FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Töpfer

Components for visualization of correlations for IVIS framework

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	prof. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2020

I would like to thank prof. RNDr. Tomáš Bureš, Ph.D. for supervising this thesis and giving me advise. I would also like to express gratitude to my family and friends who supported me during my studies.

Title: Components for visualization of correlations for IVIS framework

Author: Michal Töpfer

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: As the number of IoT devices connected to the internet grows, the amounts of data which need to be analysed and visualized also increase. One of the frameworks for creating complex configurable visualizations is IVIS, a web-based open-source framework developed at D3S, MFF UK.

In this thesis, we develop and implement components for scatter plot, bubble plot, heatmap chart and histogram chart, which did not exist previously in the framework. These components can be used to visualize correlations among data and to display properties of data distribution.

Special emphasis is given to interactivity and configurability of components and a detailed description of the configuration options is provided. We also create a set of examples to show how to use the newly added components together with existing parts of the framework. Existing charts in the framework are also enhanced with the newly introduced concepts.

Keywords: data visualization, scatter plot, bubble plot, histogram, heatmap

Název práce: Komponenty pro vizualizaci závislostí pro framework IVIS

Autor: Michal Töpfer

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: prof. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: S rostoucím počtem zařízení připojených k internetu věcí roste i množství dat, které je potřeba analyzovat a prohlížet. Jedním z frameworků pro tvorbu všestranných a konfigurovatelných vizualizací je IVIS, který je vyvíjen na D3S, MFF UK.

Cílem této práce je vyvinout a implementovat pro IVIS komponenty pro korelační diagram (XY bodový graf), bublinový graf, histogram a 2D histogram. Tyto komponenty se dají použít pro vizualizaci korelací v datech a znázornění distribuce dat.

Všechny komponenty jsou interaktivní a snadno nastavitelné, přičemž možná nastavení jsou popsána v textu práce. Použití komponent je ukázáno na několika praktických příkladech, které mimo jiné demonstrují, jak lze komponenty provázat s už existujícími částmi IVISu. Nově použité koncepty jsou také doplněny do už existujících typů grafů.

Klíčová slova: vizualizace dat, korelační diagram, bublinový graf, histogram, 2D histogram

Contents

1. Introduction

As the number of devices connected to the internet grows, the amounts of data which need to be analysed also increase. We can see this trend especially with the IoT devices, cheap sensors of all kinds connected to the internet. These devices can be set up to monitor almost anything, for example temperature, pressure, movement and chemical reactions. In the field of Industrial IoT, the sensors can be set up to monitor the manufacturing process and the data from them can be analysed in real time.

When analysing data, visualizations are an indispensable tool for understanding patterns, trends and outliers in them. Charts and plots are the most common ways to create visualizations. Picking the right chart for the data is one of the key decisions of creating a useful visualization.

There are many frameworks for creating complex configurable visualizations of data. Some of the most known frameworks are Kibana and Grafana, which are also mentioned later in this thesis. Another visualization framework is IVIS, which we focus on here. This web-based open-source framework can be used to create and display highly customized visualizations in a web browser. The IoT sensors can also be connected to send the data directly to the framework.

1.1 Problem statement

IVIS framework provides components for some of the basic chart types, for example line chart, area chart and pie chart. However, it lacks charts for deeper analytics insights such as correlations among data and displaying properties of data distribution.

For the displaying of correlations among data, scatter plot, bubble plot, heatmap chart and other types of charts can be used. These are completely missing in the framework.

Regarding visualizations of data distribution properties, the framework provides a basic histogram chart component. However, this component can be enhanced to offer more user interactions and configuration options. The histogram can only display distribution of numeric signals, there are no components for showing distributions of categorical data in the framework.

1.2 Goals

The goal of this thesis is to implement components for statistical charts for IVIS framework, mainly focusing on correlation of two or more variables and visualizing data distribution properties. In particular, the thesis will provide components for histogram, scatter plot, bubble plot, heatmap chart (histogram of two variables), and bar chart.

The development of these components includes analysis of their possible uses in order to provide useful configuration options, so that they can be easily used as a part of complex visualizations of data. The usability of the components will be evaluated by creating example visualizations.

Special emphasis is given to interactivity of the plots, such as the ability to pan and zoom in the chart. The newly introduced concepts will also be added to the existing charts in order to provide unified user experience. For example, the navigation in a line chart will be simplified by implementing the pan and zoom controls introduced in the newly implemented charts.

1.3 Structure of the text

First, we briefly describe the IVIS framework and its technological background in Chapter 2.

Chapter 3 of this thesis focuses on examples of use cases in data visualizations, which currently cannot be created in IVIS framework. We focus mainly on visualizations of correlations among data and visualizations of properties of distributions of the data. For each of the use cases, we propose charts which can be used to create the visualization. We elaborate more on the proposed charts in Chapter 4, describing possible configurations and user interactions with them.

Details about the architecture of the project and implementation of the new components are described in Chapter 5 and structure of the source code is outlined in Appendix B.

For IVIS admins and template creators, we provide a detailed description of the API of the newly created components in Chapter 6. Then, in Chapter 7, examples of the usage of these components are given. In Appendix D, we present the source code for these examples.

Chapter 8 focuses on other frameworks which can be used for data visualizing and on their comparison with IVIS. Namely, we compare IVIS with Grafana, Kibana and InfluxDB.

For users who use the visualizations to look at data, a guide to the possible interactions with the charts is given in Appendix C.

2. Technological background

2.1 IVIS framework

IVIS is a open-source web-based data processing and visualization framework that provides components to setup highly customized visualizations. Visualizations can be created and then viewed from a web browser. The framework offers visualization components (different types of charts) and other core functionalities at the client and the server side. IVIS can be tailored for domain-specific applications through extensions. More information can be found on the project's website [1] and in *IVIS: Highly customizable framework for visualization and processing of IoT data* [2].

2.1.1 IVIS concepts

This section provides a general description of basic IVIS concepts needed for this thesis. For more information including permissions system, please refer to the project's website [1, CONCEPTS.md].

Namespace

Namespaces are the mechanism to manage the organizational structure of the whole project. They can be used to group entities (panels, sensors, templates, ...) together. The user can then be granted permissions for all entities in particular namespace. For example, we can give a user the permission to view all panels in a given namespace. The permissions can also be granted for each entity separately.

Workspace and panels

Workspace can contain several panels and its purpose is to present visualizations and group related panels together. Each panel contains one visualization, which can consist of several charts and other UI elements.

SignalSet (Sensor)

SignalSet typically contains data from one source. Each signalSet is a dataset which groups signals together. The data are saved in form of records (data points, observations), each of which contains values for the signals in the signalSet. One of the signals in a signalSet is usually the time of the observation, so the data can be treated as a time series.

Signal

Signals are the columns in the dataset. IVIS supports different types of signals including numbers (both integers and floating point numbers), strings and time stamps.

Template

Templates are a method to create custom reusable visualizations in IVIS framework. They can be parametrized and then displayed in panels. Each template can be used repeatedly with a different set of parameters.

Each template consists of JavaScript code (JSX), styles (SCSS), parameters and possibly other files. The JSX code is then compiled as a React[1] component and rendered inside a panel.

More information on template creation can be found in Appendix D and examples of templates will be shown in Chapter 7.

2.1.2 IVIS installation

The installation procedure for CentOS 7 and Ubuntu 18.04 LTS operating systems is described in the official documentation [1, README.md]. To install exactly the version of the framework described in this thesis, the third step of the installation (*Download IVIS*) must be altered as follows. All changes related to this thesis are also merged into the `devel` branch in the SmartArch repository [1].

```
cd /opt
git clone https://github.com/mnaukal/ivis-core.git
cd ivis-core
git checkout --track origin/correlation_charts
```

2.1.3 Technologies used in IVIS

IVIS framework is built on top of several technologies from the JavaScript ecosystem. The framework itself is mostly written in the modern version of ECMAScript (standardized version of JavaScript), which introduces better variable declaration, classes, and many other useful functionalities.[2]

Following technologies are used in the framework [1]:

- Frontend:
  - User Interface: React
  - User Interface Design: Bootstrap
  - Visualizations: D3.js
- Backend:
  - IVIS routes, and API: Node.js
  - Database: MySQL
  - Indexing & Searching: Elasticsearch
  - Security: Passport

As the goal of this thesis is to implement components for the framework, it uses the same core technologies. However, not all of them were directly needed during the development. The rest of this chapter describes the most important technologies for this thesis in more detail.

[1]React and JSX will be discussed later in this chapter in Section 2.3.
[2]Engelschall [3] describes ECMAScript features in more detail.

2.2 D3.js

As described on the official website [4]:

> D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

The D3.js library allows to bind data (often from JS array) to the Document Object Model (DOM) and create new HTML or SVG elements based on the items. It also efficiently handles modification of the source data and updates the corresponding elements (or adds new or remove unnecessary ones if needed).

The elements can be modified in various ways. The most used technique in this thesis is modification of attributes of the SVG elements using the `attr` function. When calling the `attr` function, a function can be specified, which will be evaluated for each item and the corresponding element's attribute will be set to the returned value. Similarly, styles, inner text and inner HTML of the elements can be also set.

On top of all this, D3.js contains a large library of modules for specific purposes, which can be utilized during development. The most important modules for this project are

d3-array	functions for array manipulation,
d3-scale	mapping a domain to a range (for example using linear scale),
d3-axis	chart axes with marks in human readable form,
d3-zoom	pan and zoom using mouse or touch input,
d3-brush	region selection using mouse or touch,
d3-transition	animated transitions of data modification.

2.3 React

React is a JavaScript library for building user interfaces, which is developed and used by Facebook. It is component-based, which means that it encourages the developers to create encapsulated components that manage their own state and logic. The components are then combined together to create complex user interfaces. More information can be found in the official documentation [5].

Each React component can have properties (`props`), which are set from outside of the component, usually by a parent component. It can also manage its own state (in the `state` variable set by the `setState` method) and then use the values saved in the state for example as the properties to a child component. The library takes care of changes of props and state and updates the necessary parts of DOM and calls callback functions (`componentDidUpdate` method), so the component's logic can react to the changes.

Components in IVIS-CORE project are created as React components.

2.3.1 JSX

It is common to use JSX syntax in React. It is an extension to JavaScript syntax which allows writing HTML-like tags directly inside JS code without using any quotation marks or other notation. As an example, consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

It produces a React element, which can later be rendered to the DOM. React components usually return JSX from their `render` method. The tags used in JSX can be HTML tags, but also other React components.

2.4 Elasticsearch

Elasticsearch is a distributed search and analytics engine. It can efficiently store and index all types of data in a way that supports fast searches. Apart from simple data retrieval, it can also perform complex aggregations over the data to discover trends and patterns or to summarize the data.

Specifically for this project, histogram aggregation and the ability to return random samples of the dataset are key features of Elasticsearch. Another feature important for IVIS-CORE framework is fast filtering of time series data (selecting data indexed in time order based on a range of time). More about all features of Elasticsearch can be found in official documentation [6].

3. Analysis

While IVIS has some components for visualizing time-series data, predominantly in the form of a line chart, it lacks visualization components for deeper analytics insights such as correlations among data. This chapter focuses on common use cases in data visualization, which cannot be created using only the components currently available in IVIS framework.

For these example use cases, we propose charts which might be used to create such visualizations. Detailed description of the suggested charts and discussion about their implementation as IVIS components is provided in Chapter 4.

Here is a list of visualization use cases we focus on in this chapter with links to the sections, in which they are described in detail, and example figures:

- properties of distribution of discrete data (3.1.1, Figure 3.1),
- properties of distribution of continuous data (3.1.2, Figure 3.2),
- comparing distributions of data (3.1.3, Figure 3.3),
- correlation of two signals (3.2.1, Figure 3.4),
- correlation of three or more signals (3.2.2, Figure 3.5).

3.1 Visualizing data distribution properties

When working with a signal, only knowing its minimum, maximum and mean is usually not enough to determine enough about the data. We often want to know the distribution of the values of the signal. That is, we want to see how frequent are all the possible values of the signal.

For each of the possible values in the range of the signal, we can compute how many times does this value occur in the data. Visualizations of these counts can be done differently based on the type of the data we work with.

3.1.1 Categorical (discrete) data

Discrete data have a limited number of possible values. We can usually treat these values as categories. An example of a discrete data signal would be *current weather condition* with values such as “Sunny”, “Cloudy”, “Rain” and “Snow”.

As there are only finitely many possible outputs, we can compute frequency of each output. Then, we create frequency distribution, which is a list of the possible outputs, each with a count of its occurrences in the data. Instead of count, we can also display the relative count (percentage).

The number of categories is usually small, so we can visualize the frequency distribution using a **pie chart** or a **bar chart**. For higher number of categories with similar values, **lollipop chart** might be more appropriate than a bar chart in order to prevent Moiré effect, as discussed by Holtz [7, Lollipop chart] and in Data Viz Project [8, Lollipop chart]. Section 4.4 of the next chapter gives more information about bar charts and pie charts in IVIS.

A complete example of a visualization of discrete data are results of an election. They are usually presented using a bar chart, sometimes also using a pie chart. Figure 3.1 shows results of elections[1] visualized using bar chart.

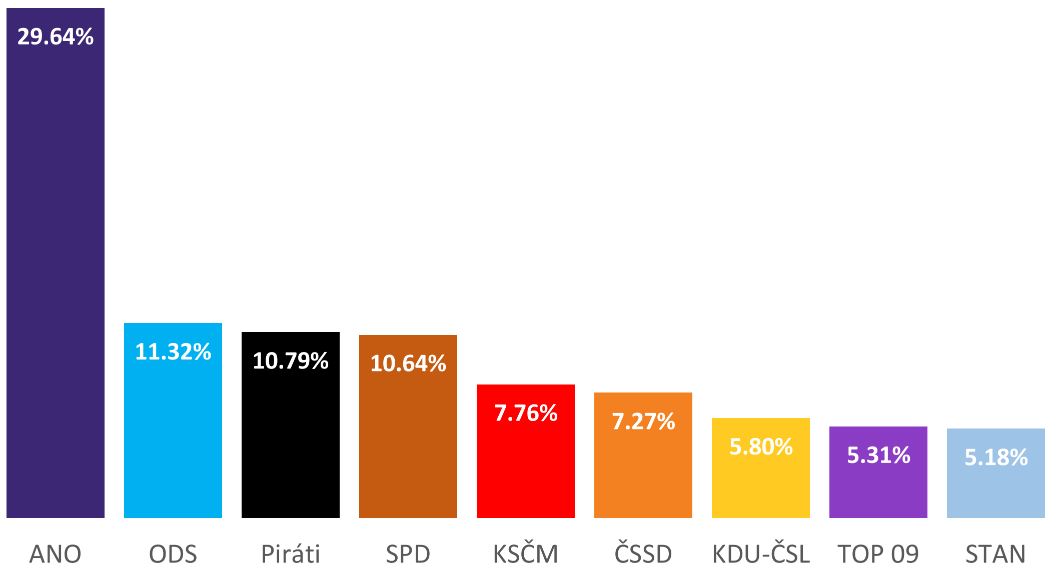

Figure 3.1: Bar chart showing results of elections. Created in Microsoft Excel based on data from Czech Statistical Office.

3.1.2 Numerical (continuous) data

For continuous data, such as real numbers, we usually cannot count each different output separately, because theoretically there can be a infinite number of them. In practice, we only work with finite datasets, but the problem remains. It rarely happens that we have the same value more than once. On the other hand, these values can often be really close to each other so visualizing each of them separately would yield only in a confusing overplotted chart.

Solution to this problem is to group similar values together. Is it usually done by creating disjunct bins (sometimes called *buckets*) across the range of the variable. Then, each sample is counted to the bin it belongs to. Counts in bins are then visualized using a **histogram chart**, i.e. a bar chart without padding around bars and with a continuous x-axis. More details about histogram chart can be found in the next chapter in Section 4.3.

When the bin size is small and the top sides of the bars are smoothed, we call this chart **density plot**. We focused on the histogram chart; implementation of the smoothing for density plot is beyond scope of this thesis.

As an example of visualization of distribution of continuous data, we provide a histogram showing wind speed (red) and generated energy (blue) for all of 2002 at the Lee Ranch facility in Colorado. This histogram is displayed in Figure 3.2.

[1]Elections to the Chamber of Deputies of the Parliament of the Czech Republic held on 20 – 21 October 2017, data downloaded from website of Czech Statistical Office (`https://volby.cz/`), showing only results of parties which got at least 5% of the votes.

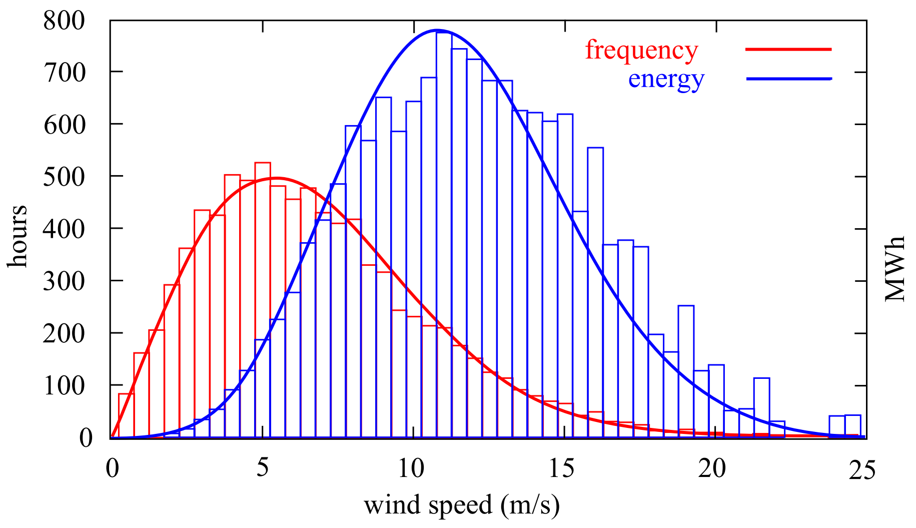

Figure 3.2: Histogram showing distribution of wind speed and generated energy. Source: Wikimedia Commons, User: Saperaud, CC-BY-SA license.

3.1.3 Comparing distributions

One way of comparing distributions of more signals is to plot them into the same histogram, density plot or other chart. This, however, can lead to a overcluttered chart when comparing too many distributions. Holtz [7, Too many distributions] recommends using other types of charts for this, such as **box plot** (shown in Figure 3.3), **violin plot** and **ridgeline plot**, or drawing the distributions in smaller separate charts. These plots are beyond scope of this thesis and we did not implement them. Thus they are not described in more detail in the next chapter. More information about them can be found for example in From Data to Viz [7] and Data Viz Project [8].

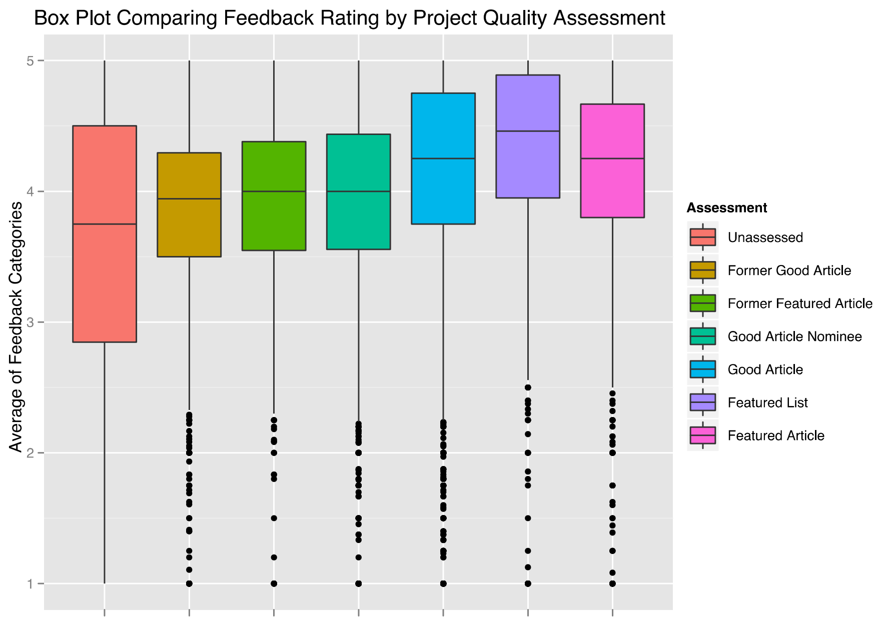

Figure 3.3: Boxplot comparing results of Article Feedback research. Source: Wikimedia Commons, User: Protonk, CC-BY-SA license.

3.2 Visualizing data correlation

Apart from observing distribution of one signal at the time, we might also want to compare two or more signals to see if they are related. That is to tell whether values of one signal depend on values of the other. The statistical relationship between two signals is called correlation.

3.2.1 Two signals

IVIS currently allows to display two lines in a **line chart** at the same time, but this approach has problems when the data do not have the same range. We can use two separate axes in the line chart, but this can be misleading as we can choose the scales arbitrarily. More on this topic can be read in Lisa Charlotte Rost's blog post in Chartable [9]. The line chart in IVIS can also display only time series data (the x-axis is always time), so we cannot visualize correlation of signals without time stamps.

