Visualizations for the web

Bram Stein
Abstract

Visualizations for the web

Bram Stein

This thesis report describes the implementation of a chart library for use in client-side browser visualizations. The library is completely written in the JavaScript programming language and supports five chart types: bar or column charts, line charts, histograms, scatter plots, and function plots. Function plotting is performed using interval arithmetic resulting in accurate and guaranteed correct function plots with one or two variables. Other charts are drawn according to the guidelines set out in Stephen Few's "Show Me the Numbers: Designing Tables and Graphs to Enlighten" book in order to create clear and consistent visualizations of data.

The design of the library is based on the concept of components and containers which are laid out using layout managers. Three layout managers are included by default: border, grid, and a flexible grid. The charts are drawn using the HTML5 Canvas element and maintain their own transformation stack in order to create crisp lines and shapes. The design of the library also contains several extension mechanism which enables the creation of custom components and chart plug-ins as well as customization of the colours and fonts used in the charts.

The library also extends the JavaScript language by adding support for several functional programming constructs. Examples of these constructs are: functional pattern matching, iterators such as map and reduce, mixins, getter and setter properties, function currying, and others.
## Contents

1 Introduction .................................................. 1  
  1.1 Background ................................................. 1  
  1.2 Goals & Limitations ........................................... 2  

2 Architecture & Design ....................................... 3  
  2.1 Core ........................................................... 3  
    2.1.1 Array extensions ........................................ 3  
    2.1.2 Function extensions ..................................... 4  
    2.1.3 Object extensions ....................................... 4  
    2.1.4 Mathematics extensions ................................ 5  
    2.1.5 Getter and Setter Properties ......................... 6  
    2.1.6 Functional Pattern Matching ......................... 6  
  2.2 Layout ....................................................... 9  
    2.2.1 Border layout .......................................... 10  
    2.2.2 Grid layout ............................................. 10  
    2.2.3 Flex grid layout ....................................... 11  
  2.3 Axis .......................................................... 11  
  2.4 Title .......................................................... 12  
  2.5 Legend ......................................................... 12  
  2.6 Canvas ........................................................ 12  
  2.7 Data ........................................................... 14  
  2.8 Graphics ....................................................... 16  
    2.8.1 Custom view-ports ..................................... 16  
    2.8.2 Crisp line drawing .................................... 17  
    2.8.3 Fonts/Text support .................................... 17  
  2.9 Defaults ....................................................... 18  
  2.10 Chart ........................................................ 19  

3 Chart types .................................................. 21  
  3.1 Bar Chart ..................................................... 22  
  3.2 Scatter Chart ............................................... 25  
  3.3 Line Chart .................................................... 26  
  3.4 Histogram ..................................................... 27  
  3.5 Function plotting .......................................... 27  

4 Related work ................................................ 31
5 Future work

5.1 Scalable Vector Graphics .................................................. 33
5.2 Server Side Charts ............................................................ 33
5.3 Support for more chart types ............................................. 34
5.4 Interactive charts ............................................................ 34
5.5 Web page integration ....................................................... 34
5.6 ASCII Math parser .......................................................... 35

6 Conclusion ................................................................. 37
List of Figures

2.1 Binary tree ................................................................. 8
2.2 Border layout with all five components present ......................... 10
2.3 Grid layout with four components ...................................... 11
2.4 Flex grid layout with six components .................................. 11
2.5 Different cartesian axis configurations .................................. 13
2.6 Polar coordinate system .................................................. 13
2.7 Grid for cartesian and polar coordinate systems ....................... 13
2.8 Outlines of the data area and surrounding area ....................... 14
2.9 Approximated line and crisp drawn line ................................ 18

3.1 Bar chart without sub categories ....................................... 23
3.2 A ranking bar chart ....................................................... 23
3.3 A bar chart with subcategories and legend ............................ 24
3.4 A bar stacked bar chart .................................................. 25
3.5 Scatter chart with multiple categories .................................. 25
3.6 Line chart with single category ........................................ 26
3.7 Line chart with multiple categories ..................................... 27
3.8 Histogram with individual axis labels ................................... 27
3.9 Function plots in Cartesian and polar coordinates .................... 28
3.10 Quadtree generated by the interval arithmetic algorithm ............. 29
Chapter 1

Introduction

This thesis report describes the design and implementation of a browser-based chart library written in JavaScript. The report consists of two main parts, which closely correspond to the structure of the chart library. The second chapter is a discussion of the architecture and design of the core components of the library. The third chapter is a discussion and showcase of the chart "plug-ins" that are built using the core components. The fourth chapter contains an overview of other open source web visualization and chart libraries written in JavaScript. The fifth chapter discusses future enhancements to the library, and the sixth and final chapter contains the pro’s and con’s of the library and other concluding thoughts.

1.1 Background

The FIRST system (Facts on International Relations and Security Trends) is a free-of-charge system for politicians, journalists, researchers and the interested public\(^1\). The FIRST system contains documented information from research institutes around the world. It covers areas in the field of international relations and security, such as hard facts on armed conflicts and peace keeping, arms production and trade, military expenditure, armed forces and conventional weapons holding, nuclear weapons, chronology, statistics and other reference data. FIRST is a joint project of the International Relations and Security Network (ISN) and the Stockholm International Peace Research Institute (SIPRI).

The data are stored in roughly 50 different datasets. These datasets range from relational databases to HTML or XML generated by partner institutes in other geographic locations. The FIRST system provides an easy to use web-based interface and offers transparent search methods for all of these datasets.

A new version of the FIRST system has been under development since early 2007. It features a completely redesigned data back end based on the REST architecture [14], support for multiple languages, and a new dataset plug in system. It also exports data and meta-data in the Resource Description Format (RDF) [3] and various other formats generated from RDF, such as XHTML and JavaScript Object Notation (JSON) [7]. The user interface built on this new system is however heavily influenced by the user interface of the old system, and thus unable to showcase the full power of the new back end. A new user interface is currently in the design phase and calls for the development of a data visualization library with support various types of charts. This thesis project involves the design and implementation of such a chart library.

\(^1\)Facts on International Relations and Security Trends website: http://first.sipri.org/
1.2 Goals & Limitations

The goal of this thesis project is to create a chart library prototype which meets the following requirements:

- the library should run client side, using JavaScript;
- visualizations are implemented using the HTML5 Canvas element;
- the input for charts should be either a XHTML table, or a JSON object;
- the following chart types must be supported: bar charts, line charts, and scatter plots.

Beyond these technical requirements, the visualizations should also be visually pleasing and meet the guidelines set out in the book "Show Me the Numbers, Designing Tables and Graphs to Enlighten" [11].

