jsGraph Documentation

Norman Pellet

Feb 25, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO JSGRAPH

jsGraph is a browser-based data plotter. It allows you to display scientific-looking charts and export them in the SVG vector format for high-quality scientific publishing. However, it is also a lot more than that. It implements fast algorithms that redraw your datasets as fast as possible, allowing redrawings to occur in a fraction of a second. Because it is implemented in SVG, jsGraph is an ideal candidate for highly interactive browser-based graphs. You can select series, data points, subscribe to events, zoom in and out, drag the viewing window around, . . .

Here are some highlights of the library:

- **Scientific-style** : Display graphs that are sober looking, but that show exactly what you need.
- **Fast rendering** : Millions of points can be drawn in a fraction of a second. We implemented multilevel downsampling algorithms that allow the fastest rendering time possible. It is expecially effective with the zoom plugin. When zoomed out, only the downsampled version of your data set gets displayed. As you zoom in, the resolution gets finer up to the one defined by your data set.
- **Rich API** : Control exactly how you want your axes, series and annotations to be displayed. We hand you the keys to the finest possible level of control. Of course, you can also use a higher level API for simpler manipulation.
- **Multiple axes** : Series are referenced to axes, which can be placed at the top, bottom, left or right of the graph. You can even have multiple axes on the same side, master-slave axes (for unit conversion, for example) or floating axes, displayed in the middle of the graph.
- **Various series** : jsGraph allows you to display line plots, scatter plots, category plots or box plots, or to combine them together.

1.1 TL;DR Show me an example !

Here is the exact source code that generated this example, with comments

```
// Let's take the j-V curve of a solar cell as an example:

// Let's assume we have it in a javascript array, in the format [ [ x1, x2, ... xn ],␣
↪[ y1, y2, ... yn ] ]
const data = [[-1, -0.95, -0.8999999999999999, -0.8499999999999999, -0.
↪7999999999999998, -0.7499999999999998, -0.6999999999999997, -0.6499999999999997, -0.
↪5999999999999996, -0.5499999999999996, -0.4999999999999996, -0.4499999999999996, -0.
↪39999999999999963, -0.34999999999999964, -0.29999999999999966, -0.24999999999999967,
↪ -0.19999999999999968, -0.1499999999999997, -0.09999999999999969, -0.
↪049999999999999684, 3.191891195797325e-16, 0.05000000000000032, 0.10000000000000032,
↪ 0.15000000000000033, 0.20000000000000034, 0.25000000000000033, 0.3000000000000003,␣
↪0.3500000000000003, 0.4000000000000003, 0.4500000000000003, 0.5000000000000003, 0.
↪5500000000000004, 0.6000000000000004, 0.6500000000000005, 0.7000000000000005, 0.
↪7500000000000006, 0.8000000000000006, 0.8500000000000006, 0.9000000000000007, 0.
↪9500000000000007, 1.0000000000000007, 1.0500000000000007, 1.1000000000000008, 1.
↪1500000000000008, 1.2000000000000008, 1.2500000000000009, 1.300000000000001, 1.
↪350000000000001, 1.400000000000001, 1.450000000000001], [-20.499747544838275, -20.
↪499659532985874, -20.499540838115898, -20.49938076340126, -20.499164882847428, -20.
↪4988737412163, -20.498481100712695, -20.497951576424207, -20.497237447419362, -20.
↪49627435611903, -20.494975508366757, -20.493223851506187, -20.490861525550883, -20.
↪48767563678195, -20.483379071684652, -20.477584622168607, -20.469770090226536, -20.
```

(continues on next page)

(continued from previous page)

```
// Create a waveform, which is used to represent data in a general sense. It has also␣
↪a few cool tricks
const wave1 = Graph.newWaveform().setData(data[1], data[0]);

// For example, you can duplicate it into a second wave and do some point-to-point␣
↪mathematics, in this case calculate the power density
const wave2 = wave1.duplicate().math((x, y) => x * y);

// Let's create a new example and place it in the placeholder <div id="graph-example-1
↪" />
var g = new Graph("graph-example-1", {});

// You need to set the size if the container doesn't have one already
g.resize(400, 300);

// Let us create a new serie, called "jV", with red thick line and markers
var jV = g
    .newSerie("jV", {
        lineColor: 'red',
        lineWidth: 2,
        markers: true,
        // Setting the style of the markers
        markerStyles: {
            // For the default "unselected" style
            unselected: {
                // Default look
                default: {
                    shape: 'rect',
                    strokeWidth: 1,
                    x: -2,
                    y: -2,
                    width: 4,
                    height: 4,
                    stroke: 'rgb( 200, 0, 0 )',
                    fill: 'white'
                }
            },
            // Maybe we want to display only one marker every five points. Nothing␣
↪easier !
            modifiers: (x, y, index, domShape, style) => index % 5 == 0 ? style :␣
↪false
        }
    })
    .autoAxis() // Assign it automatically to the left and the bottom axis (which are␣
↪created by default if they don't exist)
    .setWaveform(wave1) // Assign a waveform to this serie

// How about a second serie ?
var pV = g
    .newSerie("pV") // Give it a unique name, otherwise you'll overwrite the first one
    .setXAxis(g.getXAxis()) // The x axis is the same...
    .setYAxis(g.getRightAxis()) //  But the y axis is different, let's get the first␣
↪right axis (created by default)
    .setWaveform(wave2); // And assign the second waveform

// How about some styling of the axes ?
```

(continues on next page)

