific types of data, like module relationships in complex software, genetic sequences, or other complex types of data. Such software applications allow making comparisons for specific use cases and are optimized for that purpose, but each use case needs a custom solution.

However, common comparison issues exist, which are independent of the data that has to be compared. These help with developing data comparison methods for general visual analytics applications, which are not built for comparing special kinds of data. The same difficulties appear when comparing different objects with each other. According to (Gleicher et al., 2011), three general visual categories of comparison can be defined:

1. juxtaposition - showing different objects separately, e.g. besides each other
2. superposition - overlaying objects
3. explicit encoding

All three methods have advantages and disadvantages and should be used in the appropriate situation.

Furthermore, an important factor when comparing different objects, or different data subsets, is their complexity. This complexity may either come from the size of the objects and the abstractness of these objects.

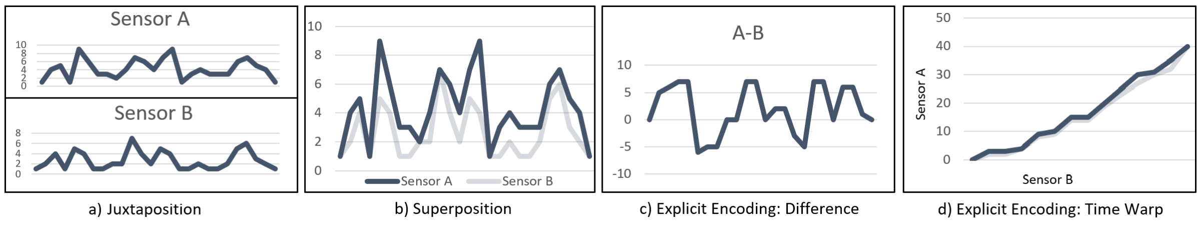


Figure 4: Comparing two time series by using measurements from two sensors (X and Y) taken over time. Multiple approaches are being shown: (a) juxtaposition, (b) superposition, (c)(d) explicit encoding. Information partially taken from (Gleicher et al., 2011).

Another important factor is the scalability. The comparison may get more difficult to understand, which depends on both the complexity of the objects and the number of compared objects.

2.3 Advantages and disadvantages of comparison strategies

The following section presents the advantages and disadvantages of the different visual comparison strategies as observed by (Gleicher et al., 2011).

Comparing objects by using a juxtaposition design means showing the objects separately in either time or space. This strategy relies on the user's memory to see the difference between the compared objects. From a usability perspective, it makes the most sense to show the objects as close as possible to each other, so that the user can see both at the same time. Often, juxtaposition is being implemented in visualization software by showing two or more views, each containing one object, next to each other. This is also called a small multiples design, and depending on the way it is being represented, can be called dual-views or side-by-side views. An example of a comparison by juxtaposition can be seen in Figure 4(a). Scalability using a superposition design is easy to achieve, simply by placing different objects next to each other. The number of objects to compare is limited by the available space and by the size and detail of the objects, but this design allows comparing potentially many objects at the same time.

Using a superposition design means overlaying multiple objects by showing them at the same place and time. To be able to distinguish between the different objects, one possible strategy is to color them differently. After that, one possible way to render the objects in superposition is by partially obscuring them, which can also be seen in Figure 4(b). An alternative approach is to make one of the objects be semi-transparent. Comparing objects by using a superposition design makes sense when either spatialization is important for the specific dataset, or when the objects are similar enough to view them together in a way that makes sense. This is difficult with data that is very dense, such as images. Because of this, it may be difficult to achieve proper scalability, and clutter may also become a problem, depending on the complexity of the objects. For example, even overlaying two objects may be difficult to understand if the objects are complex and/or dense, so overlaying even more objects may make the visualization impossible to understand. On the other hand, comparing simple and/or very similar objects can easily be achieved, and the number of simultaneously compared objects may be much higher than when doing a comparison by juxtaposition.

Explicit encodings are based on the relationships between objects and can be computed and displayed in different ways, for example by subtracting two objects as seen in Figure 4(c), or by showing a time warp as seen in Figure 4(d). Computing explicit encodings makes the most sense when the relationships between the different objects are known - this way, the most useful results can be achieved. However, if the relationships are not known, explicit encodings can also be computed, but the user needs to find the relationships instead. Another possibility if no relationships are known is to compute them - this way, some heuristics can be used to determine which relationships make the most sense, although algorithmically finding these relationships may not always work properly. More exactly, by computing relationships between two objects, a new object that consists of this relationship is being created, so one may see explicit encodings as a replacement of the original object with a new, computed object.

The main difference between these comparison strategies is the way the user has to make connections between the objects and their parts. Juxtaposition makes use of the user's memory to make connections between the objects, but this type of comparison does not help the user make connections between the parts of the objects - which may make it difficult for the viewer to see relationships between them. Superposition makes use of visuals to differentiate between the objects, and proximity helps to make connections between its parts. Explicit encodings compute new graphs to help the user determine relationships between the objects, and use other visual encodings to connect its parts.

The choice for the right comparison strategy depends on the use case and what needs to be achieved. For example, using an explicit encoding design makes sense when the goal is to visualize the relationship between two objects. However, this comparison design makes it difficult for the user to connect these relationships back to the original objects. Also, if information other than the relationships may be interesting, using an explicit encoding will hide these differences from the user.

Furthermore, it is even possible to combine two or even three of these strategies and create hybrid comparison strategies. This way, the advantages of different strategies can be used to minimize the shortcomings of the other ones. For example, explicit encodings can be used to make connections between juxtaposed views, and the juxtaposed views can give context for the encoded relationships. When designing hybrid comparisons, it is very important to avoid issues with clutter as much as possible, since combining strategies also increases the complexity of the visualization. Since combinations of different strategies are not relevant for this master's thesis, these will not be further explained.

2.4 Data comparison using stack zooming

Another technique presented in (Javed & Elmqvist, 2010) is stack zooming. The following section will be based on information found in that paper.

There are multiple ways of comparing objects by using a juxtaposition design, but usually, this means showing objects besides each other, either horizontally or vertically. In the cited paper, the stack zooming technique is used to analyze a large-scale temporal dataset, so we will use the same kind of datasets for our discussions. That means that one axis contains the time, and the other axis consists of values that, most probably, changed over time. This is also often called a time-series visualization.

The stack zooming technique uses the concept of a juxtaposition design, but takes it a step further by creating a stack of zoomed in subsets of the selected data (Figure 5). Precisely, at the very beginning, the whole time series visualization is being shown as one large single strip (a focus region). Usually, since big datasets are being used, where not all points may fit on the screen, zooming is needed to see individual points and solve certain tasks.

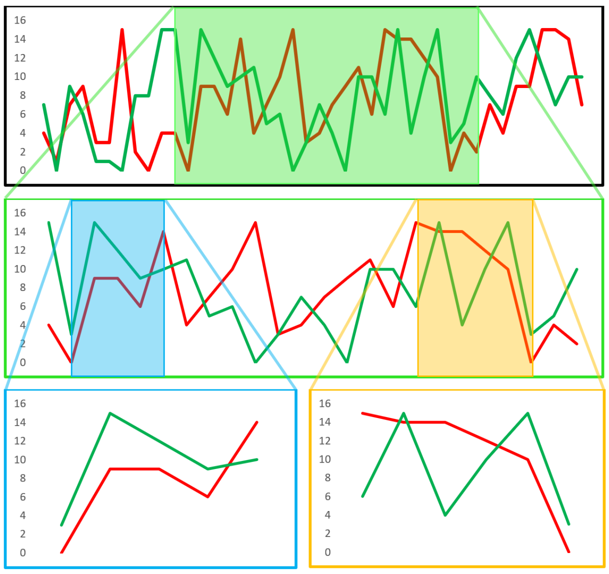


Figure 5: The stack zooming technique, in this case used to analyze line graphs. Information partially taken from (Javed & Elmqvist, 2010).

By dragging the mouse over the strip, a child strip is being created, which only includes the selected part of the parent strip, but has the same size as its parent. By dragging the mouse over another part of the parent strip, a new child strip is being created and placed next to the other child strip, but both child strips have the same size, even if the selections within the parent strip have different sizes.

Because all zoom stack levels are always visible, it is important to understand which part of a parent strip corresponds to a certain child strip. For this purpose, the selected part within the parent strip is highlighted, also called color-coded selection area, as well as the frame of the child strip, also called color-coded frame.

The stack zooming technique makes use of hierarchical zoom stacks (or Z-stacks). This hierarchy of focus regions can be technically seen as a tree with a single root node which contains all the other nodes. In a zoom stack, these are called zoom nodes. Each zoom node captures a single strip.

The technique does not specify the visual layout of the zoom stacks on the screen. The stack hierarchy can be displayed vertically, as shown in Figure 5, but it could also be displayed horizontally, instead. The prototype implementation used in (Javed & Elmqvist, 2010), TraXplorer, presents a few design and usability concepts used to improve the user experience when using stack zooming. In TraXplorer, the vertical space is being split by the amount of existing levels, so each level has an equal amount of space. All child strips always have an equal amount of available horizontal space for that level, no matter how big the selected part of the parent strip is. Additionally, it is possible to change the available space for each strip by dragging the borders of that strip.

By dragging the border of the selection in the parent strip, it is possible to change the selection itself, and the interval of the child strip will be adjusted as well. Stack zooming also allows moving the selection within the parent strip, which changes the displayed interval in the child strip. In both cases, it could happen that two or more zoom selection areas overlap. This can also happen if the user creates a new selection which overlaps with an existing selection. In (Javed & Elmqvist, 2010), when two zoom selections overlap, the layout does not change to maintain stability of the visualization, but if the temporal order of the two zoom selections change, the child strips will also change the order to reflect the changes. Furthermore, if two zoom selections have the same size and they overlap, the child strips merge - but this only works when the two intervals of the zoom selections have the same size. Some of this knowledge also helped with developing our comparison methods for Visplore.

The child strips are always kept in the same order as the selections in the parent strip to help the user better understand the connection between the color-coded selection area within the parent strip and the color-coded frame of the child strip.

Both in (Javed & Elmqvist, 2010) and in Figure 5, stack zooming is presented and discussed by using temporal visualizations, but this technique is not limited to that kind of data. However, stack zooming can be applied to any one-dimensional visual space.

2.5 Compression and expansion of areas in Visual Studio 2015

Although this does not count as a comparison, the way the scroll bar works in Visual Studio 2015 is similar to our concept of compression and expansion of areas. In Visual Studio, it is possible to configure how much information the scroll bar should show, by configuring how wide it should be. If it is wider, it is also possible to identify some class methods and some comments as well, although it is still too small to be able to read something.

Visual Studio allows functions and methods to be collapsed to occupy less space, and to be expanded again, to view the code of that function or method. This way it is much easier to navigate through the code of a file which contains many function implementations. Because it is possible to collapse and expand functions, the scroll bar works similar to our intervals, but also somewhat different.

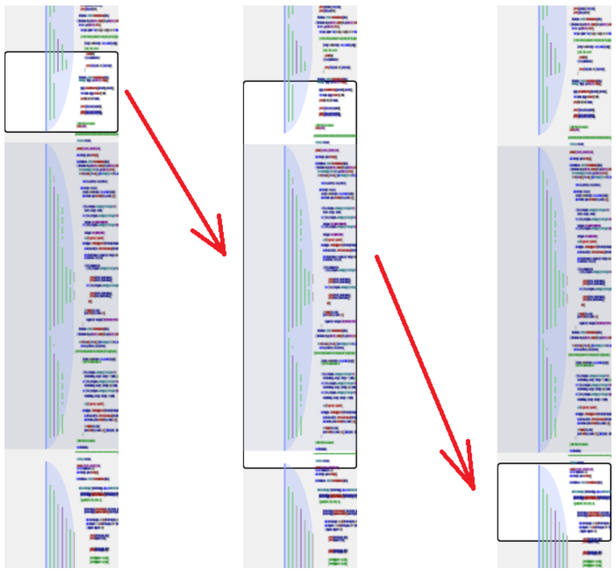


Figure 6: The scroll bar of Visual Studio 2015, while one function has been collapsed. Within the scroll-bar, the function is not collapsed, but has the original size. However, it is shown with a darker background.

Figure 6 shows the scrollbar after collapsing one of the longer methods within the 2D Scalar Grid, which we will talk about later. The function looks collapsed within the text editor (not shown in Figure 6), but the scroll bar shows it at its original size, but with a darker background color. On the left side the collapsed function is not yet visible, but the area above it, which is marked with a black rectangle. In other words, the visible part of the screen is always marked with the black rectangle. After scrolling down slightly, the editor shows the area before the collapsed function, the collapsed function itself, which occupies very little space, as well as the area after that function. The picture in the middle shows a black rectangle around the visible space within the text editor, but because the function has been collapsed in the text editor, but the whole section is still visible within the scrollbar, the black rectangle gets much bigger. In the picture on the right side we can see what happens when scrolling further down - when the collapsed function is no longer visible, the rectangle showing the visible space returns to its original size.

Although this approach is not directly related to intervals or comparison of data, it is partly similar to our approach of compressing the areas between intervals and expanding the intervals themselves. Since the function is shown as being collapsed in the editor, but expanded in the scrollbar, one may argue that it works as an expansion from the collapsed size to the original size, which is in some ways similar to our expansion of intervals. The way the black rectangle expands and gets compressed is a very similar phenomenon to what happens with our implementation of interval-based time comparison, when trying to move a selection from one interval to another one. More about that in Section 5.4.

2.6 Time comparison with overlapped curves in Visplore

Recently, a curve comparison feature has been added to Visplore. This is done by overlapping curves over each other (Figure 7), so it uses a superposed comparison design. Using overlapped curves helps with visually detecting outliers, but also makes it possible to detect small differences between two or more curves. Although this type of comparison was already implemented and worked very well, it only properly worked with very similar data: for example, overlapping many curves that look very differently makes the graph look complex and confusing. On the other hand, comparing similar curves is much easier than by doing a side by side (juxtaposed) comparison, where small details may not be easily found.

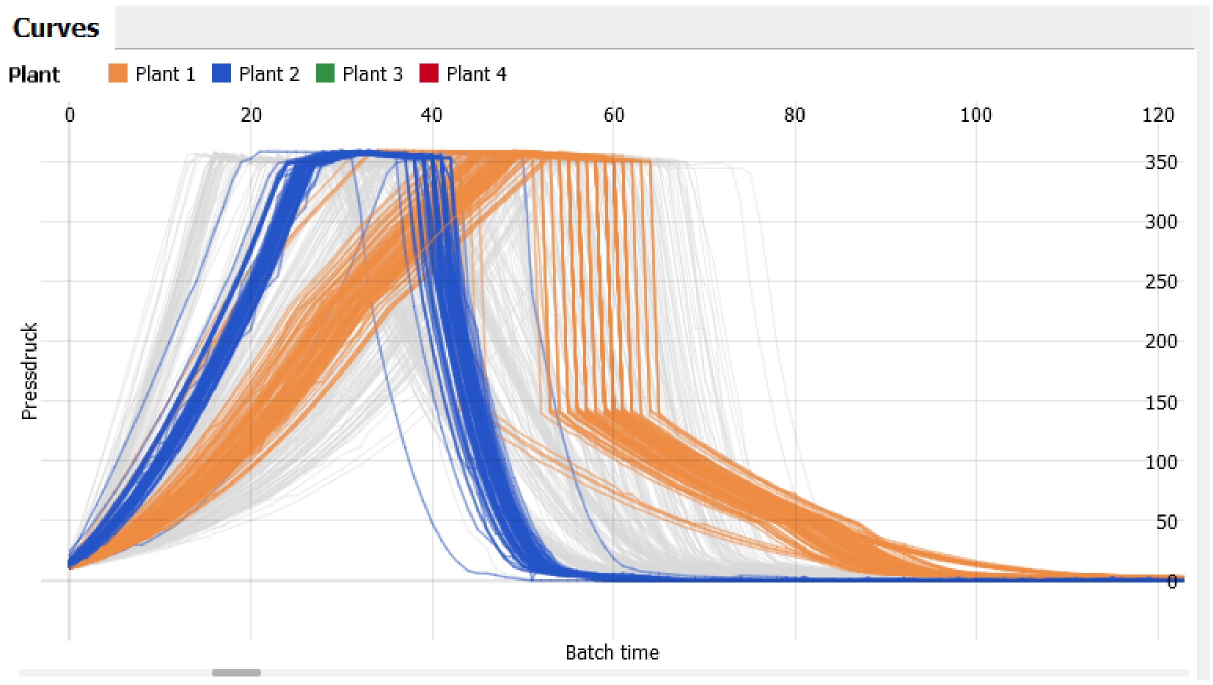


Figure 7: Time comparison using overlapped curves in Visplore. Image taken from (VRVis, n.d.*b*).

A small disadvantage when comparing many curves at once is the difficulty of seeing differences between individual curves. This could be improved, for example, by keeping the superposed design, but binning some curves together, when multiple curves are so similar that they are hard to differentiate anyway. More about the advantages of data binning and how it works can be found below. However, the data binning is not yet used for this feature. Also, although it would also be possible to use a juxtaposed comparison design to compare curves side-by-side, the comparison is even more difficult because of the space needed to show many intervals side by side. With similar curves, using the existing superposed design allows displaying much information at once on the screen, while still showing it with a high accuracy (especially when binning the data), and shows it in a way it makes sense to the user.

2.7 Time comparison with intervals in Visplore

This comparison strategy, which will be developed as part of this master's thesis, uses a juxtaposed design. Compared to the superposed design shown above, where curves are overlapped, this kind of comparison shows time intervals side by side. Although much more space is needed for such a comparison strategy, it makes possible to analyze time intervals that are very different from each other. On the other hand, comparing similar data is possible, but since the intervals are shown side by side, it is not easy to see differences between similar looking intervals, where only small details are different.

Currently, there are two use cases where such a comparison strategy can be used in Visplore, and both of them are not necessarily related to a specific view. The first use case is the comparison of two or more intervals. The user selects the time intervals and adds them to a filter, so that all the other data points are removed. Before implementing the interval-based comparison, this resulted in gaps between the selections, except before the first and after the last selection, because these parts were cut out. However, this wasted much space and although comparing the selections was theoretically possible, they were visually too far apart from each other. The new comparison feature helped by removing the gaps and replaced them with separation lines to improve the usability.

The second use case is the removal of gaps within datasets with big gaps. This can happen when a dataset with discrete data does not contain any data points for a certain period of time. This use case is not directly related to data comparison, but is still a valid use case to save space. Before implementing the interval-based comparison, a lot of space was wasted with showing ranges without any data, although the space could be used more efficiently.

3 Time comparison use cases in Visplore

3.1 The 2D Scatter Plot

According to (Fink et al., 2013), scatter plots are diagrams used to visualize two-dimensional data as sets of points in the plane, and help users detect correlations and clusters in the data.

The 2D Scatter Plot is also a very important feature of Visplore, where numerical values, categories or dates can be assigned to the X and Y axes, creating different graphs. The 2D Scatter Plot plays a significant role when analyzing data: one may analyze and interpret only one property of the data, but the relationship between one property and another one can be much more informative. Technically speaking, the 2D Scatter Plot is a plugin of Visplore.

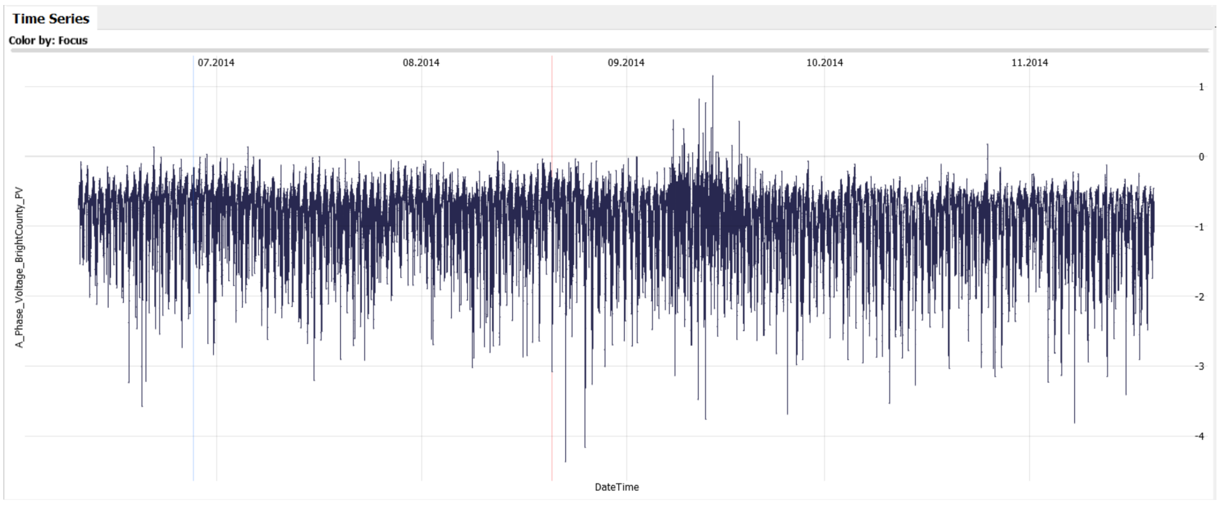


Figure 8: The 2D Scatter Plot without intervals.