The prototype only has to work under controlled circumstances, i.e. targeting a specific browser or operating system (preferably Mozilla Firefox) is acceptable.
Chapter 2

Architecture & Design

The architecture of the chart library is based on the concept of modules; pieces of self-contained code that interact with each other through public interfaces\(^1\). The chart library is split in two sections: one part provides the core functionality for creating charts in the form of modules, and the other consists of a set of plugins that actually implement the desired chart functionality using the core modules. The core modules are described in this chapter, while the actual chart plugins will be described in the next chapter.

2.1 Core

The core module extends the JavaScript language with various new features and helper functions. Due to its prototypical nature, the core JavaScript language is easily and naturally extended. The extensions fall in the following categories:

- Array extensions
- Function extensions
- Object extensions
- Mathematics extensions
- Support for getter and setter properties
- Functional pattern matching

2.1.1 Array extensions

The \texttt{Array} prototype is extended in two ways: support for JavaScript’s so-called array extra’s introduced in version 1.6 and 1.8 of JavaScript [5, 6], and a set of additional helper functions to deal with common array operations. The array extra’s includes functions like: \texttt{forEach}, \texttt{map}, \texttt{reduce} (equal to \texttt{foldr} and \texttt{foldl} in functional programming languages), \texttt{filter}, \texttt{every}, \texttt{some}, and several other methods. The array extra functions are new—but unofficial—extensions to the JavaScript language defined by the Mozilla developers. Nevertheless these functions have wide support in browsers such as Firefox, Opera, Chrome and Safari. Unfortunately, Internet

---

\(^1\)This is similar to object oriented programming. I have not used that term here to avoid confusion with classical object oriented programming, which is not supported in JavaScript in favour of prototypical object oriented programming. For more information on classical object oriented programming versus prototypical object oriented programming see [8].
Explorer does not support them. The core array extra module implements these functions in pure JavaScript and adds them to the Array prototype if they are not already natively supported. This enables the use of the array extra methods in all browsers.

An example of the reduce method introduced by the array extra’s module follows:

```javascript
function add(a, b) {
    return a + b;
}

[1, 2, 4, 1, 6, 5].reduce(add, 0); // returns 19
```

Without the array extra’s module this would have to be implemented using a for loop and an accumulation variable.

The additional helper functions included in the module are methods that are strangely omitted from the JavaScript specification, such as a peek function (pop and push are available), contains to check if an array contains a value, and an isEmpty method to test for empty arrays.

### 2.1.2 Function extensions

The function module extends the Function prototype with a small number of useful functions. The most notable functions are bind, curry, and defaults. Most of these function names are self-descriptive, but a short example of each follows.

Usually, object methods in JavaScript have a this property referring to the instance the method is operating on. The bind method allows binding the this property of a method to any object. Sometimes it may also be useful to bind a function to an object as in the following case:

```javascript
function print() {
    console.log(this.firstname + " " + this.lastname);
}
var myPrint = print.bind({ firstname: 'Bender', lastname: 'Rodriguez' });
myPrint(); // prints "Bender Rodriguez"
```

Given the add function used in the Array extensions sub chapter:

```javascript
var add1 = add.curry(1);
add1(3); // returns 4
```

The defaults function is similar to curry except that it only ”curries” a function when one or more of the arguments are missing. In effect this adds default values to optional function parameters. For example:

```javascript
var myAdd = add.defaults(1, 1);
myAdd(); // returns 2
myAdd(4); // returns 5
myAdd(2, 6); // returns 8
```

### 2.1.3 Object extensions

In contrast to the other extension modules, the object module does not extend the Object prototype, which is considered bad practice [2]. Instead the module extends the the global Object object. The consequence is that methods in the object extension module should be called with an Object prefix and have the object they operate on passed in as the first argument. So instead of:
We will need to write:

```javascript
Object.someMethod(obj);
```

The most important function in the object module is `extends`, which takes in two or more objects and copies all properties from the second (or third, fourth, etcetera, depending on how many arguments were passed in as arguments) to the first object. If there is a conflict in property names (i.e. the objects have one or more property names in common) it is first checked if the original property is a built-in property in which case it is kept. Otherwise it is overwritten by the last property of that name. The `extends` function is used extensively to extend the properties and prototypes of other objects.

The array extra’s functions are very useful in dealing with arrays. Unfortunately they do not work transparently on objects. For that reason, the object module introduces the same methods contained in the array extra module but tailored to deal with objects. This means that functions such as `forEach`, `map`, `reduce`, `filter`, `every`, and `some` can also be used on objects. The functionality and interface is identical to the array extra functions.

Other functions added to the global `Object` object are designed for more accurate type checking. Although JavaScript is dynamically typed, it is sometimes useful to check for the type of a variable, as the built-in `typeof` has some rather severe problems\(^2\). There are functions for checking the types of: atoms, numbers, strings, booleans, arrays, and functions. Combined with the array or object extra extensions, it is very easy to do input validation. For example, to check if all values in an array are of the type `number` we can just write:

```javascript
// returns false (third argument is a string)
[1, 2, '3'].every(Object.isNumber);
```

If this is a common desire, we could also add a new function by extending the `Array` prototype and currying the `every` method with the `Object.isNumber` method.

```javascript
Object.extend(Array.prototype, {
  isNumber: this.every.curry(Object.isNumber)
});
```

```javascript
// this now works (still returns false)
[1, 2, '3'].isNumber();
```

It is however recommended to keep the number of extensions to the built-in objects to a minimum to prevent name clashes and other interoperability problems.

### 2.1.4 Mathematics extensions

The math module extends the `Math` prototype with a set of new methods for rounding, generating correct random integers [16], and measuring number properties such as accuracy, precision and the number of digits.

The math module also implements interval arithmetic [24]. Interval arithmetic is a method to guarantee upper and lower bounds on imprecise calculations. For example, rounding errors often occur in computer arithmetic due to having limited floating point precision available in computers. Instead of returning an imprecise exact number to a calculation, interval arithmetic

\(^2\)For example `typeof Array` returns "object".
returns an interval which contains the solution. The calculation can then be performed again if more precision in the results is required.

The interval sub-module implements normal arithmetic functions (add, subtract, multiply, etcetera) as well as sinusoids, logarithms, powers, exponents, and square roots. Additionally it defines a set of helper functions for determining equality of intervals and inclusion tests. An interval is defined in the module as a JavaScript object with a `to` and `from` property which represents the lower and upper bound of the interval respectively.

The chart library uses the interval module extensively, for example as data ranges, or in plotting functions. Data plotting using interval arithmetic is described in a later chapter.

### 2.1.5 Getter and Setter Properties

The property module adds support for adding getter and setter properties to any object. In fact, it acts as a factory for creating functions that add getter and setter properties to objects. A getter and setter property is defined as a method that, when called without a parameter, returns the value of its property and when called with a parameter to set the value of the property to that value.