A better way to display correlation of two signals is to use two perpendicular axes, each with its own scale. The most common way to visualize the data is to plot a dot into the chart for each data point. The dot's x-position is defined by one of the signals and its y-position is defined by the other. The resulting chart is called **scatter plot** and we again provide more details about it in Section 4.1.

An example of scatter plot showing data about eruptions of the Old Faithful geyser in Yellowstone National Park, Wyoming, USA is shown in Figure 3.4. We can observe that there is a positive correlation between the waiting time and eruption duration (for longer waiting time, we get longer lasting eruptions).

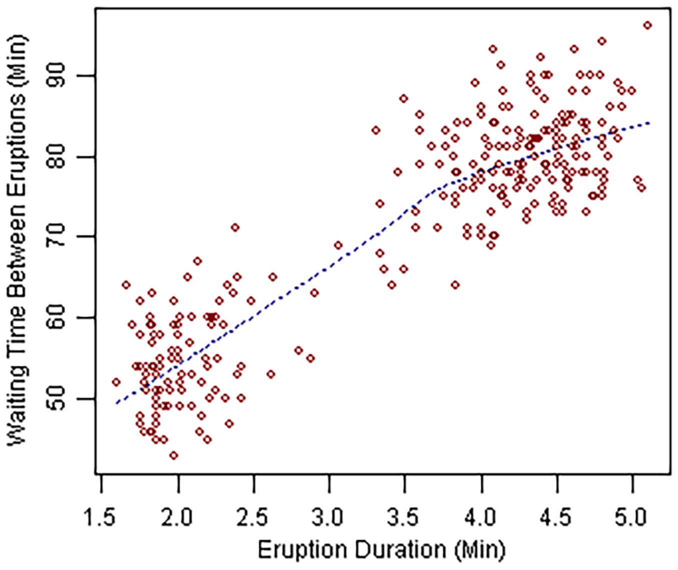

Figure 3.4: Waiting time between eruptions and the duration of the eruption for the Old Faithful geyser in Yellowstone National Park plotted in scatter plot. Source: Wikimedia Commons, User: Maksim, Public Domain.

Another way of using two perpendicular axes is a **heatmap**, which is a two dimensional equivalent of histogram or frequency distribution chart. Bins are created along both of the axes and the counts of occurrences in each bin are represented by different colors. This chart is better for displaying correlations of discrete signals, as there are usually small number of categories and all dots for one category would be on a same vertical or horizontal line in scatter plot. Furthermore, if both of the signals are discrete, then all the dots representing the same values of both signals lie on top of each other and therefore are not visible. More details about heatmap can be found in Section 4.5.

3.2.2 Three or more signals

Sometimes, we want to visualize correlation of three or more signals. This can be done by using a third perpendicular axis in **scatter plot**, but displaying three dimensional chart on two dimensional computer screen can be misleading.

We can add additional dimensions to scatter plot by using color or size of the dots. These approaches can also be combined to create a chart which can display four signals at the same time.

Colors of dots in scatter plot can be used for example to distinguish data points by their category defined by a discrete signal (different color for each category). Numerical signals can also be displayed using continuous color scheme.

Bubble plot is a variant of scatter plot where size of the dot is determined by additional signal. More details can be found in the next chapter in Section 4.2.

Combination of these approaches can be found in Figure 3.5 which shows a bubble plot of life expectancy and income of world nations with dots colored based on the region of the country.

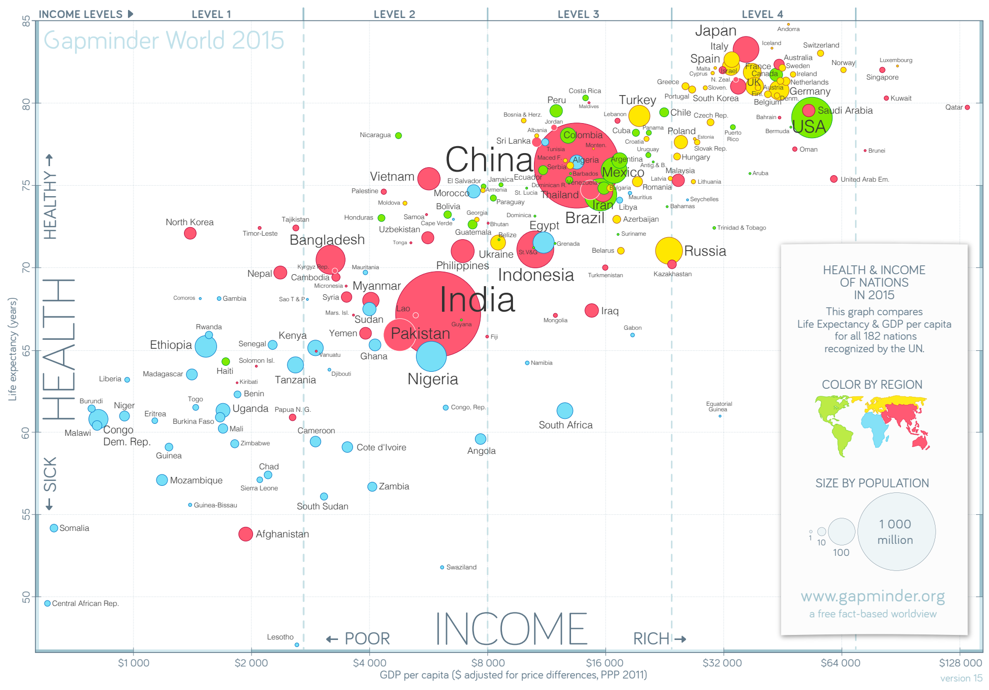

Figure 3.5: Bubble plot showing the life expectancy and income of 182 nations in the year 2015. Each bubble is a country. Size is population. Color is region. Source: Gapminder, CC-BY license.

Completely different way of visualizing correlations of more than two signals is **correlogram**. It consists of several scatter plots, one for each pair of the signals. Correlogram is described in more detail in Section 4.6.

4. Overview of the solution

In this chapter, we elaborate on charts proposed in Chapter 3 by providing a more detailed description of them. We focus only on charts implemented in this thesis. For each chart, possible component configurations and user interactions are reviewed (these are the configurations we considered before implementing the charts, the lists do not aim to be exhaustive). References to Chapter 6, which describes details about the API of implemented components, are also given.

Namely, we talk about the following charts:

- scatter plot (4.1),
- bubble plot (4.2),
- histogram (4.3),
- frequency distribution charts (4.4),
- heatmap (4.5),
- correlogram (4.6).

Additional information about the charts described throughout this chapter more images can be found for example in From Data to Viz [7] and Data Viz Project [8].

4.1 Scatter plot

Scatter plot is a chart which displays the relationship between two numerical signals. Each data point is represented by a dot in the chart. The value of first signal determines the horizontal position of the dot and the second signal determines the vertical position.

Scatter plot was implemented as `ScatterPlot` component as a part of this thesis. See Section 6.3 for details about the API of the component.

4.1.1 Overplotting

One of the problems of scatter plot can be overplotting, which means that there are too many dots in the chart so they start to form clutters and are not distinguishable anymore. Possible solutions are discussed for example by Holtz [7, How to avoid overplotting]. Three of them have been decided to be implemented in scatter plot in this thesis: data sampling, configurable dot size and configurable dot transparency.

The most important method to avoid overplotting is to sample the data, i.e. to show only part of the whole dataset. The scatter plot implemented in this thesis has a configurable limit of the maximum number of data points shown in chart. The data are sampled randomly from the dataset. The scatter plot is also implemented in such a way that the size and color (including transparency) of the dots can be changed.

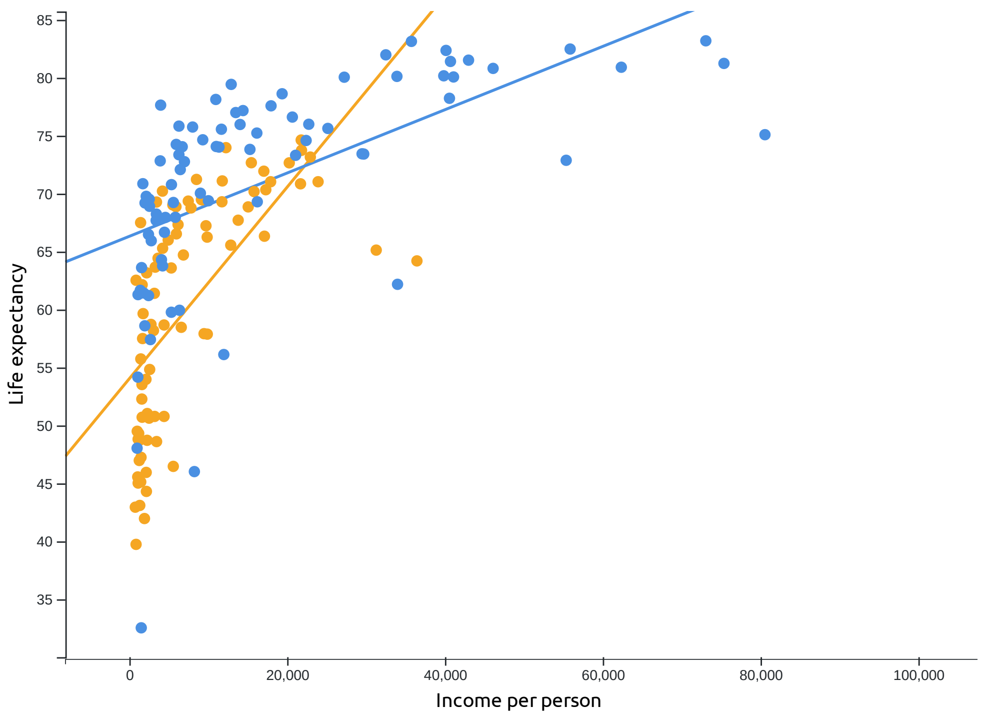

Figure 4.1: Scatter plot.

Another way to overcome overplotting is to use a heatmap chart instead of scatter plot. More details about heatmap chart can be found in Section 4.5 of this chapter.

4.1.2 Possible configurations and extensions

Here comes a list of configurations and extensions of scatter plot which we considered implementing in this thesis. For those really implemented, please see Section 6.3.

Signals

It is clear that we want to be able to configure the signals displayed along the axes. For scatter plot, only numerical signals can be used. Both of the signals also have to be from the same signalSet, so that we always have pairs of values.

The domain of each of the axes can be inferred from the data or set to a specified range.When computing domain from the data, we may either use only the records in the sample plotted in the chart, or we can use the minimum and maximum of the whole dataset (including the records which were not chosen to be displayed). The latter option gives us a better overview of the data, but there might not be any dots in outer part of the rendered chart.

Dots

There are a few possibilities of configuration of the dots displayed in the chart. First of all, their shape can be set by the creator of the visualization. We implemented a few different shapes, which are listed in Section 6.1.2. Another aspect of dots, which should be configurable is the size (radius).

Color

As mentioned earlier, we want the color of dots to be configurable by the creator of the visualization. In the simplest case, all dots will have same color.

The dots can also be colored based on another variable. This can be done both for discrete variable (color for each category) and continuous variable (continuous interpolation between colors).

Regressions

Regressions can be added to see the trends in the data. For instance, linear regression tries to predict how the values along the y-axis depend on values along the x-axis using a linear function. This model can then be rendered as a straight line in the chart. More advanced regression models can also be used.

4.1.3 User interactions

Zoom and selecting a region

When the user wants to see patterns in the data, enlarging a region of the chart might help him. One of the possible interactions is using mouse wheel or touch for zooming in and out and moving the view.

Other way to implement zoom is allow user to draw a region in the chart using his mouse and then display exactly the selected region of the chart. We call the region selection *brush* later in this thesis.

Tooltip

Tooltip with additional information can be shown to the user. It should contain details about the dot which is the closest to the user's cursor.

Time interval selection

IVIS framework contains a `TimeContext` component which allows user to select a specific time range. The data can then be filtered to contain only records within the defined time range. Newly implemented charts should be able to cooperate with this component and filter data based on the selected time interval.

4.2 Bubble plot

Bubble plot is an extension of scatter plot, so all the information written in Section 4.1 are also relevant here. Additionally, Section 6.4 enumerates features implemented in `BubblePlot` component.

Bubble plot allows to visualize an extra signal. The additional signal is represented through the size of the dots.

As written by Holtz [7], the problem with bubble plot is that the relationship between the signals represented by the x- and y-axes is much more obvious than

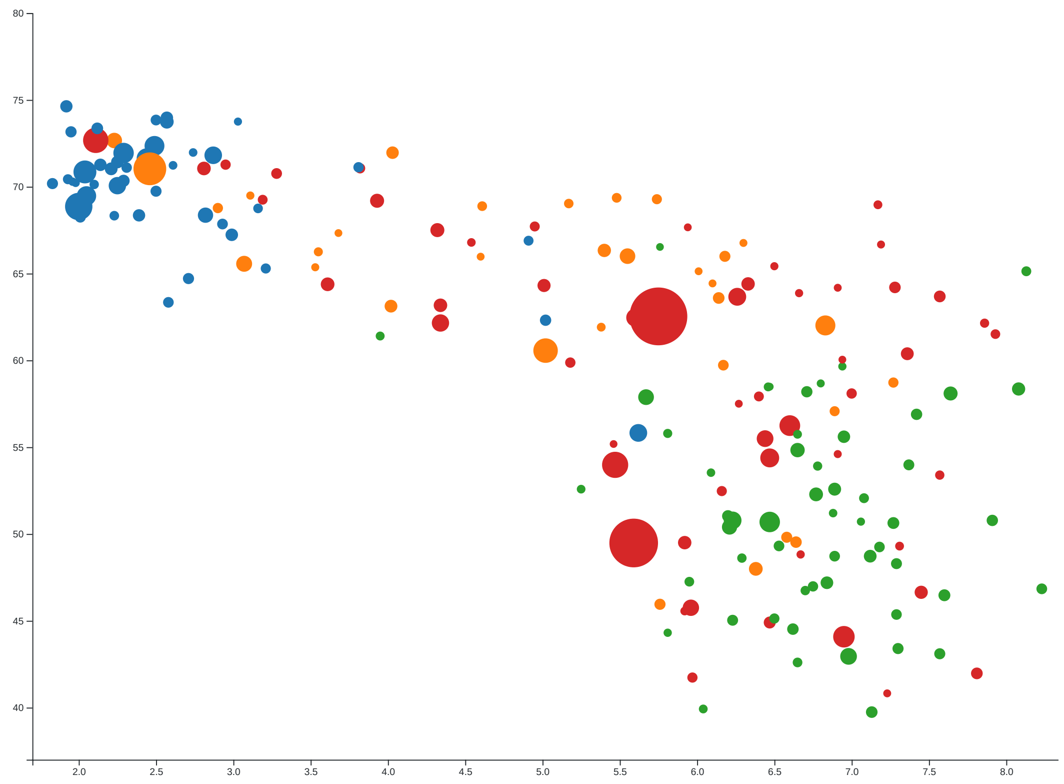

Figure 4.2: Bubble plot.

the relationship with the third signal. It is thus crucial to decide which signal should be represented by which axis.

4.2.1 Bubble size

It is the area of the bubble which is perceived by the user, not its radius. We must thus scale the area according to the signal's value and the radius according to the square root of the value. Example of a bad use of area of a circle can be found in Holtz [7, Scaling to radius or area?].

4.3 Histogram

Histogram is a graphical representation of the distribution of a numerical signal. First, the range of the signal is divided into disjunct bins. Then, we count how many observations fall into each of the bins. These counts are represented by the height of the corresponding bar in the chart.

The `HistogramChart` component existed in IVIS framework, but it lacked some of the features mentioned below. The component's API is described in Section 6.5.

4.3.1 Bin size

The size of the bins can matter a lot as when the bins are too big, we don't see much information about the data. On the other hand, too small bins are hard

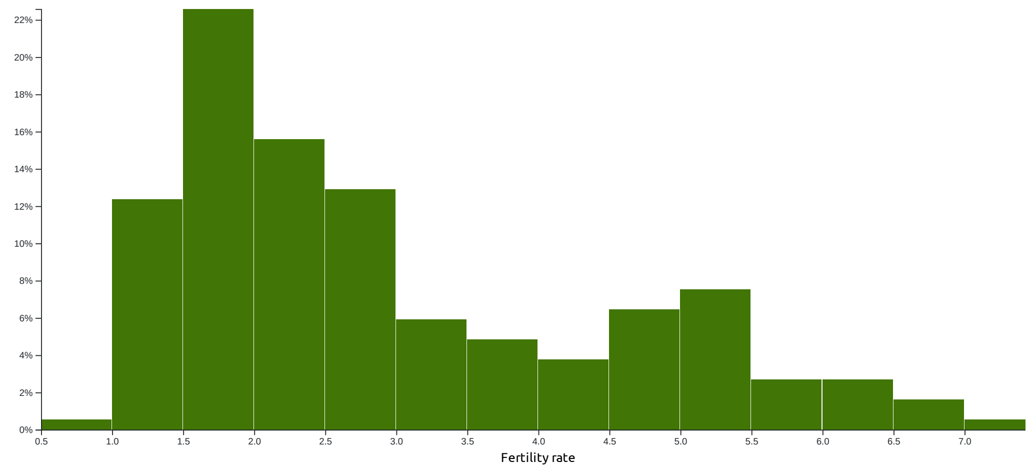

Figure 4.3: Histogram.

to visualize, because the bars representing them are too narrow. Furthermore, when using really small bins, it is quite likely that a lot of them will be empty. The problems with bin size are discussed more for example by Holtz [7, Play with your histogram bin size]. The bin size does not have to be same for all the bins.

4.3.2 Possible configurations and extensions

Here comes a list of possible configurations of the histogram chart considered for this thesis. For those implemented, please see Section 6.5.

Signal

Obviously, we want to be able to choose the signal, whose distribution we want to see. This feature is already available in IVIS framework and it currently allows to show distribution of only one signal at the time.

Bin size

One of the key aspects of creating a good histogram chart is to choose a reasonable bin size. The component currently implemented in IVIS handles this in two ways. It allow the programmer to set the `minStep`, which is the minimal size of the bins (the minimal difference between the endpoints of the bin's interval).

The other approach is to set the minimal width of the bars in pixels. By that, the histogram is always optimized for the current width of the screen.

4.3.3 User interactions

Zoom and selecting a region

Selecting part of the chart to see the distribution of the signal in a certain range can be really useful. It helps for example to filter out outliers. When we have an outlier value, which differs significantly from most of the values, it can increase the range of the variable. Histogram chart will be then rendered for the increased range, which effectively means that the bins will be wider. As a result, the part of the chart containing the outlier will be mostly empty. Furthermore, the interesting part, where most of the data points are, will be compressed to a smaller number of bars and therefore show less information.

Allowing user too zoom in and then reloading more detailed data for the currently visible region of the signals's range solves this problem. We must adapt the bin size to the level of zoom. This feature was missing in the `HistogramChart` component in IVIS and have been implemented as a part of this thesis.

There can be several methods how to enable the zoom interaction for the user. The simplest one is using mouse wheel (or pinch gesture on touchscreen devices) for changing the scale and mouse drag for panning.

Similarly to other components, we also proposed a more precise way of selecting the desired region. We do this by drawing an *overview* below the main histogram. The overview is a smaller histogram which is not changed when the user alters the visible region. Instead, a rectangle is drawn on top of the overview to indicate, which part of the chart is currently visible. The sides of the rectangle have handles which can be used to modify the visible region.

Tooltip

Tooltip with additional information can be showed to the user. It should contain details about the bin below the user's cursor.

Time interval selection

See Section 4.1.3.

4.4 Frequency distribution charts

The aim of these charts is to display distribution of a discrete signal, that is how frequent is each of the possible outputs. As discussed in Section 3.1.1, this is commonly visualized using pie chart or bar chart.

The `StaticPieChart` component already exists in IVIS, so we decided to implement only the loading of the data and then display them using the existing component. `StaticBarChart` component with similar interface to `StaticPieChart` was added as another option to display the frequency distribution.

The components for displaying frequency distribution are called `FrequencyPieChart` and `FrequencyBarChart`. Description of API of these components can be found in Section 6.7.

4.5 Heatmap

Heatmap, sometimes called 2D density plot, is a two dimensional version of the histogram chart. It creates bins for both of the axes and computes number of occurrences of signal values in each rectangle defined by bins. Then, the counts are displayed through color of each rectangle.

Heatmap was implemented as `HeatmapChart` component. Its API is described in Section 6.6.