(continued from previous page)

```
g
    .getXAxis() // Retrieve the bottom axis
    .setUnit("V") // Set the unit voltage
    .setUnitWrapper("(", ")") // Wrap in parentheses ==> (V)
    .setLabel("Voltage") // Show the axis label
    .secondaryGridOff(); // Turn off the secondary grid

g
    .getLeftAxis()  // Retrieve the left axis
    .setUnit("mA cm^-2")
    .setUnitWrapper("(", ")")
    .setLabel("Current density")
    .secondaryGridOff()
    .forceMin(-25) // Force the minimum of the axis
    .forceMax(60);  // And its maximum

g
    .getRightAxis()
    .setUnit("mW cm^-2")
    .setUnitWrapper("(", ")")
    .setLabel("Power density")
    .gridsOff(); // Do not display any grid for the right axis

// Adds some spacing to the right of the axis. This is 30% of the "data width" of the␣
↪axis (which is the max value - the min value for all series sharing this axis)
g.getBottomAxis().setAxisDataSpacing(0, 0.3);

// We want the 0 of the right axis to match the 0 of the left axis. It's much more␣
↪natural like that
// Use .adaptTo() to enforce this behaviour adaptTo( otherAxis, myRef, otherRef,␣
↪clamp )
// The clamp "min" basically says that the normal scaling behaviour applies to the 0␣
↪and to the min value of the axis. The max value is therefore the one calculated as␣
↪a function of the master axis (the left one)
g.getRightAxis().adaptTo(g.getLeftAxis(), 0, 0, "min");

// Some more styling, note how you can change the color of the axis, the ticks, the␣
↪tick labels and the axis label
g
    .getLeftAxis()
    .setAxisColor('red')
    .setPrimaryTicksColor('red')
    .setSecondaryTicksColor('rgba( 150, 10, 10, 0.9 )')
    .setTicksLabelColor('#880000')
    .setLabelColor('red');

// autoscale has to be called before the first rendering when more than one serie was␣
↪added
g.autoScaleAxes();

// And finally, let's draw it !
g.draw();
```

CHAPTER

TWO

INSTALLATION

2.1 Install with npm (recommended)

Installing jsGraph from npm is an easy way to install jsGraph and keep it up to date

```
npm i node-jsgraph
```

2.1.1 Default file

If you use a module resolver, then you should be aware that `const Graph = require("jsGraph");` will load the minified UMD file **with** ES6 polyfills.

2.2 Download from the Github repository

The project is hosted on Github and the source code is available open source and released under the MIT licence.

View on github

2.2.1 Releases

We use github releases to ship the distribution package. Check out the Release page to download the latest release.

2.3 Manual Download

jsGraph can be downloaded and installed manually or via the npm package manager. The project is hosted on Github and the distribution files can be found there. jsGraph has **zero** dependencies and does not rely on any CSS files.

We use Universal Module Definitions to ship jsGraph. That means the library is compatible with AMD definition, CommonJS definition and defaults to Browser global when none exist. Select the version to include as a function of your needs:

2.3.1 Minfied version

The full-featured, compact version of jsGraph, shipped with all plugins, shapes and series available. This version ships with Universal Module Definition (see below).

Download minified

2.3.2 Development code

The compiled but not uglified source code can be used for testing purposes and bug reporting. Similarly to the minified version, this file ships with Universal Module Definition.

Expanded version

2.3.3 ES 6 version

The source code without ES6 transpilation. Does not include any ES6 polyfills and therefore targets ES6-compatible browsers. This version ships with Universal Module Definition

Download ES6 minified

2.3.4 Module bundle

For browser that support ES6 modules (or if you want to use as such with your own packager), you can directly include the module file:

jsgraph-module.min.js

2.4 Include jsGraph in your projects

2.4.1 Universal module definitions

Browser global

Here is how to include jsGraph in your browser using the global object create. The following creates the `` `Graph` `` object on the `` `window` `` level of your browser:

```
<head>
    <script src="path/to/jsgraph/jsgraph.min.js"></script>
    <!-- Creates the public Graph object -->
</head>
```

Which allows you to use it as such:

```
<body>
    <script>
        const graph = new Graph( /* ..options.. */ );
    </script>
</body>
```

AMD definition

If you are using an AMD loader such as RequireJS, you can still use jsGraph:

The following versions are browser-ready and creates the `Graph` object on the `window` level of your browser:

```
<head>
    <script src="path/to/require/js"></script>
</head>
<body>
    <script>
        require(['path/to/jsgraph.min'], function( Graph ) {
            <!-- Creates the local Graph object -->
            const graph = new Graph( /* ..options.. */ );
        });
    </script>
</head>
```

Obviously, in a real-life example, you would use `define` in your module and load jsGraph as a dependency.

CommonJS definition

If you create your own bundle using Webpack, Rollup, Gulp or other, then you can simply the CommonJS definition:

```
const Graph = require('path/to/jsGraph');
// Use Graph
const graph = new Graph( /* ..options.. */ );
```

2.4.2 ES 6 module definition

If you’re working with ES6 modules you can use the module files as such:

In a browser

```
<body>
<!--Your page-->
</body>
<script type="module" src="./path/to/jsGraph/jsgraph-module.min.js"></script>
```

From another module

In this case, Graph is default export of the module:

```
import Graph from 'path/to/jsGraph/jsgraph-module.min.js';
```

CHAPTER

THREE

GETTING STARTED

Note: TL,DR

```
let graph = new Graph( htmlDivID, someOptions );
let wave1 = graph.newWaveform().setData( yDataAsArray, xDataAsArray );
let serie = graph.newSerie('someSerieName').setWaveform( wave1 ).autoAxis();
graph.draw();
```

After you managed to install jsGraph and load into your browser, it's time to display your first graph.

3.1 Graph Constructor method

The graph constructor takes the following possible forms

```
const graph = new Graph( wrapper?, options?, axes? );
```

where all three options are actually optional

3.1.1 DOM Wrapper

The Wrapper is the DOM element into which jsGraph will inject some SVG and HTML code. It can be the id of the container, or the html element itself:

```
<div id="graph-container" />
```

```
const graph = new Graph( "graph-container" );
// ...or
const container = document.getElementById( 'graph-container' );
const graph = new Graph( container );

// ...or, from jQuery
const container = $("#graph-container");
const graph = new Graph( container.get() );
```

You can delay the use of the wrapper and call the `setWrapper` method later on:

```
graph.setWrapper( domWrapper );
```

If no width / height options are passed in the constructor, jsGraph will attempt to find out the dimension of the container, using getComputedStyle.

If it doesn't work out, you will need to call `graph.resize( widthInPx, heightInPx )` before you call the `draw()` method.