Figure 8 shows a 2D Scatter Plot which uses the Photovoltaic (PV) Solar Panel Energy Generation dataset. The X axis contains the DateTime, the Y axis contains A_Phase_Voltage_BrightCounty_PV. In other words, the graph in Figure 8 shows the change in phase voltage in the Bright County over time. Each DateTime entry from the X axis has a corresponding value on the Y axis, so each data point has one X and one Y coordinate, which are then rendered to the screen. If one entry in the dataset does not contain either the value for the X axis, or the one for the Y axis, that value is considered missing. The valid points can either be drawn as single points, or as points connected by lines, as seen in Figure 8.

The 2D Scatter Plot can display both continuous and discrete data. For example, in Figure 8, the X axis contains continuous data, while the Y axis discrete data. However, both axes can contain either continuous or discrete data. Because the time axis is usually the X axis, and because the comparison of time intervals is the most important use case, the main focus was to implement interval comparison of time ranges (on the X axis), but this concept can also be further extended to other types of continuous data, but also to discrete data. For example, when both the X and Y axes have discrete data on them, clusters of data points may appear, eventually with large gaps between them, which could be solved by using intervals.

Both use cases presented in Section 2.7 - the actual comparison of two specific periods of time, and the removal of gaps in datasets with big gaps - are relevant for the 2D Scatter Plot, which has been adapted to work in both cases. In both cases, if no data is shown, the areas just remain white, so collapsing the areas by creating intervals and saving space makes sense. The following chapters will explain in detail how this process works.

3.2 The Horizon Graph View

A horizon graph is a compact way to visualize a graph, created to allow visualizing many graphs at the same time. It has been originally developed by Reijner (Reijner & Panopticon Software, 2008) to analyze financial data, but it is now also used for analyzing other types of data. Because graphs can occupy much space on the screen in a visual analytics software, it can be difficult to visualize many graphs at the same time, especially while preserving the amount of information and preventing data distortion. Furthermore, from a usability standpoint, the user should still be able to understand the displayed data, even if it is shown in a very compact way.

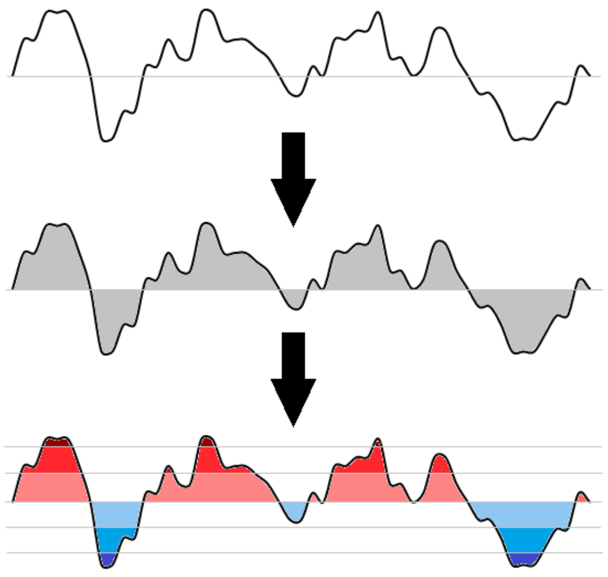


Figure 9: The creation of horizon graphs from a curve.

Figure 9 shows the way a horizon graph is being created. First of all, a time series curve is needed. The curve is split into two equally sized sections parallel to the time axis (e.g. the curve was split vertically in Figure 9). One section is the positive side of the curve (in our example the upper half), and the other is the negative side (in our example the lower half). After that, the parts between the splitting line and the curve line are used for the actual horizon graph. Additionally to the splitting line, further lines are added to divide both sides of the curve and create further subdivisions. Each subdivision gets its own color, but each half gets very different colors. One possibility is to assign warmer colors, which are closer to red, to the positive side, and colder colors, closer to blue, to the negative side - but of course, any colors can be used as long as the difference is clear. These subdivisions on both sides of the curve are also called colored bands, or sectors.

To save even more space, the negative side of the curve is being shown together with the positive side. There are two strategies for that:

1. applying an offset, which means that the negative part just gets an offset to be on the positive side of the graph (e.g. in our example, the negative parts are moved half of the

graph to the top)

2. mirroring the negative side, so it works the same way as the positive side, but with another color

After applying one of the two strategies, the last step is to place the sectors on top of each other: the ones closer to the splitting line are further behind, the ones higher or lower are gradually put on top of each other, with the highest and lowest values in the front.

The result of the whole process is a horizon graph which only needs 1/(number of sectors) of the space a classic curve diagram. Because of the proximity of the data and because of the color-coded sectors, comparing values which are normally far apart is much easier, and finding outliers and correlations is easier as well.

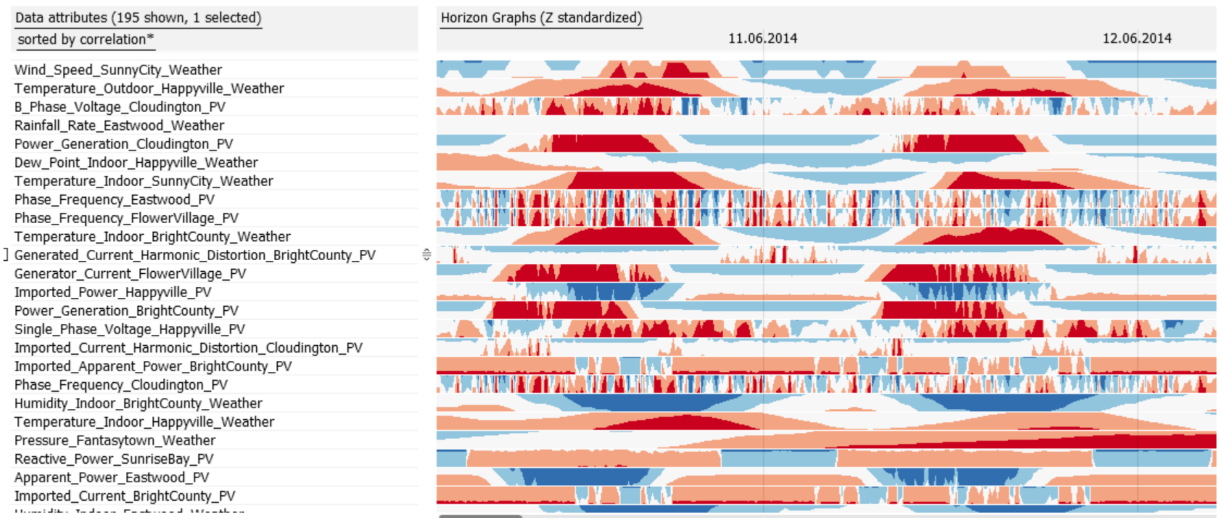

Data attributes (195 shown, 1 selected)
sorted by correlation*
Horizon Graphs (Z standardized)
11.06.2014
12.06.2014
Wind_Speed_SunnyCity_Weather
Temperature_Outdoor_Happyville_Weather
B_Phase_Voltage_Cloudington_PV
Rainfall_Rate_Eastwood_Weather
Power_Generation_Cloudington_PV
Dew_Point_Indoor_Happyville_Weather
Temperature_Indoor_SunnyCity_Weather
Phase_Frequency_Eastwood_PV
Phase_Frequency_FlowerVillage_PV
Temperature_Indoor_BrightCounty_Weather
Generated_Current_Harmonic_Distortion_BrightCounty_PV
Generator_Current_FlowerVillage_PV
Imported_Power_Happyville_PV
Power_Generation_BrightCounty_PV
Single_Phase_Voltage_Happyville_PV
Imported_Current_Harmonic_Distortion_Cloudington_PV
Imported_Apparent_Power_BrightCounty_PV
Phase_Frequency_Cloudington_PV
Humidity_Indoor_BrightCounty_Weather
Temperature_Indoor_Happyville_Weather
Pressure_Fantasytown_Weather
Reactive_Power_SunriseBay_PV
Apparent_Power_Eastwood_PV
Imported_Current_BrightCounty_PV

Figure 10: Horizon graphs in Visplore. The X axis contains the time. The Y axis contains multiple data attributes, each of them having an own horizon graph.

Figure 10 shows the Horizon Graph View in Visplore. The X axis contains the time, while the Y axis contains multiple data attributes. Each data attribute has its own horizon graph. It is possible to zoom in on the X axis, but it is also possible to change the vertical size of the data attributes, to be able to analyze the small details of each horizon graph. Also, for easier comparison of two or more data attributes, the user can order the attributes either by name, similarity, or even manually by dragging them around. In this sense, the Horizon Graph View already allowed comparisons, but only between different attributes, not between time intervals.

The data on both the X axis and on each data attribute can be either continuous or discrete. Although the time (X axis) is usually considered being continuous, datasets with discrete time data may also be used. Furthermore, theoretically, the X axis may also contain some data attribute other than the time - this would be supported by the Horizon Graph View itself. However, Visplore is currently configured in such a way that the Horizon Graph View uses the time as the attribute for the X axis.

Both use cases presented in Section 2.7 - the actual comparison of two specific periods of time, and the removal of gaps in datasets with big gaps - are relevant for the Horizon Graph View, which has been adapted to work in both cases. When filtering out data, the data which has been filtered out is shown in gray, so it makes sense to cut that area out to save space, because no data is shown in that area. When the original dataset contains a time gap in its data, the Horizon Graph View does not show a gray area like when the data is filtered, but interpolates between the last known and the next value. However, because the data is artificially computed, it is better to save some space and cut out such areas as well. If some data attributes have missing data, intervals are not created. Each situation will be explained in detail in the following chapters.

3.2.1 Data binning

According to (Novotny & Hauser, 2006), binning is the process of dividing data into multiple intervals, called bins. This process is used to transform datasets with big amounts of data by replacing the actual (possibly large) data, which may be too dense and hard to understand, with bins. This process should help with simplifying very dense data by creating aggregate data, which results in data with potentially lower accuracy, which should be, however, easier to visualize and understand. Binning is a frequency-based representation which takes a number of values and computes an aggregate value. Additionally, a bin also saves the number of data records which have been aggregated.

Data binning plays a very important role in the Horizon Graph View in Visplore. By default, the view shows the whole graph without zooming, so usually many values need to be displayed at once. Because it is not possible to draw more points than the available number of pixels on the screen, binning is used to create as many bins as the available space for the Horizon Graph View - in our case, the horizontal space, so the width of the view. In Visplore, this kind of binning is called pixel binning.

A bin contains multiple values, but some of them may be faulty. There can be multiple reasons why that happens:

1. The value is missing - when a certain entry either does not contain a value, or when the value has been marked as corrupted, for example when it contains invalid characters or when it has been marked as corrupted during the measurement. We will describe such entries as missing.
2. The timestamp is missing - when an entry does not contain its corresponding timestamp. We will describe such entries as unavailable.
3. The value has been filtered - this can happen, for example, when a user selects only a part of the data. We will describe such entries as filtered.

3.2.2 The Horizon Graph Pipeline

The Horizon Graph Pipeline has multiple stages, some of which are important for the implementation of the new comparison feature. To understand what parts of it needed to be modified and why, one must first understand how the pipeline works. The following stages and substages exist:

1. Parameter preparation - this stage gets as input the timestamp column and the data attribute, with their meta information. After this, further global and local attributes need to be determined:
   - Global attributes - parameters which are used for all data attributes (for all columns) are determined. These are, for example, the number of sectors, whether the negative values should be offset or mirrored, or whether the timestamp column is continuous or discrete.
   - Local attributes - for each data attribute, parameters only applicable to that data attribute are determined. Examples are whether the attribute contains continuous or discrete data, the upper and lower value limits.
2. Aggregation - this stage gets as input the attributes determined in the last stage, as well as the current data attribute. In this stage, the values of a bin are aggregated, and the result is one value per bin. There are three substages:
   - Projection - for each bin, the minimum and maximum values are computed, as well as their first and last temporal occurrence. The number of valid, missing, filtered and unavailable values, as well as the number of all values, are computed as well. After that, the sum of all valid bin values is computed. The result is one aggregate per bin.
   - Reduction - this substage gets as input the aggregates computed during the projection and the local and global attributes. The output is the reduction of the computed aggregate values to one value per aggregate. This considers the type of aggregate. The computed output value can also be missing. The next substage only gets one value per bin as output.
   - Interpolation - this substage gets as input the reduced values per bin and the global and local attributes. The unavailable values (entries without a timestamp) are processed in this substage. If the timestamp column contains continuous data, unavailable values are interpolated. Depending on whether the data attribute column contains continuous or discrete data, different interpolation strategies are being used. The output of this stage is one value per bin. This value can be missing, but not unavailable.
3. Rendering - this stage gets as input the reduced values (one value per bin) and the local and global attributes. This stage prepares the values for the final rendering on the screen,

and the final output of this stage is the actual rendering of the data on the screen. It is divided in three substages:

- Parameter preparation - the local and global attributes are used to create the sectors, determine the value range, as well as the color palette for the final rendering of the horizon graph.
- Mapping - with the help of the local and global attributes and the sector ranges, the actual values which need to be rendered on the screen have to be mapped from the data space to the image space, so they are mapped from the original values to pixels on the screen.
- Rendering using OpenGL - this substage gets the mapped values from the previous substage and sends them using OpenGL to the GPU.

3.3 The 2D Scalar Grid

An important part of both the 2D Scatter Plot and the Horizon Graph View is the 2D Scalar Grid. Its main purpose is to draw regular and suggestive lines for both the X and Y axes, and additionally, to draw the values or dates (for time series) corresponding to these lines. This can be observed, for example, in Figure 8. From now on, the lines drawn by the scalar grid will be called ticks. Although one could also hover specific data points and see their details, being able to see the scale of the whole visible range at once, as well as using ticks to indicate the position of a certain value over the whole view, has a big impact on the usability. The ticks help the user find certain values or dates very fast, especially when not a special value is being searched, but an approximate range of values. Also, zooming in creates additional ticks to get a refined grid which provides even more detailed information.

The 2D Scalar Grid also needed some modifications to support intervals, also because the ticks have to be computed somewhat differently. Depending on the situation, the ticks may be computed by the data mapping itself, but that may not always be the case. This will be explained later in detail.

The 2D Scalar Grid has also been used to display the separation lines between intervals, because this way it was possible to create a view-independent rendering of the separation lines, so it would also work in future views without further adjustments. This, of course, is a trade-off between reusability and customization: on the one hand, implementing the separation lines in the 2D Scatter Plot makes it reusable in multiple views, but on the other hand, different views may also want to display it slightly differently. Because of this, the implementation needed to also be customizable to allow the view to draw it differently depending on the requirements.

4 Mapping data without intervals

The task of a data mapping is to map data from a source to a target domain. For example, some data source may contain values between [15000, 35000], but these may be needed in another range, for example between [-1000, 1000]. A data mapping should be able to efficiently map values from the source to the target range. Mapping the target range back to the source range should also be possible.

There are multiple ways to map the data. The simplest way is to do a linear mapping. In this case, when mapping a value which exists, for example, at 35% of the source range, it will be mapped to the value at 35% of the target range.

Another possibility is to map values logarithmically. In this case, before the actual source to target mapping is being done, the source values are first being logarithmically transformed, and then mapped. When mapping target to source values, first an inverse transformation is being done, and after that the actual mapping. This intermediate transformation step results in logarithmically modified values.

However, more features are being supported, which may not be so obvious at first. One of them is zooming, which is used when only a part of the data needs to be shown. This is explicitly supported by the data mapping to correctly transform the values which should get mapped when logarithmically mapping data, but also, for example, to compute the ticks when using a logarithmic mapping. The minimum and maximum of the visible range are being saved, and the view only displays the actual visible range.

Another feature supported by the data mapping is a special kind of zooming, which is part of the focus-filter widget (Arbesser et al., 2014) in Visplore (see Figure 11). The data mapping additionally allows showing the whole range while displaying a certain part of it as being zoomed in, and showing the other parts without zoom. Because of this, the mapping needs to take this special kind of zooming into consideration when mapping the values. The focus-filter widget is internally called data mapping widget, because it is a way for the user to interact with the data mapping, e.g. by using the widget to zoom in or out.

Other features include setting the mapping as being symmetric when the source range and the mapped values should be symmetric around zero, or setting the mapping as being fixed, when the source range should not be allowed to change.

So why is it not possible to use the same mapping instead of creating a new one? The already existing mapping was only able to map all the data in the same way, e.g. when mapping linearly, each of the values is mapped linearly. When mapping logarithmically, although each value gets mapped differently based on a logarithmic curve, they are still mapped similarly because all of them get mapped using the same mathematical formula (that is, using a logarithmic curve). Since mapping to intervals cannot be calculated using a mathematical formula, but is instead handled differently depending on whether a data point is inside an interval, between two intervals, or before or after all intervals, each data point is mapped differently, so using the already existing data mapping class for interval-based mapping is not possible. Although it

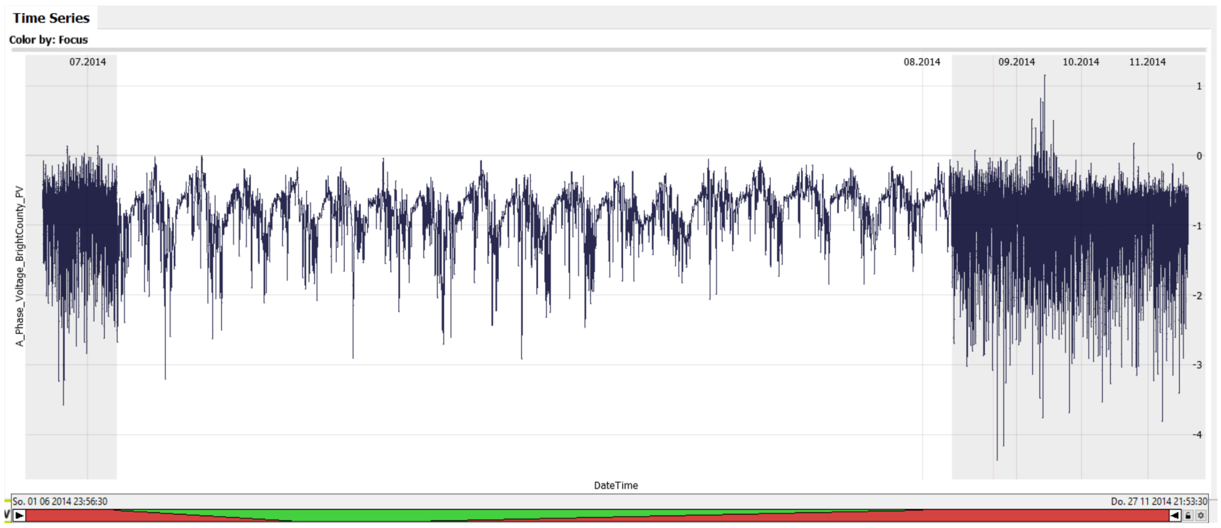


Figure 11: The 2D Scatter Plot used with the focus-filter widget.

would be theoretically possible to add a multi-interval mode to the existing mapping, doing so would make the existing mapping very complex and hard to maintain - which is the reason for creating a new mapping which supports intervals.

One of the reasons why the data mapping is so important in Visplore, is because it is used by many views, including the 2D Scatter Plot and the Horizon Graph View, to map values from the data space to the image space, and then render them on the screen. Although the data mapping can be used to map any range to any other range, because Visplore is a visual analytics software and renders many graphs on the screen whose points need to be mapped from the data space to the image space, the data mapping is a very important part of Visplore. What the two different spaces mean will be explained in the following section.

4.1 Image and Data Space

As already explained, mapping can be done from an arbitrary source range to an arbitrary target range, depending on the use case. However, one use case is to render data on the screen, and for that to happen, mapping values from the data space to the image space is needed. The following paragraphs will explain what these spaces are and the difference between them.