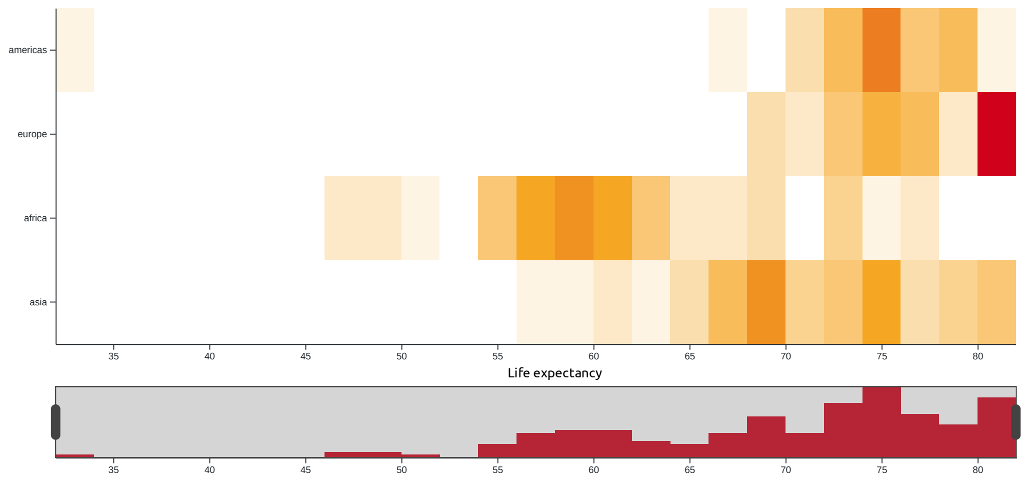

Figure 4.4: Heatmap.

4.5.1 Possible configurations and extensions

Here comes a list of possible configurations of heatmap with short descriptions. For those implemented in this thesis, please see Section 6.6.

Signals

As in other charts, we want be able to configure which signals are shown along each of the axes. This time, the implemented `HeatmapChart` component works with both numerical and discrete signals.

Bins

As in histogram (see 4.3.2), setting a reasonable bin size is crucial to creating good visualization. Size of the bins should be configurable for each axis separately as they don't have to use the same scale. The bins in heatmap can also have different shapes than rectangles. Heatmaps with hexagonal bins are quite common.

Colors

The readability of this type of chart depends heavily on the choice of the colors. It should be clear for the user which bins have the most and the least occurrences. Specifying a color scheme with more than two colors can help that. More on color scheme choices can be found in Chartable [9, What to consider when choosing colors for data visualization].

Marginal histograms

Marginal histograms are histograms placed around the chart parallel to the axes. They show the distribution of values for each signal separately.

4.5.2 User interactions

Zoom and selecting a region

As described in Section 4.3.3, being able to zoom to a specific region of the chart is quite useful. Again, this should be possible for each axis separately for the best user experience. To manage this, we propose adding overviews (as described in the last paragraph of 4.3.3) to both of the axes, serving also as marginal histograms mentioned earlier.

Tooltip

Tooltip with additional information can be showed to the user. It should contain details about the bin (rectangle) below the user's cursor.

Time interval selection

See Section 4.1.3.

4.6 Correlogram

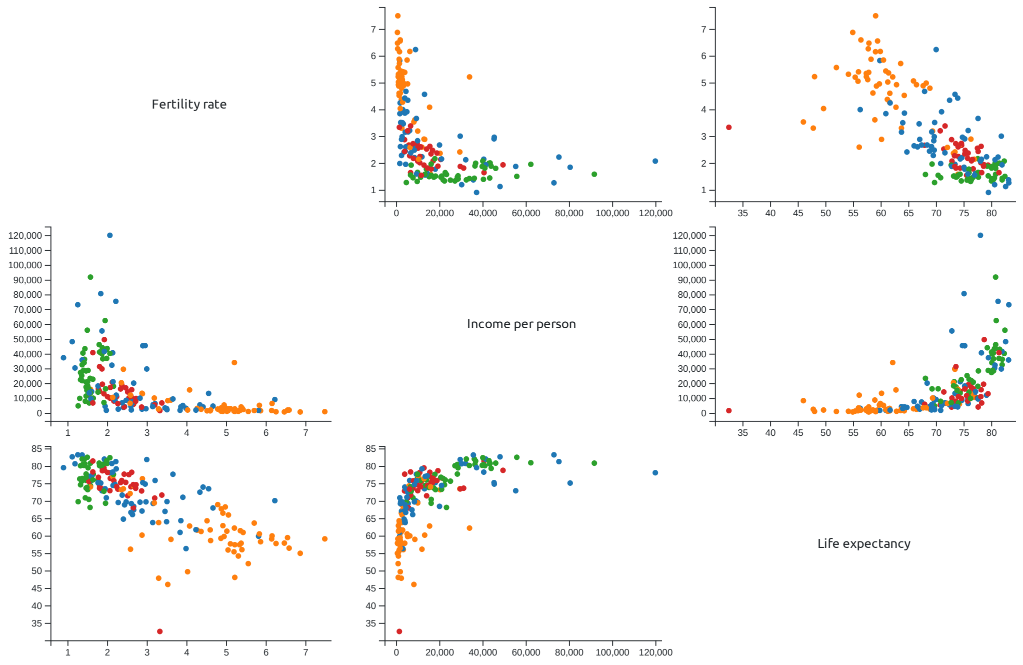

Figure 4.5: Correlogram.

Correlogram (sometimes called *scatter plot matrix*) is a chart which consists of several scatter plots. It can visualize correlation among more than two signals by creating a scatter plot for each pair. For k signals, k rows and k columns are created. The plot located at the intersection of the i-th row and the j-th column renders the i-th signal along its y-axis and the j-th signal along its x-axis. Also, a separate histogram for each of the signals can be added.

Correlogram is not implemented as a component in this thesis, but a simple version of it is shown as one of the examples in Chapter 7 (Section 7.3).

5. Implementation

The following components were implemented as a part of this thesis. API of the most important ones is described in Chapter 6. Structure of the source code files is described in Appendix B.

- `ScatterPlot` (6.3),
- `BubblePlot` (6.4),
- `HistogramChart` (6.5),
- `HeatmapChart` (6.5),
- `FrequencyDataLoader`, `FrequencyBarChart`, `FrequencyPieChart` (6.7),
- `ScatterPlotBase` – common component for `ScatterPlot` and `BubblePlot`,
- `StaticBarChart` – bar chart with data specified directly in `config` (not queried from server),
- `MinMaxLoader` – loads the minimum and the maximum of given signal.

We first provide a brief look at the architecture of the IVIS framework in figure 5.1. In the following sections of this chapter, we focus on main concerns which are common across components and pose a greater technical challenge.

The main interest of this thesis is in creating new components for charts to be used in templates. Figure 5.1 only shows parts of the framework relevant to that.

These newly created components are highlighted as yellow squares labelled "Visualization component" in the figure. Some modifications in other parts of the project were also needed to be made in order for new components to work. For example, random sampling of data for scatter plot (described in Section 5.3) and fetching data distribution for discrete signals (5.4) were added to the Indexer. These modified parts are marked with a yellow dotted border.

Visualization components are used inside a template, which is the code written by visualization creator. The creator can simply use the components as they are, or combine them using additional JavaScript code. Each template can also have parameters, which can be specified when it is used.

Templates are displayed in panels. One template can be displayed in more than one panel, each time with a different set of parameters.

The data for most of the visualization components are retrieved from the server through `DataAccess` component. This components sends the queries to the server, where they are first checked in the Model and then translated to Elasticsearch queries inside the Indexer.

The inner structure and lifecycle of the visualization components is derived from the lifecycle of the components in React. The two most important methods are `fetchData` and `createChart`.

The `fetchData` method loads the data from the server through `DataAccess` component. When the data are loaded, they are stored in React's `state` variable.

Each update of the `state` or the properties causes `componentDidUpdate` method to be called in React. In our visualization components, the `createChart` method is then called to update the chart.

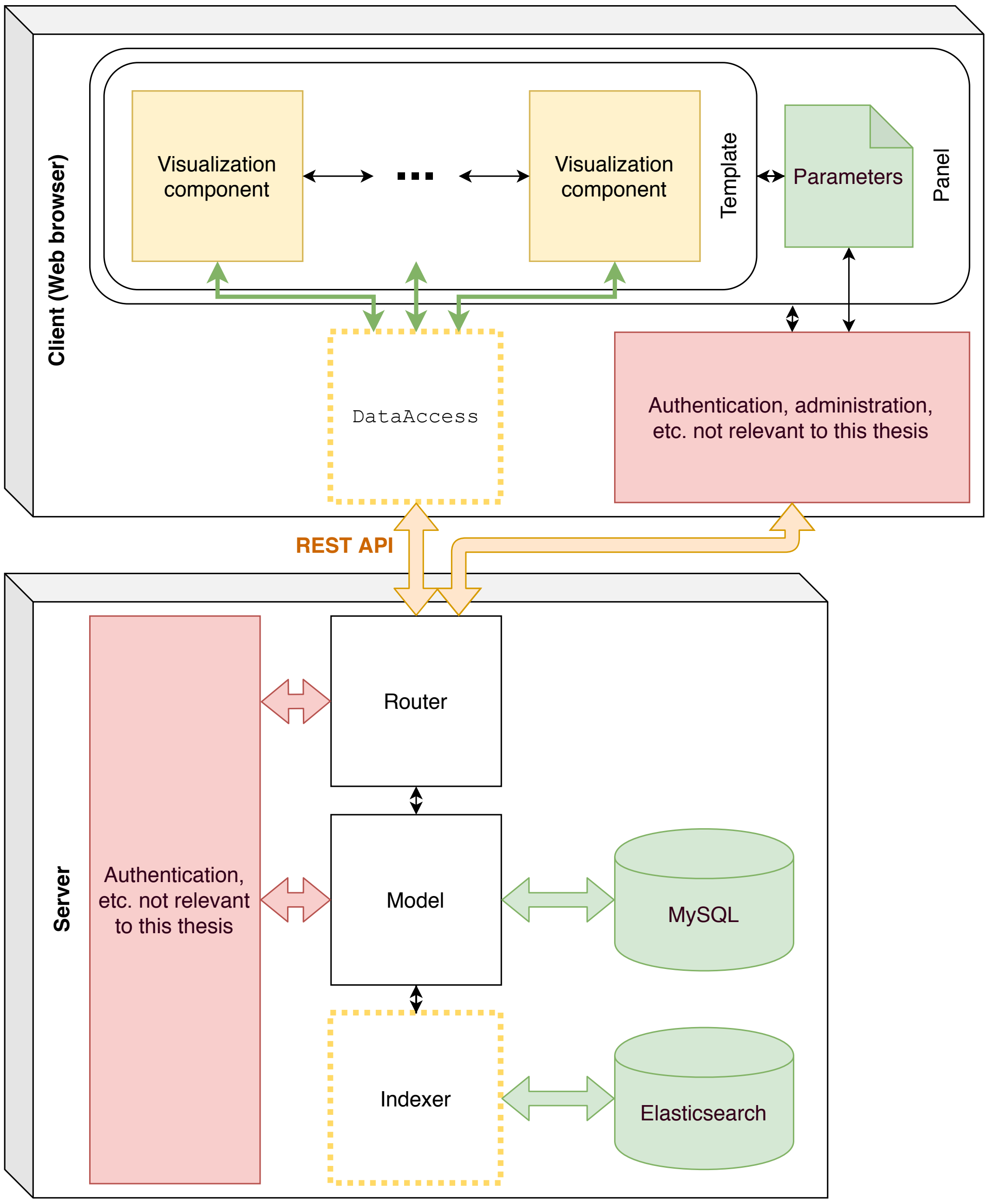

Figure 5.1: IVIS framework architecture.

5.1 Zoom

The zoom in all components is implemented using d3-zoom, which is a part of the D3.js library. It provides tools for both handling the user input (such as mouse wheel, mouse drag and touch gestures) and the manipulation of the data before drawing them. We recommend visiting its documentation [10] if some parts of this section are not clear.

The essential code for zoom is implemented in the `createChartZoom` method in all the charts. We create a `d3Zoom.zoom` object and set all desired extents, which restrict the panning and prevent the user from moving outside of the chart, and event handlers.

When the user changes the zoom, the `zoomTransform` value gets updated in the component's `state`. In this value, the current scale factor and the x and y translation of the visible region are stored. The zoom transform is later used to rescale the axes of the chart. We can imagine this as enlarging the original chart area by the scale factor, then moving in by the translation offsets and cropping it back to the original area so that everything drawn outside is not visible. The rescaling of the axes does exactly that and produces the axes for the transformed region.

Figure 5.2 shows examples of rescaling of the axis. The numbers along the bottom of the squares are the domain of the axis; the numbers along the top of the squares are the range of the axis. The left picture shows the initial state (identity transform). In the middle picture, everything is enlarged by factor 1.5 (the top left corner has coordinates $(0, 0)$, so that is the point which stays at the same place). In the right picture, we also add a translation by 75 to the left, so the final transformation along horizontal axis can be represented by function $t(x) = 1.5 \cdot x - 75$.

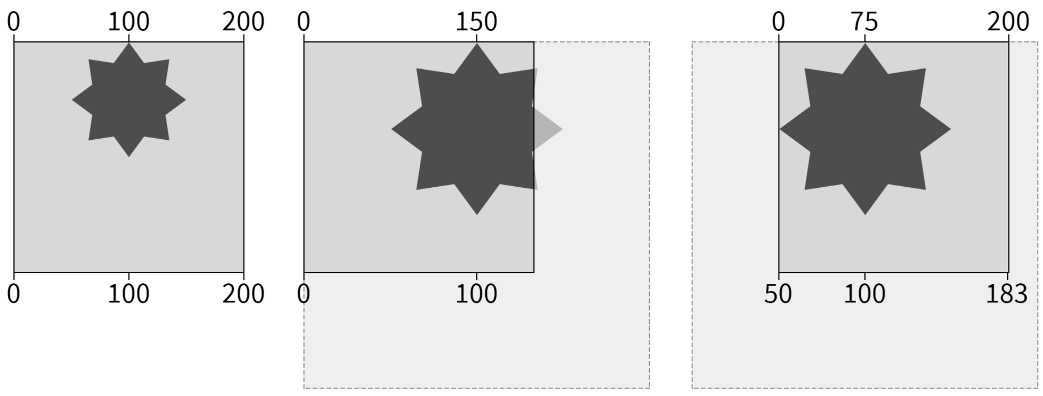

Figure 5.2: Rescaling of the axis.

5.1.1 Two dimensional zoom

The library unfortunately does not support zoom along two axes simultaneously. The `zoomTransform` has only one scale factor value. Our implementation of the two dimensional zoom using the d3-zoom library is described in this section.

Alongside the `zoomTransform`, we also save the `zoomYScaleMultiplier` variable. It can be interpreted as the ratio between zoom factor along the y-axis and the zoom factor along the x-axis. When creating the chart axes, the x-axis is rescaled exactly the same way as for one dimensional zoom. For rescaling the y-axis, the `zoomTransform` is first scaled up by the `zoomYScaleMultiplier`. This creates a zoom transform whose scale factor is the product of the scale factor of the original `zoomTransform` and the `zoomYScaleMultiplier`. We then use the modified zoom transform to rescale the y-axis.

One last thing which is needed to be done is updating the extent so that it allows the user to move up and down along the whole range of the y-axis. This is done by multiplying the vertical size of the `translateExtent` by the `zoomYScaleMultiplier`.

5.1.2 Setting the zoom from code

We use the `setZoomToLimits` method to set the boundaries of the visible region. Setting the zoom transform from code is quite straightforward. We set the scale factor to the ratio between the width of the chart and the width of the desired region. The translate values are then computed so that when the transformation is applied to the top left corner of the desired region, the result is the origin, i.e. the point with coordinates $(0, 0)$.

5.1.3 Zoom and brush

Existing components of the framework were also enhanced with the zoom functionality. We added this to the `TimeBasedChartBase` component, on top of which the line chart and area chart are built.

These components previously used mouse drag for selecting a region using brush (drawing a rectangle with the mouse button held down). The brush was edited to be enabled only when a control key is held. The same behaviour is used in the scatter and bubble plot.

5.2 Regressions in scatter and bubble plot

The statistical regressions, which can be displayed in scatter plot, are implemented using `d3-regression` library. This library was chosen because of its compatibility with the rest of the D3.js framework, especially its ability to produce the regression line in format accepted by the D3.js line generator, which then converts it to an SVG path.

The regressions are computed inside the `createRegression` method in `ScatterPlotBase` when new data are fetched. The computation of regressions might significantly increase the loading time of the chart.

All regression types provided by the library are supported: linear, quadratic, polynomial (with configurable degree of the polynomial), exponential, logarithmic, power law and LOESS regressions.

5.3 Scatter plot data sampling

The `ScatterPlotBase` component has a `maxDotCount` property which limits the maximum number of dots in the chart. This is done in order to reduce the amount of data transferred from the server to the client and to prevent overplotting[1] as there might be thousands of records.

This is implemented by specifying the `function_score` to `random_score` in the Elasticsearch queries. A random number is assigned to each of the records and then those with the highest numbers are retrieved (the number of retrieved records is limited by the `maxDotCount`). We do not specify the seed in the queries, so the fetched records are different every time.

5.4 Fetching data for histogram and frequency distribution charts

When fetching data for histogram, heatmap and frequency distribution charts, the Elasticsearch aggregations are used. For heatmap chart, the aggregations are nested, so that we get two dimensional array of buckets (bins) as the result.

For numerical data, we use the Histogram aggregation. The size of the buckets is computed on the server side but can be influenced using properties listed in Section 6.5.2 for histogram and in Section 6.6.2 for heatmap. This was already present in IVIS before this thesis.

For discrete data (keyword type in IVIS), the Terms aggregation is used. It returns the counts of unique values of the signal. By default, it only returns the 10 most frequent unique values, but the limit can be increased (that is done for example in `FrequencyDataLoader`). The Terms aggregation was added to IVIS as a part of this thesis.

For all types of data, the histogram query can be used from the client. The correct aggregation is determined by the server depending on the type of the signal. Both `HistogramChart` and `HeatmapChart` use the `getLatestHistogram` method from `DataAccessSession` (part of `DataAccess`). The `FrequencyDataLoader` uses a more general `"aggs"` query with aggregation type set to Terms.

The returned values differ slightly for the two types of data, although both contain `buckets` array with the aggregated bins (each with `key` and `count`). For Histogram aggregation, the computed bin size (`step`) is returned. For Terms aggregation the `doc_count_error_upper_bound` and `sum_other_doc_count` values specified in the documentation linked above are also returned.

[1]Overplotting was discussed in Section 4.1.1.

6. Description of components API for template designers

This chapter contains documentation of the components' API for template creators and programmers, who will use them for building visualizations. We chiefly describe the public properties and methods of the components. To see examples of usage of these components, please read Chapter 7.

Throughout this whole chapter, properties marked with * have already been described earlier (with the asterisk linking to the description). Properties marked with † are required and the component won't work properly without them.

Later, properties of the components will be listed. Property descriptions have this format:

property name (*property type*) [default value]

6.1 Common concepts

6.1.1 Colors

When later sections of this chapter talk about colors, the colors created by d3-color library are meant. Here are some examples of how a color can be created using this library:

```
import * as d3Color from "d3-color";

const fromName = d3Color.color("red");
const fromHex = d3Color.color("#ffe000");
const fromRGB = d3Color.rgb(255, 0, 0);
const fromRGBA = d3Color.rgba(255, 0, 0, 0.5);
const fromD3color = d3Color.color(fromName);
```

Usually, the color can also be specified in the component's properties by using only the string as most of the components call `d3Color.color` method inside them.

For more information, see appropriate section of D3.js documentation [10].

6.1.2 Dot shapes

The term *dot* is used in this documentation for visualization of one data point (record), for example in scatter plot. Dot can be represented by a circle or by other shapes. List of currently available shapes in IVIS is shown in figure 6.1. Particular shapes can be addressed by their names.

Using dot shapes in custom components or visualizations

To use dot shapes, import `dotShapes` from the `ivis` package. Complete list of the implemented shapes' names can be found in `dotShapeNames` array, which can be imported from the same package. This array can also be used when checking properties with React's PropTypes by using `PropTypes.oneOf(dotShapeNames)`.

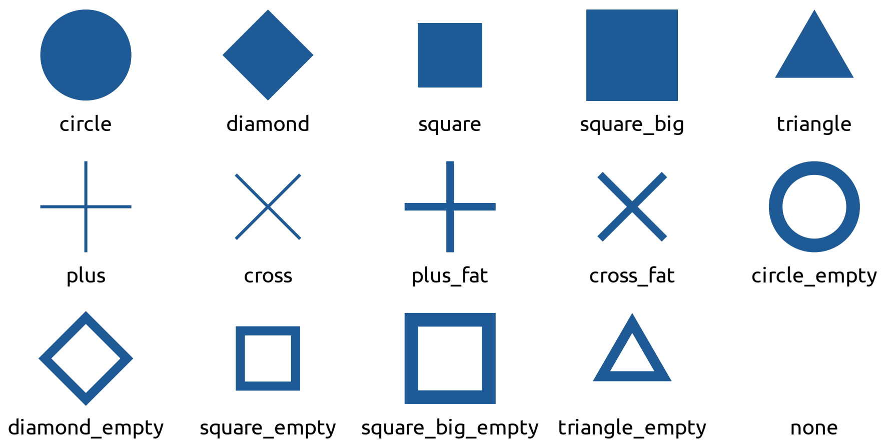

Figure 6.1: Dot shapes.