3.1.2 Options

jsGraph receives a variety of options that can be set in the constructor. Most options occur with the axes or the series, but the graph itself takes a few of them. Here's a full example of them with their default value:

```
const GraphOptionsDefault = {
    title: '', // The title of the graph

    paddingTop: 30, // Top padding, important if there's a title
    paddingBottom: 5,
    paddingLeft: 20,
    paddingRight: 20,

    // If you want to add dummy lines to make the graph appear as a rectangle
    close: {
        left: true,
        right: true,
        top: true,
        bottom: true
    },

    // Color of the closing lines
    closeColor: 'black',

    // Default font size and font used for the whole graph. Can be overridden for␣
↪each component
    fontSize: 12,
    fontFamily: 'Myriad Pro, Helvetica, Arial',

    // Refer to the interaction documentation to understand those
    plugins: {},
    mouseActions: [],
    keyActions: [],

    // Where clicking somewhere on the graph unselects the shape
    shapesUnselectOnClick: true,

    // Whether there can be only one shape selected at the same time
    shapesUniqueSelection: true,

    // Axes. Continue reading to understand this syntax
    axes: {}

};
```

3.1.3 Axes

The axes settings can also be part of the constructor. Either use them as part of the options, under the key `axes` , or, for legacy reasons, as part of the third argument in the constructor.

Here's the syntax to use:

```
// Do not use const, read why
let axes = {
    top: [
        { /* axis definition */ },
        { /* a second top axis definition */ }
    ],
    bottom: [],
    left: [
        { /* a right axis definition */ }
    ],
    right: []
};
```

jsGraph does **not** make a copy of this object. Also, options are pretty dynamic, so it may be that jsGraph fills those objects with internal values. This is useful when you want to dump the axis object, save it, and reload it some other time.

Axis definition

Here are the default options that you may override for each axis:

```
let axisDefault = {
    // Give it a unique name to retrieve it later
    name: undefined,

    // Value of the axis label
    labelValue: '',

    // You can put false here if you decide to use the axis but not to display it
    display: true,

    // Flip the axis, where the high-end value is to the right or the bottom, and the␣
↪low-end value to the left / top
    flipped: false,

    // Use this to draw a vertical or horizontal line at that value. For example, for␣
↪a straight line at 0, use lineAt: 0
    lineAt: false,

    // Adds a certain percentage of padding to the axis, with respect to the min/max␣
↪values provided by the series.
    axisDataSpacing: { min: 0.1, max: 0.1 },

    // This can be used to display the value differently. More on that later
    unitModification: false,

    // Display the primary grid, corresponding to the primary ticks (the ones with␣
↪labels)
    primaryGrid: true,
```

(continues on next page)

(continued from previous page)

```
    // Display the secondary grid, corresponding to the secondary ticks
    secondaryGrid: true,

    // Self-exlanatory grid styling
    primaryGridWidth: 1,
    primaryGridColor: '#f0f0f0',
    primaryGridDasharray: undefined,
    primaryGridOpacity: undefined,
    primaryTicksColor: 'black',

    secondaryGridWidth: 1,
    secondaryGridColor: '#f0f0f0',
    secondaryGridDasharray: undefined,
    secondaryGridOpacity: undefined,
    secondaryTicksColor: 'black',

    // Use true to hide the axis when all the series associated to it are hidden
    hideWhenNoSeriesShown: false,

    // Offset the low-end value of the graph to 0
    shiftToZero: false,

    // Tick positions with respect to the axis line: TICKS_INSIDE, TICKS_CENTERED,␣
↪TICKS_OUTSIDE are possibilities
    tickPosition: Graph.TICKS_INSIDE,

    // Approximate number of primary ticks to display on the whole axis. It is an␣
↪indication that jsGraph works with, but when working with decimal values,␣
↪variations can occur
    nbTicksPrimary: 3,

    // Approximate number of secondary ticks to display between each primary tick
    nbTicksSecondary: 10,

    // Use scientific scaling, where values of the ticks are displayed in the␣
↪scientific notation
    scientificScale: false,

    // Use a value to force the scientific exponent, rather than letting jsGraph␣
↪determine the best one
    scientificScaleExponent: false,

    // Engineering scale is similar to scientific scale, but only with exponents in␣
↪multiples of 3. (ug, mg, g, kg, ...)
    engineeringScale: false,

    // The following three options scale the value of the ticks, when scientific␣
↪scaling is off

    // Scale the value of the tick by that factor. Useful for unit conversion
    ticklabelratio: 1,
    // Multiplies the tick values by 10^x, where x is the exponential factor
    exponentialFactor: 0,
    // Same as the exponentialFactor, but also applied to the label itself, when␣
↪using scientific scaling
    exponentialLabelFactor: 0,
```

(continues on next page)

(continued from previous page)

```
    // Display the axis as a log scale
    logScale: false,

    // Force the min and the max value. Zooming is still possible, but the min/max␣
↪values provided by the series become irrelevant
    forcedMin: false,
    forcedMax: false,

    // You can use this setting to not display the axis over the full width / height␣
↪of the graph. Value in normalized percentage (0 = 0%, 1 = 100%)
    span: [0, 1],

    // Set the unit of the axis
    unit: false,

    // Wrap it in the following string
    unitWrapperBefore: '',
    unitWrapperAfter: '',

    // Add the unit in each tick
    unitInTicks: false,

    // Adjust the offset between the tick and its label
    tickLabelOffset: 0,

    // You can display a katex formula as the label, more on this later
    useKatexForLabel: false,

    // Sets the upper bond that the axis can reach and disregards the value given by␣
↪the series if their are higher/lower
    highestMax: undefined,
    lowestMin: undefined

};
```

3.2 Adding a serie

Obviously the first thing we'll want to do is to create a new serie. But before that, we need to understand the concept of waveforms

3.2.1 Understanding waveforms

Waveforms are not much more than a suger coating over standard javascript arrays. In their more general sense they represent actual data to be plotted. However they provide a bunch of useful features which target handling the data, and therefore are independent of the serie itself, which aims to display that data.