In computer games and modeling, multiple spaces exist: the object space (or model space), the world space, the camera space and the image space. In Visplore, data usually exists in either the data space or the image space. So why are different spaces needed? A simple example: a dataset (e.g. in CSV format) is loaded in Visplore, and this dataset contains different values. The space these values exist in is called data space. However, the screen almost always has different coordinates. Usually, the screen coordinates are between 0 and the width of the view (in pixels) for the X axis, and between 0 and the height of the view (in pixels) for the Y axis.

One may wonder why the data cannot just be converted to the right space and always used like that, instead of having to switch between the two spaces each time, which may seem inefficient at first sight. A simple example: the user has the possibility to zoom in, both in the 2D Scatter Plot and the Horizon Graph View. To be able to show the data at the right position on the screen, the data sent to OpenGL must exist in the image space - otherwise the data will most probably not be shown at all, since the coordinates are wrong. However, to be able to show tooltips with detailed information, such as the actual value of the rendered value, or the timestamp the value corresponds to, and to do that while zooming as well, both the values in the data space and image space need to be known or mapped when necessary. That means that, while zooming for example, although the data in the image space is "distorted" because of the zooming, the original data in the data space still needs to remain the same.

As already mentioned, for rendering, mapping the data from the data space to the image space is needed, both in the 2D Scatter Plot, to map the data points to the right space, and in the Horizon Graph View, to map the bins to the right space. However, mapping from the image space to the data space is often needed as well. Taking the same example from above, when the user hovers a data point with the mouse, a tooltip is shown. But how does this work? A mouse move event is received, with the actual mouse coordinates inside the view. These are the coordinates in the image space, so to find the actual data point, the coordinates need to be mapped to the data space. Since a graph usually consists of two axes (X and Y), each coordinate needs to be mapped using the right mapping - the X mapping for the width, and the Y mapping for the height. After mapping the values to the data space for each axis, it is possible to display detailed information in the tooltip of the hovered button, e.g. for a time-series 2D Scatter Plot it is possible to show the actual date and time of the data point, as well as the value of the other axis.

5 Design of an interval-based time comparison

This introduction will give an overview of all the modifications and additions made to Visplore while developing the interval-based time comparison feature. First, as already mentioned, the data mapping maps the values from the data to the image space. Since we want to show multiple intervals next to each other, we need some kind of data mapping that maps the values differently to show them next to each other. However, another problem appears: the views currently only work with the old data mapping implementation, and if no intervals are needed, the old implementation provides a simpler data mapping without any overhead. That means that both the old and the new implementations need to be supported. This has been implemented by developing a common interface, which will be explained in detail below.

Another possible approach would be to create a derived class from the actual data mapping implementation and extend it with methods specific to the multi-interval data mapping, as well as override the existing methods which need a different implementation than the base class. While this would have been a possibility to develop our architecture, we decided that the implementations were different enough, so that a new implementation would make more sense. Furthermore, having an interface also helps with future implementations of new data mappings: even if our implementation would be similar enough to derive from the old data mapping, it would still mean that a very different implementation which may be developed in the future would need an interface - which is why we decided to develop it right away. If other new data mapping implementations will be developed in the future, it can still be decided whether it makes more sense to create a completely new implementation by deriving from the data mapping interface, or deriving from the old data mapping, or even from our new multi-interval data mapping.

After creating the new interface, the old data mapping must be changed to inherit from it. As part of this step, some of the functionality common to the two data mapping implementations, which will probably also be the same for other data mapping implementations which may be created in the future, is moved from the old mapping class to the interface. This includes the data mapping listener, some common types, and some general methods for computing ticks, but more about that later.

Additionally, a multi-interval data mapping needs to be created, which implements the new data mapping interface. This new type of mapping needs to map the ranges inside the intervals differently than the ones between intervals. In addition to the interface methods, the new mapping needs to provide specific methods for managing intervals, for example for adding and removing intervals, or to check the number of intervals.

For the user to be able to interact with the new type of mapping as well, the data mapping widget needs to be adapted to work with the new interface, instead of working with an actual implementation. This, however, creates a new problem: the data mapping widget should work with all implementations, which is why the interface must be used, but for some use cases it makes the most sense for the user to find certain UI controls directly in the data mapping widget, where all the other mapping-related options exist. Think about the following use case: depending on the view (e.g. 2D Scatter Plot and the Horizon Graph View, in our case), different strategies for splitting the visible range into intervals exist. Because of this, the splitting logic is obviously something that the view itself should do. Now look at Figure 12, where the class dependencies are shown. The arrows show which classes know about other classes, e.g. the view knows the data mapping, but not the other way around. Because the data mapping does not know anything about the view, that means that although the view could itself do the splitting into intervals, it would not be able to provide UI controls for the data mapping widget. On the other hand, the data mapping itself could provide the UI controls, but the controls may be different for each view, so that would also not work. The solution: create a data mapping computation interface, common to all future mapping computations, and create for each view an own computation implementation. That way, the view, the data mapping widget and the data mapping itself have access to the computation. Other view-specific UI controls for the data mapping computation (e.g. enabling or disabling the interval splitting) can also be additionally implemented.

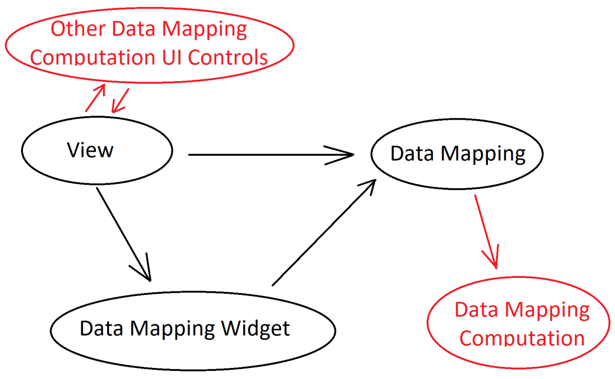


Figure 12: The black elements already existed, the red elements have been added. This is how the data mapping computation works together with the other classes in Visplore. The arrows show which classes know about other classes, e.g. the view knows the data mapping, but not the other way around.

The 2D Scalar Grid also needed some modifications to properly draw intervals. First, the right amount of ticks should be drawn even when displaying intervals, but not more than needed. For example, calculating the ticks the same way as before would result in many ticks being computed for the space between intervals, while the actual visible space would get less ticks. Also, the scalar grid now additionally draws separation lines between intervals and shows the text around interval borders somewhat differently, to emphasize that an interval ends or begin there.

In addition to that, for the implementation of the new feature in the Horizon Graph View, the pixel binning needs to be adapted as well to work with the new interface, instead of the actual implementation. Since the pixel binning implements a binning interface, not only the pixel binning and the interface had to be adapted, but all other binning implementations as well. Also, the pixel binning always assumed that the bins are next to each other without any gaps, which is now not the case anymore.

Finally, the views - in our case, the 2D Scatter Plot and Horizon Graph View - have to be adapted to support multiple intervals. Since the views must know the actual data mapping implementation, the mappings must be changed to a multi-interval data mapping. Additionally, the right computation has to be set in the data mapping. Also, other view-specific UI controls may also be needed for a proper user experience.

Other views have not been adapted yet, but that may come in the future.

5.1 Design of a data mapping interface

Newer programming languages like Java or C# have the concept of interfaces. In such programming languages, interfaces are abstract classes, but without any implementations. Abstract classes are classes which cannot be instantiated, but may contain default implementations which can be used in classes which inherit it. Such languages have their own keywords for abstract classes and interfaces. In comparison, C++ does not have keywords for that, and the language itself does not have the concept of interfaces, but nevertheless, the same concept can be applied to C++ as well. One of the reasons why C++ does not necessarily need interfaces is because it supports multiple inheritance. Other languages like Java and C# allow a class to implement multiple interfaces, but it is only possible to inherit from one class at most. C++ supports multiple inheritance, so actual interfaces are not needed.

In C++, when a derived class inherits a base class, methods which should be overridden in the derived class must be made virtual in the base class for polymorphism to work properly. Furthermore, to force a derived class to implement a method, the method can be made pure virtual. Although C++ does not have keywords for interfaces or abstract classes, a class with at least one pure virtual method is an abstract class, so it cannot be instantiated. Theoretically, a class which only contains pure virtual methods could be called interface in C++. However, often enough, abstract classes are still called interfaces in C++ if they only have a few default method implementations which are common to all implementations of that interface.

These concepts are important for the design of our data mapping interface. First, some classes must work without knowing the actual implementation, so without knowing if the current implementation is the old kind of data mapping, or the new multi-interval data mapping. A problem can be noticed already: if the interface should be used as a drop-in replacement for the old kind of mapping to allow a smooth migration to the new interface, the interface must contain the same public methods as the old mapping. Changing these would of course be an option, but that would mean that multiple thousand lines of code in Visplore would need to be changed and tested again. Although this is a possibility, refactoring the code and testing it would require huge amounts of time - and this without implementing any new feature. Because of this, an interface similar to the already existing data mapping has been created, to ensure both compatibility existing code and efficiency while developing the new features.

5.2 Developing a multi-interval data mapping

To simplify the implementation of the new multi-interval data mapping, a simple data mapping has been additionally used. Instead of mapping source values directly to target values, an intermediate step is required, but without exposing any implementation details. This way it is less complicated to do the mapping, and some functionality of the existing mapping, such as zooming, can be reused, without implementing it again for the multi-interval data mapping. From a software design standpoint, this also means that the more code can be reused, and the simpler the implementation is, the less bugs will exist. Also, when a bug will be fixed for the simple mapping implementation, the multi-interval data mapping will automatically get the fix as well. Instead of directly mapping the source range to the target range, the multi-interval mapping works this way:

1. The multi-interval mapping itself does not have a source range. The source minimum is the minimum of all intervals, and the source maximum is the maximum of all intervals. The source range of the multi-interval mapping is therefore only defined by the source range of the intervals, as explained above.
2. When setting the target range, the target range of the internal mapping gets set instead.
3. The internal mapping source range is set to the range [0,1], so it does a mapping from [0,1] to the target range.
4. When mapping a value, the multi-interval data mapping first maps the value to [0,1] considering the intervals, then the internal mapping maps the values from [0,1] to the target range.

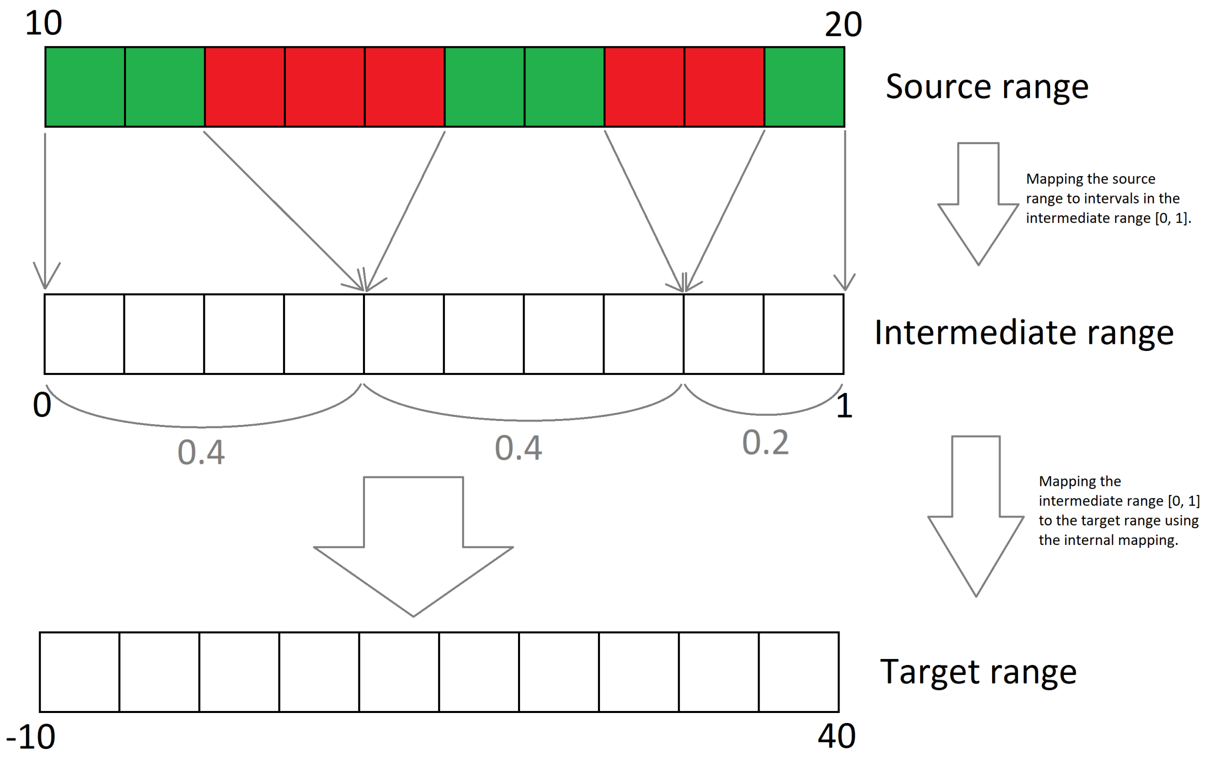

10
20
Source range
Mapping the source range to intervals in the intermediate range [0, 1].
Intermediate range
0
1
0.4
0.4
0.2
Mapping the intermediate range [0, 1] to the target range using the internal mapping.
Target range
-10
40

Figure 13: The multi-interval data mapping does not directly map a source value to the target range, but uses an intermediate mapping for that purpose. Three intervals are defined in this example, highlighted in green. The red parts are areas between intervals.

Figure 13 shows in detail how each of the steps presented above work. The example above shows a source range containing 3 intervals: [10, 12], [15, 17] and [19, 20], which are shown in green. The red sections are parts which are not defined as intervals. Our target range is [-10, 40], and, of course, we also have our intermediate range [0, 1]. To be easier to understand, the range has been divided in unit blocks to exactly show which parts gets mapped to which intervals. For demonstration purposes and to keep the graph as simple to understand as possible, we omitted the gap between intervals which should usually be displayed to improve the usability, by showing some space and a dividing line between the intervals. For example, if the value 11.5 needs to be mapped to the target range, the right interval needs to be searched first - and it is the first one we show in our example. The value 11.5 would be in the middle of the second green block, which is part of the first interval. Looking at the intermediate range, 11.5 would be mapped exactly between the third and fourth block of the intermediate range, containing the value 0.3. The last thing we need to do is the mapping to the target range, which is done by the internal mapping. Because values within the intermediate range [0, 1] are mapped to the target range, which is in our case [-10, 40], the value 0.3 within our intermediate range will be mapped to 5 within the target range. That way we mapped the original value 11.5 to 0.3 with the help of the defined intervals, and at the end to 5 within the target range.

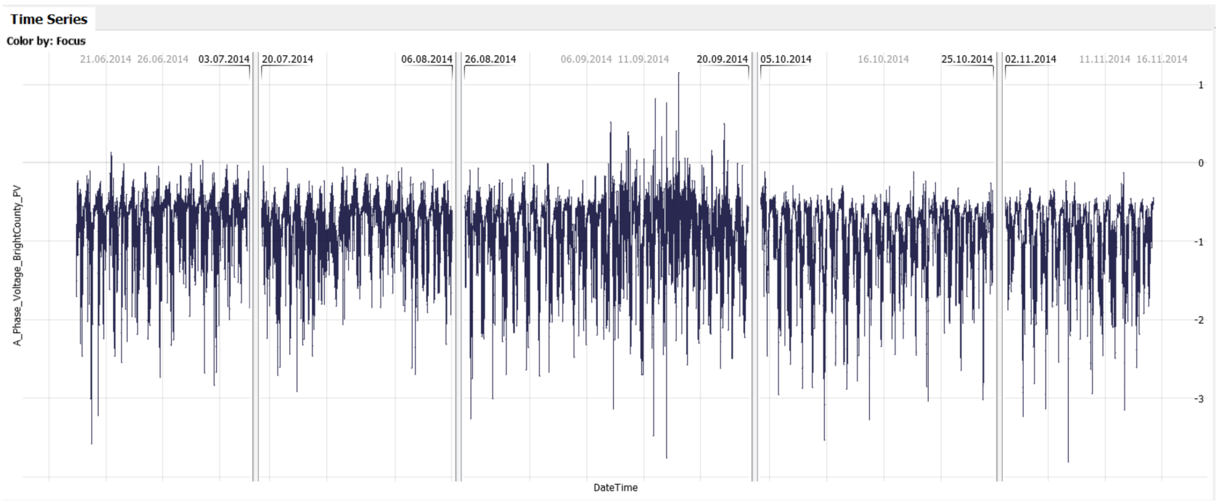


Figure 14: The 2D Scatter Plot with multiple intervals.

For this to work, intervals need to be defined. Each interval has a source minimum and maximum. Also, during the design of our multi-interval data mapping, we decided that it must always be guaranteed that a value gets mapped, so for one source value there must always exist one target value, no more and no less. The old data mapping worked the same way, so changing the way it works would have broken the compatibility with the existing code, which assumes that every input value gets successfully mapped to exactly one output value. Because of that, the mapping must be unique, so intervals cannot overlap. The reason why it worked like that for the already existing mapping was simple: when linearly or logarithmically mapping values, for every x value passed to the mapping function, one f(x) value is returned - regardless of whether a linear or logarithmic mapping is done.

Of course, theoretically we could make it possible for intervals to overlap. It would theoretically be possible to design the multi-interval data mapping the following way:

- values between intervals are not mapped at all, so for one input value, there is no output value
- values inside one interval are mapped to that interval, just as in our implementation - so for one input value, there is one output value
- values inside multiple intervals are mapped to each of the intervals, so for one input value, there are multiple output values

First, this approach would increase the complexity. If we just need to map many values, while the number of values or the relationship between the original and the mapped values are not important, a new array with a different number of values can simply be returned. However, one may want to know the relationship between the original and the mapped values, which means, whether an original value has not been mapped, or has been mapped to one or multiple values, and which original values got mapped to which mapped values. Additionally to the increased complexity, this would also mean a performance hit: we cannot search anymore for the first (and only) interval which contains the value, but need to search for all intervals which may contain the value. To properly display overlapping intervals without combining them, it would theoretically be possible to display two intervals next to each other, although they overlap, because, although they partially have the same source range, they could be assigned different intermediate ranges. This would mean, however, that some parts of the graph would be visible twice (or even more than that, if multiple intervals overlap). Also, this may also cause odd display issues, for example with selections, because the same selection would be visible more than once, and would change in multiple parts of the graph with only one move. Because of all these reasons, we decided that we should not support overlapping intervals at all. There are two strategies which can be applied to overlapping intervals:

1. If an already existing interval overlaps with the new interval which should be added, the new interval is ignored and not being added anymore. This also means that, if multiple intervals overlap, only one of them will be added as an actual interval.
2. Combine the overlapping intervals, creating one bigger extended interval. This strategy seems logical, because the whole selection done by the user would be part of the new, bigger interval. It also works as expected if only one axis uses multiple intervals. However, as soon as both X and Y axes use multiple intervals (which we do not support at the moment, but would theoretically be possible), it may happen that the overlapping intervals need to be extended on both axes, potentially including parts which have not been selected at all, as seen in Figure 15.