This allows developers to define standard properties and mix them into objects that they desire to have those properties. Instead of redefining an often used property many times, it is created once and added to other objects as desired [8].

```javascript
// create the property name, parameters, and default values
var addSize = property('size', {
    width: 100,
    height: 100
});

// add the property to a new (empty) object
var myObject = addSize({});

// returns {width: 100, height: 100}
myObject.size();
// sets width to 200 and returns {width: 200, height: 100}
myObject.size({width: 200});
```

The example also shows that by taking in the "setter" value as an object, the property is not restricted to a fixed number of arguments or the order of arguments.

### 2.1.6 Functional Pattern Matching

Pattern matching is a form of conditional branching which allows concise matching of data structure patterns and binding variables at the same time [38]. Pattern matching is supported in some functional languages such as ML, Haskell, OCaml, and Erlang. The `fun` module implements pattern matching for the JavaScript language in an efficient and concise way. The following is an example of pattern matching in JavaScript using the `fun` module:

```javascript
var fact = fun(
    [0, function () 1],
    [$, function (n) n * fact(n - 1)]
);
```
When `fact(10)` is called the value 10 is matched against the first pattern `0`. This match fails and the next pattern is evaluated. The `$` in the next pattern is an example of a parameter. A parameter matches anything, so the match succeeds and 10 is passed as an argument to the anonymous function. Since this is a recursive function it will match the second pattern until the argument to the function reaches zero and then terminates. Note that this example uses JavaScript 1.8 syntax, code in previous JavaScript versions will be slightly more verbose.

Another common use of pattern matching is to determine if a value is of a certain type and perform an action depending on the result. For example, say we have a `print` function which logs its value to the console. We would however like to customize the output for some data types. We can accomplish this using pattern matching as follows:

```javascript
var print = fun(
    // match and print Date values
    [Date, function (d) ...],

    // match and print String values
    [String, function (str) ...],

    // match and print any other type
    [$, function (o) ...]
);
```

If the type of the value is Date, the first anonymous function will be executed and the value passed as argument. The same applies to values of type String. Any other value will be passed to the last anonymous function whose pattern acts as a catch-all.

Object extraction

Patterns can also contain wild-cards, which can be used to mask out or ignore certain parts of the value it is matched against. This can be used to concisely extract properties for large and deeply nested objects. For example:

```javascript
var data = {
    label: 'Plot #1',
    type: 'points',
    values: [2.1, 5.6, 2.4, 3.4]
};

var parse = fun(
    [{label: _, type: _, values: $}, function (v) console.log(v)]
);

parse(data); // prints out [2.1, 5.6, 2.4, 3.4]
```

The wild-card symbol in this example is the underscore "_" used to mask out the `label` and `type` properties of the `data` object.

Algebraic data types

Pattern matching can also be used with Sjoerd Visscher's Algebraic data type library [36]. In the following example we define a simple binary tree data type. The tree can contain either `Void` or a `Bt` tuple. The tuple consists of a value, and two branches called left and right. In the ML programming language we would define a polymorphic binary tree like this:
datatype 'a binarytree = Void | Bt of 'a * 'a binarytree * 'a binarytree

In JavaScript—using the ADT library—it looks like this (note that this binary tree is not polymorphic, its values are numbers.)

```javascript
var BinaryTree = Data(function (binarytree) {
    Void : {},
    Bt: { v: Number, L: binarytree, R: binarytree }
});
```

We can then create a simple binary tree instance using this definition. A visual representation of this binary tree is shown in Figure 2.1.

```javascript
var bt = Bt(4,
    Bt(2, Bt(1,Void,Void),
        Bt(3,Void,Void)),
    Bt(8, Bt(6, Bt(5,Void,Void),
        Bt(7,Void,Void)),
        Bt(9,Void,Void)));
```

![Binary Tree Diagram](image)

Figure 2.1: Binary tree

We can now define various functions, for example to calculate the number of leafs (nodes without children) in the tree, or a function to test whether a value is a member of the binary tree. It is possible to use the data types in patterns, for example the `numLeafs` method first matches on an empty node, then a leaf node and finally on any other kind of node. The `isMember` method shows that the matching is not limited to one function parameter; the value to search for is taken as the first parameter and the binary tree as the second parameter.

```javascript
var numLeafs = fun(
    [Void, function() 0], // empty node
    [Bt(_, Void, Void), function() 1], // leaf node
    [Bt(_, $, $), function(L, R) numLeafs(L) + numLeafs(R)]
);

var isMember = fun(
    [_, Void, function() false],
    [$, Bt($, $, $), function(x, v, L, R) x === v ||
        (isMember(x, L) || isMember(x, R))]
);
numLeafs(bt); // 5
isMember(10, bt); // false
isMember(3, bt); // true

The following two functions return a list of all the elements using in order and pre order traversals of the binary tree.

```javascript
var inorder = fun(
  [Void, function() []],
  [Bt($, $, $), function(v, L, R) inorder(L).concat([v], inorder(R))]
);

var preorder = fun(
  [Void, function() []],
  [Bt($, $, $), function(v, L, R) [v].concat(preorder(L), preorder(R))]
);

inorder(bt); // [1,2,3,4,5,6,7,8,9]
preorder(bt); // [4,2,1,3,8,6,5,7,9]
```

A real implementation of a binary tree would of course not use the data types and functions defined in this article for performance reasons, but a binary tree serves as a good introduction to both pattern matching and algebraic data types in JavaScript.

### 2.2 Layout

The layout module provides layout algorithms for laying out components. A component is an abstraction; it can be implemented in many ways, for example as items in a HTML5 Canvas drawing or as HTML elements. The module currently provides three layout algorithms: border, which lays out components in five different regions; grid, which lays out components in a user defined grid, and flex-grid which offers a grid with flexible column and row sizes.

The design of the layout module is inspired by the Java Swing layout managers [22] and my own OpenGL based User Interface library [29]. The module itself is not part of the core charting library but distributed as a separate library [30] and bundled with a jQuery plug-in for laying out HTML elements.

We start with the definition of a component; a component is something that has a minimum size, a preferred size, and a maximum size. It also has a method to set its size and position. A container is a component that contains other components and lays them out according to a layout algorithm (which are provided in the layout module.) Components need to satisfy the following interface requirements in order to be used with the layout algorithms; the component should have a:

- **bounds** property which returns the location and dimension of the component;
- **preferredSize**, **maximumSize**, and **minimumSize** properties which return the preferred, minimum and maximum size of the component respectively;
- **insets** property which returns the offset between a container and its contents;
- **isVisible** method which returns true if the component is visible;
• `doLayout` method which might call a nested layout manager.

Note that the distinction between containers and components is artificial, both implement the same interface.

2.2.1 Border layout

The border algorithm lays out components in five different regions. These regions are called center, north, south, east and west. The center component will be laid out in the center of the container, north on top of it, south beneath it and west and east on the left and right side respectively. The layout can only contain one of each region, but all are optional. Figure 2.2 shows a visualization of a layout using all five regions.

![Figure 2.2: Border layout with all five components present](image)

The following example lays out a west, center and north component with a vertical space of 5 units between each component. There may be additional space between the components and the container if the container returns non-zero insets.