Create a waveforms

Nothing simpler than creating a waveform. The Graph object exposes a shortcut to the constructor using

```
let waveform = Graph.newWaveform();
```

XY waveforms

XY waveforms are perhaps the most obvious one. It's a bunch of Y data corresponding to a bunch of X data. Whether they represent scattered data or should be linked with a line is irrelevant.

Note: When used to display a line data and when it can be determined that the x values are monotoneously increasing, jsGraph decreased the rendering time by ignoring the data before the minimum bound of the x axis and the data above the maximum bound of that same axes. Obviously there are a lot more optimisation at play, but that's just one of them…

To set the XY data to a waveform, use the `setData` method:

```
waveform.setData( yArray, xArray );
```

X as a waveform

In this format, jsGraph actually maintains two waveforms, the main one for the y data set, and one for the x dataset. It therefore also allows you to do the following

```
// given xWaveform, yWaveform
xWaveform.setData( xArray );
yWaveform.setData( yArray );
yWaveform.setXWaveform( xWaveform );
xWaveform.math( /*...*/ ) // to apply math of the x data set
```

Y waveforms

Y waveforms occur when the interval between each data point is constant. The offset and scaling between the points can be set either in the constructor or using the `.rescaleX` method

```
let wave1 = Graph.newWaveform( yDataAsArray, offset, scale );

// or
wave1.rescaleX( offset, scale );
```

The first `y` value will be at `offset`, the second at `offset + scale`, the third at `offset + scale * 2`, etc.

Note: A third waveform type exists: Hash waveforms. They are used to represent series that go in a bar chart (or category plot). As its name indicates, it doesn't take `(x,y)` values, but a hashmap, or more generally a javascript object:

```
const wave = Graph.newWaveformHash(); // Create the waveform
wave.setData({ categoryA: yVal, categoryB: yVal2 }); // Setting the data
```

3.2.2 Creating a new serie

To create a new serie, simply use the `graph.newSerie` method:

```
let serie = graph.newSerie( serieName, serieOptions, serieType );
```

The first argument is required, while the other two are optional and default to `serieOptions: {}` and `serieType: Graph.SERIE_LINE`.

- The serie name **must be unique**. If you try to use the name of an existing serie, `newSerie` will simply return the existing serie, and you may override it
- **The `serieType` describes which type of serie you're trying to add. Valid values are:** ** `Graph.SERIE_LINE` or `"line"` ** `Graph.SERIE_SCATTER` or `"scatter"` ** `Graph.SERIE_CONTOUR` or `"contour"`: To create contour lines ** `Graph.SERIE_BAR` or `"bar"`: To use with bar charts ** `Graph.SERIE_BOX` or `"box"`: Box plots ** `Graph.SERIE_LINE_COLORED` or `"color"`: Colored line where each segment can have a different color (lower performance than `Graph.SERIE_LINE`) ** `Graph.SERIE_ZONE` or `"zone"`: Typically used to display min/max values as a greyed area ** `Graph.SERIE_DENSITYMAP`: A density map (see the tutorial about how to use density maps)

Hint: Most methods that apply to the series return the serie itself, allowing API calls to be chained:

```
let serie = graph.newSerie('name', {}, 'line').methodA().methodB().methodC();
```

Assigning axes to the serie

A serie needs to have an x and a y axis. They might not be displayed, but they must exist. Most jsGraph axis getters create axes if they don't exist, so don't worry too much about that. If you would like to use the default axes, use

```
serie.autoAxis();
// or:
serie.autoAxes();
```

Important: The default axes are the `left` axis at index `0` and the `bottom` at index `0`. They will be created automatically if they don't exist.

You may of course use other axes. For that, the `setXAxis( axis )` and `setYAxis( axis )` exist:

```
serie.setXAxis( graph.getBottomAxis( 1 ) ); // Get the second bottom axis
serie.setYAxis( graph.getRightAxis() ); // Get the first right axisDataSpacing
```

(continues on next page)

(continued from previous page)

```
// Don't do that:
serie.setXAxis( graph.getLeftAxis() ); // Error ! Assigning an y axis while the serie␣
↪expects and x axis
```

3.2.3 Drawing the graph

So far you haven't asked the graph to draw anything. You merely created object and told jsGraph how you wanted to render them. For the final rendering use:

```
graph.draw();
```

3.2.4 Boilerplate example

Summing up everything we've done, it all boils down to a few lines code. Consider the following complete example:

```
var g = new Graph("graph-example-gettingstarted-1", {});
g.resize(400, 300).newSerie('serieName')
    .setWaveform(Graph.newWaveform().setData([1, 2, 3], [4, 5, 6]))
    .autoAxis();
g.draw();
```

This code would display the following basic graph:

3.2.5 Redrawing methods

To redraw the method, you can rebind the data to the waveform, and the waveform to the serie:

```
waveform.setData( dataY, dataY ); // Rebinding arrays
serie.setWaveform( waveform );  // Rebinding waveform
graph.autoscaleAxes(); // Optional, but rescales the axes to fit the new (?) min/max␣
↪values
graph.draw();
```

Rebinding the data is not an computationnally expensive data. However, sometimes you may loose track of `dataY` and `dataX`.

In this case, you can also directly **mutate** the arrays and not rebind them to the serie. It may be useful if you lose track of where the arrays are. That's not a problem for jsGraph, but it becomes your responsability to tell the waveform, the serie and the graph that the data has changed.

If you're not sure whether the min / max values have changed:

```
waveform.mutated(); // Tell the waveform to recompute the min/max
serie.dataHasChanged(); // Tell the serie that the data has changed
graph.updateDataMinMaxAxes(); // Tell the graph that there may be new min max values
graph.autoscaleAxes();
graph.draw();
```

If you are sure that the min/max values haven't changed:

```
s.dataHasChanged(); // Flag the serie for a redraw
graph.draw();
```

Warning: Even if you do not wish to do call `autoscaleAxes`, and in the case where the min/max of the data may have changed, you **need** to call the `mutated` method on the waveform and the `updateDataMinMaxAxes` on the graph object.