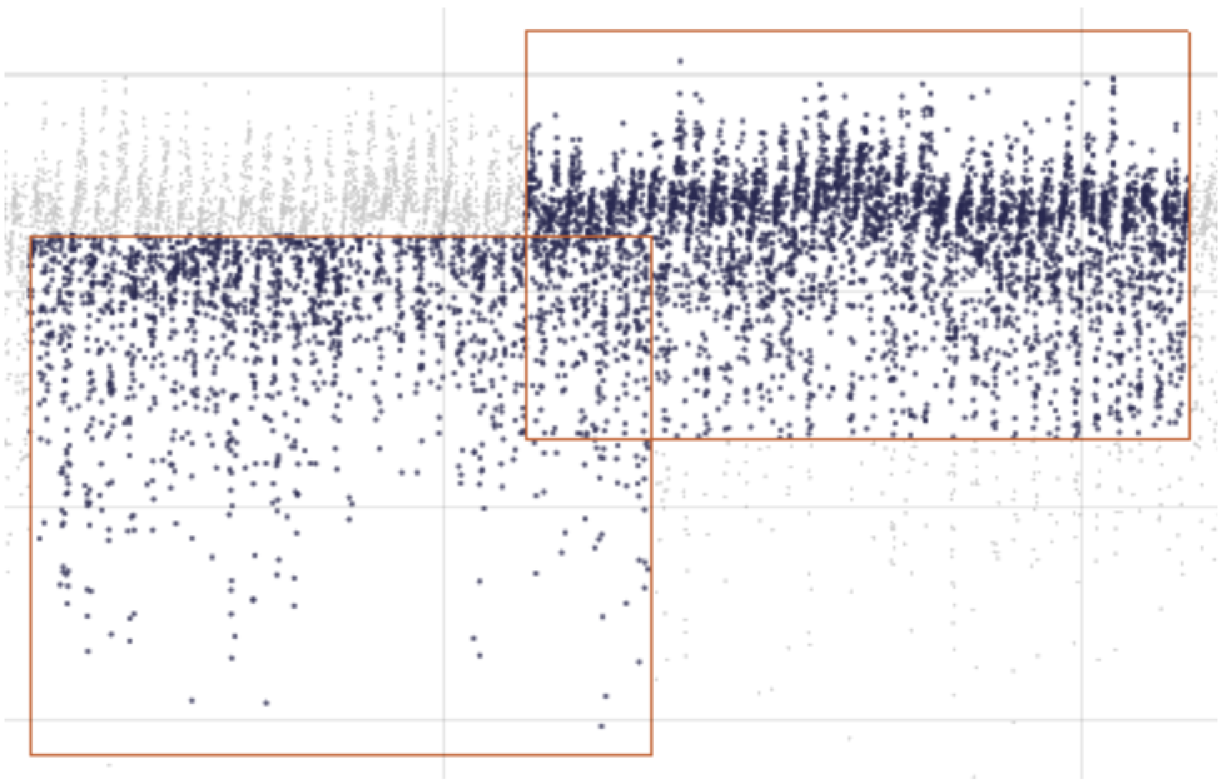


Figure 15: Overlapping selections in the 2D Scatter Plot. These could be used to create intervals, which would then overlap. Theoretically, it would be possible to create one extended interval out of the two selections, but extending the interval on both axes would mean that larger parts which have not been selected by the user would be part of the interval.

Because of this, we decided to ignore overlapping intervals at the moment. If a new interval should be added, but the new interval overlaps with an existing one, the new one is not added anymore. We may return to this decision in the future and improve the usability so that the issue above does not occur. We also decided that it would be the easiest to just keep the same rule as with the existing mapping, so one input value always has one output value - also when a value should be mapped between two intervals.

However, it may visually look odd if the user can modify the already created intervals, but the intervals cannot overlap, even if the user tries to make them overlap (e.g. by dragging the borders of the created interval). Fortunately, the way the intervals are created and displayed in Visplore does not allow the user to change the intervals after they have been created, so that way we managed to avoid the issue - more about the exact way it works in the 2D Scatter Plot and the Horizon Graph View will be explained later.

For efficiency reasons, the intervals are always being internally sorted from the lowest to the highest ranges. Each time one or multiple intervals are being added, they are sorted again

For usability reasons, a target gap between intervals is considered when mapping. That way the mapping lets a small space between intervals, which helps to easier differentiate between intervals.

When mapping a value from the source range to the internal range, there are four cases which need to be considered:

1. the source value is before all intervals
2. the source value is after all intervals
3. the source value is in one of the intervals
4. the source value is between two intervals

Since the mapping must always return valid mapped values, one of the above cases must always apply. Each of these cases are also shown in Figure 16. The first bar represents a source range containing 3 intervals: [10, 12], [15, 17] and [19, 20], which are shown in green. The red sections are parts which are not defined as intervals. Each dot represents one of the cases explained above: the blue dot shows a value before all intervals, the orange dot represents a value after all intervals, the purple dot is a value between two intervals, and the yellow dot is a value within an interval. All these values must be properly mapped, and the exact way it works will be explained below.

Depending on which case applies, different strategies are used to map the values. When a value exists either before all intervals or after all intervals, it is being extrapolated - that means, the distance to the first, and respectively the last interval, is calculated. Other strategies could have been used, but we decided that extrapolating the value made the most sense in this case.

When a value is inside one of the intervals, a simple linear mapping is done. When a value is between two intervals, there are again multiple strategies which can be used to map the value. Since a target gap between intervals (which may be bigger than 0) is being considered, doing a linear mapping between the maximum of the last interval and the minimum of the next interval made the most sense. If the target gap is 0, the point gets then mapped exactly at the border between the two intervals.

From a technical point of view, mapping values between intervals is not a problem, because the multi-interval data mapping has been designed in such a way that mapping values must always work. However, from a usability standpoint, creating intervals in such a way that values exist between them results in a bad usability. Because the important data should exist inside intervals and not within the target gap between intervals, the target gap is visually narrow, because it should not occupy much screen space. That also means that, if many values exist between intervals, they have very little space and it is not possible to distinguish between them. It even gets worse when the target gap is 0, because all points get mapped to the same coordinate. Fortunately, although having data between intervals is technically possible, the way the multi-interval data mapping is used in the 2D Scatter Plot and the Horizon Graph View - that is, intervals are only created when the gap between two values is large enough - ensures that only empty areas are removed, so values will not exist between intervals.

One of the challenges was to map the values as efficient as possible. Imagine having to map 10000 values. First, checks are needed to determine whether the value is before all intervals,

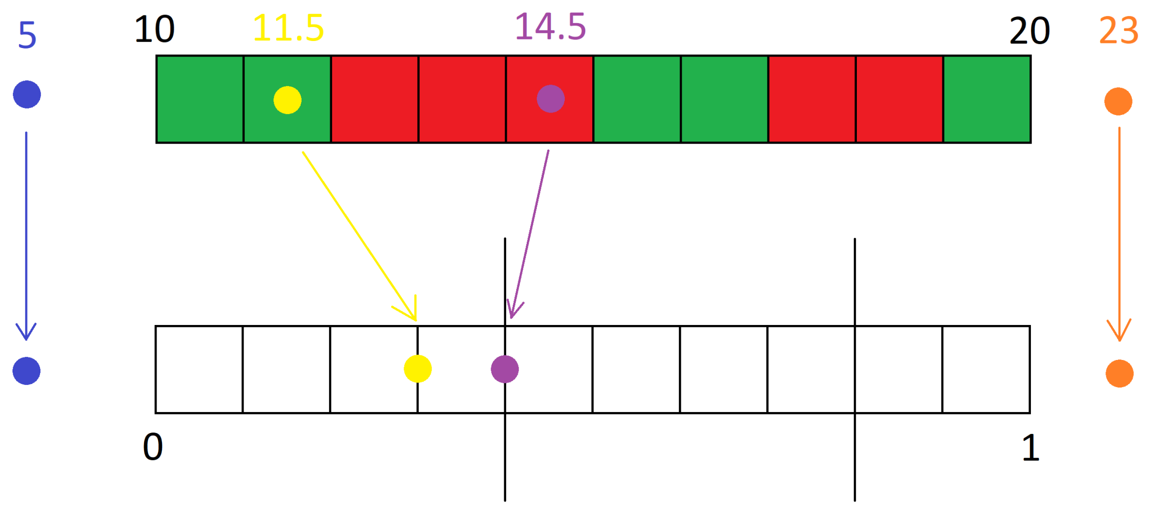


Figure 16: The multi-interval data mapping needs to map each source value to exactly one target value, or each target value to exactly one source value. Because of this, every case needs to be handled: whether a value is before all intervals (the blue dot), or after all intervals (the orange dot), or between two intervals (the purple dot), or in one of the intervals (the yellow dot).

after all intervals, inside one of them, or between two of them. For the last two cases, the correct interval needs to be found first. Doing the same for each of the values would slow down the mapping significantly. Instead, a smarter way can be used: instead of doing a full search to find the right interval for every single value, the last found interval can be saved and, if possible, reused for the next value. Often enough the values are more or less in the same area, so the probability of being able to reuse the last interval is high - but this may not always be the case. Even when the last found interval cannot be used to map the next value, a full search is still not needed: since the intervals are sorted, if the value is located before the last found interval, it is only needed to search to the left of that interval, and if the value is located after the last found interval, it is only needed to search to the right of it.

One feature present in the normal data mapping, but missing from the multi-interval data mapping, is the ability to do logarithmic mapping. This may make sense for a simple mapping, but since mapping with intervals works entirely different, this mode is not supported.

5.3 The data mapping computation interface

The reason the computation itself is needed has already been explained above and shown in Figure 12, but a computation interface has also been created to allow for multiple computation implementations for different use cases. For example, although the final result of splitting data into intervals looks very similar between the 2D Scatter Plot and the Horizon Graph View, the actual implementation is quite different, because the way they work is also very different. The computation interface is part of the Visplore Core, since it should be possible to implement it in every other part of Visplore, if needed.

By creating multiple implementations, it is also possible to create different UI to configure the data mapping within the data mapping widget. Because the user should be familiar with the options in both views, they both have a similar look and feel, but still, having two different implementations allows to extend each of them in the future, e.g. with new features or specific customizations.

5.4 Integration of the multi-interval data mapping in the 2D Scatter Plot

The main use case of showing multiple intervals in the 2D Scatter Plot was with time-series (continuous) data. Of course, it is also possible to create intervals by using discrete data, for example by sorting the data and splitting it into intervals the same way it is done for time-series data, but this was not our main use case. Because the date and time are always shown on the X axis, we decided to only create intervals on the X axis, and only when the 2D Scatter Plot is used as a time-series view, but prepared the 2D Scatter Plot to also work without time-series data, and also with the Y axis, whenever we will decide to support it. Theoretically, it is possible to display intervals on both the X and Y axes at the same time, but this may lead to confusion for the user. However, depending on the use case, this may be useful for displaying clusters: if many data points are nearby each other, but with big gaps between them, using intervals on both axes reduces the space needed for the potentially big gaps between the clusters, and allows to view the clusters in more detail - which is even more important in this case, because a cluster contains many data points in a very dense space, which are easier to analyze when more space is available. On the other hand, it depends on the goal the user tries to achieve: if analyzing the distance between the clusters is important, cutting the gaps and showing intervals is counter-productive - but for that use case, the user can also disable the interval and see the graph without intervals, and enable them again when needed.

Because the new multi-interval data mapping does not support all the features of the old data mapping, and because we still have many cases where no intervals are needed, we decided that we needed to support both mapping implementations. There are two possibilities of implementing this:

- The X and Y mappings of the 2D Scatter Plot save a pointer to the interface, instead of a pointer to the actual implementation. Depending on whether intervals are needed or not, the appropriate type of mapping implementation is used. If the user interacts with the software and a switch of the mapping implementation is needed, a new instance of the other implementation can be created, the relevant information can be copied and the old implementation can be deleted. This, however, has some disadvantages: information needs to be copied every time, and the listeners need to be re-registered. Also, it would be needed to differentiate in many places between the different implementations - in other words, dynamic casts to the actual implementations would be needed in some places, as well as further tests to insure that the correct implementation is used.

- The X and Y mappings of the 2D Scatter Plot save a pointer to the new multi-interval data mapping, so the 2D Scatter Plot always knows which implementation it uses. Depending on whether intervals have been added or not, the multi-interval data mapping acts either as the old mapping, if there are no intervals, or as the multi-interval data mapping, if intervals have been defined. The multi-interval data mapping can act as the old mapping by forwarding the calls to the internal mapping whenever there are no intervals defined, and use its own implementation instead, when intervals have been defined.

We decided to use the second option primarily because of the easier implementation, but also because of the potential performance improvements over the first option.

Basically, intervals are created when a gap exists in the data which is big enough to create intervals. This can either happen when the dataset contains gaps in the data (for example, when data is only registered during work days, or when the production of a product is stopped for a certain time), or when the user only selects certain data to either be contained within the filter (so only the selected data is considered), or when the selected data is filtered out (so everything except the selected data is considered), if the gaps between the data within the filter are big enough. Per default, the multi-interval mode is active, so even without user interaction, if the dataset contains gaps which are big enough, they get cut out by default. The way intervals are created is shown in Figure 17.

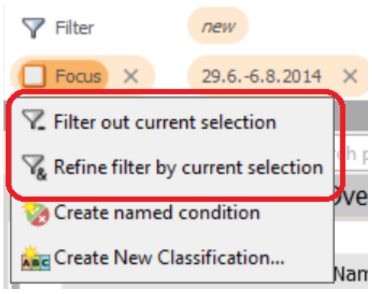
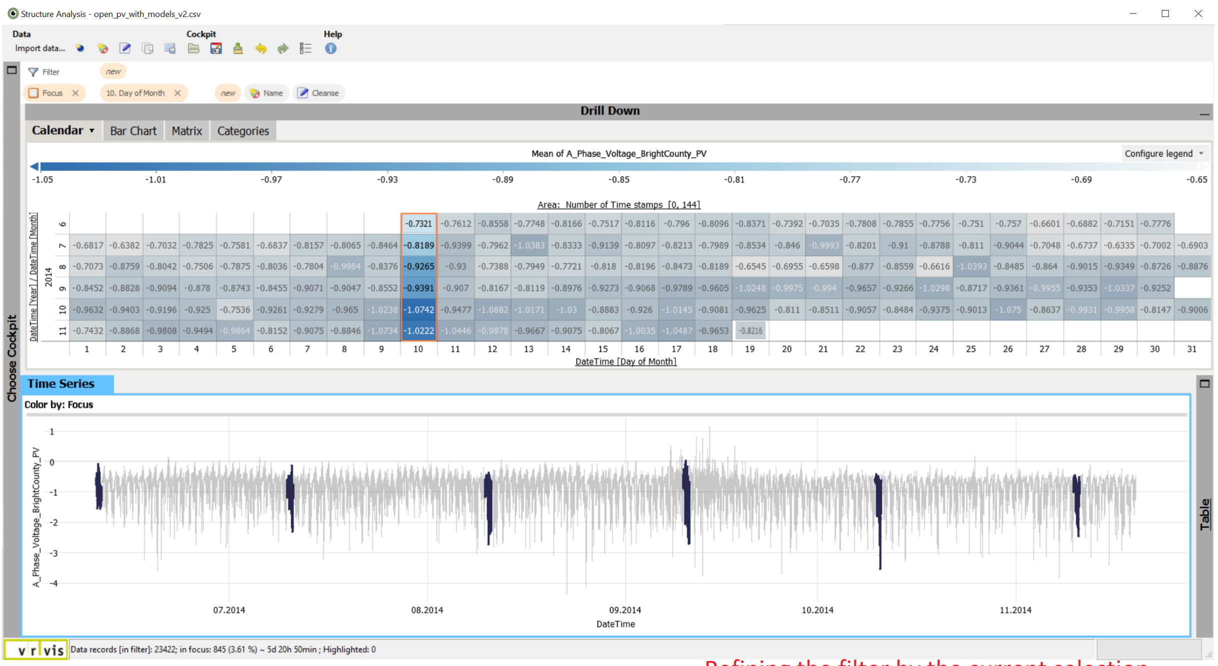

Filter
new
Focus
29.6.-6.8.2014
Filter out current selection
Refine filter by current selection
Create named condition
Create New Classification...

Figure 17: After doing some selections, the user has the possibility to add these to the filter. There are two possibilities: the current selection can either be filtered out, so that everything except the data within the selection is added to the filter, or the filter can be refined by the current selection, which means that only the data within the selection is added to the filter. If the gaps created are big enough, intervals are created.

Of course, on the one hand, the user may just want to consider one part of the data, and use intervals to save space instead of seeing empty gaps after the data has been cut out. On the other hand, a very important use case is also the comparison. Figure 2 shows how comparison of random selections works. This may be useful if the user sees something interesting and wants to select it directly. However, the user may want to compare specific parts - for example, specific periods of time. It may be interesting to compare two different months with each other, or to compare the Christmas sales of two different years by comparing the corresponding quarters of the two years. For such a use case, the user needs to select the exact parts which are important, without having to do a manual selection, because it is hard to do manual selections with very high accuracy. For example, Figure 18 shows a comparison of the 10th day of the months June until November. By adding this selection to the filter, the filter only contains the selected dates, which creates gaps big enough between them so that intervals get created. That way, the user can compare the days with each other and see exactly the data which is interesting for the required comparison, without gaps and/or wasted space.



2014	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
6										-0.7321	-0.7612	-0.8558	-0.7748	-0.8166	-0.7517	-0.8116
7	-0.6817	-0.6382	-0.7032	-0.7825	-0.7581	-0.6837	-0.8157	-0.8065	-0.8464	-0.8189	-0.9399	-0.7962	-1.0383	-0.8333	-0.9139	-0.8097
8	-0.7073	-0.8759	-0.8042	-0.7506	-0.7875	-0.8036	-0.7804	-0.9984	-0.8376	-0.9265	-0.93	-0.7388	-0.7949	-0.7721	-0.818	-0.8196
9	-0.8452	-0.8828	-0.9094	-0.878	-0.8743	-0.8455	-0.9071	-0.9047	-0.8552	-0.9391	-0.907	-0.8167	-0.8119	-0.8976	-0.9273	-0.9068
10	-0.9632	-0.9403	-0.9196	-0.925	-0.7536	-0.9261	-0.9279	-0.965	-1.0238	-1.0742	-0.9477	-1.0882	-1.0171	-1.03	-0.8883	-0.926
11	-0.7432	-0.8868	-0.9808	-0.9494	-0.9864	-0.8152	-0.9075	-0.8846	-1.0734	-1.0222	-1.0446	-0.9878	-0.9667	-0.9075	-0.8067	-1.0035

2014	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
6	-0.796	-0.8096	-0.8371	-0.7392	-0.7035	-0.7808	-0.7855	-0.7756	-0.751	-0.757	-0.6601	-0.6882	-0.7151	-0.7776	
7	-0.8213	-0.7989	-0.8534	-0.846	-0.9993	-0.8201	-0.91	-0.8788	-0.811	-0.9044	-0.7048	-0.6737	-0.6335	-0.7002	-0.6903
8	-0.8473	-0.8189	-0.6545	-0.6955	-0.6598	-0.877	-0.8559	-0.6616	-1.0393	-0.8485	-0.864	-0.9015	-0.9349	-0.8726	-0.8876
9	-0.9789	-0.9605	-1.0248	-0.9975	-0.994	-0.9657	-0.9266	-1.0298	-0.8717	-0.9361	-0.9955	-0.9353	-1.0337	-0.9252	
10	-1.0145	-0.9081	-0.9625	-0.811	-0.8511	-0.9057	-0.8484	-0.9375	-0.9013	-1.075	-0.8637	-0.9931	-0.9958	-0.8147	-0.9006
11	-1.0487	-0.9653	-0.8216												

Data records [in filter]: 23422; in focus: 845 (3.61 %) ~ 5d 20h 50min ; Highlighted: 0

Refining the filter by the current selection. After this step, the 10th day of the months June until November are compared with each other by using intervals.

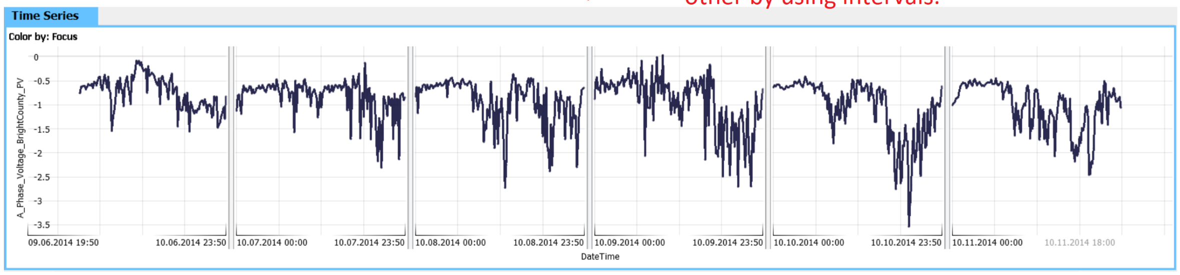


Figure 18: Selecting the 10th day of each month in another view in Visplore, it is possible to add these to the focus with ease, as seen in the top picture. Furthermore, by adding these days to the filter, it is possible to compare them with each other, as seen in the bottom picture.

Figure 19 presents some selection modes available in Visplore. These worked properly with the existing mapping and should also work with the new multi-interval data mapping. On the left side the horizontal 1D interval selection mode has been used, which is used to select everything between the shown bars. In our case, this has been done on the horizontal axis (the X axis), so only one part of the X axis is selected, while everything within that range on the Y axis is selected as well. However, a vertical 1D interval selection mode is available as well, for only selecting a part of the Y axis. In the middle of the figure, the rectangular selection mode is shown, which is very similar to the 1D interval selection mode, except that it is done on both the X and Y axes simultaneously, so both axes are only partially selected within the specified ranges. On the right side of the figure the lasso selection mode has been used, which allows the user to freely define the selection using the mouse - so the user moves the mouse, and the path created by the movement is used to create a polygon containing many points. After a selection has been created, the user also has the possibility to click on it and drag it around.