The shapes are implemented as SVG definitions to be used in `<use>` element. Before use, the dot shapes must be set up by adding the following code to the `<svg>` element. It is a JSX code snippet.

```
<defs>
    {/* dot shape definitions */}
    {dotShapes}
</defs>
```

Then, an `<use>` tag can be added to the same `<svg>` to display desired shape. The `href` attribute of the `<use>` tag must be set to "#" (hash) followed by the shape's name (i.e. one of the elements from `dotShapeNames`), for example "#circle". More information about `<defs>` and `<use>` can be found in MDN Web Docs [11].

All shapes (except "square_big" and "square_big_empty") are created to have radius exactly 1 pixel. To enlarge them to desired size, the `transform` attribute with `scale()` value must be added to the `<use>` tag. We recommended also including `transform-origin` as an inline CSS with the same values as are the x and y position of the `<use>` element. This makes it easier to control the behaviour of scaling.

This example creates a circle with radius 10 pixels on specified x and y position (assuming it is placed inside `<svg>` tag with definitions as described above):

```
<use href="#circle" x="50px" y="100px" transform="scale(10)"
     style="transform-origin: 50px 100px;" />
```

6.2 Common properties

Some of the properties serve the same function in more than one component. Those will be described in this section and later only mentioned in given component's description.

6.2.1 Signal configuration

The most important property for all charts in IVIS is the `config`. It contains information about the signalSets and signals from which the chart loads the data.

It usually contains `signalSetCid` property (identification of the signalSet) and one or more `sigCid` properties (identification of the signal).

The config property might also contain information specific for rendering each of the signalSets, such as colors and labels.

6.2.2 Size and margins

height (*number*)
The height of the chart in pixels.

margin (*object* { top, bottom, left, right : *number* })
Margins of the area in which the chart is displayed, in pixels. The axes of the chart are drawn inside the margins, so setting the margins to zero will hide the axes. When specifying `margin`, all four sides must be set.

The **width** of components is determined by the width of its parent components and the browser window. It can also be modified using CSS (see 6.2.3).

6.2.3 CSS

The `className` and `style` property are available in most of the components with the same functionality as they have in React. See React documentation [5, Styling and CSS] for more information.

className (*string*)
Name of the CSS class to be applied to the component.

style (*object*)
Inline CSS to be applied to the component. The names of the properties are camel cased (instead of standard CSS notation with hyphens).

6.2.4 Enabling features

These properties are boolean values used to enable or disable functionalities in components.

withZoom (*bool*)
Enables zooming and panning by mouse and touch. User can enlarge desired region of the chart.

withBrush (*bool*)
Enables brush. Brush lets user draw a region with their mouse while holding the mouse button down. It invokes an action when the mouse button is released, for example zooming to the selected region.

withTooltip (*bool*)
Displays tooltip with extra information about the data. The tooltip moves with cursor and typically shows information about the data points closest to the cursor.

withCursor (*bool*)
Adds extra visual clues to see where the cursor is. This is usually done by adding extra lines perpendicular to the axes to the chart.

withTransition (*bool*)
Enables animations (smooth transitions) of zoom, movement, data update, etc.

6.2.5 Limits

Setting these props will set the limits (minimum and maximum) of the axes. All the queries done to fetch data will respect these limits. The initial view of the chart will be set to these limits.

xMinValue (*number*)
xMaxValue (*number*)
yMinValue (*number*)
yMaxValue (*number*)

6.2.6 Chart axes

These are properties which affect the axes of the chart.

xAxisTicksCount (*integer*)
Preferred number of ticks (reference marks) to be displayed along x-axis. The real number of rendered ticks might be slightly different in order to make them more readable. This prop will be passed to `d3-axis.ticks`.

xAxisTicksFormat (*function*)
Formatter function for the x-axis ticks. It will receive each tick value and the value it returns will be rendered as reference mark. This prop will be passed to `d3-axis.tickFormat`.

xAxisLabel (*string*)
Name of the x-axis signal to be displayed under the chart.

yAxisTicksCount (*integer*)
Analogous to `xAxisTicksCount`, but for the y-axis.

yAxisTicksFormat (*function*)
Analogous to `xAxisTicksFormat`, but for the y-axis.

yAxisLabel (*string*)
Name of the y-axis signal to be displayed to the left of the chart.

6.2.7 Zoom

These properties control the behaviour of zooming (enlarging region of the chart) and panning.

withZoom* (*bool*)

zoomLevelMin (*number*)
zoomLevelMax (*number*)
Zoom level is the factor by which the chart is scaled (enlarged). The properties `zoomLevelMin` and `zoomLevelMax` define the minimal and maximal value of zoom level which can be achieved using mouse wheel, touch pinch

and zoom buttons in the chart's toolbar. Brushing (selecting a region) can lead to even higher enlargement. It is not recommended to reduce the `zoomLevelMin` below 1.

6.2.8 Setting the visible region

Components with zoom functionality also have a way to set the visible region from code. It can be done through the `setView` method. To get the currently visible region, the `getView` method can be used. A function passed to the `viewChangeCallback` property will get called every time the visible region changes.

Example on how these methods and properties can be used are shown in Section 7.5, which describes how to create two charts with synchronized visible regions.

`setView` method

The signature of this method is similar in all components:

```
setView(xMin, xMax, yMin, yMax, source, causedByUser = false)
```

The first four arguments (in some components only two) are the desired boundaries of the view. The `xMin` is the left boundary, `xMax` right, `yMin` bottom, `yMax` top. The values are in the same units as the data, not in pixels. To keep some of the boundaries unchanged, pass the `undefined` value as the corresponding argument.

The `source` argument should be set to the object which caused this view change. It is used inside the `setView` method to prevent updating the view when the change was caused by the component itself.

The `causedByUser` is a boolean argument which tells whether the view update was caused by the user or if it is a result of updating another component. The value will be passed to the `viewChangeCallback` call (described below).

`getView` method

Returns the boundaries of the currently visible region of the chart. The boundaries are returned as an object of shape { `xMin, xMax, yMin, yMax` } with all fields being numbers or string depending on the data displayed by the chart.

`viewChangeCallback` property

A function with following signature can be assigned to this property:

```
function(component, view, causedByUser)
```

It will be called whenever the view is changed.

The first argument is the component whose view was changed. The second one is the current view (in same format as returned by the `getView` method). The last argument is a boolean telling if the view change was caused by user. If the view change is a result of `setView` method call, the last argument will be set to the `causedByUser` value in that call.

6.3 `ScatterPlot` component

The `ScatterPlot` component implements a scatter plot described in more detail in Section 4.1. The chart is initialized with data from the specified signals. If the dataset is too large, only a random sample of the data is loaded. The user can later fetch additional data for a specific region of the chart. The data fetched during initialization will remain visible to keep the overall picture of the data, but will be rendered differently than the newly fetched data for specified region.

The term *dot* is used in this documentation for visualization of one data point. It can be represented by a small circle or other shape as described in Section 6.1.2. We use the term *global dot* for data points fetched during the initialization of the chart, i.e. those records which were not fetched additionally by the user for a specific region of the chart.

6.3.1 Signal configuration

The `config` property of `ScatterPlot` has only one field called `signalSets` which is an array of signalSet configurations. Each such configuration has the properties described in the following subsections.

Signals

All signals must be from the same signalSet. All these fields are specified inside the `config`.

cid † (*string*)
SignalSet identifier.

x_sigCid † (*string*)
Identifier of the signal to be used as horizontal position of the dot.

y_sigCid † (*string*)
Identifier of the signal to be used as vertical position of the dot.

colorContinuous_sigCid (*string*)
Identifier of the signal to be used to determine color of the dot. It must be a numerical (continuous) signal. This property is mutually exclusive with `colorDiscrete_sigCid`.

colorDiscrete_sigCid (*string*)
Identifier of the signal to be used to determine color of the dot. It must be a categorical (discrete) signal (keyword type in IVIS). This property is mutually exclusive with `colorContinuous_sigCid`.

tsSigCid (*string*)
Identifier of the signal defining the date and time of the records. It is used to filter the data if the component is placed inside a `TimeContext`.

label_sigCid (*string*)
Identifier of the signal to be used for labels in tooltip. See also Tooltip section below (6.3.1).

Rendering

These are properties specific for each signalSet which influence how the data are rendered. More information about how the final color and size of the dot is determined is written in sections 6.3.4 (color) and 6.3.3 (size).

color (*color* or *array* of *colors*)
The color of the dots for this signalSet. If not specified, one of the colors from component's `colors` property[1] will be used.

If `colorContinuous_sigCid` or `colorDiscrete_sigCid` is specified, this property must be an array (or undefined).

label (*string*)
Name of the signalSet to be displayed in tooltip, etc.

enabled (*bool*)
Determines whether the signalSet should be rendered or not. Useful for example in combination with `Legend` component as described in example in Section 7.4.

dotSize (*number*)
Size (radius) of the dots for this signalSet, in pixels. If not specified, value of `dotSize` property of the component[2] is used.

dotShape (*string*, one of dot shape names[3]) ["circle"]
The name of the shape to be used for displaying dots in chart.

globalDotShape (*string,* one of dot shape names[3]) ["circle_empty"]
The name of the shape to be used for displaying global dots in the chart.

getGlobalDotColor (*function*)
Function to be applied to the color of each global dot before rendering. It is applied to each dot separately. It takes a *color* as argument (the color the dot would have if it wasn't global) and should return *color* (which is then used to render the dot). By default, the opacity of global dots is decreased to half.

tooltipLabels (*object*)
Configuration of information shown in tooltip, described below in 6.3.1.

regressions (*array* of *objects*)
Configuration of regressions shown in chart, see Section 6.3.1.

Tooltip

The tooltip can display additional information about the records. For each signalSet information about the dot closest to the mouse cursor is shown. To enable tooltip, use `withTooltip` property of the `ScatterPlot` component.

withTooltip* (*bool*) [true]
This property will get copied to the internal state of the component in the

[1]It will be described in Section 6.3.4.
[2]It will be described in Section 6.3.3.
[3]See Section 6.1.2.

constructor. Changing it later will **not** update it. Use `setWithTooltip` method[4] to change the property after the component was created.

The tooltip displays information about each signalSet. On the first line of the tooltip text for a signalSet, the `label` is displayed. Then, the values of signals are displayed. To change how signal values are displayed in the tooltip, set the `tooltipLabels` property of signalSet configuration. It has the following format.

label_format (*function*)
Function to format the label on the first line of the tooltip. It gets the signalSet's `label` and the value of `label_sigCid` for the closest dot if defined. The value returned from this function will be rendered in the tooltip.

x_label (*string* or *function* or `null`)
Label for the signal displayed on X-axis.

y_label (*string* or *function* or `null`)
Label for the signal displayed on Y-axis.

color_label (*string* or *function* or `null`)
Label for the signal used for determining color of dots.

Each of the properties (except `label_format`) can be a string or a function. Setting the property to `null` will hide information about the corresponding signal in the tooltip. If the property is set to string, this string is displayed before the value of the signal with a colon between them. If it is set to a function, the value will be passed as a parameter to the function and the returned value will be rendered in the tooltip. It is possible to set only some of the fields mentioned above, the default values will be used for the unspecified.

Regressions

The regressions can be specified in `regressions`* property of the signalSet configuration. The property accepts an array of regression configurations. Each such configuration has the following format.

type † (*string*)
The type of regression to be computed. Currently supported types are "linear", "quadratic", "polynomial", "loess", "exponential", "logarithmic", "power".

color (*color* or *array* of *colors*)
Color of the rendered regression line. If not specified, the color of the signalSet will be used.

createRegressionForEachColor (*bool*) [false]
Only valid when `colorDiscrete_sigCid` is specified. When set to `true`, the regression is computed and rendered independently for all records of each unique value of the `colorDiscrete_sigCid` signal.

bandwidth (*number* between 0 and 1)
Only valid for "loess" regression type. Defines the level of smoothing.

[4]It will be described in Section 6.3.10.

order (*integer*)
Only valid for "polynomial" regression type. Defines the degree of the polynomial.

Please note that some types of regression (especially high degree polynomials) can take long time to compute, which will significantly prolong the loading time of the chart.

To show or hide the coefficients of the regressions, set the `withRegression-Coefficients` property of the `ScatterPlot` component.

withRegressionCoefficients (*bool*) [true]
If set to `true`, the coefficients of the computed regressions will be displayed below the chart.

6.3.2 Limits

To set the maximum number of data points fetched for each signalSet, use the `maxDotCount` property of the `ScatterPlot` component:

maxDotCount (*integer*) [100]
Maximum number of records to be fetched in one query. This property will get copied to internal state of the component in the constructor, so changing it later will **not** update it. Use `setMaxDotCount` method[5] to change the property after the component was created.

Setting these props will set the limits (minimum and maximum) of axes. All the queries done to fetch the data will respect these limits. The initial view of the chart will be set exactly to these limits (unless `xAxisExtentFromSampledData` or `yAxisExtentFromSampledData` described in Section 6.3.5 are set to `true`).

xMinValue (*number*)
xMaxValue (*number*)
yMinValue (*number*)
yMaxValue (*number*)

6.3.3 Dot size

These are the properties of `ScatterPlot` component which influence the size of the rendered dots. If `dotSize` is specified in signalSet configuration, it is used. If not, the `dotSize` property of the chart is used.

dotSize (*number*) [5]
Size (radius) of the dots (circles or other shapes), in pixels. This is used for signalSets which don't have `dotSize` in their configuration.

highlightDotSize (*number*) [1.2]
Factor by which the radius of highlighted dots is multiplied. Highlighted dot is the one closest to the mouse cursor. If tooltip is enabled, it shows information about the data point represented by the highlighted dot.

[5]It will be described in Section 6.3.10.

6.3.4 Dot color

Dot color can be specified using these properties and also those inside the `config` property (as describe earlier in Section 6.3.1). Below, we describe how color is determined.

colors (*array* of *colors*) [schemeCategory10[6]]
Colors to be used for dots from signalSets, which don't have specified `color` property in `config`. The colors will be taken from this array in the same order as signalSets have in the `config`.

minColorValue (*number*)
maxColorValue (*number*)
Only valid for signalSets which have `colorContinuous_sigCid` property specified. These properties define the minimum and maximum of the domain for determining color (if not specified, the domain is inferred from the data). Color equal to the first element of `colors` array will be used for records with `colorContinuous_sigCid` value equal to `minColorValue`. Similarly, the last element of `colors` array will be used for records with `colorContinuous_sigCid` value equal to `maxColorValue`. If only one of these properties is specified, the other is inferred from the data.

colorValues (*array* of *strings*)
Only valid for signalSets which have `colorDiscrete_sigCid` property specified. If we know the possible outputs of `colorDiscrete_sigCid` signal, this property can be used to ensure that same colors are used for the same outputs every time. The colors from `colors` property will be used for the values specified in `colorValues` in the same order.

For each signalSet, color of the dots is determined as follows. Always, the `color` property from signalSet's `config` is preferred. If it is not specified, the `colors` property of the component is used.

First, lets assume that `colorContinuous_sigCid` nor `colorDiscrete_sigCid` is specified for the signalSet. If `color` property inside the signalSet's config is specified, it is used (or its first element if it is an array). Otherwise the element from `config` property of the component with the same index as this signalSet has in the `config` is taken. So, when we don't specify `color` inside `config.signalSets`, the signalSets will get colors from `colors` property of the component in the same order in which they are in `config.signalSets`.

If `colorContinuous_sigCid` is specified, the `color` property in `signalSets` in `config` must be an array or not defined. If it is not, the `colors` property of the component is used. A linear interpolator is made from all the colors for values between minimum and maximum of the signal or `minColorValue` and `maxColorValue` props if specified (with all colors evenly spaced). The final color is then computed using this interpolator.

Similarly, if `colorDiscrete_sigCid` is defined, each output value of this signal will get one of the colors (with cyclic indexing if there are more values than specified colors). The order of the values is not guaranteed by the component if `colorValues` property is not specified, so it might change (for example when the time interval is changed).

[6]See Color Schemes (d3-scale-chromatic) page of D3.js documentation [10] for more details.

6.3.5 Chart axes

These are properties which affect the axes of the chart.

xAxisExtentFromSampledData (*bool*) [false]
If set to `false`, the domain of x-axis is determined by the minimum and maximum values of the signal, even if the corresponding records were not retrieved in the fetched sample of data. This can lead to empty space around the dots, but shows the real extent of the data, including those data points which were not randomly selected to be displayed.

If set to `true`, the domain of x-axis is zoomed in to show only the data displayed in chart and no extra space around. That means, that minimum and maximum of the fetched sample are used instead of minimum and maximum of the whole signal.

xAxisTicksCount* (*integer*)

xAxisTicksFormat* (*function*)

xAxisLabel* (*string*)

yAxisExtentFromSampledData (*bool*) [false]
Analogous to `xAxisExtentFromSampledData`, but for the y-axis.

yAxisTicksCount* (*integer*)

yAxisTicksFormat* (*function*)

yAxisLabel* (*string*)

6.3.6 Zoom

These properties describe behaviour of the chart when user zooms in.

withZoom* (*bool*) [true]

zoomLevelMin* (*number*) [1]
zoomLevelMax* (*number*) [10]

zoomLevelStepFactor (*number*) [1.5]
The factor by which zoom level is multiplied (resp. divided) when user clicks on the Zoom In (resp. Zoom Out) button.

updateColorOnZoom (*bool*) [false]
If set to `true`, minimum and maximum values of `colorContinuous_sigCid` signal are recomputed based on current view to always use the whole range of colors.

6.3.7 Brush

A brush can be enabled, so that user can select a region of the chart to zoom to. The brush is not active by default and must be activated by the user using a button in toolbar[7] or by holding down the control key.

withBrush* (*bool*) [true]

[7]See also 6.3.8.

withAutoRefreshOnBrush (*bool*) [true]
Brushing automatically fetches new data for the selected region.

6.3.8 Toolbar

A toolbar above the chart can be displayed to allow the user to set the zoom level, enable brush and set exact boundaries of the visible region. See also the user guide for toolbar in Section C.3.1.

withToolbar (*bool*) [true]
Shows toolbar with buttons above the chart.

withSettings (*bool*) [true]
The toolbar allows user to alter the configuration of the chart's properties, namely the `withTooltip`, `maxDotCount`, and the view boundaries can be set. This option is only valid when `withToolbar` is set to `true`.

6.3.9 Common properties

These props, already described in subsections of Section 6.2, are available in the `ScatterPlot` component.

height* † (*number*)

margin* (*object* { top, bottom, left, right : *number* }) [{ left: 40, right: 5, top: 5, bottom: 20 }]

withCursor* (*bool*) [true]

withTransition* (*bool*) [true]

viewChangeCallback* (*function*)

className* (*string*)

style* (*object*)

6.3.10 Methods

This is a list of public methods of `ScatterPlot` component with descriptions of their arguments.

getView() *

setView(xMin, xMax, yMin, yMax, source, causedByUser = `false`, withTransition = `false`) *

If `withTransition` is set to `true` (and `props.withTransition` is also `true`), the view change will be animated.

setWithTooltip(newValue *: bool*)

Sets the value of `withTooltip` property.

setMaxDotCount(newValue *: integer*)

Sets the value of `maxDotCount` property.

6.3.11 Advanced configuration

The `ScatterPlot` component also provides a way to modify the fetching and the rendering of the data. The following properties can be used to replace the built-in methods. We recommend seeing the source code of the `ScatterPlot` and `ScatterPlotBase` components as a reference for how these methods are called.

If the desired behaviour is only slightly different to the built-in one, the built-in methods can also be used inside their replacements. This can be used for example to save the results of the built-in method and work with them later as we can see in the example in Section 7.2.2.

The filter property can be used to create more specific queries which retrieve only a subset of the signalSet.

filter (*object* or *function*)
The filter will be added to each data query. If this property is a function, it will be evaluated with signalSet configuration as an argument and the result will be used as a filter. Examples of filters can be found in the `getQueriesForSignalSet` method in `ScatterPlotBase` component.

All the following properties are functions which replace the built-in methods. We only provide a brief description here; the arguments and return values are documented in the source code.

getQueries (*function*)
This method prepares queries for retrieving data through the `DataAccess`. The built-in implementation first sets the limits based on visible region and then calls the `getQueriesForSignalSet` method for each signalSet and concatenates the results.

getQueriesForSignalSet (*function*)
This method creates queries for one signalSet. It gets the config for the signalSet as one of the arguments.

prepareData (*function*)
This method is used to process both the records and the extents of data returned from server before saving them to component's state. Calls the `processDocs` and `computeExtents` methods inside.

processDocs (*function*)
Processes the records returned from server. This method can be used to alter the returned data. Called from `prepareData`.

computeExtents (*function*)
Computes the extent (minimum and maximum) of each signal. Called from `prepareData` after `processDocs`.

filterData (*function*)
Before drawing the chart, the data are filtered so that the dots which would be outside of the visible region are not rendered.

drawChart (*function*)
The method for drawing the chart. It calls `drawDots` for each of the signalSets. This method can be used to add elements to the chart.

drawDots (*function*)
This method draws the dots for one signalSet. It can be used to alter the way the dots are rendered.

drawHighlightDot (*function*)
The method for highlighting the dot closest to the cursor. It gets called once for each signalSet. The built-in implementation draws another dot on top of the original using the `drawDots` method with darker color.