Demonstration

```
const graph = new Graph("graph-example-gettingstarted-2");
graph.resize(400, 300);

let x = [1, 2];
let y = [1, 2];
let w = Graph.newWaveform().setData(y, x);
let s = graph.newSerie('s').setWaveform(w).autoAxis();
graph.draw();

let i = 0;
setInterval(function () {

    if (i % 100 < 50) {
        x.push(i % 100 + 3);
        y.push(i % 100 + 3);
    } else {
        x.pop();
        y.pop();
    }
    i++;

    w.mutated();
    s.dataHasChanged();
    graph.updateDataMinMaxAxes();
    graph.autoscaleAxes();
    graph.draw();

}, 200);
```

This code would display the following basic graph:

CHAPTER

FOUR

STYLING SERIES

Without style, a serie defaults to either a black line (for line series), or round dots for a scatter serie. In most cases this behaviour is not enough, and we would like to be able to change the color, thickness, dashing/dotting, fill color, etc… of the various series that we want to display.

To this purpose, jsGraph employs the notion of style, which is nothing else than a javascript object. A style is defined by a `name` and a collection of attributes

Note: There are two default styles that ship with jsGraph and that are involved in the functionning of the library:

- `unselected`. This is the basic style that renders the serie by default. By default, when performing API calls without setting the style, jsGraph assumes that you mean the `unselected` style
- `selected`. The style that is displayed when a serie, or, for a scatter plot, some points of the serie, is/are selected. It **inherits** the style `unselected` (see below)

4.1 Style inheritance

jsGraph allows you to derive some styles from others. For example, is the style `unselected` and `red` look like that:

```
// unselected
const unselected = {
    line: {
        width: 3,
        color: 'black'
    }
}

// red
const red = {
    line: {
        color: 'red'
    }
}
```

and that the following API calls are used:

```
serie.setStyle( unselected );
serie.setStyle( red, "red", "unselected"); // <== Red style inherits from Unselected␣
↪style
serie.setActiveStyle( "red" );
```

Then a **thick** red line that is 3px width will be displayed.

Warning: All styles must inherit from another one, up and until the style `unselected`. If no name is provided as the third parameter of the `setStyle` API call, `unselected` is assumed to be the base style.

4.1.1 Runtime inheritance

Styles are computed at runtime. In other words, taking the example that is discussed above, if the `unselected` style is changed using an API call:

```
serie.setLineStyle( "4,4", "unselected" ); // <== Second parameter is optional here,␣
↪because it defaults to "unselected"
graph.draw();
```

The serie with the style `red` will automatically determine that it should be dashed, because it inherits from the style `unselected`, which we just said should be dashed.

```
const graph = new Graph("example-1");
graph.resize(400, 300);

let x = [1, 2, 3, 4];
let y = [1, 2, 1, 2];

let w = Graph.newWaveform().setData(y, x);
let s = graph.newSerie('s').setWaveform(w).autoAxis();

const unselected = {
    line: {
        width: 3,
    }
}
const red = {
    line: {
        color: 'red'
    }
};

s.setStyle(unselected);
s.setStyle(red, "red", "unselected");
s.activateStyle("red");

graph.draw(); // You won't see this, but at this stage, the serie is not dashed

s.setLineStyle("4,4", "unselected");
graph.draw(); // Now it will be dashed
```

This code would display the following graph:

4.1.2 Object mutation

jsGraph allows you to mutate the style objects, but it asks that you warn him that the style has changed. This is done for optimisations purposes: the style is not recomputed, nor applied, if it hasn't changed. Because we don't want to start observing your objects, we just ask that you notify the serie of the change:

```
let coloredStyle = { };
s.setStyle( coloredStyle, "colored" );

// Later
coloredStyle.color = 'green';
s.styleHasChanged( "colored" );
```

Even if you change a base style that your derived style might be using, that's fine, jsGraph will look at whether any ancestor has been modified and if so, recalculate the whole derived style.

4.1.3 Activating a style

To activate a style, use the `activateStyle` with the name of style you want to apply. jsGraph **does not** redraw by default, for optimisation purposes. You need to call `graph.draw()` afterwards.

```
s.activateStyle("selected");
// or:
s.setActivateStyle("selected");
// Finally, when you're ready to render:
g.draw();
```

4.2 Line series

For line series, three styles are available:

- The color
- The width
- The dashing

They are available through individual API calls or via the `setStyle` method.

4.2.1 API calls

You may use the following method signatures to set the style of the line serie:

```
// Sets the line width
s.setLineWidth( widthInPx, styleName = "unselected" );

// Sets the line color
s.setLineColor( '#FF0000', styleName = "unselected" );
s.setLineColor( 'red' /*, styleName... */ ); // Or use a name
s.setLineColor( 'rgba(1.0, 0.0, 0.0, 1.0)' /*, styleName... */ );

// Set the line dashing
s.setLineStyle( 1, styleName="unselected" ); // Sets mode 1. See below
s.setLineStyle( "4,4" /*, styleName... */) // Directly sets the stroke-dasharray SVG␣
↪property
```

(continues on next page)

(continued from previous page)

There are 12 different modes, for you to use with a counter for example.

- `1`: Straight line
- `2`: 1px dots spaced by 1px
- `3`: 2px lines spaced by 2px
- `4`: 3px lines spaced by 3px
- `5`: 4px lines spaced by 4px
- `6`: 5px lines spaced by 5px
- `7`: Long dashes, short gaps
- `8`: Short dashes, large gaps
- `9`: Long dashes, alternating short and long gaps
- `10`: Alternating dashed and dots
- `11`: Very long dashes, short gaps
- `12`: Short dashes, very long gaps

4.2.2 setStyle method

Use the setStyle method to set all three parameters at once. Consider the following snippet:

```
let style = {

    line: {
        color: 'red',
        width: 4,
        style: 2
    }
};

s.setStyle( style, "myStyleName", "unselected" ); // <== Inherits from the unselected␣
↪style, but that parameter is optional
```

4.3 Scatter series

Scatter series offer a lot of styling possibilities. You affect the shape, size and appearence of a “marker”, which is the element that is displayed at each point of the serie. What’s interesting is that all markers of a serie need not be the same ones. We offer a lot of possibilities to apply modifications.