```javascript
var borderLayout = jLayout.border({
  west: myWestComponent,
  center: myCenterComponent,
  north: myNorthComponent,
  vgap: 5
});

borderLayout.layout(myContainer);
```

If a region is not specified or the component is not visible its space will be taken up by the other components.

2.2.2 Grid layout

The grid algorithms lays out the components in a grid, and resizes each component to the same size. The number of columns and rows can be specified by the user. Figure 2.3 shows a visualization of a grid layout with four components in a 2x2 grid.

The following example lays out four components in a 2x2 grid, without any spacing between the components.

```javascript
var gridLayout = jLayout.grid({
  rows: 2,
});
```
columns: 2,
items: [myComponent1, myComponent2, myComponent3, myComponent4]
});

gridLayout.layout(myContainer);

![Grid layout with four components](image)

Figure 2.3: Grid layout with four components

If the number of rows is given, the number of columns is calculated automatically by taking the total number of components into account. If the number of rows is not given (or set to zero), and the number of columns is given, the number of rows will be automatically calculated using the number of components. If neither is given, the number of rows is set equal to the number of components and the number of rows is set to zero.

### 2.2.3 Flex grid layout

The flex grid algorithms lays out the components in a grid with flexible row and column sizes. The number of columns and rows can be specified by the user. Figure 2.4 shows a visualization of a flex grid layout with six components in a 3x2 grid.

![Flex grid layout with six components](image)

Figure 2.4: Flex grid layout with six components

The interface to the flex grid is identical to that of the grid layout, and thus not further explained.

### 2.3 Axis

The axis module is a factory method for creating an object representation of a chart axis. The resulting object can represent two different kinds of axes: a numeric axis or a categoric axis.
Axis themselves are not components that can be used by a layout manager. Instead they are parameters to the canvas module, which draws them (and is a component that can be laid out.)

To create a numeric axis, the user supplies the axis module with an interval indicating the range the axis should span. The user can also optionally supply either an array of minor and major tick mark locations, or the desired number of minor and major tick marks. When a number is supplied the axis module automatically calculates the tick locations in a visually pleasing manor, implemented using an algorithm described in the "Graphics Programming Gems" series [19].

The algorithm boils down to choosing tick mark locations that have "nice" round numbers starting with either 1, 2, or 5. The algorithm favours these numbers over others. If such numbers are not desired, the user could always supply the axis module with custom values instead of having them automatically calculated based on the interval.

A categoric axis can be created by supplying the axis module with an array of labels. The axis module will then create an axis with an empty interval and use the supplied categories as labels.

All axes can also have a custom set of labels for the tick marks, and a label for the axis itself.

### 2.4 Title

The title module creates a title component, which simply draws a title and subtitle string on the screen. Both values are optional. No title or subtitle will be drawn if they are not supplied. Font information is retrieved from the default settings.

### 2.5 Legend

The legend module creates a component capable of drawing legend items. It supports the following types of legend items: point (circle, cross, triangle, diamond, and square), line or bar. These three types are sufficient to represent all chart types currently supported in the library.

The legend component maintains a grid layout internally to lay out the individual legend items. This means the the items in a legend are dynamically arranged. If the legend is placed in a vertical position, the items will be laid out in one column, or given enough space multiple columns. When the legend is placed in a horizontal position, for example on top of a chart, the legend items will naturally "flow" into a row formation.

### 2.6 Canvas

The canvas module is responsible for setting up view-ports and drawing axes, labels, and an optional grid. It supports three different types of axes: Cartesian, polar and categorical (which is a special instance of a Cartesian axis.) Depending on the axes and drawing options given to the canvas module, it sets up a correct view-port and draws axes, tick marks, labels and the grid.

The canvas module accepts an options object which should at least contain two Cartesian axes, or a polar axis. It might optionally also accept a hint for the aspect ratio of the canvas, and various boolean settings for turning on or off the drawing of—for example—axis lines, tick marks, labels, etcetera. Although all drawing options can be changed by the user, the defaults are chosen sensibly so as to cover most of the use cases. This results in the default visual output for Cartesian axes shown in Figure 2.5.
A single polar axis determines the radius of the polar coordinate system. The axis is drawn in the horizontal position and duplicated in the vertical position. A visual representation of a polar coordinate system is shown in Figure 2.6.

Both coordinate system do not draw a grid by default. Figure 2.7 shows how grid lines are drawn when they are explicitly enabled. Note that the user can also choose to only draw the horizontal or vertical grid lines.
Because the canvas is a component, it has a minimum and a preferred size. The minimum size of the canvas is calculated by the width and height of the labels of its major tick marks, plus some additional spacing between the labels. A canvas’ preferred size is the size it would like to have to optimally display data. This includes the minimum size multiplied by the preferred aspect ratio. A canvas’ maximum size is a user defined setting of the maximum width and height the chart may attain. By default, a chart will try to use exactly the space given to it by its drawing area unless it violates either the minimum or maximum size. In this case the drawing area will be resized.

The aspect ratio defines the dimensions of a chart. The ratio itself is defined by either the horizontal and vertical axis or set by the user. If no ratio is given the library calculates it according to several heuristics aiming to reach an optimal 1 : 1 ratio for most charts (other charts might have the golden ratio \[37, 11\] set as default optimal value.) In order of importance, the heuristics are:

- the resolution of the data (i.e. a maximum of one data point per screen pixel)
- number and width of the axis labels
- available space

The total size of a canvas is divided into two areas: a data area surrounded by padding necessary to contain all the labels. Figure 2.8 shows the data area marked by a red outline and the surrounding area marked with a blue outline.

![Figure 2.8: Outlines of the data area and surrounding area](image)

2.7 Data

The data module defines several data relationships and their corresponding input formats and validation routines. The chart plug-ins use these relationship definitions to declare the input they accept. Because not every data relationship can be predefined the data module is made extensible so that custom relationships can be constructed while still using some of the built in functionality.

A chart plug-in is not limited to accepting a single data relationship, it can declare an array of data relationships. The following list shows the definitions the available data relationships [12]:

**Nominal comparison** To compare quantitative values for one or more categories.
Time series To display a relationship between quantitative values belonging to one or more categories and subdivisions of time.

Distribution To display the distribution of quantitative values over a range.

Correlation To display the correlation between a set of quantitative values.

Function To display an equation visually.

The three relationships which are not mentioned are: part to whole, ranking, and deviation. They have no explicit relationship type as they can be defined in terms of the other relationships. For example, deviation is a nominal comparison to a baseline, ranking is an ordered nominal comparison (ordinal), and part to whole is a nominal comparison with sub categories.

The data module accepts a single data object which contains both the data and meta-data. The meta-data properties consists of categories and subcategories, which are both optional. The overall structure of the data object is as follows:

```javascript
var data = {
    categories: ["...",
    subcategories: ["...",
    data: ["...
```;

All of the properties must be arrays. Both category and subcategory must not contain nested arrays. Sub-categories are the same for each category, which is enforced by the design of the data object. Although categories and sub-categories are nominal—and thus have no order—arrays are used because it is often desired to order them in a user defined way. The data format has no intrinsic time series type either, but because categories are ordered they can be used for that purpose. The data property can contain nested arrays, depending on if, and how, the categories and subcategories properties are used.