6.4 `BubblePlot` component

The `BubblePlot` component implements a bubble plot, which was described in Section 4.2. This chart is almost identical to the `ScatterPlot` with only additional possibility to control size of the dots with a signal. We therefore only describe differences from the scatter plot here.

6.4.1 Signal configuration

Similarly to the `ScatterPlot`, the `config` property of `BubblePlot` has only one field called `signalSets` which is an array of signalSet configurations. Each such configuration has following properties.

Signals

The properties regarding signal selection in `BubblePlot` are the same as in `ScatterPlot`, described in Section 6.3.1, with addition of the following one.

dotSize_sigCid † (*string*)
Identifier of the signal to be used to determine size of the dot. This will update the area of the dot proportional to values of this signal, not its radius, as described in Section 4.2.1.

Rendering

The `BubblePlot` has the same properties influencing rendering specific for each signalSet as the `ScatterPlot` except for the `dotSize` property, which is removed. See Section 6.3.1.

Tooltip

All properties described in Section 6.3.1 are also relevant here with addition of a new field to `tooltipLabels` property.

dotSize_label (*string*)
Label of the signal used to determine size of dots.

Regressions

These configurations are identical to the `ScatterPlot`. See Section 6.3.1.

6.4.2 Limits

These configurations are identical to the `ScatterPlot`. See Section 6.3.2.

6.4.3 Dot size

The size of the dots is determined by `dotSize_sigCid` signal for each signalSet. The following properties influence the final size of the dots.

minDotSize (*number*) [2]
maxDotSize (*number*) [14]
Minimum and maximum size (radius) of the dots (circles or other shapes), in pixels. The minimum dot size corresponds to the minimum value of the `dotSize_sigCid` signal or to the value specified in `minDotSizeValue`. The maximum dot size corresponds to the maximum value of the signal or to the `maxDotSizeValue`.

minDotSizeValue (*number*)
maxDotSizeValue (*number*)
If specified, the range for determining minimum and maximum size of the dots is not inferred from the extent of the `dotSize_sigCid` signal but is set to these values. It is also possible to specify only one of these properties.

highlightDotSize* (*number*) [1.2]

6.4.4 Dot color

These configurations are identical to the `ScatterPlot`. See Section 6.3.4.

6.4.5 Chart axes

These configurations are identical to the `ScatterPlot`. See Section 6.3.5.

6.4.6 Zoom

All properties described in Section 6.3.6 can be used. The following one is added.

updateSizeOnZoom (*bool*) [false]
If set to `true`, the minimum and maximum values of `dotSize_sigCid` signal are recomputed based on current view to always use the whole range of sizes.

6.4.7 Brush

These configurations are identical to the `ScatterPlot`. See Section 6.3.7.

6.4.8 Toolbar

These configurations are identical to the `ScatterPlot`. See Section 6.3.8.

6.4.9 Common properties

These configurations are identical to the `ScatterPlot`. See Section 6.3.9.

6.4.10 Methods

The public methods are identical to the `ScatterPlot`. See Section 6.3.10.

6.4.11 Advanced configuration

These configurations are identical to the `ScatterPlot`. See Section 6.3.11.

6.5 `HistogramChart` component

This component implements a histogram, which was described in Section 4.3. The data fetched from the server are bins covering the full extent of the signal. Each bin contains number of records belonging to it. The `HistogramChart` can only be used for numeric signals.

It is possible to enable a second histogram below the main one for better control of the zoom. We call this secondary histogram an *overview*.

6.5.1 Signal configuration

The `config` property contains information about the signalSet and signal used to create the histogram. It has following fields.

Signals

All these fields are specified inside the `config`.

sigSetCid † (*string*)
SignalSet identifier.

sigCid † (*string*)
Signal identifier. Only numerical signals are valid.

tsSigCid (*string*)
Identifier of the signal defining the date and time of the records. It is used to filter the data if the component is placed inside a `TimeContext`.

metric_type (*string*)
If specified, the bucket's value (height of the corresponding bar) is not the count of the records belonging to the bucket, but it is computed from the values of the `metric_sigCid` signal of the records belonging to the bucket. Possible values are `"sum"` (bucket's value is the sum of the signal values), `"min"` (bucket's value is the minimum of the signal values), `"max"` (bucket's value is the maximum of the signal values), `"avg"` (bucket's value is the average of the signal values).

metric_sigCid (*string*)
Identifier of the signal for the `metric_type` property.

Rendering

All these fields are specified inside the `config`.

color † (*color*)
The color of the bars in the chart.

6.5.2 Bin size

All three of these properties are used to determine the size of the bins (buckets). They all limit the maximum number of bins, e.g. when `minBarWidth` is set, the chart will fetch the highest number of buckets that will fit the chart's width. All conditions set by these properties will hold simultaneously for the final number of buckets.

minStep (*number*)
Minimal size of the histogram bin, in the units of the data.

minBarWidth (*number*) [20]
Minimal width of the bars in histogram, in pixels.

maxBucketCount (*integer*)
The maximum number of histogram bins.

6.5.3 Limits

Setting these props will set the limits (minimum and maximum) of the x-axis.

xMinValue (*number*)
xMaxValue (*number*)

6.5.4 Chart axes

These are properties which affect the axes of the chart.

xAxisTicksCount* (*integer*)

xAxisTicksFormat* (*function*)

xAxisLabel* (*string*)

6.5.5 Zoom

withZoom* (*bool*) [true]

zoomLevelMin* (*number*) [1]
zoomLevelMax* (*number*) [4]

topPaddingWhenZoomed (*number*) [0]
A number between 0 and 1 which defines how big portion of the chart will remain empty when the chart is zoomed and the bars are enlarged. The bars are never shrunk, so the actual free space might be smaller.

When the user zooms in, all bars are stretched vertically so that the height of the highest one is at least `chartHeight * (1 - topPaddingWhenZoomed)`, where `chartHeight` is the `height` of the chart reduced by `margin.top` and `margin.bottom`.

6.5.6 Overview

As stated earlier, the overview is a secondary histogram with permanent brush[8] that helps user to select the desired visible region of the main chart.

withOverview (*bool*) [true]
Displays the overview below the chart.

overviewHeight (*number*) [100]
The height of the overview chart in pixels.

overviewMargin (*object* { top, bottom, left, right : *number* }) [{ top: 20, bottom: 20 }]
Equivalent of the chart's `margin` property for the overview. The axis of the overview is drawn inside the margins, so setting the bottom margin to zero will hide the axis.

6.5.7 Common properties

These props, already described in subsections of Section 6.2, are available in the `HistogramChart` component.

height* † (*number*)

margin* (*object* { top, bottom, left, right : *number* }) [{ left: 40, right: 5, top: 5, bottom: 20 }]

withCursor* (*bool*) [true]

withTooltip* (*bool*) [true]

withTransition* (*bool*) [true]

viewChangeCallback* (*function*)

className* (*string*)

style* (*object*)

6.5.8 Methods

This is a list of public methods of `HistogramChart` component with descriptions of their arguments.

getView() *

setView(xMin, xMax, source, causedByUser = `false`) *

6.5.9 Advanced configuration

The `HistogramChart` component also provides a way to modify the data fetched from the server before rendering them. The following properties can be used to replace the built-in methods. We recommend seeing the source code of the `HistogramChart` component as a reference for how these methods are called.

[8]See Section C.2.2.

If the desired behaviour is only slightly different to the built-in one, the built-in methods can also be used inside their replacements (see the example in Section 7.2.2).

The filter property can be used to create more specific queries which retrieve only a subset of the signalSet.

filter (*object*)
The filter will be added to each data query. Examples of filters can be found in the `fetchData` method in `HistogramChart` component.

All the following properties are functions which replace the built-in methods. We only provide a brief description here; the arguments and return values are documented in the source code.

prepareData (*function*)
This method is used to process the data returned from the server before saving them to component's state. It calls the `processBucket` method for each of the buckets and then computes the frequencies of the buckets (heights of the bars) based on their values.

processBucket (*function*)
This method should return the value of the bucket which is then used to compute the frequencies of all buckets. The built-in implementation returns the count of the records inside the bucket if the `metric_type` is not specified, and the value of the metric if the `metric_type` is specified.

6.6 `HeatmapChart` component

The `HeatmapChart` is a two dimensional equivalent of the `HistogramChart`. Many similar properties will therefore exist. Details about the chart itself can be found in Section 4.5. One of the difference is that the `HeatmapChart` can work both with numerical and discrete signals.

Similarly to the histogram, an overview below the chart can be enabled to allow easier navigation. In this overview, a marginal histogram is drawn for the x-axis signal. Additionally, an overview for the y-axis signal can be also added and will be displayed to the left of the chart.

6.6.1 Signal configuration

The `config` property contains information about the signalSet and signals used to create the heatmap.

Signals

All these fields are specified inside the `config`.

sigSetCid † (*string*)
SignalSet identifier.

x_sigCid † (*string*)
Horizontal axis signal identifier.

y_sigCid † (*string*)
Vertical axis signal identifier.

tsSigCid (*string*)
Identifier of the signal defining the date and time of the records. It is used to filter the data if the component is placed inside a `TimeContext`.

metric_type* (*string*)

metric_sigCid* (*string*)

Rendering

All these fields are specified inside the `config`.

colors † (*array* of *colors*)
The color spectrum to be used for displaying densities in bins. The first color will be used for the bin with the smallest count of elements and the last color will be used for the bin with the highest number of occurrences. Between those, all colors are linearly interpolated.

6.6.2 Bin size

For numerical signals, the bin sizes are calculated the same way as in histogram (see Section 6.5.2) with the only difference being that they are set independently for each of the axes. For discrete signals, at most 10 of the most frequent outputs will be treated as bins. None of the following properties have any effect for discrete signals.

minStepX (*number*)
Minimal size of the bin along the x-axis, in the units of the data.

minStepY (*number*)
Minimal size of the bin along the y-axis, in the units of the data.

minRectWidth (*number*) [40]
Minimal width of the rectangles displayed in the heatmap, in pixels.

minRectHeight (*number*) [40]
Minimal height of the rectangles displayed in the heatmap, in pixels.

maxBucketCountX (*integer*)
The maximum number of bins along the x-axis.

maxBucketCountY (*integer*)
The maximum number of bins along the y-axis.

6.6.3 Limits

Setting these props will set the limits (minimum and maximum) of axes.

xMinValue (*number*)

xMaxValue (*number*)

yMinValue (*number*)

yMaxValue (*number*)

6.6.4 Chart axes

These are properties which affect the axes of the chart.

xAxisTicksCount* (*integer*)

xAxisTicksFormat* (*function*)

xAxisLabel* (*string*)

yAxisTicksCount* (*integer*)

yAxisTicksFormat* (*function*)

yAxisLabel* (*string*)

6.6.5 Zoom

The `HeatmapChart` allows more precise control of the zoom as it can be enabled independently for each axis.

withZoomX (*bool*) [true]
withZoomY (*bool*) [true]
Analogous to `withZoom`. Enables zooming and panning by mouse and touch for the axis. User can focus on a desired region of the chart. These two properties allow to enable zoom for each axis independently.

zoomLevelMin* (*number*) [1]
zoomLevelMax* (*number*) [4]

6.6.6 Overviews

The overviews work similarly to the overview of `HistogramChart` described in 6.5.6. This time however, the brush is optional and the overviews can be drawn without it just to display the marginal histograms for the data.

withOverviewBottom (*bool*) [true]
Displays the overview with x-axis histogram below the chart.

withOverviewLeft (*bool*) [true]
Displays the overview with y-axis histogram to the left of the chart.

withOverviewBottomBrush (*bool*) [true]
withOverviewLeftBrush (*bool*) [true]
Displays a brush on top of the overview histogram showing which region of histogram is currently visible in the heatmap. It also allows user to change the visible region.

overviewBottomHeight (*number*) [60]
The height of the x-axis overview chart, in pixels.

overviewLeftWidth (*number*) [70]
The width of the y-axis overview chart, in pixels.

overviewBottomMargin (*object* { top, bottom, left, right : *number* })
[{ top: 0, bottom: 20 }]

overviewLeftMargin (*object* { top, bottom, left, right : *number* }) [{ left: 30, right: 0 }]
Equivalent of the chart's `margin` property for the overviews. The axes of the overviews are drawn inside the margins, so setting the corresponding margin to zero will hide them.

overviewBottomColor (*color*)
overviewLeftColor (*color*)
The color to be used for bars in the overviews. If not specified, the last element of `config.colors` is used.

6.6.7 Common properties

These props, already described in subsections of Section 6.2, are available in the `HeatmapChart` component.

height* † (*number*)

margin* (*object* { top, bottom, left, right : *number* }) [{ left: 40, right: 5, top: 5, bottom: 20 }]

withTooltip* (*bool*) [true]

withTransition* (*bool*) [true]

viewChangeCallback* (*function*)

className* (*string*)

style* (*object*)

6.6.8 Methods

This is a list of public methods of `HeatmapChart` component with descriptions of their arguments.

getView() *

setView(xMin, xMax, yMin, yMax, source, causedByUser = `false`) *

If the axis data type is keyword, the arguments must be strings and both boundary values are included in the changed view.

6.6.9 Advanced configuration

Same as in `HistogramChart`. See Section 6.5.9.

6.7 Frequency distribution charts

This section covers charts which are an equivalent of the histogram for discrete data. The frequency distribution can be displayed using a bar chart or a pie chart. We discussed these charts in Section 4.4.

6.7.1 `FrequencyDataLoader` component

The `FrequencyDataLoader` has the following properties. It fetches the counts of occurrences for unique outputs of the `sigCid`. When the data are loaded, the function specified in `processData` is called.

config † (*object*)
Configuration of the signals of the component. It has following fields:

> **sigSetCid †** (*string*)
> Identifier of the signalSet.
>
> **sigCid †** (*string*)
> Identifier of the signal.
>
> **tsSigCid †** (*string*)
> Identifier of the signal defining the date and time of the records. It is used to filter the data if the component is placed inside a `TimeContext`.

maxBucketCount (*integer*)
Maximum number of unique values to be fetched. The most frequent ones are retrieved.

processData † (*function*)
Function which will get called when the results are available. It gets an object as argument with `buckets` property, which is an array containing a `key` and `count` for each element.

6.7.2 `FrequencyBarChart` component

Bar chart is one of the ways to display the frequency distribution. The `FrequencyBarChart` combines `FrequencyDataLoader` and `StaticBarChart` components. It has following properties.

config (*object*)
Identical to the `config` of the `FrequencyDataLoader` component as it gets passed directly into it.

colors (*array* of *colors*) [schemeCategory10[9]]
List of colors to be used for bars. To have all bars of the same color, pass an array with only one element here.

getLabel (*fucntion*)
Function which will get `key` and `count` of each bucket and returns a value which will be used as the label for the corresponding bar. The default implementation just returns the `key`.

getColor (*function*)
Function to get the color of the bar. It will get called for each bar with the `colors` property as its first argument and the index of the bar as the second one. The default implementation returns the `index`-th color from the array (with cyclic indexing for indices higher than number of colors).

[9]See Color Schemes (d3-scale-chromatic) page of D3.js documentation [10] for more details.

maxBucketCount (*integer*)
Identical to the `maxBucketCount` of the `FrequencyDataLoader` component as it gets passed directly into it.

otherLabel (*string*) [“Other”]
When `maxBucketCount` is specified, the number of other records (i.e. those which didn’t fit into the buckets) is also recorded and displayed. This property sets the label for the bar displaying those. To completely hide the bar, set this property to `null`.

otherColor (*color*)
Color to be used for the bar displaying record which didn’t fit into any bucket (described in `otherLabel` above). If not specified the bar will be treated as additional bucket (i.e. the `getColor` function will get called).

The `FrequencyBarChart` also has `height`, `margin`, `padding`, `withTooltip`, `withTransition` and `withZoom` properties which are directly passed to the `StaticBarChart` component. The only one of them not mentioned earlier in this chapter is the `padding` property, which accepts a number between 0 and 1 defining the left and right paddings around the bars[10].

6.7.3 `FrequencyPieChart` component

Pie chart is another way of displaying the frequency distribution. The `FrequencyPieChart` component has the same basic properties as `FrequencyBarChart` described in previous section.

What differs are the properties passed to the `StaticPieChart` component which are `height`, `margin`, `getArcColor`, `legendWidth`, `legendPosition` and `legendRowClass`. Most of them have default values, so the only one required is the `height`.

[10]See D3.js documentation [10, `band.padding` in d3-scale] for more details.

7. Evaluation – visualization examples

In this chapter, we evaluate the developed component by creating examples of visualizations using them. We also try to showcase the possibilities of configuration and cooperation of the components. Here is a list of the examples described in this chapter.

- Hans Rosling's bubble plot (7.2) – simple `BubblePlot` component usage.
- Correlogram (7.3) – shows how to use panel parameters and styles and how to create many components in a loop.
- Scatter plot with legend (7.4) – cooperation of new `ScatterPlot` component with `Legend` component in IVIS.
- Synchronized views (7.5) – shows how to set the visible region of components from code.

This chapter assumes that the reader has previous experience with creating templates in IVIS. We cover the basics of creating and using templates in IVIS in the Appendix D, so we recommended reading it first to the readers without previous knowledge of IVIS.

All examples created in this chapter can be viewed in an online demo, which has been set up on `https://ivis-mt.smartarch.cz/`. To log in, use the username "demo" and password "ivis-MT2020". The permissions are set up in a way that allows the demo user to modify templates and panels only in the `public` namespace. The demo user also has the view permission for the templates in `top` namespace, but in order to modify those, a copy of the template inside the `public` namespace must be created. The online demo runs a slightly extended version of IVIS called FIVIS. All the aspects regarding charts created in this thesis are the same on both versions.

7.1 Preliminaries

7.1.1 Dataset

The dataset used in the examples below comes from Gapminder, which is an independent Swedish foundation. Gapminder combines data about world countries from multiple sources into unique coherent time series. More information about the foundation can be found on its website [12].

The Gapminder dataset was presented by Hans Rosling in his TED talks such as The best stats you've ever seen and Let my dataset change your mindset.

Gapminder offers many different indicators (signals). Each of the indicators is a time series with yearly data points and contains information about countries in the world. The data are provided for free under CC BY-4 licence.

Only a few of the indicators have been selected to be used in this thesis as it is enough to showcase the components and present the examples. The datasets can be downloaded through Gapminder's Open Numbers project hosted on GitHub. Here comes the list of selected indicators with links to their documentation on Gapminder website, download links, and signal identifiers we chose for them (identifiers will be needed in examples).

GDP per capita, constant PPP dollars
Signal ID: `income_per_person`
Documentation: GD001
Download: Open Numbers

"GDP per capita measures the value of everything produced in a country during a year, divided by the number of people. The unit is in international dollars, fixed 2011 prices. The data is adjusted for inflation and differences in the cost of living between countries, so-called PPP dollars." [12]

Total population
Signal ID: `population`
Documentation: GD003
Download: Open Numbers

Life expectancy at birth
Signal ID: `life_expectancy`
Documentation: GD004
Download: Open Numbers

Children per woman (total fertility rate)
Signal ID: `fertility_rate`
Documentation: GD008
Download: Open Numbers

Apart from the indicators, each record in the signalSet will also have following signals with general information about the country and a time stamp of the data.

Country name
Signal ID: `country`

Year
Signal ID: `year`

Country region
Signal ID: `region`
Documentation: Four Regions

Data processing and filtering

We decided to filter the data as the complete dataset contains over 85 000 records. First, all CSV files have been merged into one spreadsheet. As not all of the indicators have the same domain of countries and years, only records with values of all indicators have been selected.

To filter the data, we chose to keep only data points of every 10th year between 1900 and 2010. This reduced the size of the dataset to 2232 records. Lastly, the data were converted to JSON and a header was added as that is the format in which IVIS API accepts requests. The final data prepared for the API request can be found in `gapminder_filtered_with_header.json` file in `dataset` folder in digital attachment (see Appendix A).

The filtered dataset has been loaded to the online demo as `top:gapminder` signalSet.

To demonstrate the ability of the charts to render multiple signalSets at the same time, we decided to also include a few smaller portions of the dataset, each with entries from only one of the years. These are the signalSets with names `top:gapminder_1950` to `top:gapminder_2010` in the demo.

Loading data to IVIS

We used an API specific for FIVIS to load the data into the online demo. A shell script used for that is also provided in the `dataset` folder of the digital attachment. Before running the script, the access token and the URL address of the FIVIS server must be specified.