4.3.1 Basic styling

You may use a javascript object that will directly be mapped to the SVG properties of the object:

```
let style = {
    shape: 'circle',
    cx: 0,
    cy: 0,
    r: 3,
    stroke: 'transparent',
    fill: 'black'
}
```

Will create the following SVG element

```
<circle cx="0" cy="0" r="3" stroke="transparent" fill="black" />
```

You can therefore use any SVG element available to you. The only reserved key in the object is `shape`, which of course is transformed into the SVG node name.

4.3.2 Setting the style

For scatter series, use the `setMarkerStyle` or `setStyle` method, with slightly more complex parameters:

```
let style = {
    shape: 'circle',
    cx: 0,
    cy: 0,
    r: 3,
    stroke: 'transparent',
    fill: 'black'
}

// Basic version
s.setMarkerStyle(style, "styleName", "inheritedStyleName" ); // <== Second and third␣
↪parameters default to "unselected"

// With modifiers (see below)
s.setMarkerStyle(style, arrayOrFuncOfModifiers, "styleName", "inheritedStyleName" ); /
↪/ <== Third and fourth parameters default to "unselected"

// Generic setStyle method
let gStyle = {
    markers: {
        all: style,
        modifiers: arrayOrFuncOfModifiers
    }
};

s.setStyle(  gStyle, "styleName", "inheritedStyleName" ); <== Again, second and third␣
↪parameters default to "unselected"
```

4.3.3 Modifiers

Maybe you want to draw attention to a specific marker. In that case, you can use the modifiers to set its properties. You have two possibilities:

- You may either feed an array of any length, but where the index of the point you want to modify contains a javascript that will extend the style at that position:

```
let modifiers = Array( n );
modifiers[ 258 ] = { fill: 'blue' }; // <== Fills the point number 258␣
↪with the color blue
```

- Use a runtime method to determine the modifier on the fly (Be careful when using time sensitive applications)

```
let modifiers = ( x, y, index ) => { return y > 0 ? { fill: 'green' } : {␣
↪fill: 'red' } };
```

The following shows an example of using the style and modifiers:

```
const graph = new Graph("example-2");
graph.resize(400, 300);

let x = new Array(200).fill(0).map((x, index) => index / 20);
let y = [...x].map(x => Math.sin(x));
let w = Graph.newWaveform().setData(y, x);
let s = graph.newSerie('s', {}, 'scatter').setWaveform(w).autoAxis();

const posModifier = { fill: 'green' };
const negModifier = { fill: 'red' };

const posNegStyle = {
    markers: {
        all: {
            shape: 'rect',
            width: 4,
            height: 2,
            x: -2,
            y: -1
        },

        modifiers: (x, y, index) => {
            return index % 5 != 0 ? false : (y < 0 ? negModifier : posModifier) //␣
↪Display every 5 marker
        }
    }
};

s.setStyle(posNegStyle, "posNeg");
s.activateStyle("posNeg");
graph.draw();
```

This code would display the following graph:

Note: When using a modifier method, return `false` to deactivate the marker rendering at that particular index.

4.3.4 Individual styles names

Modifiers work great if you want to highlight some of the markers using a special style. However, you can also assign style names to each point independently. For example, taking the example above (and adding some animation onto it), we could define two styles:

```
let neg = { shape: 'circle', r: 3, fill: 'red' }
let pos = { shape: 'rect', width: 4, height: 4, x: -2, y: -2, fill: 'green' };

s.setMarkerStyle(neg, "negative");
s.setMarkerStyle(pos, "positive");

s.setIndividualStyleNames((s, i) => s.getWaveform().getY(i) > 0 ? 'positive' :
↪'negative');
```

This would output the following graph

4.3.5 Enabling markers with line series

Line series extend from scatter series, but have markers by default disabled. If you want to use markers with a line serie, use

```
s.setMarkers( true ); // First parameter defaults to true, so s.setMarkers(); also␣
↪enable markers
```

4.4 Closing example

Obviously, we can start mixing things up and have a little bit of fun:

```
const graph = new Graph("example-4");
graph.resize(400, 300);

let date = Date.now();
let s = graph.newSerie('s', {}, 'line').autoAxis();


function d() {
    let phase = (Date.now() - date) / 1000;
    let x = new Array(200);
    x = x.fill(0).map((x, index) => index / 10);
    let y = [...x].map(x => Math.sin(x + phase));

    let w = Graph.newWaveform();
    w.setData(y, x);
    s.setWaveform(w);
    s.setIndividualStyleNames((s, i) => s.getWaveform().getY(i) > Math.cos(phase) ?
↪'positive' : 'negative');

    graph.draw();
}

let def = { line: { color: 'orange', width: 2 } };
let neg = { markers: { all: { shape: 'circle', r: 3, fill: 'red' } } }
let pos = { markers: { all: { fill: 'green' }, modifiers: (x, y, index) => index % 2␣
↪== 0 ? { fill: 'blue' } : null } };
```

(continues on next page)

(continued from previous page)

```

s.setMarkers();
s.setStyle(def, "unselected"); // In the scatter serie, this is overridden by␣
↪setIndividualStyleNames, but the line serie will take the "unselected" style
s.setStyle(neg, "negative");
s.setStyle(pos, "positive", "negative");


graph.draw();

setInterval(d, 100);
```

That’s all we got (for now) !

CHAPTER

FIVE

AXES SPANNING

jsGraph allows you to have in a single graph more than one axis at the same position (top, left, right, bottom). Axes don't actually have to take up the 100% of the space, horizontally or vertically.

For example, let us imagine that you want to display different kind of data in the y direction (for example, speed, angle and height), versus one single x axis (let's say time), it might make sense to display all three series in different regions of the graph, because the axes would have nothing to do with each other. . .