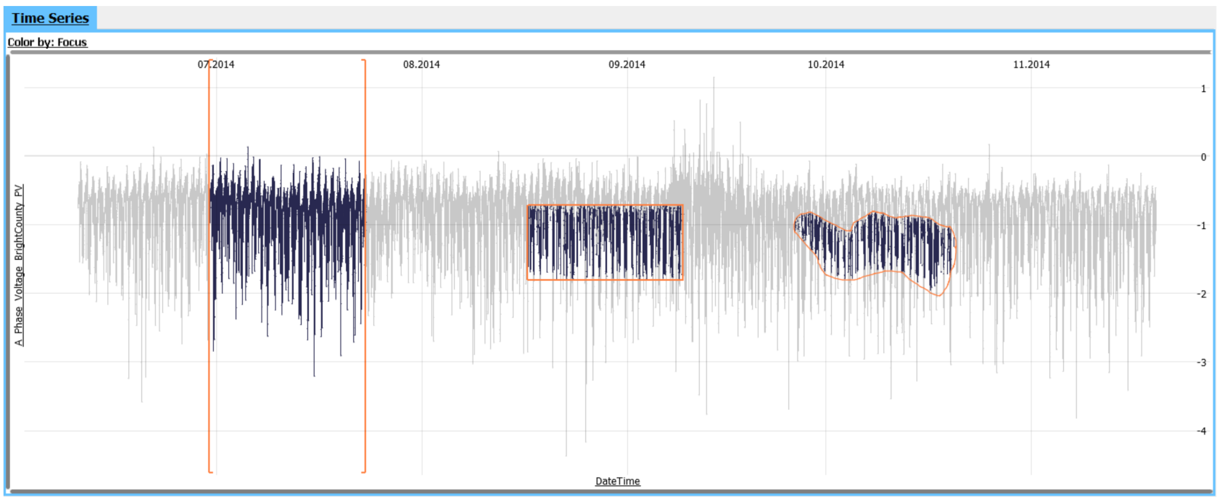
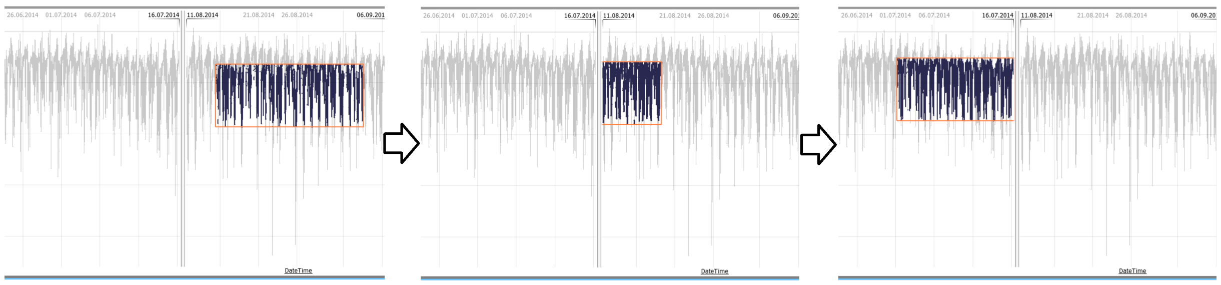


Figure 19: Different selection modes in the 2D Scatter Plot in Visplore. On the left side the horizontal 1D interval selection mode has been used, in the middle the rectangular selection mode, and on the right side the lasso selection mode.

The selections work, as expected, with the multi-interval data mapping as well. However, something interesting happens when trying to move a selection by dragging it around with the mouse, when the selection is moved between intervals. As explained above, what actually happens is that the intervals get expanded, and the areas between the intervals get compressed - however, this process is transparent to the user, because the intervals are created by Visplore and it is not possible for the user to change them afterwards, like with selections. The user is however able to create a selection and interact with it, for example by moving it around, but the same kind of compression and expansion happens also with selections, which becomes apparent when moving the selection around.

Figure 20 shows the compression and expansion of the rectangular selection, as explained above. On the left picture, a rectangular selection has been created inside the second visible interval. The picture in the middle shows how the selection is dragged to the left side, while the mouse is still inside the second visible interval. As expected, the selection gets compressed the



1. Creating a rectangular selection

2. Dragging the rectangular selection to the left, while still being inside the same interval.

3. Dragging the rectangular selection even further to the left, jumping to the interval on the left side.

Figure 20: Dragging the rectangular selection from one interval to another one.

nearer it comes to the interval separation line, until it potentially becomes as small as the space between the intervals. This, however, depends on both the original size of the area between the two intervals and the size of the selection. The picture on the right shows how the selection is already on the other side, since it moved from the second to the first interval. One can already observe that it is getting bigger the more it moves to the left side, and moving it even further to the left (not seen in the pictures above) would bring the selection back to its original size. This behavior is similar to the one in Visual Studio 2015 when collapsing functions, as explained in Section 2.5.

In Figure 2, the comparison of three selections made with the horizontal 1D selection was shown. The same can be done using a vertical 1D selection, although currently intervals are only created on the X axis. This can be observed in Figure 21, where a vertical 1D selection has been used to select the lower part of the data. Depending on the data shown on the Y axis, it may make sense to only select a part of the data for a more detailed analysis. Because continuous time-series data has previously been shown, but only a part of the data has been selected, many small gaps have been created after adding the selection to the filter. One further thing to notice is the size of the separation lines: because some intervals may contain a very low number of data points, it may be a waste of space to add many separation lines occupying a lot of space, while being much bigger than the intervals themselves. Because of this, if not much space is available, the separation line gets smaller to give more space to the actual data, which is more important. In other words, size of the separation line depends on the size of the intervals around it, to save as much space as possible.

5.4.1 The data mapping computation in the 2D Scatter Plot

The data mapping computation has two purposes: the computation of intervals, and to provide the UI controls for the data mapping widget. There are three different computation modes which are relevant to both the way the intervals are computed, and to the UI controls shown to the user: a single interval mode, a multi-interval percentage gap mode, and a multi-interval value gap mode. These are also presented in Figure 22.

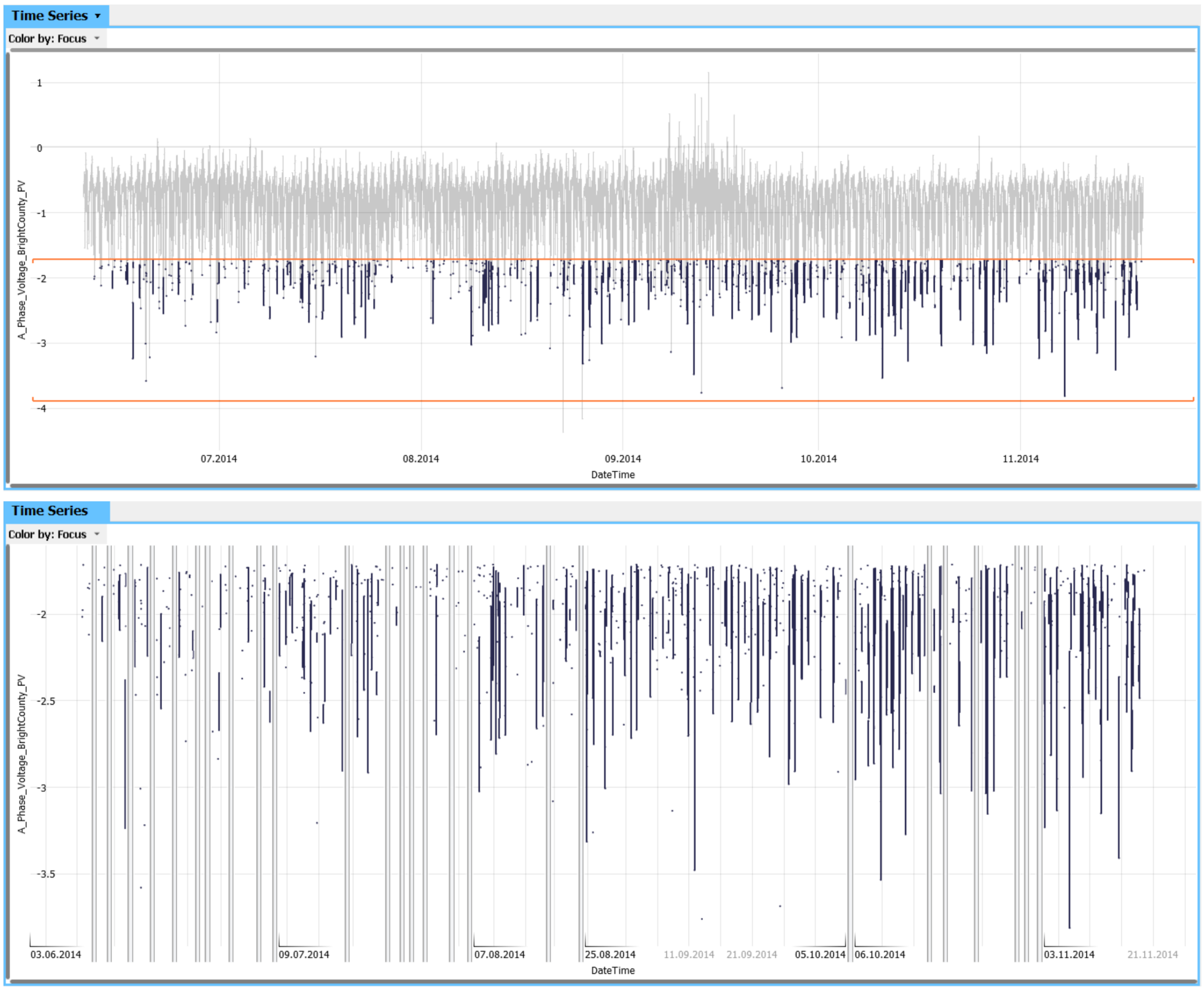
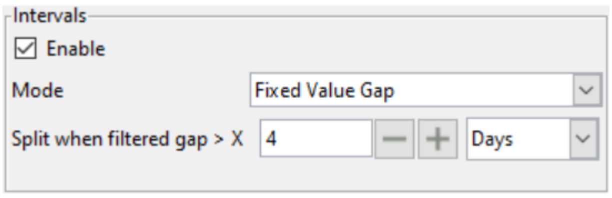
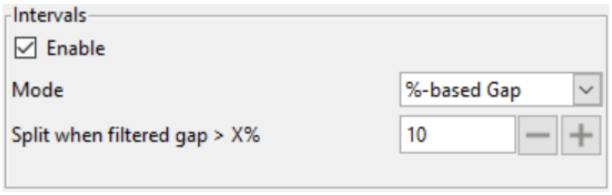


Figure 21: The 2D Scatter Plot with a vertical 1D selection, which was then added to the filter. Because only the lower part of the data has been selected, many small gaps have been introduced. The picture at the bottom shows intervals after the maximum gap has been set to 21 hours.

The single interval mode works like the already existing mapping: it removes all existing intervals and sets the source range. If the user does not want to see any intervals, this is the right mode to choose. The percentage gap and the value gap modes are similar to each other, with one important difference: the percentage gap mode works with percentages, so it creates an interval if a gap in the data, which is bigger than that percentage, exists. The value gap mode works with absolute values instead, e.g. seconds, minutes, hours, days, weeks, months, quarters, years. Of course, other units for non-time-series usage may be added in the future. Using actual (absolute) values may help the user understand the gap size better than with percentages. This is an interesting usability point, because it is usually easier for users to imagine cutting out gaps which are bigger than 1 day, rather than cutting out gaps which are bigger than 10% of the total data range. This also means that, although it is easy to find a reasonable value for the percentage gap mode, it is much harder to find one for the value gap mode. For example, after some tests we decided that a reasonable default value for the


Intervals
☑ Enable
Mode: Fixed Value Gap
Split when filtered gap > X: 4 Days


Intervals
☑ Enable
Mode: %-based Gap
Split when filtered gap > X%: 10

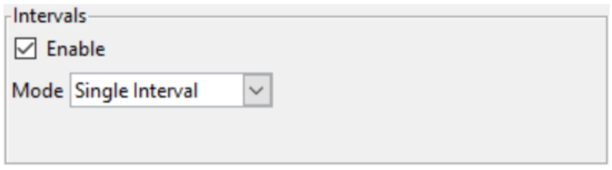

Intervals
☑ Enable
Mode: Single Interval

Figure 22: The computation modes in the 2D Scatter Plot. These can be configured within the options of the Data Mapping Widget.

percentage gap mode is around 10% - this means that, without user interaction, intervals are created if the gap between two values is bigger than 10% of the total range. Although this is easy to implement programmatically, we also wanted to offer a better usability by providing the user with reasonable default values for the value gap mode, which is easier to understand than percentages. However, finding a reasonable default value is much harder, because it has to be computed according to the data. For example, if the dataset includes data taken over a period of a few years, it may make sense to set the default value to 1 month, or, if the dataset includes data taken over one hour, it may make sense to set the default value to 5 minutes. Also, "nice" values should be computed, because users usually prefer to see a default value of 2 hours, instead of 1 hour, 58 minutes and 23 seconds - so after determining a reasonable value, it must first be converted to a "nice" value, and and be used afterwards as a default value. It is important to compute a reasonable value which can directly be used by the user, without having to change the setting first. That way, the new interval feature can also be used by new users who never used Visplore before, but also by more experienced users who want to be able to manually set the exact values when intervals should be created.

The computation calculates the maximum gap size, considering the total range of the whole (non-filtered) data. First, the previous intervals are removed. Also, although continuous data is mostly ordered in the data source, e.g. data sources with time series data first contain the older entries, and then the newer entries, data points can theoretically exist in any order in the data source. Because of that, the data points are sorted first, before comparing them with each other. After that, the gaps are being computed this way:

- If the gap between the very beginning and the first actual point is bigger than the maximum allowed gap, the gap at the beginning is cut out by taking the first actual data point. Otherwise, the very beginning is taken.

- For each point, if the gap between the next and the current point is bigger than the maximum allowed gap, that part is cut out.
- If the gap between the last actual point and the very end is bigger than the maximum allowed gap, the gap at the end is cut out by taking the last actual data point. Otherwise, the very end is taken.

If no gap was big enough, nothing has been cut out. In that case, the total source range containing all the data points is set, without setting any intervals. Otherwise, the computed intervals are set. That being said, both when setting intervals or setting none at all, an additional operation is made: the source range is extended for usability reasons. Without this operation, the visible range would begin at the very first data point and would end at the very last data point. However, because it looks nicer to let some space between the visible beginning/end of the scatter plot and the first/last data point, the range is extended.

Additionally, it may happen that only one interval gets created, because the gaps are not large enough to create further intervals. Even with one interval, it can happen that either the gap at the beginning of the original range has been cut out, or the gap at the end of the original range, or both of them, or even none of them, but otherwise no other intervals have been created. In this case, only one interval exists, even if anything has been cut out, the whole range is still being shown (including the parts which should have been cut out), just as if no intervals would exist. This was a deliberate decision to improve the usability: one of the ways a user can find out whether the multi-interval mode is enabled, is by seeing interval boundaries. These are drawn as separation lines between two intervals. However, because in our case only one interval would need to be shown, no separation lines would be drawn, so the user may get confused to see no separation lines, but also not the whole range. Because of this, we decided to only cut out the beginning and/or the end when at least two intervals are shown.

Additionally, the 2D Scatter Plot contains controls which allow the user to enable or disable intervals, as seen in Figure 23. These are the other data mapping computation UI controls, as shown in Figure 12. Although this is also possible by going to the settings of the Data Mapping Widget, when intervals should only quickly be enabled or disabled without changing the computation mode or setting any value for it, it is much easier for the user to have a button which can be quickly accessed anytime. Of course, this button is only available when creating intervals is possible, so it is only shown whenever it makes sense.

5.4.2 The 2D Scalar Grid with intervals in the 2D Scatter Plot

Some changes also needed to be done to the 2D Scalar Grid. Instead of creating a fixed amount of ticks for a certain range, the ticks must consider intervals now. This is because ticks have previously been created at a regular distance from each other over the whole source

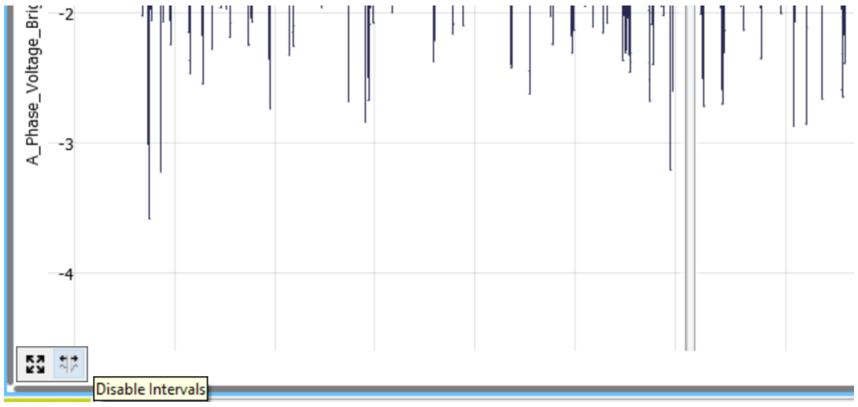


Figure 23: In the 2D Scatter Plot, intervals can easily be enabled or disabled by using buttons which are easy to find.

range. However, because of the way our multi-interval data mapping works - it expands the intervals and compresses the areas between them - it also meant that many ticks have been created in the space between intervals - something we actually do not want to happen. Between intervals, a separation line should be shown, and not tens of ticks which cannot be properly seen. Because of this, the tick computation needs to be done per interval, instead of the whole source range. Also, we decided that it makes the most sense to let the 2D Scalar Grid draw the separation lines between the intervals, because that way it must not be implemented again in every view.

The tick computation is done by the data mapping, so that it works regardless of the implementation used. That way, it works like it worked before for the old type of mapping, but also properly works with intervals for the multi-interval data mapping. Additionally to the standard ticks, some other usability improvements have been made to ensure that it is clear for the user where an interval begins and ends. More exactly, additionally to the coordinates used to draw the ticks, it is now possible to also configure whether the tick line should also be drawn in addition to the corresponding text, the priority each tick has, the alignment of the corresponding text and the color of the text.

First, to properly identify that intervals are available, it should be clear to the user where the beginning and where the end of an interval is. We decided it would be the easiest to draw these with the help of the 2D Scalar Grid, but a bit differently than the other ticks. For example, when drawing ticks without intervals, all tick lines are drawn normally, and the text corresponding to the tick is drawn in black and centered over the tick line. This way, all ticks look similar. With multiple intervals, it makes more sense to differentiate between the ticks: there are ticks which are drawn at the beginning of an interval, ticks which are drawn at the end of an interval, and ticks which are drawn in between. At the beginning and at the end of an interval, the tick lines are not useful, but we still want to see the text corresponding to them, which shows the interval minimum and maximum values. Because of this, it is now possible to configure whether the line corresponding to the text should be drawn at all, as well as the alignment of the text. The beginning of an interval should have its text drawn on the right side of the interval beginning, and the end of an interval should have its text drawn on the left side of the interval end. Also, to emphasize the beginning and end values even more, only these are drawn in black, while all the other ticks are drawn in gray. All these steps are always only done for at least partially visible intervals, so no unneeded ticks are being computed. These usability improvements can be observed in Figure 24, which shows a 2D Scatter Plot with intervals, while zooming in, where there is enough space to draw both the beginning and the end of intervals, but also the ticks in between.

Summarized, instead of treating all ticks similarly and drawing all of them in black we now do it like this:

- The ticks at the interval borders are computed first.
- The tick line at the beginning of each interval is not drawn, the tick text is aligned to the right, and it has the highest priority.
- The tick line at the end of each interval is not drawn, the tick text is aligned to the left, and it has the second highest priority.
- All the other ticks have a visible tick line, the tick text is centered, and they have the lowest priority.

The priority is related to the tick text. The text at the beginning of an interval should always be drawn if there is enough space. If there is even more space, the text at the end of an interval should also get drawn. If even more space is available, the text of the other ticks is drawn - but we always check whether there is enough space available, so if two tick texts have the same priority, the next one is only drawn if there is enough space.