7.2 Hans Rosling's bubble plot

Live version: `https://ivis-mt.smartarch.cz/workspaces/1/1`
Source code: D.2 and also in the online demo

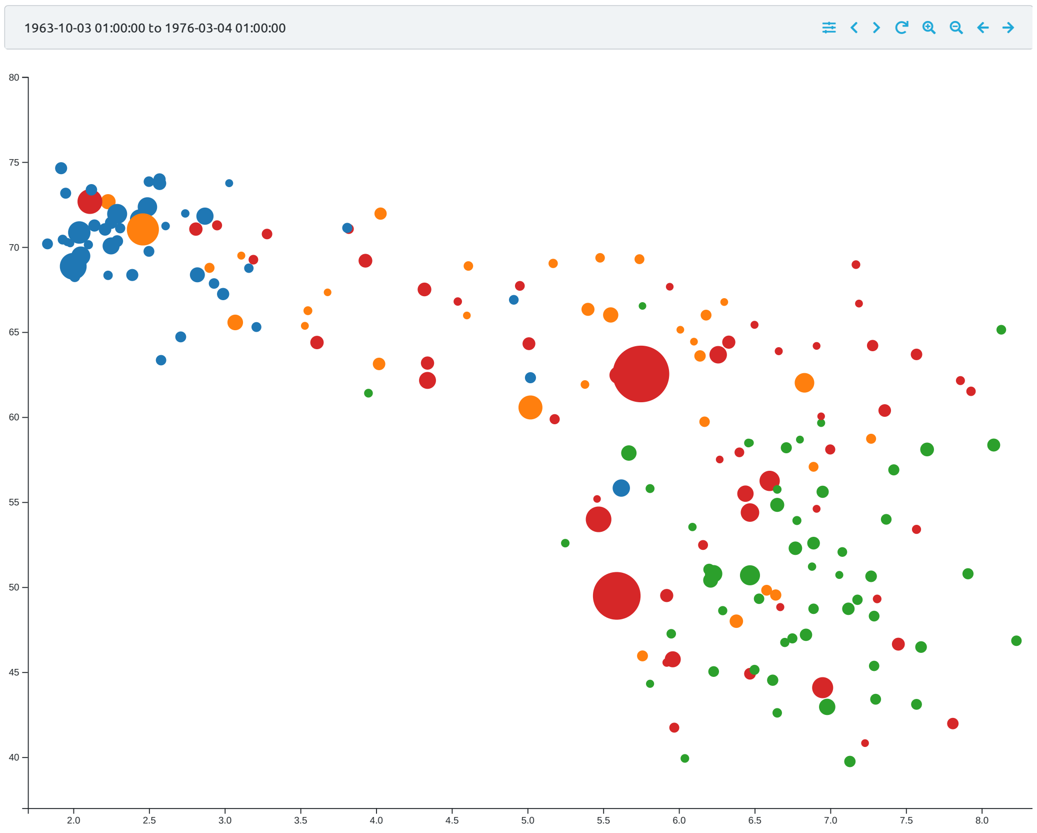

Figure 7.1: Hans Rosling's bubble plot recreated in IVIS framework.

The first example presented here will be a bubble plot, which Hans Rosling showed in his TED talk The best stats you've ever seen. This chart shows correlation of size of the family (fertility rate) and life expectancy in countries of the world in time. The size of the bubble represents the population of the country and its color shows to which region the country belongs (Europe, Americas, Africa, Asia).

As described earlier in the dataset section, we imported a subset of the Gapminder data into the IVIS framework, so we are able to recreate this bubble plot. Other components of the IVIS framework were also used to enable filtering of the data by time. Thanks to that, we have a toolbar above the chart which allows us to select the year. *Move left* and *Move right* buttons (the two rightmost icons) will change the time by 10 years (we only have data once every 10 years as described in 7.1.1, so this will take us exactly to the next point in the time series).

7.2.1 Code description

We create a React component in which we render a `BubblePlot` component. The `BubblePlot` is placed inside a `TimeContext` component alongside a `TimeRangeSelector`. These two components allow us to filter the data by time. The initial time interval is set to years 1953 to 1966, so the first records shown in the chart are those from 1960. The dates were chosen in order to make the *Move left* and *Move right* buttons of `TimeRangeSelector` move the interval by 10 years (the time intervals overlap slightly when moving left and right).

The `BubblePlot`'s `config` is specified with identifiers of all the necessary signals. The `label_sigCid` is set to "country" in order to show the name of the country in the tooltip.

```
const cnf = {
    signalSets: [{
        cid: "top:gapminder",
        x_sigCid: "fertility_rate",
        ...
        label_sigCid: "country",
```

The `tooltipLabels` except `dotSize_label` are set to `null` to hide them in the tooltip. In `dotSize_label`, we use d3-format library to render the country populations with comma separated thousands.

```
        tooltipLabels: {
            x_label: null,
            ...
            dotSize_label: p => "Population: " + d3Format.format(",")(p)
        },
```

We also specify the following properties, which ensure that when the user refreshes the chart while zoomed in, the global dots (those outside of the visible region) will be rendered the same way as the newly fetched records when the user zooms back out.

```
    globalDotShape: "circle",
    getGlobalDotColor: color => color
```

Many of the properties of the `BubblePlot` component (all of which are described in Section 6.4) are set:

- We set maxDotCount to 200 to show all the countries in the dataset at once.
- The xMinValue, xMaxValue, yMinValue, yMaxValue and xAxisLabel, yAxisLabel properties set the ranges and labels of the axes.
- The colorValues property ensures that the color of the bubbles will remain the same for the same regions when changing the time interval.
- By setting highlightDotSize to 1, the dot closest to the cursor will not be enlarged.
- The maxDotSize is the radius of the dot representing the country with the highest population.

7.2.2 Adding legend

Live version: https://ivis-mt.smartarch.cz/workspaces/1/13
Source code: D.2.2 and also in the online demo

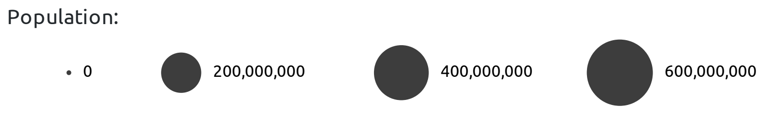

Figure 7.2: Legend for dot sizes for Hans Rosling's bubble plot.

In order to add a legend for the size of the dots, we use some of the more advanced configuration options and React's state variable (see React documentation [5, State and Lifecycle] for more information about state).

We added the computeExtents property to the BubblePlot. This function is used instead of the built-in way of computing the minima and maxima of the signals. But as we don't want to change the way the extents are computed and only save the computed values, we call the built-in implementation inside our computeExtents function:

```
const extents = BubblePlot.computeExtents(base, processedResults,
    results, queries, additionalInformation);
```

The rest of the computeExtents function just saves the minimum and maximum of the dot size signal to the state.

To draw the legend, we use the d3-scale library to generate nice human-readable values along the extent of the signal. This is done by first creating a scaleSqrt object and then calling the ticks method on it. For each of the generated values we draw an SVG circle (with radius obtained through the scaleSqrt) and a text with the value.

We also want to show an alternative function for drawing the highlighted dot (the one closest to the mouse cursor). The drawHighlightDot function in this example draws a circle with a dark grey stroke instead of darkening the color of the dot which is done in the built-in function.

7.3 Correlogram

Live version: https://ivis-mt.smartarch.cz/workspaces/1/4
Source code: D.3 and also in the online demo

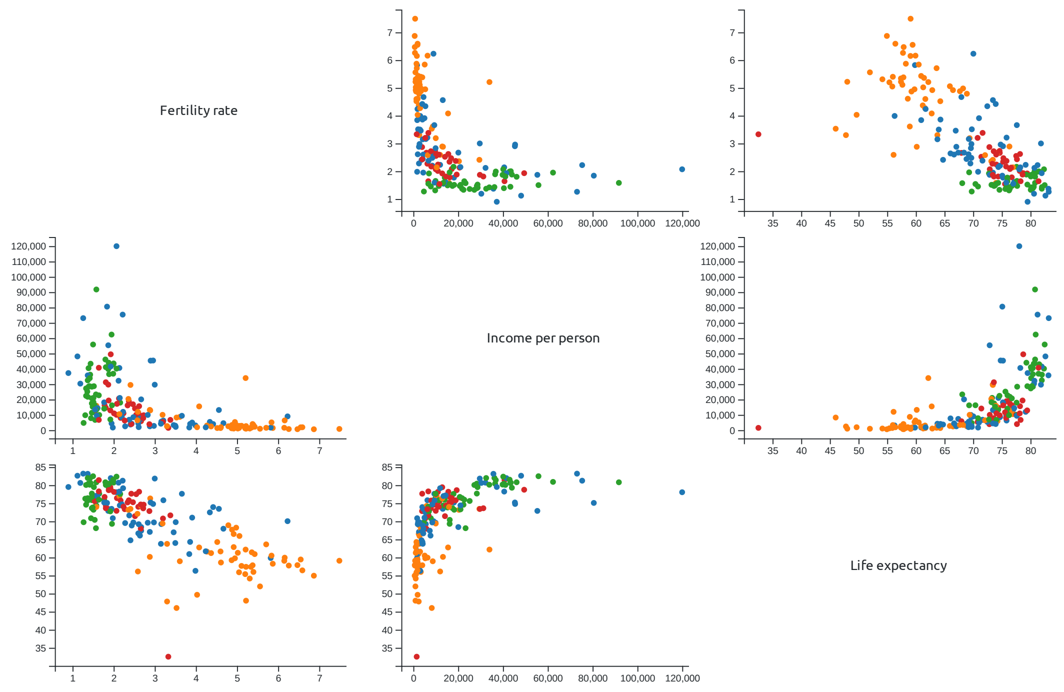

Figure 7.3: Correlogram created in IVIS.

Correlogram (also known as *scatter plot matrix*) is a chart which is composed of several scatter plots. It can visualize correlation among more than two signals by creating a scatter plot for each pair. We have described it in Section 4.6.

In this example, we will create a configurable correlogram. This example builds upon the previous one, so it is recommended to read Section 7.2 first.

7.3.1 Code description

The correlogram template consists of two classes. The first one creates the `Correlogram` component, which is then used in the second one. This template also uses parameters to specify the signals.

Similarly to the first example, we added a `TimeContext` component inside the `render` function of the `TestCorrelogram` class (which is exported as default and thus rendered by the template) to filter the data by year. This time, the `TimeRangeSelector` will only be rendered if the `ts_signal` parameter is set for the panel.

The `TestCorrelogram` class loads the configuration of the panel and converts it to a format accepted by the `Correlogram` class.

The `Correlogram` class creates a simple correlogram chart. It is a React component with only the `render` method. Inside this method, we first prepare the properties for all the scatter plots (same for all):

- We disable tooltip, regression coefficients and toolbar to keep only the chart.
- We also disable zoom and brush for the chars, so the visible region cannot be changed by the user.
- The `xAxisTicksCount` is reduced, so that the ticks do not overlap (we use income per person, which has high values, as one of the signals in the demo).

Then, we go through the signals in two nested loops (using `.entries()` to get the indices and values at the same time) and create a scatter plot of the two signals if they aren't the same. We also add color and time series signals to the config of each scatter plot to enable filtering of the data by time and coloring of the dots using the signal specified in the panel parameters.

We keep the created scatter plots in `row` and `rows` arrays. The scatter plots are then rendered inside a html `<table>`. On the main diagonal (when we have the same indices in the loops), we display a `<div>` with the name of the signal. These `<div>`s have their styles defined by a class and also by the `styles` property, in which we set the margins to the same values as the margins of the plots to center the text.

7.4 Scatter plot with legend

Live version: `https://ivis-mt.smartarch.cz/workspaces/1/6`
Source code: D.4 and also in the online demo

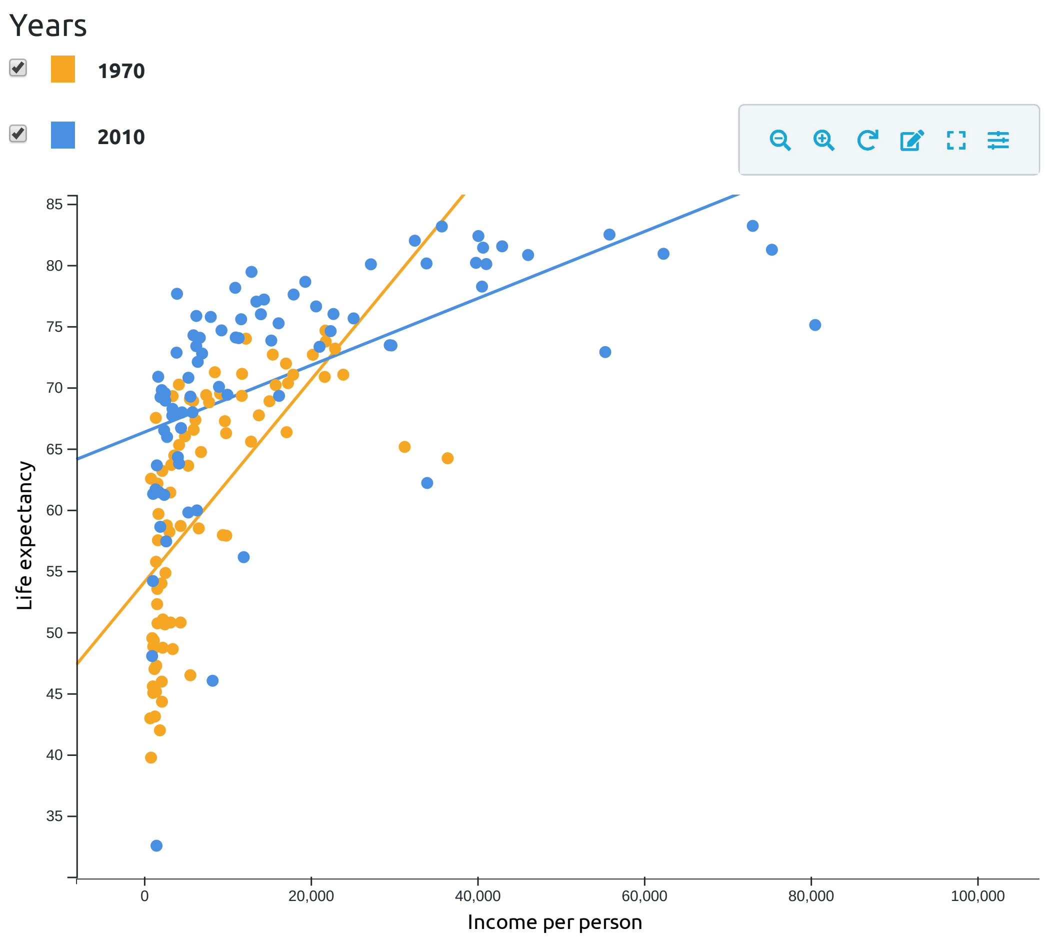

Figure 7.4: Scatter plot with legend and signal selection, created in IVIS.

This example shows how to connect the `ScatterPlot` component to the `Legend` component in IVIS to allow the user to select which signalSets are displayed.

7.4.1 Code description

The `Legend` component is not connected to the `ScatterPlot`, but it rather updates directly the configuration of the panel. We have to specify the identifier of the parameter modified by the legend in the `configPath` property and the structure of the parameter in `configSpec`. The structure is just a copy of the specification of the only parameter for this template.

In the render method, we first prepare the config for the scatter plot. Some of the properties of each `signalSetConfig` are set to the same values for all the signalSets. The rest is then added based on the panel configuration. For example, the `enabled` property is used to show or hide the specified signal in the chart. It is set by the checkbox in the legend.

The `Legend` and the `ScatterPlot` are then rendered together inside a React fragment (a way to render two DOM elements at the same time).

7.5 Synchronized views

Live version: `https://ivis-mt.smartarch.cz/workspaces/1/5`
Source code: D.5 and also in the online demo

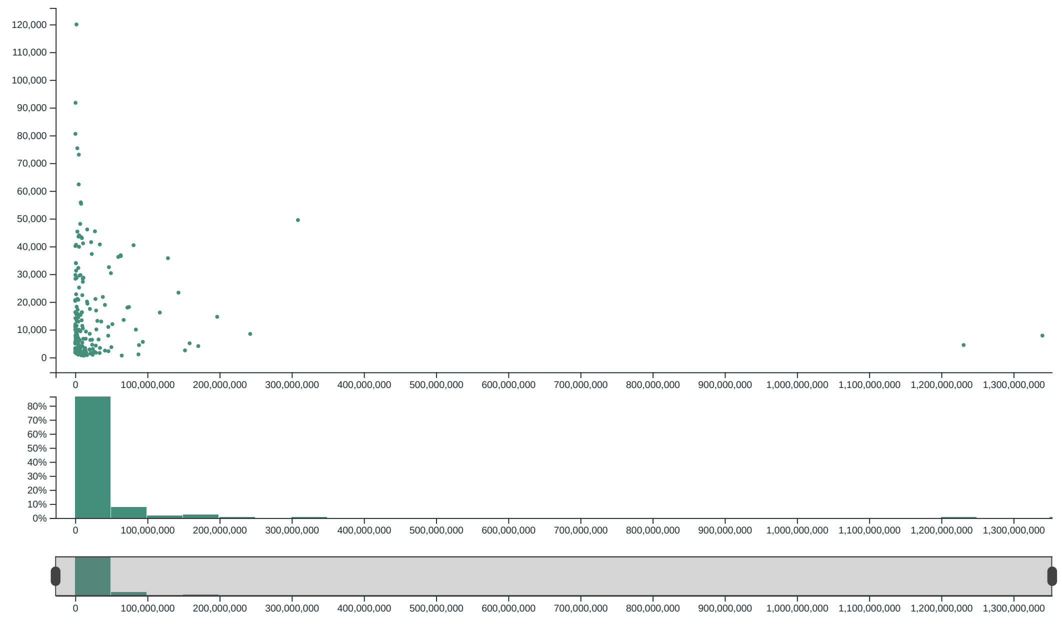

Figure 7.5: Scatter plot and histogram with synchronized views, created in IVIS.

The fourth example will show a more advanced visualization with two charts with synchronized views. That means that if the user zooms in one chart, the other chart will update to show the same extent of data.

We selected a scatter plot displaying correlation between country population and its income per person. We added a histogram below the scatter plot to filter the data by country population. This is a nice example in which we want to filter our data as we have two outlier values with really high population (China and India) and most of the data are on the opposite side of the scale. We can now use the histogram's overview to easily select the desired region of the scatter plot.

7.5.1 Code description

The setup

This template creates a more advanced React component, which uses `state` to store information and automatically update the rendered elements (see React documentation [5, State and Lifecycle] for more information about `state`). We also use a `MinMaxLoader` component, which retrieves the minimum and maximum of a signal. This is done to set the scatter plot's and histogram's range along x-axis to the same values, so the displayed data are aligned nicely. We add a slight margin to this range using `extentWithMargin` method in order to keep a space between the data and the left axis of the scatter plot.

In the constructor, we initialize the `this.state` variable to contain `xMinValue` and `xMaxValue` with `null` values. When the component is first rendered, only the `MinMaxLoader` and `<div>` with text "No data." is drawn (`MinMaxLoader` does not have any UI).

The `MinMaxLoader` component automatically starts to fetch the data and when the results are available, it calls the method specified in its `processData` property (which is `this.processMinMaxResults`). That method sets the `xMinValue` and `xMaxValue` properties of the `state` variable using React's `setState` method.

When the state is updated, the component is rerendered automatically. This time, the `ScatterPlot` and `HistogramChart` are created and the "No data." text is removed.

Views synchronization

There are two interesting properties in both of the charts. The `ref` property is a React's internal property which allows us to save the component into a variable (see React documentation [5, Refs and the DOM] for more information about `ref`). The `viewChangeCallback` is the one which allows us to synchronize the views. The method passed to it is called whenever the visible region of the chart updates. It also indicates whether this change was caused by the user (using mouse, touch, etc.) or from code, and gives us reference to the object which called the method. This helps us to prevent an infinite loop of view updates.

In `viewChanged` method, we call the `setView` method on both the `ScatterPlot` and the `HistogramChart`. This method updates the visible region of the components. Passing `undefined` as the third and the fourth parameter to the `ScatterPlot`'s `setView` means that we don't want to change the vertical extent. The last argument of this method is the object from which the `viewChanged` method was called. It is used inside the `setView` methods to prevent updating the view when the change was caused by the component itself. We pass the object we got in the `viewChangeCallback` (which is one of the charts) there.

8. Related work

In this chapter, we would like to shortly present other frameworks, which can be used when visualizing data. We will focus on three data analysis and visualizing frameworks – *Grafana*, *Kibana* and *InfluxDB*. Kibana is a visualization framework created by developers of Elasticsearch. InfluxDB is an alternative to Elasticsearch for the data storage and querying, which also has visualization capabilities. Grafana can work on top of both Elasticsearch and InfluxDB (and also other) data storage engines.

All of the visualization platforms, we present in this chapter, offer a graphical user interface for setting up the visualizations. The user selects the signals, aggregations and all configurations of the chart using their mouse. This can be really simple for new users and they can learn to create visualizations really quickly. IVIS has a completely different approach to the creation of visualizations. In IVIS, each visualization is a JavaScript class (a React component), so the template creator has to have at least basic knowledge of programming. The template can have parameters which can be then specified in the graphical user interface, so the knowledge of programming is not needed when setting up the final visualization. This approach offers more possible customizations and advantages in form of using the programming skill to create visualizations, for example creating many components in a loop as we have seen in example in Section 7.3.