```javascript
var data = {
    categories: ['West', 'East'],
    data: [10, 12]
};
```

When subcategories is specified the data looks as follows, to—for example—define a time series with categories:

```javascript
var data = {
    categories: ['2001', '2002'],
    subcategories: ['West', 'East'],
    data: [
        [131, 120],
        [119, 141]
    ]
};
```

When none of the above rules meet the data, nested arrays are treated as multiple variables. For example, when the following data is given to chart that supports multiple variables it will be treated as x, y coordinates in a single category.

15
Normal JavaScript functions can be used to plot functions:

```javascript
function sine(x) {
    return Interval.sin(x);
}
```

The function is called using an interval and must also return an interval. Function plotting is explained in more detail in Chapter 3.5.

2.8 Graphics

The graphics module is an abstract layer built on top of the HTML5 Canvas API. It abstracts away less often used methods and implements higher level functionality such the ability to create view-ports and to draw points in various shapes.

The API is built on the chaining principle; methods return the object on which they were invoked. This allows for a very succinct way of drawing graphics:

```javascript
graphics.
    beginPath().
    moveTo(10, 10).
    lineTo(20, 20).
    endPath().
    stroke('#00000');
```

The graphics API has two basic types of methods, one to create shapes and the other to create paths. Shapes include the following methods: `beginViewport, closeViewport, beginPath, stroke, fill, rect, line, points` (i.e. `circle, triangle, cross, diamond, etc.`), and `text`. Paths include the following methods: `lineTo, moveTo, arcTo, bezierCurveTo, quadraticCurveTo, closePath, and endPath`. Shapes can be considered ready-made instances of paths (although the implementation might differ) and they define a higher level interface to drawing complex paths.

The graphics object does not expose path methods directly, instead the `beginPath` method returns a set of functions that can create paths. Once the user is creating a path there is no way—other than by calling either the `closePath` or `endPath` methods (which returns the shapes set of methods)—to return to the shape group. This enforces the correct creation of paths and shapes (i.e. the API ensures that only valid method call combinations are made.)

2.8.1 Custom view-ports

The chart library implements its own transformation stack instead of using the one provided by the HTML5 Canvas API. The reason for this is to have complete control over how lines are drawn. This can only be achieved by not using the built-in transformation stack. An example of one of the problems is that line width is not implemented consistently across browsers and
transformation states. Some browsers will use the line width associated with the transformation matrix when the path was created, while others will use the line width associated with the transformation matrix when the path was stroked. The following HTML5 Canvas example demonstrates this problem.

```javascript
ctx.lineWidth = 2;
ctx.save();
ctx.scale(1.5, 1.5);
ctx.lineTo(10, 10);
ctx.restore();
ctx.stroke();
```

Some implementations will stroke the path with a line width of 2, while others will use a line width of 3 (2 scaled by 1.5). Unfortunately—and unlike other graphics APIs, such as OpenGL [27]—the HTML5 Canvas element does not specify a stack for rendering properties such as line width. To overcome this problem, it was necessary to implement a custom transformation stack. The graphics module supports this through two methods: `beginViewport` and `closeViewport`. Instead of offering more general `scale` and `translate` methods, these methods deal with the concept of a view-port: a two dimensional area with its own custom coordinate system. The `beginViewport` method takes four required parameters (`x`, `y`, `width`, and `height`) and three optional ones (a horizontal interval, a vertical interval, and a polar boolean.) When the horizontal and vertical intervals are specified, all drawing between the `beginViewport` and `closeViewport` method calls are performed in those intervals. This simplifies drawing data points by removing the need for transforming the data into the charts’ view-port; data can simply be plotted to the screen without transformation. View-ports can also be nested without problems.

The last optional parameter is polar, which, when turned on, transforms the coordinates to all drawing methods from polar to Cartesian. This makes it possible to also transparently plot polar data.

### 2.8.2 Crisp line drawing

Although it has support for some vector operations, the HTML5 Canvas element is not a vector based API, it basically draws bitmaps. As such, it becomes a necessity to manually ensure that lines are drawn on screen pixels instead of on pixel boundaries resulting in an approximation, or blurred lines [25]. Figure 2.9 shows two lines on a pixel grid. The red lines indicate the vector representation of the lines, and the blue filled area the actual screen pixels. The first line is drawn on pixel boundaries and thus results in an approximation, the right line on the other hand is drawn at exactly the center of a pixel and thus results in a crisp line drawing.

The graphics module internally ensures that all lines and rectangles are drawn on screen pixels and never on the boundaries. As this "rounding" to screen pixels is done consistently, the resulting image is not affected, other than being slightly moved up or down (depending on the implementation of the rounding.)

### 2.8.3 Fonts/Text support

Unfortunately font and text drawing support in the HTML5 Canvas element is not well developed at the moment. Only Firefox 3.1 and Safari implement a standardized text drawing API. Firefox also offers its own alternative text drawing API. Both Internet Explorer and Opera have no support for drawing text on a HTML5 Canvas (Internet Explorer does in fact not support the Canvas element at all.) The standardized text API is also severely lacking in functionality. For example, it is possible to retrieve the width of a string in pixels, but not the height. Text is not
Figure 2.9: Approximated line and crisp drawn line

treated as a set of paths, but drawn directly on the canvas through two separate `fillText` and `strokeText` functions. Strangely enough, the alternative Firefox text API does offer support for treating text as paths, and supports stroking or filling those paths. Direct drawing versus paths is not a big issue in the development of this chart library, but the font metrics problems are a big issue when aligning labels, legends and titles.

In order to overcome these problems the chart library uses the HTML DOM to insert a properly stylized string into an off-screen location in the surrounding HTML page and retrieve correct font metrics that way. In future versions of the library it might be possible to replace this with a better standardized and implemented HTML5 Canvas text API. Alternatively, it would be possible to overlay absolute positioned HTML elements on top of the HTML canvas [28], use a bitmap font, or use one of the available path based fonts [32, 9, 18].

2.9 Defaults

The defaults "module" is not really a module, but a public object containing the default settings used in rendering charts. This includes colours, fonts, spacing between labels, point types and a list of all the available chart types. Users of the library can override any value in this object to suit their own needs.

The default colours come in two varieties: qualitative and diverging[13]. The qualitative set is used for displaying sub categories, to separate the items into distinct groups. The diverging set is used for encoding ranges, for example low to high. Both varieties have three subsets, one for highlighting values with strong hues and one for non-highlighted data with medium intensity hues. The last subset is a gray-scale version for printing. The hues themselves are taken from Cynthia Brewer’s ColorBrewer application\(^3\), which features many qualitative, sequential and diverging color palettes.

Other colours are kept muted [33, 13, 31] so as to not distract from the data itself. Fonts default to simple black text, 11px in size and sans serif.

\(^3\)http://www.ColorBrewer.org by Cynthia A. Brewer, Geography, Pennsylvania State University.
2.10 Chart

Each chart plug-in is an instance of the chart super class\textsuperscript{4}. The chart module takes care of the repetitive tasks, such as maintaining a layout, validating data, setting up a graphics context, parsing user options and so forth. Chart plug-ins can "inherit" from the chart module in the following way:

\begin{verbatim}
Object.extend(defaults.type, {
    myChart: function (canvasIdentifier, data, options) {
        var that = {},
            my = {};

        that = chart(canvasIdentifier, options, my);

        Object.extend(that, {
            plot: function (g) {
                // draw the chart
            }
        });
        return that;
    }
});
\end{verbatim}

This example first extends the \texttt{defaults.type} object discussed in the last section with a new chart type called \texttt{myChart}. It then creates two empty objects, one for the chart plug-in itself and one for any shared private (protected in classical inheritance) called \texttt{my}. The \texttt{that} object is then initialized using the chart constructor. The chart constructor parses the options and returns a chart component. The code then continues by overriding the plot function, which is called internally by the chart when it is drawn. When the plot function is called, the chart has already set up a view-port and drawn the axes and labels. This makes it especially easy to create new chart plug-ins. Plug-in authors can "inherit" from the chart module, perform any custom data parsing or validation and draw their chart in the plot function.

\textsuperscript{4}The word \textit{class} here does not imply classical inheritance, but rather a module that shares the same methods and some state with another module.
Chapter 3

Chart types

Charts can be created by calling the appropriate chart plug-in constructor with: the identifier for the canvas element to use, the data, and an options object. The following example shows how to create a simple bar chart, add a title and subtitle, and set its size.