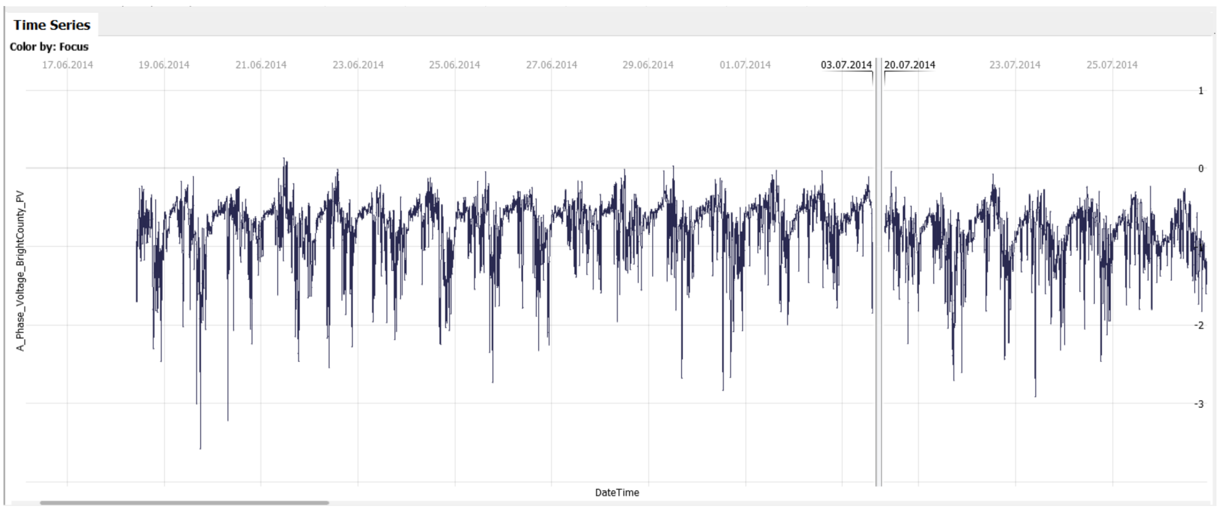


Figure 24: The 2D Scatter Plot with multiple intervals, while zooming in.

Without going too much into details, the final ticks are not computed by the mapping itself. This is because the tick computation does some more complex computations instead of just creating multiple ticks between two floating-point values (the interval range, in our case). This is done to improve the usability for the user, so the most useful ticks are computed, instead of computing random ticks. This is done by the so-called types in Visplore. There are multiple kinds of types: DateTime, Float, and many more. These can be used for different purposes, for example to convert strings to a value of this type, or convert a value to a nice-looking string, or to safely convert values of another type to this type, or, for example, to compute useful ticks. For example, for a time-series graph, a dataset may have one row of data every second. Depending on the amount of data / data density, it may make sense to create ticks for each minute, or for each hour, or maybe for each day. It may make more sense for the user to properly identify where each hour begins, than to create a random tick anywhere within our data. This way, it is possible to create more useful ticks which help pinpoint certain points in time which may be relevant for the user. Of course, all the other types (e.g. the float or the integer type) also help creating more useful ticks, depending on the kind of data loaded.

5.5 Integration of the multi-interval data mapping in the Horizon Graph View

One of the biggest differences between the 2D Scatter Plot and the Horizon Graph View is that the 2D Scatter Plot directly displays data points in space, while the Horizon Graph View creates bins out of data points to display them as compact and understandable as possible. Although intervals are currently supported only on the X axis in the 2D Scatter Plot, it is very easy to implement it for the Y axis as well, if needed. This is possible because data attributes get assigned to both the X and Y axes, so both axes contain data points, so intervals can be created if gaps appear. This however does not make sense in the Horizon Graph View: the X axis contains one data attribute, but the Y axis contains multiple data attributes, so it would make no sense to create intervals on the Y axis. Also, since the Horizon Graph View does not display traditional data points like the 2D Scatter Plot, but bins instead, intervals must be created differently than in the 2D Scatter Plot.

For the same reasons explained in 5.4, we also decided to save a pointer to the actual multi-interval data mapping implementation, instead of switching between the simple and the multi-interval data mapping implementations. In the 2D Scatter Plot, both axes use the new multi-interval data mapping, but multiple intervals are disabled on the Y axis for reasons which were already explained, even though it would be possible to create intervals on both axes. In comparison, the Horizon Graph View has only been adapted to work with multiple intervals on the X axis, while the Y axis still uses the simple data mapping implementation. This is because the Y axis contains multiple attributes, where it does not make sense to ever create intervals - also not for the future. Because of this, changing the implementation used on the Y axis would not only potentially make the mapping slightly slower, but would also not make sense at all.

In the Horizon Graph View, it is possible to create intervals either from the bins which need to be displayed, or from the original data points which were used to create the bins. The advantages and disadvantages of the two methods will be explained in the next chapter. Although using the data points for computing intervals is similar to the way the 2D Scatter Plot works, some additional work needs to be done for displaying the computed intervals within the Horizon Graph View. After computing the ranges for creating intervals, just like the 2D Scatter Plot needs to do, the bin ranges must be computed - but because the Horizon Graph View uses a Pixel Binning, it must do it for the visible range only - so either when in interval is partially or fully visible. After that, the borders of each bin are computed, and then the bins themselves. As one can see, some more work needs to be done compared to the 2D Scatter Plot. Although some of these steps also needed to be done before intervals have been implemented, the implementation was simpler (but also faster), so some steps, like the pixel binning, needed to be adapted before they could be used with intervals.

Figure 25 shows the Horizon Graph View without intervals at the top, and with intervals at the bottom. It also shows that two intervals have been created. The first interval has been created by compressing the gray area which can be observed in the top picture. It consists of data which has been filtered out by the user, for example after doing one or multiple selections. The second interval has been created by compressing the second area, which is made of straight lines. These are created whenever the dataset has no entries for a certain period of time, also called unavailable. In this case, the Horizon Graph View interpolates between the values, which creates the straight lines for the second area, instead of showing a gray area. Since the Photovoltaic (PV) Solar Panel Energy Generation dataset has continuous time data without gaps in it, it was necessary for demonstration purposes to slightly modify the dataset and create a bigger gap in the data. In our image, both the gap created after filtering the data and the one consisting of interpolated values occupy a big portion of the screen space, which is wasted, because the bins either contain no data (like the gray area) or contain artificially computed, interpolated data, both of which are usually not very interesting for the user. Creating the two intervals, as seen in the bottom image, saved much space. It can also be observed in Figure 25 that missing data of certain attributes (shown in black) is not replaced by intervals, although the gap would be big enough.

Generally, it makes sense to create intervals where some data is not available. This can happen due to multiple reasons:

- The data has been filtered out, which is shown in gray when no intervals are created. This is an action which needs to be done by the user. In the Horizon Graph View, the user can either filter a certain period of time (on the X axis), or select one or more attributes on the Y axis, but it is not possible to actually filter a part of the Y axis, or a part of a single attribute. Because the time is the same for all attributes, filtering a time period filters it for all attributes, so it is not specific to one of the attributes. Filtered data can be observed in Figure 25 on the left side of the top picture. In this case, it makes sense to create intervals, because using screen space to show no data at all wastes space and does not allow the rest of the data to be shown with higher precision, since more space is available to it.

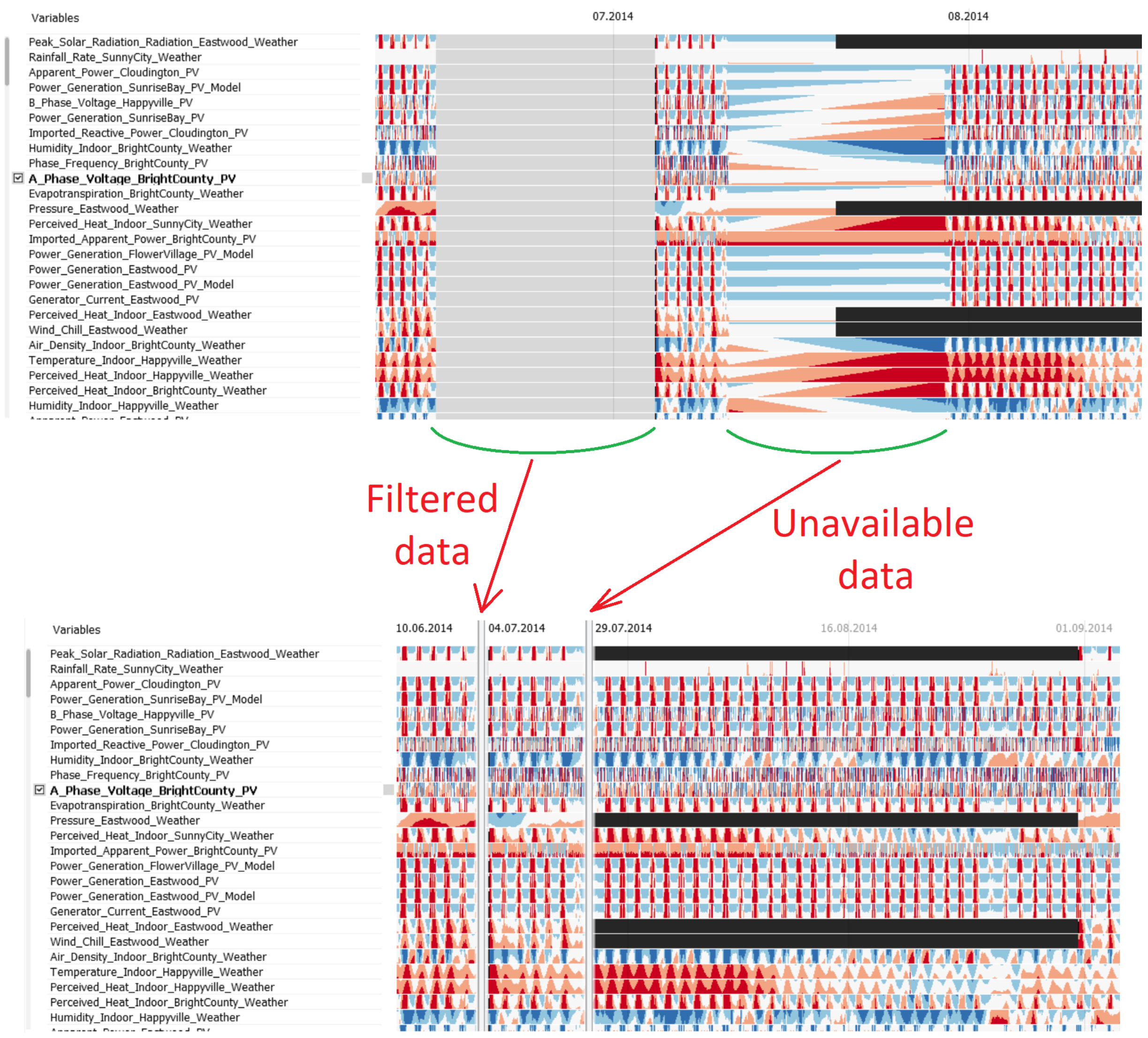

Variables
07.2014
08.2014
Peak_Solar_Radiation_Radiation_Eastwood_Weather
Rainfall_Rate_SunnyCity_Weather
Apparent_Power_Cloudington_PV
Power_Generation_SunriseBay_PV_Model
B_Phase_Voltage_Happyville_PV
Power_Generation_SunriseBay_PV
Imported_Reactive_Power_Cloudington_PV
Humidity_Indoor_BrightCounty_Weather
Phase_Frequency_BrightCounty_PV
A_Phase_Voltage_BrightCounty_PV
Evapotranspiration_BrightCounty_Weather
Pressure_Eastwood_Weather
Perceived_Heat_Indoor_SunnyCity_Weather
Imported_Apparent_Power_BrightCounty_PV
Power_Generation_FlowerVillage_PV_Model
Power_Generation_Eastwood_PV
Power_Generation_Eastwood_PV_Model
Generator_Current_Eastwood_PV
Perceived_Heat_Indoor_Eastwood_Weather
Wind_Chill_Eastwood_Weather
Air_Density_Indoor_BrightCounty_Weather
Temperature_Indoor_Happyville_Weather
Perceived_Heat_Indoor_Happyville_Weather
Perceived_Heat_Indoor_BrightCounty_Weather
Humidity_Indoor_Happyville_Weather
Filtered data
Unavailable data
Variables
10.06.2014
04.07.2014
29.07.2014
16.08.2014
01.09.2014

Figure 25: The Horizon Graph View without intervals (top picture) and with intervals (bottom picture). The first gap (gray, on the left side) has been created by filtering out that part of the data. The second gap (in the middle, consisting of straight lines), has been created by using a slightly modified version of the Photovoltaic (PV) Solar Panel Energy Generation dataset. More exactly, data has been deleted for a certain period of time, which means that the data is unavailable. Missing data of certain attributes (shown in black) is not replaced by intervals, because intervals are only created when all attributes have gaps in the data (either filtered or unavailable).

- The data is unavailable because it is not available for a certain period of time. Since this applies to time data, it also affects all attributes, just like when the user filters out data. The main difference is that the data is not available in this case, and no action needs to be done by the user. Compared to data which has been filtered out, the Horizon Graph View interpolates between unavailable data values. In other words, if there is no data for a certain period of time, the Horizon Graph View interpolates between the last known value and the next one, as described in 3.2.2. Although this improved the usability before creating intervals was possible, this still means that a part of the screen space was wasted with rendering data which did not actually exist, but was interpolated instead. Because of this, we decided to also create intervals in this case, to waste as little space as possible.
- The data is missing. This is something specific to certain data attributes, but not to time data. Because gaps are then only present in certain attributes (where missing data is shown in black), it does not make sense to create intervals in this case - neither on the Y axis, where it is not possible anyway, nor on the X axis, where other attributes which are not missing would also be affected.

Because of this, we decided that it makes sense to create intervals both when the user filters some data, and when the dataset contains gaps by itself. Because of this, creating intervals is also very similar to the 2D Scatter Plot and, if intervals are enabled in both views and both of them have the same settings, exactly the same intervals are created, which also ensures consistency between the views.

5.5.1 The data mapping computation in the Horizon Graph View

On the one hand, having a data mapping computation interface helps creating different implementations for different purposes. This is also useful in our case, because the way the intervals are being computed is somewhat different than in the 2D Scatter Plot. On the other hand, we decided to keep the UI the same, so that the users can easily find the same options and to improve the consistency within our software. Because of this, the UI is the same as in Figure 22. Although the UI looks the same at the moment, having two different implementations allows to extend and modify the two computations independently of each other in the future, to allow further options which may not be available in the other views.

It is important to mention that, although the bins could be analyzed to create intervals out of them, it makes more sense to take the original data points (which are used to create the bins). On the one hand, using the bins for computing intervals means that they need to be computed twice, which is inefficient. Using the bins to create intervals would mean that the bins need to be computed for the whole range, not considering intervals. After that, the bins are used to compute the intervals, after which the bins are discarded to create new bins for each interval. On the other hand, using them instead of the actual data points means that the accuracy of the interval computation decreases. Because a bin contains a certain amount of values depending on the amount of displayed data and its density, intervals can only be created if enough bins have no values at all. However, this also means that the invalid values may already begin inside an existing bin, so less is cut out when creating an interval. Because of this, it may even happen that some gaps are not cut out anymore. Using the original data instead of the bins improves both the performance and the accuracy of the created intervals.

Just like the computation of the 2D Scatter Plot, the computation used in the Horizon Graph View also sorts the values first, then compares them with each other and checks whether the gap between two points is bigger than the maximum allowed gap - if that is the case, the gap is cut out. Compared to the 2D Scatter Plot, the Horizon Graph View always shows the data in focus, so if the user selects a part of the data in the 2D Scatter Plot, only this data is being shows in the Horizon Graph View. That also means that, compared to the 2D Scatter Plot, the Horizon Graph View does not need a special handling of the gap before the first value, and the gap after the last value.

5.5.2 The 2D Scalar Grid with intervals in the Horizon Graph View

The same changes which were done to the 2D Scatter Plot were also needed for the Horizon Graph View, so the 2D Scalar Grid itself needed no other specific changes to properly work with the Horizon Graph View.

However, one last small change was still needed: previously, the Horizon Graph View only rendered the 2D Scalar Grid over the header of the Horizon Graph View, to show the dates and the beginning of the ticks, but was not rendered over the whole view, like in the 2D Scatter Plot. This is because the tick lines were harder to see in the Horizon Graph View, but also because seeing them was not as important as in the 2D Scatter Plot. This, however, created one problem: since the 2D Scalar Grid was drawing the separation lines between intervals, it needed to be drawn over the whole vertical space of the view in order to show their full height and cover all data attributes. On the one hand, this change allows the separation lines to be rendered over the whole view, without doing any Horizon Graph View specific changes in the 2D Scalar Grid. On the other hand, this change also causes the ticks to be drawn over the whole view, which might eventually slightly improve the usability, but the change is not significant.

6 Implementation of an interval-based time comparison

6.1 The data mapping interface

The purpose of the data mapping interface has already been explained in Section 5.1. Implementation-wise, the interface should do everything that is common for all (or for most) implementations - in our case, our data mapping interface should do whatever is common to both the simple and the multi-interval data mapping. That includes some functionality previously included in the simple data mapping, which is now common to both mappings.

One example of such functionality is the Data Mapping Listener. It implements the observer pattern and it is used to notify other classes of a change in the data mapping. These classes - or rather the instances of them - have to register themselves at the data mapping for getting updates when something changes. There are then notified when the mapping changes, and there are multiple cases when that can happen:

- When the source range changes.
- When the target range changes.
- When the parameters change (e.g. the zoom, whether the data mapping is fixed, or whether it is symmetric)
- When the source computation change - a feature which has been added as part of the newly added source computations.
- When the data mapping gets deleted.

This way it is possible, for example, for the user to make a change in the Data Mapping Widget, which then applies the change to the data mapping, which then notifies all the registered listeners that something changed - for example the 2D Scatter Plot, or the Horizon Graph View - which can then react accordingly, for example by rendering the data again according to the new settings. That is how the interaction between the data mapping, the data mapping widget and the different views work in the background, so that a change in the UI also changes the data mapping and notifies the view to allow it to react to the changes.

It also includes further general types and structs, which are not specific to any data mapping implementation. Otherwise, the interface contains mostly pure virtual functions, which need to be overridden. A few methods should work the same way for both implementations, and have been directly implemented in the interface. One example for that are the relative mapping methods. Mapping relative values means mapping a value between 0.0 and 1.0 - that is, between 0% and 100% - from the source to the target range, or the other way around. In order to map a relative source to a relative target value, we need to compute the original absolute source value - that is, the original value inside our range - and map normally, using our own methods, to the target value, then compute the percentage inside the target range. By using our own method, each of the mappings can use a different implementation, and the rest of the computations are common to both implementations. One may wonder why a relative mapping is needed at all. Indeed, when using a simple data mapping and mapping data linearly, mapping data relatively is not needed, because a value at 35% of the source range will also be at 35% of the target range. However, mapping logarithmically, or mapping with intervals does produce a different result.

One difficulty of developing the data mapping interface was that the existing classes, which worked with the already existing data mapping, but now needed to be changed to work with both implementations, still needed to work with minimal changes. This is because the data mapping class has been created almost 15 years ago, so it has already been used in many places in Visplore. However, to be able to create a derived class from our new interface, the interface needed to look very similar to our existing data mapping class - which also helps whenever an existing class in Visplore needs to be switched to work with both implementations, because often not many changes are needed. This is, however, also a disadvantage. Writing a new class in such a way that it ensures compatibility with existing code also means that not much of the class design can be changed, otherwise bigger code changes in many places in Visplore are needed. On the one hand, using a tried and tested design means that there is not much room for making mistakes when designing the new interface. On the other hand, because the class already exists for such a long time, redesigning it would also allow us to use newer programming concepts and structures, but at the cost of having to rewrite much of the code which was already using it. Examples of modern C++ features include using std::vectors (with iterators and/or range-based for loops) instead of C arrays, using smart pointers instead of raw pointers whenever suitable, or even references, when they suffice.

6.2 The multi-interval data mapping

Because Visplore is a multi-threaded software, some parts of the multi-interval data mapping needed to be properly synchronized to be thread-safe. One example of synchronization in our case are the intervals: it should never be possible to add or remote intervals on one thread, while another thread maps values, a process which uses the intervals. Because of this, a mutex has been used to synchronize different parts of the multi-interval data mapping, such as when adding or removing intervals, or when mapping values. To improve the performance and not block other threads unnecessarily, read and write lock guards have been used. This has a big performance advantage: a classic mutex blocks the resources from being used by another thread, but a read/write lock gives the programmer the possibility to specify whether a thread only needs to read, or also to write. In other words, when using a classic mutex, only one thread can read and/or write the resources at one point in time. Using a read/write lock, many threads can lock the mutex for reading at any time, because only reading from a resource is not a problem when multi-threading. However, as soon as one thread locks the mutex for writing, all the other threads which also need to lock a mutex for either reading or writing have to wait until the mutex gets unlocked.