You can do that using the method *axis.setSpan(spanFrom, spanTo)*, where *spanFrom* and *spanTo* must range between *0* and *1* and represents the position in percentage where the axis will start and end.

Note: With the **x** axis, 0 is always at the **left**, and 1 is at the **right**. With the **y** axis, 0 is the **bottom**, and 1 is the **top** (inverse of the traditional SVG coordinates).

5.1 Basic example

To illustrate the *setSpan* method, let us make a nice graph with two y axes spanning from 0 to 45% and from 55% to 100%:

```
var dataColorado = [[2015, 17559.393], [2014, 17944.255], [2013, 18881.823], [2012,␣
↪19263.158], [2011, 18744.067], [2010, 18978.981], [2009, 17351.28], [2008, 18961.
↪826], [2007, 19532.855], [2006, 19707.00899], [2005, 19013.11703], [2004, 19251.
↪20903], [2003, 19595.836], [2002, 19446.04], [2001, 19764.973]];
var dataCalifornia = [[2015, 34522.242], [2014, 39213.757], [2013, 39474.651], [2012,␣
↪38978.114], [2011, 42542.656], [2010, 41890.627], [2009, 39271.173], [2008, 42190.
↪776], [2007, 41064.161], [2006, 41937.83301], [2005, 40352.02201], [2004, 39342.
↪39199], [2003, 38521.048], [2002, 38604.897], [2001, 41305.269]];
dataColorado = Graph.newWaveform().setData(dataColorado.map(e => e[1]), dataColorado.
↪map(e => e[0]));
dataCalifornia = Graph.newWaveform().setData(dataCalifornia.map(e => e[1]),␣
↪dataCalifornia.map(e => e[0]));

var graph = new Graph("example-1");
graph.resize(700, 300);
graph.getLeftAxis(0).setSpan(0.0, 0.45).turnGridsOff().setLabel("Colorado");
graph.getLeftAxis(1).setSpan(0.55, 1).turnGridsOff().setLabel("California");
graph.getBottomAxis().setPrimaryGridColor("rgba( 100, 100, 0, 0.5 )").setLabel("Year
↪");

graph
    .newSerie("colorado")
```

(continues on next page)

(continued from previous page)

```
    .setWaveform(dataColorado)
    .autoAxis()
    .setYAxis(graph.getLeftAxis(0))
    .setLineColor("#CF4E4E")
    .setLineWidth(2);

graph
    .newSerie("california")
    .setWaveform(dataCalifornia)
    .autoAxis()
    .setYAxis(graph.getLeftAxis(1))
    .setLineColor("#3EA63E")
    .setLineWidth(2);

graph.draw();
```

This code would display the following graph:

5.2 Overlapping axes

One might wonder how jsgraph handles cases where axis spans overlap. For example `axisA.setSpan( 0, 55 )` would clash with `axisB.setSpan( 50, 100 )`. In such cases, jsGraph determines automatically such clashes and offsets one of the axis (the one with the largest index in the stack) by a sufficient amount so that visual perception is not deteriorated. In general, jsGraph will try to place as many axis in the first level, and perform iteratively for the following levels. For example, if you have *axisA.setSpan(0, 55)*, *axisB.setSpan(50, 100)* and *axisC.setSpan(60, 80)*, then axisC will be placed together with axisA because they don't overlap. It will not appear on a third level.

In this particular example, it wouldn't make much sense, but for the sake of it, here is how this example would turn out:

```
var dataColorado = [[2015, 17559.393], [2014, 17944.255], [2013, 18881.823], [2012,␣
↪19263.158], [2011, 18744.067], [2010, 18978.981], [2009, 17351.28], [2008, 18961.
↪826], [2007, 19532.855], [2006, 19707.00899], [2005, 19013.11703], [2004, 19251.
↪20903], [2003, 19595.836], [2002, 19446.04], [2001, 19764.973]];
var dataCalifornia = [[2015, 34522.242], [2014, 39213.757], [2013, 39474.651], [2012,␣
↪38978.114], [2011, 42542.656], [2010, 41890.627], [2009, 39271.173], [2008, 42190.
↪776], [2007, 41064.161], [2006, 41937.83301], [2005, 40352.02201], [2004, 39342.
↪39199], [2003, 38521.048], [2002, 38604.897], [2001, 41305.269]];
var dataKentucky = [[2015, 664.166], [2014, 878.434], [2013, 915.246], [2012, 1183.
↪112], [2011, 1539.699], [2010, 1542.78], [2009, 1521.939], [2008, 1723.062], [2007,␣
↪1752.384], [2006, 1710.887], [2005, 1676.522], [2004, 1731.218], [2003, 1727.233],␣
↪[2002, 1821.618], [2001, 1739.07]];
dataColorado = Graph.newWaveform().setData(dataColorado.map(e => e[1]), dataColorado.
↪map(e => e[0]));
dataCalifornia = Graph.newWaveform().setData(dataCalifornia.map(e => e[1]),␣
↪dataCalifornia.map(e => e[0]));
dataKentucky = Graph.newWaveform().setData(dataKentucky.map(e => e[1]), dataKentucky.
↪map(e => e[0]));

var graph = new Graph("example-3");
graph.resize(700, 300);

graph.getLeftAxis(0).setSpan(0.0, 0.55).turnGridsOff().setLabel("Colorado");
graph.getLeftAxis(1).setSpan(0.5, 1).turnGridsOff().setLabel("California");
```

(continues on next page)

(continued from previous page)

```
graph.getLeftAxis(2).setSpan(0.6, 0.8).turnGridsOff().setLabel("Kentucky");

graph.getBottomAxis().setPrimaryGridColor("rgba( 100, 100, 0, 0.5 )").setLabel("Year
↪");

graph
    .newSerie("colorado")
    .setWaveform(dataColorado)
    .autoAxis()
    .setYAxis(graph.getLeftAxis(0))
    .setLineColor("#CF4E4E")
    .setLineWidth(2);

graph
    .newSerie("california")
    .setWaveform(dataCalifornia)
    .autoAxis()
    .setYAxis(graph.getLeftAxis(1))
    .setLineColor("#3EA63E")
    .setLineWidth(2);


graph
    .newSerie("kentucky")
    .setWaveform(dataKentucky)
    .autoAxis()
    .setYAxis(graph.getLeftAxis(2))
    .setLineColor("#2F7C7C")
    .setLineWidth(2);


graph.draw();
```

CHAPTER

SIX

SCIENTIFIC AXES

While displaying scientific plots, you might find yourselves fighting with units and unit scaling. jsGraph helps you with that:

- Displaying scientific notation
- Displaying engineering notation
- Adding the SI prefix when scaling (mg, g, kg, . . .)