```javascript
var c = chart.bar('canvas', data, {
  title: 'Q1 Sales',
  subtitle: 'By Region'
});

c.bounds({
  width: 200,
  height: 100
});

c.draw();
```

Because each chart plug-in is a "subclass" of the chart component, the `preferredSize`, `minimumSize`, `bounds`, and other methods are available and can be used to—for example—set a chart to its preferred size (taking the aspect ratio into account.) The `bar` plugin name in the above example can be replaced by any of the following chart plug-ins that come with the library:

- **bar** The bar chart plug-in supports categories and subcategories, in both horizontal and vertical orientations. It can also display stacked bar charts.

- **scatter** The scatter chart plug-in supports single or multiple categories with two variables.

- **line** The line chart plug-in supports categories and subcategories.

- **histogram** The histogram supports multiple categories but not subcategories.

- **plot** The plot chart plug-in supports plotting a single function with one or two variables.

Using these chart plug-ins it is possible to display all the relationships defined in Chapter 2.7. The following list describes how each relationship maps to a plug-in appropriate for displaying that relationship.

**Nominal comparison** Nominal comparisons should almost always be encoded as a bar chart.
**Time series** When there are few time subdivisions and many categories, a line chart with a line for each category is a good solution. When there are few time subdivisions and few categories a bar chart should be used. Finally, time series with many time subdivisions should be encoded as a line chart.

**Ranking** A ranking relationship should be encoded as an ordered nominal comparison, and as such displayed in a bar or column chart.

**Part to whole** Part to whole relationships are best displayed as stacked bar charts.

**Deviation** When there are few categories a deviation relationship can be displayed as bars, but when there are many categories a line chart should be used.

**Distribution** When a distribution has a single variable and few values a histogram is recommended. When the distribution has more than one variable, it is best to use a scatter chart. If the distribution has a single variable and many values it should be displayed as a line chart.

**Correlation** Correlation relationships are best displayed as scatter charts.

**Function** Functions should be plotted using the plot chart, or if a function is very simple, a line chart.

Of course these solutions are only suggestions, as long as a chart plug-in accepts the input it will try to draw the data. If that is not sufficient it is always possible to extend either the available data relationships or add new chart plug-ins through the plug-in extension mechanism.

The following sections detail the visual output of each chart type and its input values.

### 3.1 Bar Chart

The most simple bar chart is shown in Figure 3.1, which displays the (fictitious) employee turnover in various departments. Its input data object is as follows:

```javascript
var d = {
    categories: ['Sales', 'Engineering', 'Marketing', 'HR'],
    data: [1.5, 3, 2.2, -0.3]
};
```

Note that the negative value results in a chart with a zero baseline and correctly drawn negative bar. The axis range and tick marks are automatically generated by the axis algorithm described earlier.
The width of the bars is defined by the available space, which means that creating a smaller chart creates thinner bars. The ratio between the bar width and spacing between the bars is always kept at 1 : 1, which is the recommended ratio [11].

The chart in Figure 3.2 shows a horizontal bar chart, with the categories on the vertical axis and values on the horizontal axis. In this case the chart is used to display a ranking relationship with the largest value at the top of the chart. The data input is shown below.

```javascript
var d = {
  categories: ['West', 'South', 'East', 'North'],
  data: [38, 25, 22, 15]
};
```

The default bar orientation is vertical. To draw bars in a horizontal orientation, the `reverse: true` boolean should be added to the options object passed to the bar chart constructor.

The next bar chart variation is one with subcategories, shown in Figure 3.3. This chart has two main categories, "Treasury Bonds" and "AAA Municipal Bonds", and also two subcategories "Ten-Year Yields" and "Two-Year Yields". Subcategory bars are drawn together, while the main categories are separated by the width of one bar.
Because the legend only has two items it is drawn above the chart, right below the title. If the legend had too many items it would have been drawn on the right side of the chart. The data object for this chart looks as follows:

```javascript
var d = {
    categories: ['Treasury Bonds', 'AAA Municipal Bonds'],
    subcategories: ['Two-Year Yields', 'Ten-Year Yields'],
    data: [
        [1.11, 3],
        [3.1, 5.7],
    ]
};
```

The last bar chart variant, a stacked bar chart, is shown in Figure 3.4. This chart can display a part to whole relationship, and in this case, a percentage of total sales. The input to a stacked bar chart is identical to a bar chart with multiple categories.