Some interesting implementation details will be presented in detail, also with some code examples. One of these will be the way intervals are added to the multi-interval data mapping. The first time one or multiple intervals are added, the multi-interval data mapping needs to change from a single-interval mode, which acts like the already existing data mapping (and supports features like logarithmic or symmetric mapping), to the multi-interval mode. Precisely, this means that the internal mapping gets set to the [0,1] range and both the logarithmic transformation and the symmetric mode get disabled. As previously mentioned, the data mapping always needs to map every single received value to the requested range - for example, every source value needs to be mapped to exactly one target value - no less, no more. Because of this, the new interval which should get added needs to be checked against all existing intervals first, to ensure that the new interval does not overlap with any existing ones. If that does not happen, the interval gets added, and the list of intervals gets sorted again, to ensure that all intervals are always in ascending order. The method then returns whether the interval has been added or not. The exact way the method works can be seen in Code 1. An additional method also allows adding multiple intervals at once, and returns the number of added intervals, as well as which intervals have been added - but works otherwise very similar.

```cpp
bool BAMultiIntervalDataMapping::AddSourceInterval(const double dSourceMin,
    const double dSourceMax)
{
   // the change lock guard ensures:
   //  - that the mutex is (writing-)locked as long as it exists
   //  - that notifications to be sent are collected during that time and then
       sent after unlocking the mutex
   TChangeLockGuard stChangeLockGuard(this);

   assert(IVLFloatType::IsFinite(dSourceMin) &&
       IVLFloatType::IsFinite(dSourceMax));
   assert(dSourceMin < dSourceMax);

   bool bIntervalsOverlap = false;

   if (!m_pIntermediateToTargetRangeMapping->IsFixed())
   {
      if (m_pSourceIntervals->empty())
      {
         ItlEnsureIntermediateSourceRange();
      }

      // Check that neither the new interval minimum nor the maximum are inside
          another interval (and thus overlapping).
      for (auto pInterval : *m_pSourceIntervals)
      {
         bIntervalsOverlap = vm::Overlap<double>({ dSourceMin, dSourceMax }, {
             pInterval->dSourceMin, pInterval->dSourceMax });
         if (bIntervalsOverlap)
            break;
      }

      if (!bIntervalsOverlap)
      {
         auto pNewInterval = new TItlInterval(dSourceMin, dSourceMax);

         m_pSourceIntervals->push_back(pNewInterval);

         std::sort(m_pSourceIntervals->begin(), m_pSourceIntervals->end(),
             [](TItlInterval * pFirst, TItlInterval * pSecond)
         {
            return (*pFirst) < (*pSecond);
         });

         ItlUpdateSourceToIntermediateRangeMappings();

         ItlNotifyListeners(BAIDataMapping::UPDATE_SOURCE);
      }
   }

   return !m_pIntermediateToTargetRangeMapping->IsFixed() && !bIntervalsOverlap;
}
```

Code 1: Adding an interval to the multi–interval data mapping.

Something that needs to be mentioned at this point is the type of data the data mappings get. Because of the way the data mappings work, it makes sense to only accept floating-point numbers - that means that at least the result of the mapping will be most probably a floating-point number. For historical reasons, only float to float mappings have been accepted in the past, and the data mapping only supported double values at a later point in time. Because of this, it was originally possible to map floats to floats, and later it was also possible to map doubles to doubles, but also floats to doubles and doubles to floats, and that from the source to the target range, and from the target to the source range. Because of the already quite complicated list of public methods, and to avoid code duplication, a private templated method has been created which is able to map any floating point to any other floating point. Whenever needed, the C++ compiler compiles the right version of the method, so when a float to double mapping is needed, it uses that one. Strictly speaking, two templated methods have been created - one for source to target mapping, and one for target to source mapping - because the two directions work slightly different.

There were two important optimizations we needed to make to ensure that mapping with multiple intervals works as efficient as possible. First, instead of calculating the interval positions each time when mapping, it is much better to cache them. That means that, in addition to the coordinates of the intervals within the source range, we also save the coordinates within the intermediate [0,1] range. Once added, the source coordinates never change - so they remain the same since the interval gets added until it gets removed. However, within the intermediate range, the required space changes over the time: whenever an interval gets added or removed, or whenever the gap between the intervals gets changed, or whenever the source range gets extended (usually for usability reasons, like in the 2D Scatter Plot, as explained above), the cached interval coordinates within the intermediate range need to be updated. This is because each of these changes means that the sizes of all intervals change, so each interval has different intermediate coordinates after each change. For the usual use case it means that, after an instance of the multi-interval data mapping has been created, one can define intervals with coordinates within the source range, then the mapping calculates their coordinates within the intermediate range. After that, while mapping the actual values - and if the mapping function gets called multiple times - no unnecessary computations are being done.

While mapping values to intervals, although it might happen that the received values are in a random order, they often enough are either in ascending order (e.g. for time series data, the data within the CSV file often exists in an ascending time order), or adjacent values are often similar enough to each other so that they often belong to the same interval. Because of this, one further important performance improvement has been made: the last interval which was used to map a value is additionally cached. Because adjacent values are often enough mapped to the same interval, the same interval which was used to map the last value can usually also be used to map the next value. If this is not the case, the cached interval can also help: instead of doing a full search for the right interval every time when the previous interval cannot be reused, we can start searching the right interval beginning with the previous interval, because often enough the previous or the next interval is the right one, or at least some other interval which is nearby the previously used interval. In other words, in most of the cases, this optimization will vastly improve performance, and in the worst case, the performance will remain the same, but will never get worse.

The exact way the source to intermediate range mapping will now be presented. Keep in mind, however, that mapping in the other direction - that is, from the intermediate to the source range - works similarly.

```
template <typename TIN, typename TOUT>
void BAMultiIntervalDataMapping::ItlMapSourceToIntermediateRange(
    const TIN * pSourceValues,
    TOUT * pIntermediateRangeValues,
    unsigned int nNumValuesToMap,
    unsigned int nStrideInValues /* = 0 */,
    IVL::t_index * piIntervalIndices /* = nullptr */) const
{
    IVLWriteLockGuard tGuard(*m_pMutex);

    // should only be called in case there are intervals...
```

```
assert(!m_pSourceIntervals->empty());

if (nNumValuesToMap > 0 && !m_pSourceIntervals->empty())
{
   auto currentinterval = m_pSourceIntervals->begin();

   if (m_iLastMappedSourceIntervalIndex >= 0 &&
       m_iLastMappedSourceIntervalIndex < m_pSourceIntervals->size())
   {
      currentinterval = m_pSourceIntervals->begin() +
          m_iLastMappedSourceIntervalIndex;
   }
   else
   {
      m_iLastMappedSourceIntervalIndex = 0;
   }

   auto pInputValue = pSourceValues;
   auto pOutputValue = pIntermediateRangeValues;

   for (unsigned int n = 0; n < nNumValuesToMap; ++n)
   {
      if (ext::is_finite(*pInputValue))
      {
         bool bMapped = false;
         while (!bMapped)
         {
            // check if the value is greater than the current interval range
            if ((*pInputValue) > (*currentinterval)->dSourceMax)
            {
               // check if there is a next interval
               auto nextinterval = currentinterval + 1;
               if (nextinterval != m_pSourceIntervals->end())
               {
                  // if the value is between the current and the next
                      interval, map it between them:
                  if ((*pInputValue) < (*nextinterval)->dSourceMin)
                  {
                     (*pOutputValue) = (TOUT)vm::InterpolateLinear(
                        (*currentinterval)->dSourceMax,
                        (*nextinterval)->dSourceMin,
                        (*currentinterval)->dIntermediateRangeMax,
                        (*nextinterval)->dIntermediateRangeMin,
                        double(*pInputValue));

                     if (piIntervalIndices != nullptr)
                        piIntervalIndices[n] = -1;

                     bMapped = true;
                  }
                  else
                  {
                     // otherwise, move to the next interval
                     currentinterval = nextinterval;
                     ++m_iLastMappedSourceIntervalIndex;
                  }
               }
               else
               {
                  // if this is the last interval, just map it:
                  (*pOutputValue) = (TOUT)vm::InterpolateLinear(
                     (*currentinterval)->dSourceMin,
                     (*currentinterval)->dSourceMax,
                     (*currentinterval)->dIntermediateRangeMin,
                     (*currentinterval)->dIntermediateRangeMax,
                     double(*pInputValue));

                  if (piIntervalIndices != nullptr)
                     piIntervalIndices[n] = -1;

                  bMapped = true;
               }
            }
            // check if the value is smaller than the current interval range
            else if ((*pInputValue) < (*currentinterval)->dSourceMin)
            {
               // check if there is a previous interval
               if (currentinterval != m_pSourceIntervals->begin())
               {
                  auto previnterval = currentinterval - 1;

                  // if the value is between the previous and current interval, map it into that range!
                  if ((*pInputValue) > (*previnterval)->dSourceMax)
                  {
                     (*pOutputValue) = (TOUT)vm::InterpolateLinear(
                        (*previnterval)->dSourceMax,
                        (*currentinterval)->dSourceMin,
                        (*previnterval)->dIntermediateRangeMax,
                        (*currentinterval)->dIntermediateRangeMin,
                        double(*pInputValue));

                     if (piIntervalIndices != nullptr)
                        piIntervalIndices[n] = -1;

                     bMapped = true;
                  }
                  else
                  {
                     // otherwise move to the previous interval
```

```
                        currentinterval = previnterval;
                        --m_iLastMappedSourceIntervalIndex;
                    }
                }
                else
                {
                    (*pOutputValue) = (TOUT)vm::InterpolateLinear(
                        (*currentinterval)->dSourceMin,
                        (*currentinterval)->dSourceMax,
                        (*currentinterval)->dIntermediateRangeMin,
                        (*currentinterval)->dIntermediateRangeMax,
                        double(*pInputValue));

                    if (piIntervalIndices != nullptr)
                        piIntervalIndices[n] = -1;

                    bMapped = true;
                }
            }
            else
            {
                // the value is inside the current interval: map it!
                (*pOutputValue) = (TOUT)vm::InterpolateLinear(
                    (*currentinterval)->dSourceMin,
                    (*currentinterval)->dSourceMax,
                    (*currentinterval)->dIntermediateRangeMin,
                    (*currentinterval)->dIntermediateRangeMax,
                    double(*pInputValue));

                if (piIntervalIndices != nullptr)
                    piIntervalIndices[n] = m_iLastMappedSourceIntervalIndex;

                bMapped = true;
            }
        }
    }
    else
    {
        (*pOutputValue) = (TOUT)(*pInputValue);
    }

    pInputValue += nStrideInValues + 1;
    pOutputValue += nStrideInValues + 1;
  }
 }
}
```

Code 2: Mapping source to intermediate range values in the multi–interval data mapping.

Looking at Code 2, some of the most interesting parts will be explained in detail. First, the method gets an array of values within the source range, which should be mapped to the intermediate range. For the values which should be mapped, a second array is provided. Both must be externally allocated, and both are templated, so that both doubles and floats can be mapped, in any combination. One interesting feature is the stride in values, which allows an array containing values to specify the distance between the values which should get mapped. For example, specifying a stride of 1 means that it skips every second value. This is especially useful, for example, when having an array with multiple (x,y) coordinates, in the format: x,y,x,y,x,y, (...). Because different axes should always use different mappings, each axis should get mapped by its own mapping, so one cannot map both axes using the same mapping. If the mapping would not support a stride in values, one would need to split an array containing (x,y) coordinates into two arrays containing only x and only y coordinates. To avoid this, the mapping supports specifying how much distance exists between the values which need to be mapped. Finally, an array can be provided to save the interval indices each value belongs to, or -1 if a value exists between two intervals.

Now to the mapping process itself: first, the last interval used is tried again, for the case that the next value to be mapped can use the same interval as the last mapped value. If the right interval has been found, the value is mapped to that interval. To map a value to an interval, a linear interpolation is done. This works like this because the multi-interval data mapping always maps values linearly (for example, a logarithmic mapping is not available). A linear interpolation is done in our case by knowing the source range of the interval, as well as the intermediate range. Having the value within the source range which we need to map, it is possible to calculate the value within the intermediate range we are searching for. The exact way the linear interpolation works is shown in Code 3. Keep in mind that the method is generally available in Visplore, and not part of the multi-interval data mapping, so the names of the variables are generic.

```
/// Interpolates an unknown Y value between two known Y values,
/// given two X values to interpolate between and the X value to interpolate for.
template <class Tx, class Ty>
Ty InterpolateLinear(const Tx& rXFirstValue, const Tx& rXSecondValue,
   const Ty& rYFirstValue, const Ty& rYSecondValue,
   const Tx& rXValueForInterpolation)
{
   static_assert(std::is_arithmetic<Tx>::value,
      "The␣x␣axis␣of␣the␣linear␣interpolation␣method␣needs␣to␣be␣arithmetic");
   static_assert(std::is_arithmetic<Ty>::value,
      "The␣y␣axis␣of␣the␣linear␣interpolation␣method␣needs␣to␣be␣arithmetic");
   assert(rXSecondValue > rXFirstValue);

   double dWeightToFirstPoint = double(rXValueForInterpolation - rXFirstValue) /
       double(rXSecondValue - rXFirstValue);
   return static_cast<Ty>((1.0 - dWeightToFirstPoint) * rYFirstValue +
       dWeightToFirstPoint * rYSecondValue);
}
```

Code 3: The linear interpolation in Visplore.

If it is not possible to reuse the interval which was last used to map the last value, the next best interval is searched. Remember that the intervals are always sorted in ascending order after being added, so we always know that the intervals with lower coordinates are the first ones, and the ones with higher coordinates are the last ones. Because of this, if it is not possible to reuse the last used interval, if the value is lower than the last interval, the new interval is searched at the left of the previous interval. Otherwise, if it is higher, the new interval is searched at the right of the previous interval. For values which exist before the first or after the last interval, the first, respectively the last interval are used to interpolate - or, in this case, extrapolate - the value. Finally, it can also happen that a value lies between two intervals. In this case, the maximum of the lower interval and the minimum of the higher interval are taken, and the value is also interpolated - but this time, instead of interpolating within an interval, the value gets interpolated to the gap between intervals. This has an actual effect if the gap between intervals is bigger than 0, otherwise the value gets mapped exactly between the two intervals.

6.3 Adapting the 2D Scatter Plot to work with intervals

As previously explained, there are multiple splitting modes provided by the data mapping computation. The value gap mode is probably the most logical for the user, because a user may want to cut out gaps which are, for example, bigger than 1 day, instead of cutting out gaps, for example, bigger than 10%. Usually, the user thinks in absolute values (e.g. days) instead of percentages of the whole range. For the sake of simplicity, we will discuss the percentage gap mode. The value gap mode works very similar, with the only notable difference being that the percentage gap mode first has to calculate the actual maximum gap size out of the percentage, while the value gap mode already has the maximum gap size and can continue right away.

```
std::vector<double> vIntervalMinValues;
std::vector<double> vIntervalMaxValues;

if ((pMultiIntervalMapping != nullptr) && (pChannelType != nullptr) &&
   pChannelType->IsClass(IVLDateTimeType::ClassID()))
{
   IVLStringVector vFlags({BADataMatrixQueryDefinition::FLAG_SORTED_ASC});
   std::shared_ptr<BADataMatrixQueryDefinition> pQueryDefinition =
      BADataMatrixQueryDefinition::Create();
   bool bOk = pQueryDefinition->AppendResultColumnQueryFromChannel(
      pChannel, pChannel->GetName(), &vFlags);

   if (bOk)
   {
      // create a new query and a result matrix
```

```
    auto pQuery = BADataMatrixQuery::Create(pChannel->GetTable(),
        pQueryDefinition);
    std::unique_ptr<IVLInMemoryDataMatrix> pResultMatrix =
        std::unique_ptr<IVLInMemoryDataMatrix>(IVLInMemoryDataMatrix::Create());

    bThreadContinues =
        pQuery->Execute(pResultMatrix.get(), nullptr, pFilterMask, nullptr,
            pThread);

    if (bThreadContinues && pResultMatrix->GetNumRows() > 1)
    {
        unsigned int nNumRows = pResultMatrix->GetNumRows();

        auto * pChannelValues = new double[nNumRows];
        bOk = pResultMatrix->GetAsDoubleArray(pChannelValues, 0, nNumRows);
        assert(bOk);

        // Split the data into intervals, when the gap is higher than this value
        // (e.g.when relatively splitting, this is in percent of the whole
            range,
        // between 0.0 and 1.0).
        double dMaximumGapSize = (m_dPercentageGap / 100.0) * dTotalRange;
        double dAverageDistance = 0.0;

        if ((pChannelValues[0] - dContextMin) > dMaximumGapSize)
            vIntervalMinValues.push_back(pChannelValues[0]);
        else
            vIntervalMinValues.push_back(dContextMin);

        for (unsigned int n = 0; n < nNumRows - 1; ++n)
        {
            if ((pChannelValues[n + 1] - pChannelValues[n]) > dMaximumGapSize)
            {
                vIntervalMaxValues.push_back(pChannelValues[n]);
                vIntervalMinValues.push_back(pChannelValues[n + 1]);
            }

            dAverageDistance = dAverageDistance +
                ((pChannelValues[n + 1] - pChannelValues[n]) - dAverageDistance) /
                (n + 1);
        }

        if ((dContextMax - pChannelValues[nNumRows - 1]) > dMaximumGapSize)
            vIntervalMaxValues.push_back(pChannelValues[nNumRows - 1]);
        else
            vIntervalMaxValues.push_back(dContextMax);

        size_t nNumIntervals = vIntervalMinValues.size();
        for (size_t n = 0; n < nNumIntervals; ++n)
        {
            if (vIntervalMinValues[n] == vIntervalMaxValues[n])
            {
               vIntervalMinValues[n] -= (dAverageDistance / 2.0);
               vIntervalMaxValues[n] += (dAverageDistance / 2.0);
            }
         }

         delete[] pChannelValues;
      }
   }
}
```

Code 4: Calculating the intervals in the 2D Scatter Plot in the percentage gap mode.

Code 4 shows exactly how the percentage gap computation works. First, a query is being prepared. This is needed because not all entries should always be used - some of them could either be missing or filtered out. Because of this, the data matrix query gets a mask (called pFilterMask in Code 4) which specifies which values have been selected by the user. Furthermore, the data matrix query filters out the missing values, so the result only contains valid values. Also, for performance reasons, it is better to sort the values, which is something the data matrix query does as well. The result is an array of double values, which are our coordinates of the axis where intervals should be created (currently, only the X axis).

After this, the actual gap size is calculated from the percentage. Because the percentage is shown to the user as part of the UI, it is saved as a number between [0.0, 100.0], so the user can understand it with ease. However, internally we need it as a number between [0.0, 1.0], so it gets divided by 100. Finally, it is multiplied with the total range, to get the absolute gap value. This is then used for the actual computation: gaps between two values which are bigger than this value get split into intervals.

The exact way the intervals are split has already been explained in 5.4.1. Shortly explained, there is an expected range - in Code 4 called dContextMin and dContextMax - which is the range the data is expected to be within. The exact way this is calculated is more complicated, but without going into too much detail, it is generally the minimum and maximum values within one axis, plus some space at the beginning and at the end - which is done to improve the usability, so that the first and last data points displayed are not at the very beginning and very end of the graph, but some space is added for aesthetic purposes. This preferred range also contains missing values, but only the values which have been selected by the user. Because of the missing values it can happen that the preferred minimum and maximum may have a gap to the nearest points which is big enough and could be cut out.

So, if a gap big enough has been found either between the preferred minimum and the first data point, or between any of the data points, or between the last data point and the preferred maximum, intervals are being created. However, a workaround for one issue in the multi-interval data mapping had to be found. It may sometimes happen that exactly one data point exists in one interval, because the gaps before and after that point are too big. This results in an interval where the interval minimum is the same as the interval maximum. This is, however, currently a problem with the multi-interval data mapping, when we get to the point that the values get mapped (which happens after the intervals are created). Looking at Code 3 will clarify why this causes problems. In our case, calculating dWeightToFirstPoint means we need to divide by dIntervalSourceMaximum minus dIntervalSourceMinimum (variable names have been adapted for better understanding). The problem is that, if the maximum and minimum of an interval are the same, we have a division by 0, which brings us to wrong results. At the moment, a workaround has been applied: an average distance between points is calculated and, if an interval has exactly one value so the same minimum and maximum, the interval is made a bit larger to avoid a division by zero, which results is slightly wrong results, but are, however, good enough to use.