8.1 Grafana

Grafana is an analytics platform for observing metrics and logs. It focuses mainly on time series data. It was first designed for analysing and visualizing metrics such as system CPU, memory, disk and I/O utilization.[13] Grafana can use data from many data sources including Graphite, Elasticsearch and InfluxDB. Many information about the framework can be found on its website [14].

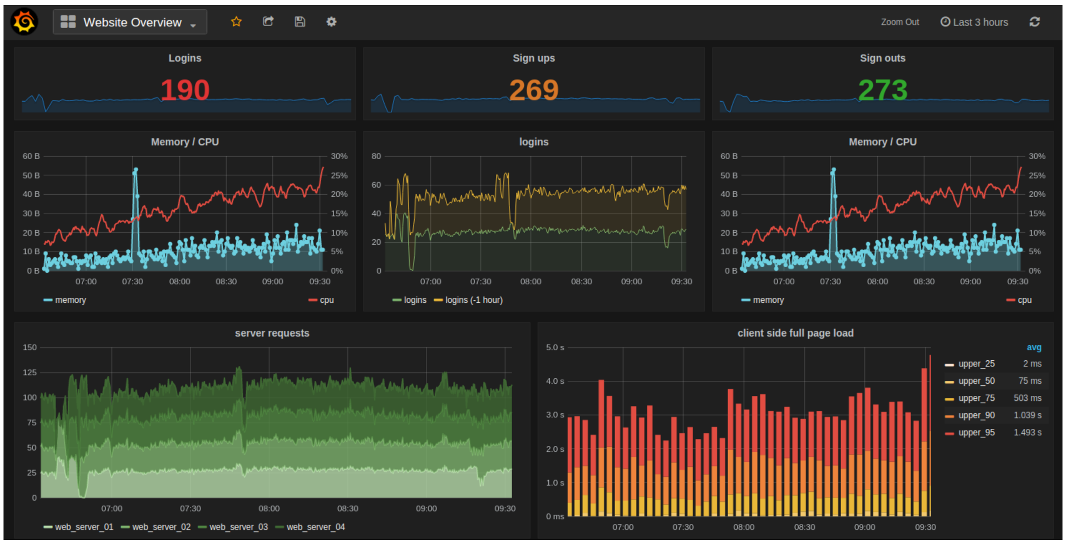

Figure 8.1: Example of Grafana dashboard. Source: Wikimedia Commons, User: Linux Screenshots from USA, CC-BY license.

The approach to creating visualizations is different than in IVIS. The user first selects the data to query and then the form of presenting them. That can be a graph, gauge, number, table, etc. When graph is selected, the user can enable or disable drawing of bars, lines and points independently. At the time of writing, there were no native charts for displaying correlations of two metrics in Grafana, although some plugins, which add them to the ecosystem, exist. In IVIS, the user first has to select the component (usually one type of chart) that he wants to use for displaying the data. The data source is then specified in the component's configuration and the component itself loads the data.

Many custom filters can be used when querying data in Grafana. The results can also be modified using functions – transformations of the data series – which can be applied sequentially. Although it is also possible to combine data from more queries, setting it up in the graphical user interface may take a lot of time. As all IVIS components are written in JavaScript, they offer the template designer more freedom in working with the data.

Grafana offers the users lots of configurations of the charts through the graphical user interface and REST API. The charts are then stored in JSON format. For creating many similar templates, Grafana offers Jsonnet, which is a superset of JSON with variables, conditions, functions, etc.[15]

One of the advantages of Grafana is a built-in alerting engine that, as written by Yigal [13], "allows the users to attach conditional rules to dashboard panels that result in triggered alerts to a notification endpoint of their choice (e.g. email, Slack, PagerDuty, custom webhooks)". In IVIS, alerts can be created in the data processing pipeline (Tasks and Jobs).

8.2 Kibana

Kibana is developed by the same company as Elasticsearch, marketed as "Your window into the Elastic Stack". List of the frameworks features and other information can be found on its website [16]. Kibana runs on top of Elasticsearch and is used primarily for analysing log messages. [13]

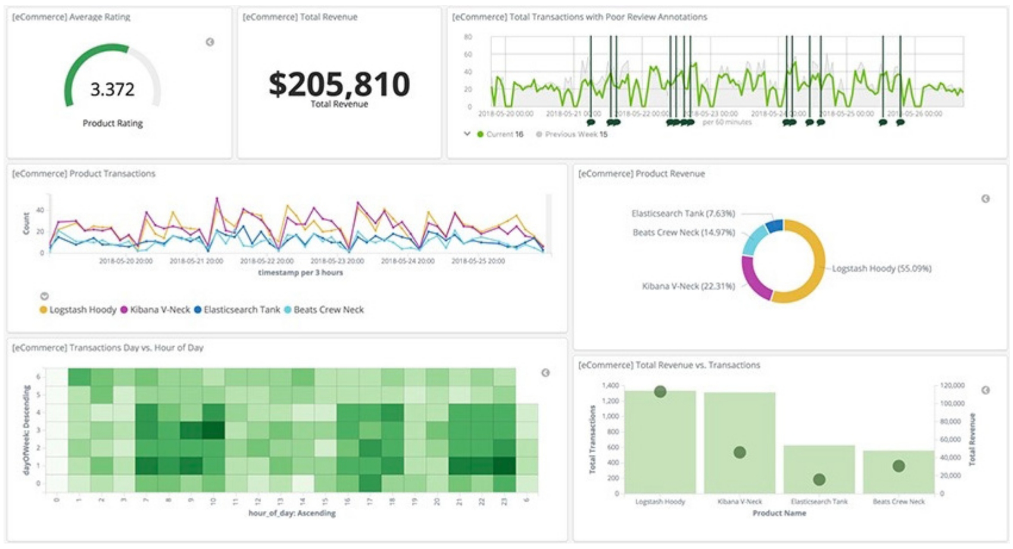
[eCommerce] Average Rating
3.372
Product Rating
[eCommerce] Total Revenue
$205,810
Total Revenue
[eCommerce] Total Transactions with Poor Review Annotations
[eCommerce] Product Transactions
Logstash Hoody
Kibana V-Neck
Elasticsearch Tank
Beats Crew Neck
[eCommerce] Product Revenue
Elasticsearch Tank (7.63%)
Beats Crew Neck (14.97%)
Kibana V-Neck (22.31%)
Logstash Hoody (55.09%)
[eCommerce] Transactions Day vs. Hour of Day
hour_of_day: Ascending
[eCommerce] Total Revenue vs. Transactions
Total Transactions
Total Revenue
Product Name

Figure 8.2: Example of Kibana dashboard. Source: [16]

The visualizations can be created similarly to Grafana, using a graphical user interface. Standard charts like line, area, bar and pie chart are supported. Kibana also offers charts for visualizing correlations, namely a heatmap. The user can select what data to plot on the X and Y axis. Custom visualizations can be set up through Vega visualization grammar (in JSON).

Kibana also supports visualizations of geographical data. The data are drawn directly on top of a map. This can be used for example to draw points on specific locations based on the data, or to aggregate the data over a region and then display the values with different colors for the regions.

Machine learning can be used in Kibana for time series data forecasting and anomaly detection. This can be connected to trigger an alert when anomaly occurs.

Besides creating visualizations, direct queries on Elasticsearch can be run from the Kibana dashboard. This can be used to search and filter the data and view the records.

8.3 InfluxDB

InfluxDB is a database built for time series data, which is designed to handle high write and query loads. It is a part of the TICK (Telegraf, InfluxDB, Chronograf, Kapacitor) stack, which groups together data collection, monitoring and analytics. More information can again be found on the official website [17].

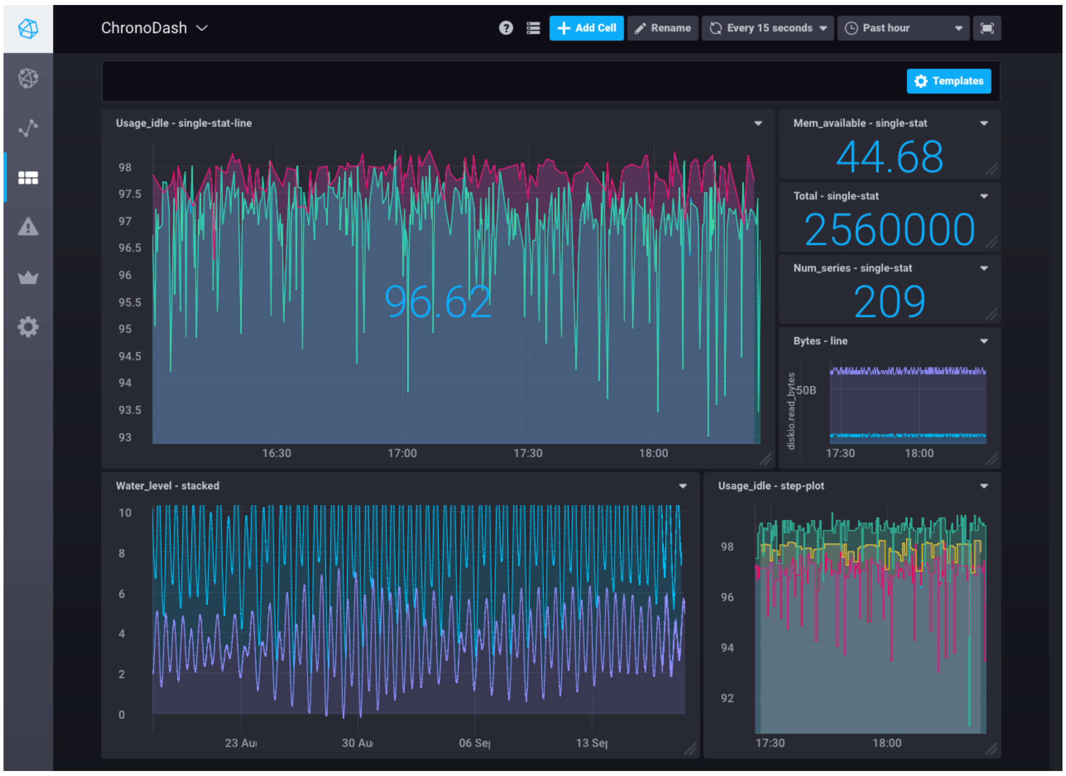

Figure 8.3: Example of InfluxDB (Chronograf) dashboard. Source: [17]

The user interface for version 1 of InfluxDB is called Chronograf. Its capabilities are similar to the frameworks mentioned above. It allows to create customizable visualizations and execute direct queries on the InfluxDB in data explorer. When creating multiple similar visualizations, template variables can be used to allow signal selection directly from the user interface. Chronograf only supports time series data and doesn't have any charts to visualize correlations.

Version 2 of InfluxDB (in beta at the time of writing) uses a different front end called InfluxDB UI. This version has many additional features including native scatter plot, heatmap and histogram charts. The charts can be set up and configured through the graphical user interface. The version 2 of InfluxDB also supports custom user-built visualization templates, but they are only packaged InfluxDB configurations saved in a file. Although Jsonnet[1] can be used to create InfluxDB templates, it still cannot offer the same capabilities as custom programmed visualizations in IVIS.

[1]Mentioned earlier in the Grafana section (8.1). Additional information can also be found on its website.

9. Conclusion

As stated in the goals of this thesis, we developed and implemented new components for IVIS framework for visualizing correlations of variables and properties of data distribution.

We first analysed which of these visualizations cannot be created with components previously present in IVIS and we proposed new components to be created. Namely, we suggested creating components for scatter plot, bubble plot and heatmap chart for visualizing correlations, and bar chart for visualizing properties of distribution of discrete data. We also proposed enhancements of the existing component for histogram chart, which is often used to show properties of distribution of continuous data. For all these charts, we discussed possible user interactions, such as zoom to a specified region of the chart, and configurations.

These components were then implemented as React[1] components. This included understanding of the relevant parts of IVIS framework and the technologies it uses. We provided code to fetch the data from the IVIS server, process the data, and then display them as SVG in the web browser using the D3.js[2] library. The components then react to the user input and update accordingly. Some changes to the server side of the project were also needed in order for the components to work.

The configurable properties of the newly implemented components were then described in detail, so that they can be easily used by the template creators in IVIS in their visualizations. We also evaluated the components by creating examples of their possible usage. The examples also cover how the components can be used together with existing parts of the IVIS framework.

All the newly created components have support for zoom, which allows user to enlarge a region of the chart. When the chart is zoomed in, the user can use mouse or touch to pan the view. As suggested earlier, histogram chart was enhanced with zoom too. Furthermore, we implemented the zoom and pan behaviour also to the line chart in order to ensure unified user experience in all components.

The newly created components are integrated into the IVIS framework so they can be used in the projects which use the IVIS framework, such as AFarCloud [18], and FitOptiVis [19].

[1]The React framework was described in section 2.3.
[2]The D3.js library was described in section 2.2.

Bibliography

[1] SmartArch. IVIS-CORE. URL `https://github.com/smartarch/ivis-core`.

[2] Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Václav Čamra, Petr Siegl, and Michal Töpfer. IVIS: Highly customizable framework for visualization and processing of IoT data. *to appear in proceedings of SEAA 2020, Portoroz, Slovenia, IEEE, August 2020*, 2020.

[3] Ralf S. Engelschall. ECMAScript 6: New features: Overview & comparison, 2017. URL `http://es6-features.org/`.

[4] Mike Bostock. D3.js – data-driven documents, 2019. URL `https://d3js.org/`.

[5] React developers. React documentation, 2020. URL `https://reactjs.org/docs/`.

[6] Elasticsearch developers. Elasticsearch documentation, 2020. URL `https://www.elastic.co/guide/index.html`.

[7] Yan Holtz. From data to viz, 2018. URL `https://www.data-to-viz.com/`.

[8] Ferdio. Data viz project. URL `https://datavizproject.com/`.

[9] Datawrapper. Chartable. URL `https://blog.datawrapper.de/`.

[10] Mike Bostock. D3.js documentation, 2020. URL `https://github.com/d3/d3/wiki`.

[11] Mozilla. MDN web docs, 2020. URL `https://developer.mozilla.org/en-US/docs/Web`.

[12] Gapminder Foundation. Gapminder. URL `https://www.gapminder.org/`.

[13] Asaf Yigal. Grafana vs. kibana: The key differences to know. URL `https://logz.io/blog/grafana-vs-kibana/`.

[14] Grafana Labs. Grafana: The open observability platform. URL `https://grafana.com/`.

[15] Joey Bartolomeo. How to configure grafana as code. URL `https://grafana.com/blog/2020/02/26/how-to-configure-grafana-as-code/`.

[16] Elasticsearch developers. Kibana: Explore, visualize, discover data. URL `https://www.elastic.co/kibana`.

[17] InfluxData. Influxdb: Purpose-built open source time series database. URL `https://www.influxdata.com/`.

[18] Aggregate FARming in the Cloud. URL `http://www.afarcloud.eu/`.

[19] From the cloud to the edge – smart IntegraTion and OPtimisationTechnologies for highly efficient Image and VIdeo processing Systems. URL `https://fitoptivis.eu/`.

List of Figures

A. Attachments

This is the structure of the digital attachment of this thesis:

- `ivis-core` – source code of the IVIS framework, details are described in Appendix B
- `dataset` – files related to the Gapminder dataset described in Section 7.1.1

B. Structure of source code

The source code of this project is available in https://github.com/Mnaukal/ivis-core/ GitHub repository, which is a fork of the official IVIS framework repository [1]. All code related to this thesis can be found in `correlation_charts` branch. The changes have been merged to the official IVIS-CORE repository in pull requests #12, #20 and #25. The code is also attached to this thesis as mentioned in the list of attachments (Appendix A).

The main interest of this thesis was in creating new components in the client side of the project. Each such component is written in its own JavaScript file with some common code written in additional files. Some changes to the server side of the project were also needed in order for the components to work.

Here is an excerpt of the project's source code structure with files relevant to this thesis.

```
ivis-core
└ client ............................................ Client related code.
  └ src
    └ ivis
      └ BarChart.js
      └ BubblePlot.js
      └ FrequencyDataLoade.js
      └ FrequencyBarChart.js
      └ FrequencyPieChart.js
      └ HeatmapChart.js
      └ HistogramChart.js
      └ FrequencyPieChart.js
      └ MinMaxLoader.js
      └ ScatterPlot.js
      └ ScatterPlotBase.js .............. Common code for ScatterPlot
                                                       and BubblePlot.
      └ ivis.js ................ List of classes exported to the ivis package.
      └ DataAccess.js .............. Interface for fetching data into charts.
      └ common.js ....... Common code used by more than one component.
      └ dot_shapes.js .................... Implementation of dot shapes.
      └ CorrelationCharts.scss .... Stylesheet for all charts in this thesis.
    └ lib
      └ CustomPropTypes.js ............ Additional property type checkers
                                               for React's PropTypes [5].
└ server ............................................ Server related code.
  └ lib
    └ indexers
      └ elasticsearch-query.js ........... Runs queries on Elasticsearch.
  └ models .. Models check the queries before they are passed to the indexer.
    └ signal-sets.js
└ examples .................................... Examples of templates.
  └ templates
```

```
├─ hans_rosling_bubble ...................... Code for example in 7.2
│  ├─ jsx.js
│  └─ jsx_withLegend.js
├─ correlogram .............................. Code for example in 7.3
│  ├─ jsx.js
│  ├─ params.json
│  └─ styles.scss
├─ scatterplot_legend ....................... Code for example in 7.4
│  ├─ jsx.js
│  └─ params.json
├─ sync_view ................................ Code for example in 7.5
│  ├─ jsx.js
│  └─ params.json
└─ heatmap ......................... Code for HeatmapChart example.
   ├─ jsx.js
   └─ params.json
```

C. User guide

This chapter contains description of the components for users viewing the visualization and interacting with the charts.

C.1 IVIS introduction

Each visualization in the IVIS framework is drawn inside a *panel.* The panels with related visualizations can be grouped together inside a *workspace.*

Navigating through the IVIS framework website is simple. Workspaces are listed in the horizontal menu at the top of the page. On a mobile device, the workspaces menu is accessible through the button in the top right of the page. Panels from currently selected workspace can be navigated through the menu on the left.

C.2 Common concepts

Many of the components feature similar interactions. These will be described in this section.

C.2.1 Zoom

If zoom is enabled for the component, the user can control which region of the charts is visible. The controls use the standard intuitive concepts. On a desktop device, mouse wheel can be used to change the scale level of the chart. Then, the visible region can be panned by clicking and dragging. On a mobile device, pinch gestures with two fingers can be used to alter the scale.

C.2.2 Brush

Brush is a technique for precise selection of the visible region. There are two types of brushes used depending on the type of the chart.

Click-and-drag brush

This type of brush is used for example in the scatter plot. When enabled, user can use the mouse to draw a region on top of the chart. To start drawing the region, mouse button (or touch) must be pressed and held. Moving the mouse while holding the button selects the region. A grey rectangle is drawn over the chart to represent the selected region. When the mouse button is released, the chart zooms to the selected region and the grey rectangle disappears.

Permanent brush

In histogram and heatmap charts, a different type of brush is used. This brush is visible for the whole time (again represented by a grey rectangle) in an additional chart under or to the left of the main chart. The ends of the rectangle have handles which can be used to modify the selected region by dragging them. The whole rectangle can also be moved by mouse drag.

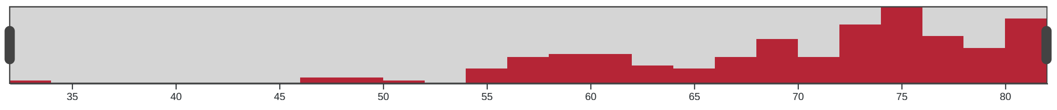

Figure C.1: Example of permanent brush.

C.2.3 Tooltip

In most of the charts, a tooltip with additional information can be enabled. When visible, it typically shows information about the data directly below the mouse cursor or those data points closest to it. The selected data points are usually highlighted, for example by changing color.

C.3 Scatter and bubble plot

The scatter plot and bubble plot charts come with zoom, click-and-drag brush and tooltip. All these features can be enabled or disabled by the visualization creator, so they might not be always available. The brush is initially disabled and can be enabled using a button in the toolbar or by holding down CTRL key.

C.3.1 Toolbar

If toolbar is enabled for the chart by the visualization creator, it is rendered in the top right-hand side of the chart. Is shows up to six buttons depending on the configuration of the chart. The two buttons represented by magnifying glass with a plus or minus sign can be used to zoom in or out. The *Reload data* button will fetch new data for the currently visible region of the chart. The *Select area* button enables brush (and disables zoom). This button keeps its state and changes color when brush is enabled. To disable it, click the button again. The *Reset zoom* button will zoom out completely. The *Open settings* button will open settings dialog which can be used to alter the current view by settings exact boundaries.

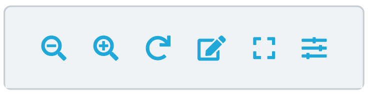

Figure C.2: Scatter plot toolbar. Buttons from left to right: *Zoom out*, *Zoom in*, *Reload data*, *Select area*, *Reset zoom*, *Open settings*.