```javascript
var d = {
    categories: ['Q1', 'Q2'],
    subcategories: ['West', 'East', 'South', 'North'],
    data: [
        [38, 22, 15, 25],
        [25, 27, 20, 28]
    ]
};
```

The user can add the `stacked: true` boolean to the options object to draw any bar chart with subcategories as a stacked bar chart. The plug-in automatically adds up the values to create the maximum value of the "whole" and creates an appropriate axis. A legend is added as usual.
3.2 Scatter Chart

The scatter chart plug-in displays the correlation between values. Figure 3.5 displays Anscombe’s quartet of datasets [1, 33]. Each set has an identical mean, variance, correlation and linear regression. These datasets were used by Anscombe to argue the usefulness of plotting data in charts.

The chart plugin uses different symbols for each dataset and also draws each symbol in a different colour to help distinguish them from each other. Again a legend is used to label the
3.3 Line Chart

Line charts can be used to plot time series and simple functions. Figure 3.6 is an example of a simple time series with a single category: "Sales per year". The data object for this chart is very basic, two arrays, one for the categories and one for the values belonging to each category.

```javascript
var d = {
    data: [13000, 13400, 14000, 16000, 14100, 12700, 15000, 16000]
};
```

Note that the line does not start at the zero point and the end point of each line segment is located directly above the category label, indicating the line symbolizes discrete values instead of continuous values.

![Sales per year](image)

Figure 3.6: Line chart with single category

Figure 3.7 shows a line chart used to draw a time series with multiple categories. The lines for the different categories leave enough space for drawing the legend directly onto the chart, a practice recommended over a normal legend [11]. If the lines ended at the same points or very close to each other the line chart plug-in would have opted to use a legend instead. The data object is shown below:

```javascript
var d = {
    subcategories: ['Engineering', 'Marketing', 'HR', 'R&D'],
    data: [
        [1200, 2000, 4000, 4500, 5000, 6300],
        [2000, 4000, 3200, 6000, 7000, 7500],
        [1000, 1500, 1000, 1000, 1800, 1800],
        [1800, 2000, 2100, 2300, 2000, 2700]
    ]
};
```
3.4 Histogram

The histogram chart plug-in supports plotting simple histograms. It accepts categories but not subcategories. Figure 3.8 shows a histogram that plots the frequency versus the height (in feet) of black cherry trees. Because the histogram chart type does not support subcategories a simple grey colour is chosen as the fill colour.

Note that this chart also uses axis labels for both the horizontal and vertical axis, a feature that is supported on all chart types (because it is part of the canvas component, and not specific to the histogram plug-in.)

3.5 Function plotting

The plot chart plug-in supports plotting of mathematical equations in a certain range. Figure 3.9 shows: a) a simple plot of $y(t) = \sin(t)$ and b) a plot of the "Rhodonea curve" defined by
\[ r = \cos(4\theta) \], plotted in polar coordinates. The sine plot also shows non-numeric and Unicode support in labels.

These figures are plotted using interval arithmetic. The basic idea behind plotting using interval arithmetic is to subdivide the range in which to plot a function into four sections. Each section is evaluated in turn for a possible solution using the user supplied function. If a solution is found, the subdivisions with a solution are again subdivided into four sections and the algorithm repeats. If no solution is found in a particular section it does not need further examination \[26, 10, 21\].

The effect of these recursive subdivisions is that the algorithm basically operates as a quad-tree algorithm \[15\]. It keeps subdividing until it either reaches a user-specified depth level or the subdivisions are smaller than one screen pixel. At that point it will plot a pixel on that location. This pixel is then guaranteed to contain the correct solution. This also means that plots are always correct, in that they have no missing singularities or artifacts that are difficult to plot using other methods.
Figure 3.10 shows the algorithm at work for a simple sine wave. The big empty rectangles do not have a solution in them, the smaller rectangles are further subdivided until the subdivision becomes smaller than a screen pixel, at which point the pixel is coloured red. The number of subdivisions in this image is limited to a size of four pixels so as to clearly demonstrate the algorithm.
Chapter 4

Related work

This survey of related work focuses on open-source visualization toolkits which are based on non-proprietary platforms. The multitude of Flash based toolkits and libraries are thus not taken into consideration. The survey is further limited to chart libraries that draw on the client side as that is one of the requirements of this thesis project.

Flot

Flot is a pure JavaScript plotting library for use with the jQuery JavaScript library. Its goals are ease of use, attractive looks and user interaction such as zooming and mouse tracking\(^1\). It is built upon the HTML5 Canvas element and has support for line, bar, and point charts. For text drawing it uses absolute positioned div elements.

PlotKit

PlotKit is a JavaScript plotting library for the MochiKit library with a focus on time series\(^2\). It has support for bar, line and pie charts. There are several render back-ends available with support for both the HTML5 Canvas element and SVG. Additionally it provides render back-ends for basic charts and charts with extra decorations (borders, shadows, etcetera.) PlotKit also draws text labels using absolute positioned div elements.

Flotr\(^3\)

Flotr\(^3\) is a JavaScript plotting library which draws great inspiration from the Flot library mentioned earlier. It has similar features, but this time it is built for the Prototype JavaScript library. In contrast to the Flot library, it has support for negative values, styling graphs via CSS, and several kinds of events for user interaction.

\(^2\)PlotKit: http://www.liquidx.net/plotkit/, MochiKit: http://mochikit.com/
\(^3\)Flotr: http://solutoire.com/flotr/, Prototype: http://prototypejs.org/
ProtoChart

ProtoChart is yet another Prototype based chart library\textsuperscript{4}, inspired by the Flot, Flotr and PlotKit libraries. It features line, bar, pie, curve, and area charts. ProtoChart also supports interaction, and combining different chart types into one chart.

Bluff

Bluff is a JavaScript port of the Gruff Charting library originally written in the Ruby programming language\textsuperscript{5}. It has support for line, bars, pie, area, and spider charts. It uses the HTML5 Canvas element for drawing the charts and overlays absolute positioned div elements for labels and legends. A distinct feature for Bluff is that it supports (simple) HTML tables as input for the charts.

FGCharting

The FGCharting library is a demonstration of how accessible charts can be made by parsing HTML tables using the jQuery library\textsuperscript{6}. It supports bar, pie, line and area charts, and renders these using the HTML5 Canvas element. Again absolute positioned div elements are used for labels, titles and legends.

\textsuperscript{5}Bluff: http://bluff.jcoglan.com/, Gruff: http://rubyonrails.com/pages/gruff
\textsuperscript{6}FGCharting: http://www.filamentgroup.com/lab/creating_accessible_charts_using_canvas_and_jquery/
Chapter 5

Future work

This chapter details some of the feature enhancements and development directions that could be taken. Where possible it also tries to detail how such features could be implemented and gives pointers to related work or projects.

5.1 Scalable Vector Graphics

The current implementation of the chart library uses the HTML5 Canvas element to render its charts. Instead it would also have been possible to use the Scalable Vector Graphics (SVG) support available in some browsers. SVG has better support for text than the Canvas element, which might be an advantage. Browser support is near identical; both require a plug-in or emulation on Internet Explorer, and have decent support on the other browsers. A future version of the chart library could also support an SVG rendering back-end, depending on which approach to client-side graphics on the web gains more traction. A simple implementation could replace the graphics module with one that creates an SVG image.

5.2 Server Side Charts

While rendering charts on the client side has many advantages, such as the possibility for user interaction, and adapting to the users’ environment, it might also be advantageous to render the charts on the server side, for example as a fall-back for browsers that do not natively support the HTML5 Canvas element. A server side implementation can be done with only minor changes to the chart library, by simulating a browser environment on the server. Server side implementations of the browser Document Object Model (DOM) already exist in the form of a JavaScript implementation [23] running on top of the Mozilla Rhino JavaScript engine\(^1\). Because the Rhino JavaScript engine is written in the Java programming language it has the capability to call "native" Java functions from JavaScript.

The DOM implementation cited above could be enhanced with support for the HTML5 Canvas element which simply maps the Canvas API calls to the Java 2D API. From there it would be possible to convert the Java 2D frame buffer into an image and serve it to the client. Apart from retrieving the frame buffer and serving it as an image, the chart library would be completely unaware that it was running on the server and as such only requires minimal modification.

\(^1\)Rhino: JavaScript for Java, http://www.mozilla.org/rhino/
5.3  Support for more chart types

The chart library currently only supports a handful of chart types (bar, line, scatter, histogram, and function plots). This is not a limitation of the design of the library, in fact the modular design and flexible standard components make it very easy to develop additional chart types. For example, the following types would also be useful to support:

- Small multiples
- Spark lines
- Range charts
- Stacked Graphs

Small multiples [33] are a number of charts of the same type and same range, but of different datasets displayed in a grid formation so as to quickly spot the differences between them. This can be implemented in the chart library by creating a composite chart type that internally uses a grid layout to order the individual charts. Spark lines [34] are small in-line charts that display a trend. This might simply be a variant on the line chart code, without drawing the axes. Range charts can take various forms, for example range bars, or high-low charts, and can be implemented by a custom chart plug-in. Although the bar chart plug-in supports stacked bars, support for stacked (or area) charts [4] could be extended to the other chart types as well.

5.4  Interactive charts

The chart library does not currently offer any user interaction. This could also be a future development direction, for example adding support for tool tips and mouse rollovers for data points or labels could significantly enhance its usability. Another type of user interaction could be navigation (for example zooming in and out) on some chart types to reveal details or overall trends. Another use of interactive charts could be for data that is continuously updated such as server statistics or status information for use in a so-called information dashboard [12].

5.5  Web page integration

Currently the chart library takes a JSON object as data input. To enhance ease of use it would be interesting to take a HTML table as input to the charting engine. This would involve the creation of a HTML table parser and a chart recommendation engine.

The HTML table parser could make heavy use of the pattern matching implementation by rewriting a JSON serialization of the table element tree into a format that the charting library supports. The thead and tfoot sections of a table could be used as a source for the meta data and the tbody section as the source for the actual data. Depending on the structure of the thead and tfoot section, both categories and subcategories could be reconstructed and the data parsed accordingly. Both sections and use of the th element can also be used to detect whether a table is unidirectional or bidirectional.

Once the table data is in a format that the charting library understands, another component would take over and decide which chart type would be most suitable for displaying the data. This component could be built using a rules engine (an expert system.) This engine should naturally be extensible by the user of the charting library because it is impossible to predict all the types of data it could encounter. It should however have standard heuristics and parsers for recognizing dates, nominal and qualitative data.
The classification of attributes is however only a suggestion and can be overridden by adding additional class attributes to rows or columns, in order not to restrict the user in her choice of visualization.

Attempts to narrow the range of relationships that may be considered, restrict the transformations that may be applied, or proscribe the statistics that may be computed limit our ability to detect anomalies. Text books and computer programs that enforce such an approach to data mislead their readers and users [35].

Data elements are treated as qualitative when they contain text values and quantitative when they contain numeric values. This simple classification will be supplanted by additional heuristics to detect—for example—numerical nominal values such as product ID numbers. This can be done by calculating the Gini value for these and reclassifying low values as nominal or ordinal (ordinal is more likely here, as numbers have an intrinsic order and can thus always be considered ordinal.)

5.6 ASCII Math parser

At the moment the functions plotted by the plot chart plug-in must be written using the functions in the interval module. Although the module contains most mathematical constructs in the form of methods, writing equations this way is complicated and prone to errors.

Instead the plot plug-in could integrate a ASCIIMathML parser [20], which accepts an ASCII formatted equation and returns a MathML tree. MathML is an XML specification for encoding mathematic equations. The MathML tree could be rewritten to output equations using interval arithmetic. These equations could then be used as input to the plot plug-in.

\[^2\text{For more information, see: http://www.w3.org/Math/}\]
Chapter 6

Conclusion

One of the main advantages of the chart library is its component based nature. All of the components such as titles, legends, and even charts themselves can be moved around easily and laid out with a few lines of code. This makes the library very easy to modify and maintain. New components can be developed without any changes to existing components and—as long as they meet the simple component interface requirements—can be integrated and laid out just as easily. The use of layout managers also proved to be very useful as it separates the layout algorithms from the components.

Another advantage is the plug-in structure, which enables plug-in authors to easily add new chart types. Authors will have all the standard components at their disposal, and basically only have declare which input they accept and then iterate through the data and draw the chart.

Firefox, Opera and Safari all support various subsets of the HTML5 Canvas element. Fortunately there is a common subset of functionality that is sufficient for use in the chart library (apart from a text API.) Unfortunately, the only browser that does not support the HTML5 Canvas element is also the browser with the largest market share: Internet Explorer. There are however JavaScript and ActiveX implementations available that provide a HTML5 Canvas element API for Internet Explorer and convert the API calls into Vector Markup Language (VML) elements [17]. By using one of these implementations, the chart library supports all major browsers without modification to the source code.

Whether or not the choice of using the HTML5 Canvas element for drawing charts is correct remains to be seen. Both SVG and Canvas currently have problems with browser support and interoperability (either something is not supported or implementations differ.) Currently the only safe choice would be to develop for both platforms until a clear winner has emerged (if there is such a thing.) The graphics module makes it easy to support both drawing APIs as it encapsulates and abstracts all drawing calls, shielding the rest of the chart library from "platform" specific APIs. Alternatively, SVG could be used for static charts, while Canvas, with its low level API, makes animation and performance easier to achieve.

Compared to other chart libraries, features that stand out are the aforementioned component structure, accurate function plotting, crisp line drawing, and the plug-in extension mechanism. The current code base is modular and easily extended, thus making it a good platform for further development. The library is also independent of other JavaScript frameworks, unlike other chart libraries which depend on libraries such as jQuery, Prototype and MooTools.
Bibliography