Let us consider the following example:

```
const years = [1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991,
↪ 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
↪ 2006, 2007, 2008, 2009, 2010, 2011, 2012];
const coalConsumption = [0.020494449, 0.020919039, 0.021248864, 0.021850876, 0.
↪02315959, 0.024146973, 0.024530887, 0.025317155, 0.026103415, 0.026055829, 0.
↪026108312, 0.025206791, 0.024850082, 0.025047841, 0.025224058, 0.025698446, 0.
↪026416749, 0.026247128, 0.02613963, 0.026560849, 0.027977463, 0.028029715, 0.
↪02855675, 0.030803576, 0.033055718, 0.034708407, 0.036478632, 0.038019204, 0.
↪038728538, 0.038927756, 0.041658958, 0.043534821, 0.043029446];

const g = new Graph('example-1');
g
    .resize(500, 300)
    .newSerie('coal_consumption')
    .setWaveform(Graph.newWaveform().setData(coalConsumption, years))
    .setLineColor('#A52B11')
    .setMarkers(true)
    .setMarkerStyle({ shape: 'circle', r: 4, fill: '#A52B11', stroke: 'white',␣
↪strokeWidth: '2px' })
    .autoAxis();

g.getBottomAxis().setLabel('year');
g.getLeftAxis().setLabel('World coal consumption');
g.draw();
```

Although years are fairly easy to understand, there's no unit in the y axis. What are we talking about ? Oranges ? Burgers over Bald Eagles ? Or kWh ? Or btu ? Of course you could include it directly in the label, but we can do better than that.

6.1 Setting axes units

Let's start setting the unit of the axis:

```
// Setting the SI unit
g.getLeftAxis().setUnit('Wh').setUnitWrapper('[', ']');
```

Obviously, the data we're dealing with is not in Wh, but it is given to us in some retarded unit, the **btu**, or british thermal unit. More specifically, in quadrillions of btus. (algthough we may wish for a coal-free world, we're actually going into the opposite direction). So we must scale the data accordingly. First of all, let's transform all of this into some normal SI unit, the Wh.

We could change the original data, but we could also make use of the `waveform.setScale` method:

```
// Setting the SI unit
w.setScale(
    ( 0.000293071 * 1000 )   // Conversion to Wh
    * 1e15                   // there are quadrilions of them
)
```

Ok, it's not looking pretty, but at least everything is in SI units, which is good. Let us see how we can play around with the units.

Warning: If you change the scaling of a waveform after binding it to a serie, you should rebind it with `serie.setWaveform( w )`.

6.2 Enabling the scientific scaling

First up, enable the scientific scaling:

```
// Setting the SI unit
g.getLeftAxis().setScientific( true );
```

Alright, now we're talking ! The axis now uses scientific units, and adds the exponent in the label.

Note: Scientific scaling allows jsGraph to try to optimize the unit display. If you want to keep complete control of the graph, you have access to the method `setExponentialFactor`, which scales the axis by `10^x` but does not affect the label. You may use it in the following condition for example:

```
// Force GWh display
g.getLeftAxis()
    .setScientific( false )
    .setUnit('GWh')
    .setExponentialFactor( -9 ); // Need to remove 1 billion to the Wh unit
```

However, note that there is also a possibility to force the exponent factor with scientific scaling (read on to find out how).

6.2.1 Enabling the engineering scaling

Engineering scaling is a type of scientific scaling, but where the exponent is always a power of 3. This eliminates the weird 10 ^{13} and replaces it by 10 ^{12} and multiplies all labels by 10. That's normally a lot more natural if you're dealing when quantities, like time, mass, etc.

6.2.2 Forcing the exponent

When in scientific scale, force the exponent of the axis using `axis.setScientificExponent( 3 )`, for example, to display the axis in `x1000 Wh`.

6.3 Using unit decades

Instead of displaying `10^x` in the scale, you may be interested in displaying a letter (e.g. n for nano, T for tera, etc...). Use the `setUnitDecade` together with `setScientificScaling` for that purpose:

```
g.getLeftAxis()
    .setScientific( true )
    .setUnitDecode( true );
```

See how now the data displayed is in `TWh`, instead of `Wh` x10 ^{13} .

jsGraph will replace the unit scaling by the following letters when appropriate:

- 10 $^{-15}$: `f`
- 10 $^{-12}$: `p`
- 10 $^{-9}$: `n`
- 10 $^{-6}$: `µ`
- 10 $^{-3}$: `k`
- 10 ^{3} : `k`
- 10 ^{6} : `M`
- 10 ^{9} : `G`
- 10 ^{12} : `T`
- 10 ^{15} : `E`

6.4 Log scale

We finish our discussion of the scientific axes by mentionning that you can ask the axis to display a logarithmic scale (only in base 10 for now). Use:

```
const g = new Graph('example-7');

let w = Graph.newWaveform()
    .setData(new Array(20)) // Use an empty array to define the length
    .rescaleX(-3, 0.5) // x = -3, -2.5, -2, etc...
    .math((y, x) => { return Math.exp(x) });

```

(continues on next page)

(continued from previous page)

```
let s = g
    .resize(500, 300)
    .newSerie('log')
    .setWaveform(w)
    .setLineColor('#3f9169')
    .setMarkers(true)
    .setMarkerStyle({ shape: 'circle', r: 4, fill: '#3f9169', stroke: 'white', ␣
↪strokeWidth: '2px' })
    .autoAxis();

g.getLeftAxis().setLogScale(true);
g.draw();
```

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search