6.4 Adapting the Horizon Graph View to work with intervals

To compute bins very fast, the pixel binning has originally been created with performance in mind. Because of that, only one range could be used at once, so defining another range would overwrite the old range. This was problematic because, to support multiple intervals, the pixel binning needs to work for multiple intervals at the same time. The simple implementation had, however, a performance advantage: when searching which bin contains a certain value, calculating the right bin was an easy task. Knowing the range (minimum and maximum values) of the bins and the number of bins means that it is possible to calculate the size of one bin, as seen in Code 5. Since we know the value which we need to find the bin for, and the minimum of all bins, as well as the size of each bin, it is possible to calculate which bin contains the searched value. For this to work, all the bins must be continuously created in the same range - and since only one range could be defined until now, and because all bins are created continuously in the specified range, this computation was very fast and efficient.

```
auto nNumBins = m_vBins.size();
//Get the bin size!
double dBinSize = (m_dBinningMax - m_dBinningMin) / nNumBins;

if (dData >= m_dBinningMin && dData <= m_dBinningMax)
{
   //Put the entry into the right bin!
   auto nBinIndex = static_cast<size_t>((dData - m_dBinningMin) / dBinSize);
   if (nBinIndex < nNumBins)
   {
      iBin = static_cast<int>(nBinIndex);
   }
   // put the last entry into the last bin
   else if (nBinIndex == nNumBins)
   {
      iBin = static_cast<int>(nNumBins - 1);
   }
}
```

Code 5: Calculating the bin containing a certain value, without intervals.

Adding support for multiple ranges to the pixel binning means that no bins exist between different ranges, so searching for a value between two ranges cannot return a valid bin. Also, it is possible that some ranges contain no bins at all, e.g. when there are bigger gaps between single values, bins should be created for these values, but if the bin is smaller than 1 pixel (because the view is zoomed out), no bins are created at all. Since doing the same calculation can no longer work as it did above, the right bin needs to be searched within all created bins. This is a much more expensive computation, but some optimizations can be made here as well.

Although the search of the bin containing a certain value could theoretically be done in any order, most of the time, values within the same bin will be searched, and from time to time just going to the next bin will suffice. That means that, instead of doing a full search each time a new value is searched, it is possible to cache the last bin used and either reuse it next time, or go to the next bin. Although this already improves the performance significantly, further improvements can still be easily made. For example, instead of checking every single bin, this strategy can be combined with the original one. Instead of only saving the minimum and maximum of all bins, one could also save the minimum and maximum of each range added to the pixel binning (that is, each interval). This way, instead of searching whether a value exists inside of a bin, the bin ranges (which are the same as the intervals) could be checked instead, which is a much faster operation. So instead of saving the bins themselves, we could save bin ranges, and these bin ranges could contain the actual bins computed within the range. As soon as we know the bin range, the exact bin within that range can be calculated as it was done before multiple ranges were supported, so the calculation is very fast.

Surprisingly, after changing the way the pixel binning works, we started getting precision errors. As seen in Code 5, the minimum and maximum values of the bins are doubles and are casted to a size_t - that is the way it worked before the new changes. The problem was that, while computing the minimum and maximum values contained in a bin, the minimum and the maximum of each bin was calculated based on the values of the previous bin. Although this was very efficient, calculating hundreds of bins also meant that the more bins got calculated, the higher the precision error was - so the last bins had the highest precision error. Because previously a cast has been done, the error was always small enough that it did not matter. However, finding the right bin for a value now meant comparing the value with the minimum and maximum values of a bin, so some values were wrongly assigned to the wrong bin, or were suddenly outside any of the bins. The solution was to compute the minimum and maximum of each bin each time, which was a bit more expensive, but allowed for the best precision.

Another change to support multiple intervals was done at the preparation of the global parameters. Because all pixels were next to each other, because only one range was supported before, the maximum of a bin was always the same as the minimum of the next bin. To use less memory, instead of computing and saving the minimum and maximum of each single bin, only the minimum of all bins was saved, as well as the maximum of the last bin. Because this cannot work while supporting multiple intervals, both the minimum and the maximum are now saved for each bin. Of course, optimizing it further and introducing bin ranges which contain the bins, would also make it possible to only save the minimum of each bin and also the maximum of the last bin, like it was done before.

Also, the way the bins are computed had to be changed for the intervals to be considered. Previously, the bins were computed this way:

- The old bins and the old bin borders were invalidated.
- The new bin borders were computed first - that is, the minimum and maximum values of each bin, without computing the bins themselves. This is needed because each screen pixel contains a bin, so no matter how many values a bin contains, the bin is created anyway. The computation is only done for the visible range, so zooming in creates the same number of bins (because the same amount of pixels are visible), but each of the bins contain less values, creating more detailed bins.
- The bins themselves were computed, considering the values inside the bin.

The way the computation works now is more complex:

- The old bins and the old bin borders are invalidated.
- The ranges the bin borders should be computed for, are computed first. This always happens for the mapped range, which is, as explained above, the visible range. It means that, for example, whenever the user zooms in to further analyze the details of the data, the bins should only be created for the visible part. They need to be recomputed to create more detailed bins when zooming in, and it is only done for the visible range to improve the performance. The total available space in pixels is specified as well, which is used for all bins which will be created. More exactly, it works this way:
  - If there are no intervals at all, the whole visible range is used to create the bins, as it was done before. The whole available space is used for this range.
  - If intervals are available, for each interval, the bins are created. Also, to keep the implementation simpler, bins are also created between intervals, so for n intervals, n bin ranges are being computed, as well as n-1 bin ranges between intervals.
  - For each of the ranges which should be computed, the following is checked:
    - First, the bin borders for this range are only computed if the range is either fully, or at least partially visible (which could happen if the user zoomed in).
    - If the range is fully visible, the whole range is taken into consideration. Otherwise, the actually visible part of that range is computed - so the range can either be fully visible (either when not zooming in, or for ranges which are still fully visible, also when zooming in - which can happen when multiple ranges are visible,

for all visible ranges except the very first and the very last), or partially visible on one of the sides (when zooming in, where multiple ranges are visible, which can only happen to the very first and the very last visible range), or partially visible on both sides (when zooming in, where only one range is visible, but only partially). After the visible part of the range is known, the visible minimum and maximum are mapped to the target range - that way we know the visible part of that particular range (either interval range, or the one between intervals), additionally to the already known whole visible range for the intervals. Knowing that, we can calculate how big the visible part of that one interval (or the space between two of them) is, compared to the whole visible range of all intervals. Also knowing the available pixels, we can calculate how many pixels this range has, which is also the number of bins which need to be computed (because one bin per pixel is created).

        * After this step, we know whether the range is visible or not. If it is, we know its visible minimum and maximum values in the target range, as well as its width in pixels. This information is added to a vector of ranges which is later used to create the actual bins.

    - After this step, we have a vector of all visible ranges which will be used to create the actual bins.

- At this point, we have all the information needed to compute the actual bins. We know the exact ranges needed to compute the bins for, as well as the available space for each of the ranges.

- Before the final computation of the bins, one further computation is needed. For each of the computed ranges, we need to compute the bin borders - that is, the minimum and maximum values of each bin, without computing the bins themselves.

- The last step is the computation of the bins themselves, as explained in Section 3.2.2.

6.5 Interaction possibilities between the 2D Scatter Plot and the Horizon Graph View with intervals

Both the 2D Scatter Plot and the Horizon Graph View are views which allow an easy and user-friendly visualization of the data, and provide many options for doing an advanced analysis of the data, such as selections and filtering, as well as interactions with the displayed selections. However, even more advanced use cases are allowed by linking two or more views with each other. This means that a change in one view is reflected in another view. This is very useful because the main reason why different views exist is to show the same data in different ways, because each visual representation shows different properties of the same data, so one view may be able to highlight some information out of the loaded data, while another view will emphasize other information. The combination of the advantages of different ways of visualizing the same data creates a powerful visualization software. However, for this to work, views must be linked, so a change within a view must be reflected in all other linked views where such a change is relevant.

These features can also be very helpful when comparing different objects with each other, because they might assist with rearranging and repositioning objects so that a comparison is much easier to do (Gleicher et al., 2011). The same can also be applied to intervals.

One example of linked views in Visplore consists of the link between the 2D Scatter Plot and the Horizon Graph View. As explained above, the Horizon Graph View tries to display the data as detailed but as compact as possible. Because of that, the whole range of the data is only shown when it is needed. For example, the user can select one or multiple parts of the data in the 2D Scatter Plot. If the beginning and/or the end of the total data range has not been selected, the Horizon Graph View cuts it out automatically - something that also worked before intervals have been added. Furthermore, big gaps within the selected data are cut out by using intervals, saving even more space. That means that interacting with the 2D Scatter Plot changes the way the Horizon Graph View looks. Compared to the 2D Scatter Plot which always lets some space at the beginning and the end of the graph, the Horizon Graph View always displays the bins from the very beginning to the very end of the graph. One difference between the views is, for example, that a selection in the 2D Scatter Plot makes the Horizon Graph View only create bins for that specific range, without any additional interaction needed by the user. This also means that a selection in the 2D Scatter Plot is sufficient to change the data displayed in the Horizon Graph View, and to create intervals, if needed. Because the new range is a different one than the original one, more intervals may be created, because gaps which were previously too small to be removed may now be large enough, because only a part of the data can be seen. So, in addition to the interactions done in the view itself, it is also possible to interact with one view, while the other view also reflects the changes done by the user - also when intervals are created. On the other hand, selecting a time range in the Horizon Graph View also creates the same selection in the 2D Scatter Plot, but does not add it to the filter, so no intervals are created.

Just because the views are linked it does not mean, however, that the views must be configured the same way. For example, different settings for displaying intervals can be made, so the two views might look very different, but if the intervals are configured the same way, the same intervals are created. This can be seen in Figure 26. For example, both the 2D Scatter Plot and the Horizon Graph View have the time on the X axis. The 2D Scatter Plot has the phase voltage in the Bright County on the Y axis, so the change in the phase voltage over the time can be analyzed. Because specific parts of the data have been selected, intervals have been created to cut out the parts without data. In the Horizon Graph View, the same time period and the same intervals are shown, but the Y axis contains many different data attributes, so a comparison between one important data attribute shown in the 2D Scatter Plot and many other attributes

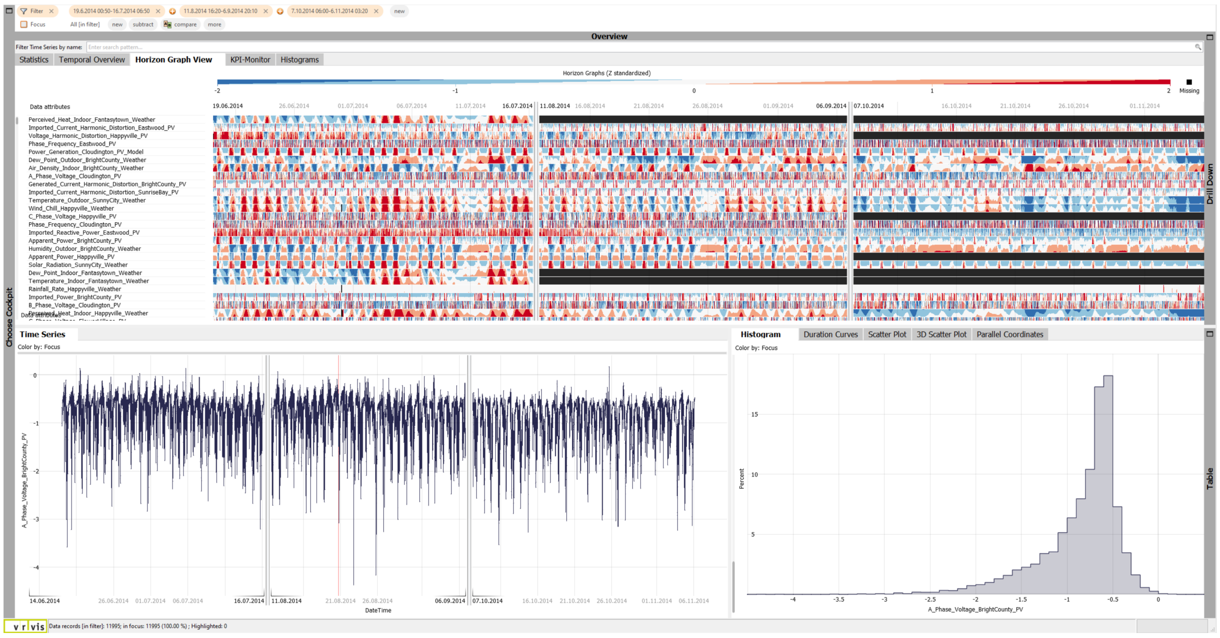


Figure 26: The 2D Scatter Plot and the Horizon Graph View, both containing the same intervals, after a few interactions done by the user. They look similar, because the two views are linked to each other.

shown in the Horizon Graph View is possible. If needed, the same data attribute shown in the 2D Scatter Plot can also be dragged around in the Horizon Graph View to allow a for a closer comparison, by placing one data attribute under the other one.

7 Conclusion

To improve the usability of Visplore, a new comparison technique has been researched as part of this master's thesis. To achieve results as fast as possible, existing research and methods have been studied, including different papers about comparison strategies, as well as actual implementation of such strategies in other software and usability tips to improve the user experience. This was important to see the advantages and disadvantages of each comparison strategy without needing to try out each of them and decide which one works the best for our use case. Also, some papers about Visplore, as well as comparison strategies which are already implemented in Visplore have been analyzed as well, to properly understand what Visplore already supports, as well as what can be improved and why. Although only one new comparison strategy has been implemented as part of this master's thesis, we decided this is the right one for what we wanted to achieve. Other tools may offer other comparison strategies and features, but are out of scope for what Visplore tries to achieve. In other words, the goal was to find the best strategy for our use case, as well as implementing it in such a way that it is useful, understandable and usable, and thus creates value for the users of Visplore.

The paper written by (Gleicher et al., 2011) presents multiple comparison strategies, such as comparing different objects by placing them in juxtaposition (that is, placing them besides each other), or in superposition (that is, placing them over each other), or by doing an explicit encoding. These methods have advantages and disadvantages and can be used for different use cases. For example, Visplore already had the option of displaying superposed curves, which is very useful when having data which is very similar to each other, where only small differences exist. Such a technique also helps in detecting outliers. Also, this can only properly work when the number of objects - curves, in our case - is limited, because displaying thousands of curves will make the visualization hard to understand. On the other hand, such a comparison approach allows to easily compare potentially many curves at the same time, while a juxtaposed comparison would only allow the comparison of a very limited number of curves. In other words, in case of the curves, it made more sense to use a superposed comparison. However, since we wanted to compare potentially very different data, a juxtaposed comparison strategy was more appropriate for what we wanted to achieve, although it needed much more screen space.

There were two reasons why the new comparison strategy was needed in Visplore. On the one hand, the user previously had the possibility to do one or multiple selections in certain views, and then to either filter out the selected data, or to include only the selected data, but then big gaps without any actual values were still shown to the user. This was not only a waste of space, but it was often enough hard to compare different time intervals with each other, because they were far apart, although no data was present in between - so comparison was already possible, but far from ideal. On the other hand, many datasets included data with many missing values. This can happen, for example, when the production in a factory is stopped for some time. Cutting out bigger gaps helped not only by using the space more efficiently, but also by displaying the actual data more detailed, because more space is available for it.

By developing the new comparison technique in Visplore, it is now possible to compare areas of interest with each other by using interval-based comparisons. The new feature uses a juxtaposed (side-by-side) comparison design, which makes it easy to compare data that is not similar, so a superposed (overlaid) design would not properly work. This new feature can help the production industry, but also many others, visualizing not only one exact area of interest, or one area of interest at a time, but simultaneously comparing multiple areas of interest with each other to see differences and potentially find anomalies in the data. Such comparisons can only work when comparing smaller amounts of data with each other. For example, now it is possible to compare the industrial production of two different days by displaying them besides each other, and ignore the rest of the data. This was possible by creating a new multi-interval data mapping, which compresses the unimportant (empty) areas and expands the areas of interest.

Additionally, the new multi-interval data mapping had to be integrated in the existing software, and in two views of Visplore: the 2D Scatter Plot and the Horizon Graph View. In order to make many of the existing classes compatible with both implementations, an interface which both the old and the new mapping derive from had to be created and used instead of the actual data mapping implementation which was used before. Of course, this can only be done whenever the same behavior is expected for both implementations, because otherwise one may need to differentiate between the two implementations.

Although usability was partially an issue with the newly introduced feature, it had to be solved in a way that the user interface remains both intuitive and not overwhelming for the user. Because of this, in addition to the integration in the 2D Scatter Plot and the Horizon Graph View, the 2D Scalar Grid also needed some adjustments to display the beginning and the end of each interval, as well as separation lines between them, so that it is clear when an interval begins and ends. Also, for proper support of multiple ranges, the Pixel Binning also needed to be adjusted to support the newly added intervals from the Horizon Graph View.

Some implementation details as well as certain programming difficulties and their solution have been presented in the implementation part. Finally, the way the two views can interact with each other has also been explained. Although this is not directly something new, adding intervals to both views helped with comparing different data attributes in both views as the same time, combining the strength of both views.

In conclusion, implementing our new juxtaposed comparison technique in both the 2D Scatter Plot and the Horizon Graph View improved the usability of Visplore, saved screen space, and allowed the users to compare time periods which are far away from each other. For example, comparing the Christmas sales from two or more years is much easier than before. All of this has been solved by designing the new features to be as simple, flexible and intuitive as possible.

> “... with proper design, the features come cheaply. This approach is arduous, but continues to succeed.” - Dennis Ritchie (Ritchie, n.d.)

Bibliography

Arbesser, C., Rafelsberger, O. & Piringer, H., 2014. The focus-filter widget: A versatile control for defining spatial focus + context in 1d. In: *Proceedings of IEEE InfoVis 2014*.

Fink, M., Haunert, J., Spoerhase, J. & Wolff, A., 2013. Selecting the aspect ratio of a scatter plot based on its delaunay triangulation. *IEEE Transactions on Visualization and Computer Graphics*, 19(12), pp.2326–2335.

Gleicher, M., Albers, D., Walker, R., Jusufi, I., Hansen, C. D. & Roberts, J. C., 2011. Visual comparison for information visualization. *Information Visualization*, 10(4), pp.289–309. Available at: <https://doi.org/10.1177/1473871611416549> [Accessed February 21, 2020].

Javed, W. & Elmqvist, N., 2010. Stack zooming for multi-focus interaction in time-series data visualization. In: *2010 IEEE Pacific Visualization Symposium (PacificVis)*, pp.33–40.

Keim, D. A., Mansmann, F., Stoffel, A. & Ziegler, H., 2009. *Visual Analytics*. Boston, MA: Springer US. Available at: <https://doi.org/10.1007/978-0-387-39940-9_1122> [Accessed November 17, 2019].

Novotny, M. & Hauser, H., 2006. Outlier-preserving focus+context visualization in parallel coordinates. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), pp.893–900.

Reijner, H. & Panopticon Software, 2008. The development of the horizon graph. *Vis08 Workshop From Theory to Practice: Design, Vision and Visualization*, .

Ritchie, D., n.d.. *Dennis Ritchie quote*. Available at: <http://www.azquotes.com/quote/673778> [Accessed August 30, 2020].

UK Power Networks, 2011. *Photovoltaic (PV) Solar Panel Energy Generation data*. [Online] Available at: <https://data.london.gov.uk/dataset/photovoltaic--pv--solar-panel-energy-generation-data> [Accessed March 27, 2020].

VRVis, n.d.*a*. *Visplore - Technology for custom analysis solutions*. Available at: <https://www.vrvis.at/site/assets/files/3713/vrvis_visplore_en.pdf> [Accessed November 17, 2019].

VRVis, n.d.*b*. *Visplore project presentation*. Available at: <https://www.vrvis.at/research/projects/visplore/> [Accessed April 2, 2020].

List of Figures

List of Code

List of Abbreviations

API	Application Programming Interface
CSV	Comma-Separated Values
GPU	Graphics Processing Unit
GTK	GIMP Toolkit
GUI	Graphical User Interface
OpenGL	Open Graphics Library
UI	User Interface
Visplore	VISual exPLORation

A Appendix A

License Statement for Photovoltaic and Weather dataset:

"Contains public sector information licensed under the Open Government Licence v3.0."

Source of Dataset (in its original form): https://data.london.gov.uk/dataset/photovoltaic–pv–solar-panel-energy-generation-data

License: UK Open Government Licence OGL 3: http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/

———

Dataset was modified to fit the needs and purposes of demonstrating USPs of the Visplore software

- geographic places were anonymized,
- time rasters were changed and synchronized,
- a few data quality issues were introduced for showcase purposes
- technical column names were modified for simpler explanation + as not to confuse at first glance with original, unmodified dataset.