C.3.2 Loading data

After a click on the Reload data button, the scatter plot and bubble plot fetch new data for the currently visible region and display them. The data fetched during chart's initialization (without any zoom) might also remain visible depending on the configuration (they will usually be rendered with less opaque color and possibly be represented by a different shape). All the data fetched for other specific region will disappear.

C.4 Histogram

The histogram chart has zoom and tooltip enabled by default (although they can be disabled in configuration). The bars will also get stretched up when the highest one gets out of the current view; the displayed scale will update accordingly.

When enabled, an additional histogram will appear under the main one. This additional histogram works as an overview of the data and will not be zoomed. A permanent brush is drawn on top of the overview histogram to show what range of data is currently visible in the main histogram.

C.5 Heatmap

The heatmap chart works similarly to the histogram. It can be configured to show overview histograms along both of the axes. The zoom can be also limited to only one axis.

C.6 Note on web browser compatibility

The IVIS framework uses modern technologies which might not be available in all browsers. On desktop, Firefox and Chrome browsers have been tested and all components should work in them. On mobile devices with Android operating system, we experienced some problems with zoom in Firefox browser. We thus recommend using Chrome browser on Android phones.

D. Examples of IVIS templates

In this appendix, we add more information to Chapter 7. That includes basics of creating templates in IVIS and source codes of the examples.

D.1 Creating IVIS templates

First, we provide a tutorial on how to create and render a template in IVIS framework. This thesis does not focus on the permission system of IVIS, so this section assumes that the user has all the needed permissions.

To begin, open *Settings* and navigate to *Templates* page. Use the *Create Template* button and select desired name, description and namespace for the template. *Type* should be set to *JSX template*. To allow the template to use signals not specified via panel parameters, check the *Elevated Access*. Elevated access is needed for most of the examples shown in Chapter 7. Now save the template.

Using the buttons in the top right-hand corner, navigate to *Code* editing page. To create a simple "Hello World!" template, just copy the following JSX code to the text editor and save it.

```
'use strict';

import React, {Component} from "react";

export default class HelloWorld extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <div>Hello World!</div>;
  }
}
```

This code creates a React component which will be rendered as a `<div>` element in the web browser. For more information on creating React components, see the React documentation [5].

We have now created an IVIS template and we want to display it. For that, we need a panel. Panels can be created in *Workspaces* page in *Settings*. If there are no workspaces, create a new one. Then use the *Panels* icon in the workspace's row in the table (the second icon in the rightmost column).

Create a new panel and select the previously created template in the *Template* field. The *Save and leave* button will take you back to the list of panels in your workspace and clicking on the name of the panel will display it. After a few seconds, the "Hello World!" text should appear in the browser.

D.1.1 Template parameters

The *Parameters* tab in the template code editor can be used to specify configurable parameters for the template. These parameters can then be set in the bottom section of panel configuration.

Parameters are encoded in JSON format as an array. Each parameter must contain several properties. The `id` property will specify identification of the parameter. We will need the `id` later to get the parameter's value. The `label` and `help` texts are shown to the user when specifying the parameters in panel settings.

The most interesting property is `type`. It specifies which type of data can be saved to this parameter. Basic data types such as *boolean*, *string*, *number*, *color*, *text* and also *html* and *json* are supported. Advanced types specific for IVIS are *signalSet* and *signal*. We can also set the `type` property to *fieldset*, which will create a nested set of parameters specified in the `children` property. This is useful in combination with the `cardinality` property, which allows us to determine the number of parameters of the same type (allowing us to get a list of parameters instead of just one).

If the `type` property is set to *signal*, we get a signal picker in the settings. To determine from which signalSet we want the signal to be picked, we can specify the `signalSet` or `signalSetRef` property. The `signalSet` is just the identifier of the signal set. The `signalSetRef` property defines id of the parameter from which the signal set identifier should be taken. The example in Section D.3.3 shows, that the `signalSetRef` can be relative path to the parameter ("`sigSet`" in `color_signal` and `ts_signal`) or an absolute one (we have to use "`/sigSet`" with leading slash in `sigCid` inside `signals` fieldset children).

The parameters in example in Section D.3.3 also show the creation of a fieldset with cardinality of at least two items (with no upper bound). Each item will consist of a string label and a signal. The last two parameters are optional, which can be done by setting the cardinality to "`0..1`" (at least 0, at most 1).

Accessing parameters inside JSX code

To access the parameters, the React component must be decorated with `withPanelConfig` mixin, which is done by adding `@withPanelConfig` before the class declaration.

Then, the current configuration of panel parameters can be retrieved by calling `this.getPanelConfig()` function. It returns an object with properties named by the identifiers of the parameters.

To see an example of parameter usage, see the `TestCorrelogram` class in Section D.3.

D.1.2 CSS styles in templates

The *SCSS* tab of the code editor allows us to write CSS styles. The styles must by imported into the template. To use CSS classes and ids, you must access them through the `styles` object:

```
import styles from './styles.scss';

<div className={styles.label} >
```

D.2 Source code – Hans Rosling's bubble plot

This is source code for the example described in Section 7.2.

D.2.1 JSX

```
'use strict';

import React, {Component} from "react";
import {BubblePlot, TimeContext, TimeRangeSelector, IntervalSpec} from "ivis";
import * as d3Format from "d3-format";

export default class HansRoslingBubblePlot extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    const cnf = {
      signalSets: [{
        cid: "top:gapminder",
        x_sigCid: "fertility_rate",
        y_sigCid: "life_expectancy",
        colorDiscrete_sigCid: "region",
        dotSize_sigCid: "population",
        tsSigCid: "year",
        label_sigCid: "country",
        tooltipLabels: {
          x_label: null,
          y_label: null,
          color_label: null,
          dotSize_label: p => "Population: " + d3Format.format(",")(p)
        },
        dotGlobalShape: "none"
      }]
    };

    return (
      <TimeContext initialIntervalSpec={
        new IntervalSpec("1953-10-30", "1966-04-01", null, null)}>
        <TimeRangeSelector/>
        <BubblePlot
          config={cnf}
          height={600}
          margin={{ left: 45, right: 5, top: 5, bottom: 40 }}
          maxDotCount={200}
          maxDotSize={30}
          minDotSizeValue={0}
          colorValues={["europe", "americas", "africa", "asia"]}
          xMinValue={0.5}
          xMaxValue={8.7}
          yMinValue={20}
          yMaxValue={87}
          xAxisLabel={"Fertility rate"}
          yAxisLabel={"Life expectancy"}
          withToolbar={false}
          zoomLevelMax={3}
          highlightDotSize={1}
        />
      </TimeContext>
    );
  }
}
```

D.2.2 JSX – with legend

```
'use strict';

import React, {Component} from "react";
import {BubblePlot, TimeContext, TimeRangeSelector, IntervalSpec} from "ivis";
import * as d3Format from "d3-format";
import * as d3Scale from "d3-scale";

export default class HansRoslingBubblePlot extends Component {
  constructor(props) {
    super(props);

    this.minDotSize = 2;
    this.maxDotSize = 30;

    this.state = {
      minPopulationValue: null,
      maxPopulationValue: null
    }
  }

  generateLegend() {
    const legend = [];
    if (this.state.minPopulationValue === null ||
        this.state.maxPopulationValue === null)
      return legend;

    const sizeScale = d3Scale.scaleSqrt()
      .domain([this.state.minPopulationValue, this.state.maxPopulationValue])
      .range([this.minDotSize, this.maxDotSize])
      .nice();
    const ticks = sizeScale.ticks(4);

    for (const [i, tick] of ticks.entries()) {
      const radius = sizeScale(tick);
      legend.push(
        <svg key={i} height={2*this.maxDotSize} width={tick === 0 ? 80 : 180}>
          <circle cx={radius} cy={this.maxDotSize} r={radius} fill={"#3d3d3d"}/>
          <text y={this.maxDotSize} x={2*radius + 10} dominantBaseline="middle">
            {d3Format.format(",")(tick)}
          </text>
        </svg>
      );
    }
    return legend;
  }

  computeExtents(base, processedResults, results, queries, additionalInformation) {
    const extents = BubblePlot.computeExtents(base, processedResults, results,
      queries, additionalInformation);
    const sizeExtent = extents[2];
    this.setState({
      minPopulationValue: sizeExtent[0],
      maxPopulationValue: sizeExtent[1]
    });

    return extents;
  }

  drawHighlightDot(base, record, selection, xScale, yScale, sScale, cScale) {
    selection.selectAll("circle").remove();
    if (record) {
      selection.append("circle")
        .attr('cx', xScale(record.x))
        .attr('cy', yScale(record.y))
        .attr("r", sScale(record.s))
        .attr("stroke", "#3d3d3d")
        .attr("stroke-width", 1)
        .attr('fill', cScale(record.d));
    }
  }

  render() {
    const cnf = {
      signalSets: [{
        cid: "top:gapminder",
        x_sigCid: "fertility_rate",
        y_sigCid: "life_expectancy",
        colorDiscrete_sigCid: "region",
        dotSize_sigCid: "population",
        tsSigCid: "year",
        label_sigCid: "country",
        tooltipLabels: {
          x_label: null,
          y_label: null,
          color_label: null,
          dotSize_label: p => "Population: " + d3Format.format(",")(p)
        },
        globalDotShape: "circle",
        getGlobalDotColor: color => color
      }]
    };

    const legend = this.generateLegend();

    return (
      <TimeContext initialIntervalSpec={
        new IntervalSpec("1953-10-30", "1966-04-01", null, null)}>
        <TimeRangeSelector/>
        <BubblePlot
          config={cnf}
          height={600}
          margin={{ left: 45, right: 5, top: 5, bottom: 40 }}
          maxDotCount={200}
          minDotSize={this.minDotSize}
          maxDotSize={this.maxDotSize}
          colorValues={["europe", "americas", "africa", "asia"]}
          xMinValue={0.5}
          xMaxValue={8.7}
          yMinValue={20}
          yMaxValue={87}
          xAxisLabel={"Fertility rate"}
          yAxisLabel={"Life expectancy"}
          withToolbar={false}
          zoomLevelMax={3}
          highlightDotSize={1}
          computeExtents={::this.computeExtents}
          drawHighlightDot={::this.drawHighlightDot}
        />

        <h5>Population:</h5>
        <div id={"legend"}
              style={{
                display: "flex",
                justifyContent: "center"
            }}>
          {legend}
        </div>
      </TimeContext>
    );
  }
}
```

D.3 Source code – Correlogram

This is source code for the example described in Section 7.3.

D.3.1 JSX

```
'use strict';

import React, {Component} from "react";
import {ScatterPlot, withPanelConfig, TimeContext, IntervalSpec, TimeRangeSelector}
  from "ivis";
import styles from './styles.scss';

class Correlogram extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    const scatterPlotProps = {
      withTooltip: false,
      withZoom: false,
      withRegressionCoefficients: false,
      withToolbar: false,
      withBrush: false,
      height: this.props.height / this.props.config.sigCids.length,
      margin: { left: 55, right: 3, top: 5, bottom: 20 },
      dotSize: 3,
      maxDotCount: 200,
      xAxisTicksCount: 8
    };

    const rows = [];
    for (const [i, sig1] of this.props.config.sigCids.entries()) {
      const row = [];
      for (const [j, sig2] of this.props.config.sigCids.entries()) {
        if (i !== j) {
          const cnf = {
            signalSets: [{
              cid: this.props.config.sigSetCid,
              x_sigCid: sig2,
              y_sigCid: sig1,
              dotGlobalShape: "none"
            }]
          };
          if (this.props.config.color_sigCid)
            cnf.signalSets[0].colorDiscrete_sigCid = this.props.config.color_sigCid;
          if (this.props.config.ts_signal)
            cnf.signalSets[0].tsSigCid = this.props.config.ts_signal;

          row.push(<ScatterPlot config={cnf} {...scatterPlotProps}/>);
        }
        else
          row.push(
            <div className={styles.label}
              style={{
                marginLeft: scatterPlotProps.margin.left,
                marginRight: scatterPlotProps.margin.right,
                marginTop: scatterPlotProps.margin.top,
                marginBottom: scatterPlotProps.margin.bottom}}>
              {this.props.config.labels[i]}
            </div>);
      }
      rows.push(row);
    }

    return (<table><tbody>
      {rows.map((r, i) => <tr key={i}>
        {r.map((sp, j) => <td key={j}>{sp}</td>)}
      </tr>)}
    </tbody></table>);
  }
}

@withPanelConfig
export default class TestCorrelogram extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    const config = this.getPanelConfig();

    const cnf = {
      sigSetCid: config.sigSet,
      sigCids: config.signals.map(x => x.sigCid),
      labels: config.signals.map(x => x.label),
      color_sigCid: config.color_signal,
      ts_signal: config.ts_signal
    };

    return (
      <TimeContext initialIntervalSpec={
        new IntervalSpec("2003-10-30", "2016-04-01", null, null)}>
        <Correlogram
          config={cnf}
          height={700}
          margin={{left: 0, right: 0, top: 0, bottom: 0}}
        />
        {config.ts_signal && <TimeRangeSelector/>}
      </TimeContext>);
  }
}
```

D.3.2 SCSS

```
.label {
  text-align: center;
}

table {
  width: 100%;
  margin-bottom: 15px;
}

td {
  width: 1%; /* trick to make all columns same width */
}
```

D.3.3 Parameters

```
[
  {
    "id": "sigSet",
    "label": "Signal Set",
    "help": "Select the desired signal set. We recommend the Gapminder dataset for
      this template.",
    "type": "signalSet"
  },
  {
    "id": "signals",
    "label": "Signals",
    "cardinality": "2..n",
    "type": "fieldset",
    "children": [
      {
        "id": "label",
        "label": "Label",
        "type": "string"
      },
      {
        "id": "sigCid",
        "label": "Signal",
        "type": "signal",
        "signalSetRef": "/sigSet"
      }
    ]
  },
  {
    "id": "color_signal",
    "label": "Color signal (category)",
    "cardinality": "0..1",
    "type": "signal",
    "signalSetRef": "sigSet"
  },
  {
    "id": "ts_signal",
    "label": "Time series signal",
    "cardinality": "0..1",
    "type": "signal",
    "signalSetRef": "sigSet"
  }
]
```

D.4 Source code – Scatter plot with legend

This is source code for the example described in Section 7.4.

D.4.1 JSX

```
"use strict";

import React, { Component } from "react";
import { ScatterPlot, Legend, withPanelConfig } from "ivis";

const signalSetsStructure = [
  {
    labelAttr: 'label',
    colorAttr: 'color',
    selectionAttr: 'enabled'
  }
];

const signalSetsConfigSpec = {
  "id": "years",
  "type": "fieldset",
  "cardinality": "1..n",
  "children": [
    {
      "id": "label",
      "label": "Label",
      "type": "string"
    },
    {
      "id": "color",
      "label": "Color",
      "type": "color"
    },
    {
      "id": "enabled",i
      "label": "Enabled",
      "type": "boolean",
      "default": true
    },
    {
      "id": "sigSetCid",
      "label": "SignalSet",
      "type": "signalSet",
      "help": "Choose one of the \"top:gapminder_YEAR\" signalSets."
    }
  ]
};

@withPanelConfig
export default class ScatterPlotWithLegend extends Component {
  constructor(props) {
      super(props);
  }

  render() {
    const config = this.getPanelConfig();

    const signalSets = [];
    for (const year of config.years) {
      let signalSetConfig = {
        x_sigCid: "income_per_person",
        y_sigCid: "life_expectancy",
        label_sigCid: "country",
        tooltipLabels: {
          label_format: (year, country) => country + ", " + year,
          x_label: "Income per person",
          y_label: y => "Life expectancy: " + y + " years"
        },
        regressions: [ {type: "linear"} ]
      };
      signalSetConfig.label = year.label;
      signalSetConfig.color = year.color;
      signalSetConfig.enabled = year.enabled;
      signalSetConfig.cid = year.sigSetCid;
      signalSets.push(signalSetConfig);
    }
    const cnf = {
      signalSets
    };

    return (
      <>
        <Legend label="Years" configPath={['years']} withSelector
          structure={signalSetsStructure} withConfiguratorForAllUsers
          configSpec={signalSetsConfigSpec}/>
        <ScatterPlot
          className={"asd"}
          config={cnf}
          height={500}
          margin={{ left: 45, right: 5, top: 5, bottom: 40 }}
          maxDotCount={100}
          dotSize={4}
          xAxisLabel={"Income per person"}
          yAxisLabel={"Life expectancy"}
        />
      </>
    );
  }
}
```

D.4.2 Parameters

```
[
  {
    "id": "years",
    "type": "fieldset",
    "cardinality": "1..n",
    "children": [
      {
        "id": "label",
        "label": "Label",
        "type": "string"
      },
      {
        "id": "color",
        "label": "Color",
        "type": "color"
      },
      {
        "id": "enabled",
        "label": "Enabled",
        "type": "boolean",
        "default": true
      },
      {
        "id": "sigSetCid",
        "label": "SignalSet",
        "type": "signalSet",
        "help": "Choose one of the \"top:gapminder_YEAR\" signalSets."
      }
    ]
  }
]
```

D.5 Source code – Synchronized views

This is source code for the example described in Section 7.5.

D.5.1 JSX

```
'use strict';

import React, {Component} from "react";
import {ScatterPlot, HistogramChart, withPanelConfig, TimeContext, IntervalSpec,
  TimeRangeSelector, MinMaxLoader} from "ivis";

@withPanelConfig
export default class SynchronizedViews extends Component {
  constructor(props) {
    super(props);
    this.state = {
      xMinValue: null,
      xMaxValue: null
    }
  }

  viewChanged(target, view, causedByUser) {
    if (!causedByUser)
      return;

    this.scatter.setView(view.xMin, view.xMax, undefined, undefined, target);
    this.histogram.setView(view.xMin, view.xMax, target);
  }

  processMinMaxResults(results) {
    const config = this.getPanelConfig();
    this.setState({
      xMinValue: results[config.x_sigCid].min,
      xMaxValue: results[config.x_sigCid].max
    });
  }

  extentWithMargin(min, max, margin_percentage) {
    const diff = max - min;
    const margin = diff * margin_percentage;
    return [min - margin, max + margin];
  }

  render() {
    const config = this.getPanelConfig();
    const color = "#448e7c";

    const scatter_config = {
      signalSets: [{
        cid: config.sigSet,
        x_sigCid: config.x_sigCid,
        y_sigCid: config.y_sigCid,
        tsSigCid: config.tsSigCid,
        label_sigCid: config.label_sigCid,
        color: color,
        dotGlobalShape: "circle"
      }]
    };
    const histogram_config = {
      sigSetCid: config.sigSet,
      sigCid: config.x_sigCid,
      tsSigCid: config.tsSigCid,
      color: color
    };
    const margin = {
      left: 40, right: 5, top: 5, bottom: 20
    };
    const [xMinValue, xMaxValue] =
      this.extentWithMargin(this.state.xMinValue, this.state.xMaxValue, 0.02);

    let charts;
    if (this.state.xMinValue === null || this.state.xMaxValue === null)
```

```
      charts = <div style={{textAlign: 'center'}}>No data.</div>
    else
      charts = (<>
        <ScatterPlot config={scatter_config}
                     height={400}
                     margin={margin}
                     maxDotCount={200}
                     dotSize={3}
                     xMinValue={xMinValue}
                     xMaxValue={xMaxValue}
                     withToolbar={false}
                     viewChangeCallback={::this.viewChanged}
                     ref={node => this.scatter = node}
        />
        <HistogramChart config={histogram_config}
                        height={200}
                        margin={margin}
                        xMinValue={xMinValue}
                        xMaxValue={xMaxValue}
                        topPaddingWhenZoomed={0.25}
                        viewChangeCallback={::this.viewChanged}
                        ref={node => this.histogram = node}
        />
      </>)

    return (
      <TimeContext initialIntervalSpec={
        new IntervalSpec("2003-10-30", "2016-04-01", null, null)}>
        <TimeRangeSelector/>

        <MinMaxLoader
          config={{
            sigSetCid: config.sigSet,
            sigCids: config.x_sigCid,
            tsSigCid: config.tsSigCid
          }}
            processData={::this.processMinMaxResults}
        />

        {charts}
      </TimeContext>);
  }
}
```

D.5.2 Parameters

```
[
  {
    "id": "sigSet",
    "label": "Signal Set",
    "help": "Select the desired signal set. We recommend the Gapminder dataset
      for this template.",
    "type": "signalSet"
  },
  {
    "id": "x_sigCid",
    "label": "X axis signal",
    "type": "signal",
    "signalSetRef": "sigSet"
  },
  {
    "id": "y_sigCid",
    "label": "Y axis signal",
    "type": "signal",
    "signalSetRef": "sigSet"
  },
  {
    "id": "tsSigCid",
    "label": "Time series signal",
    "type": "signal",
    "signalSetRef": "sigSet"
  },
  {
    "id": "label_sigCid",
    "label": "Labels signal",
    "type": "signal",
    "signalSetRef": "sigSet",
    "cardinality": "0..1"
  }
]
```