Advanced Workshop

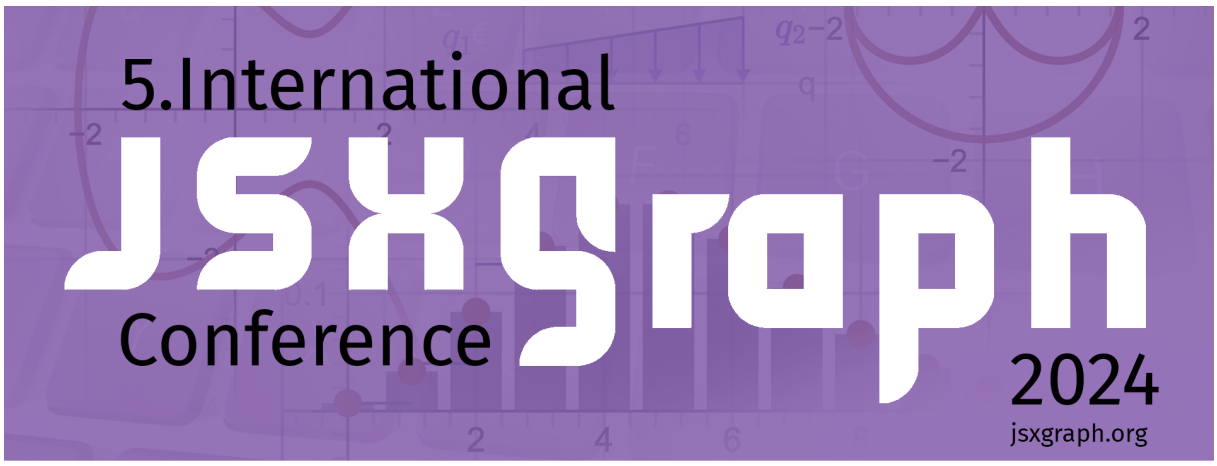
5. International JSXGraph Conference

Alfred Wassermann

UNIVERSITÄT BAYREUTH

10-10-2024, 6:30pm

Contents



October 8 – 10, 2024

UNIVERSITÄT BAYREUTH

1 Introduction

- I'm the JSXGraph lead developer, University of Bayreuth, Germany
- The plan of this workshop is to show more advanced features of JSXGraph, with a focus on elements which have been introduced since October 2023 and - maybe - learn some lesser known features of JSXGraph.

2 Preliminaries

2.1 Where to get information

- JSXGraph wiki: https://jsxgraph.org/wiki
- JSXGraph examples database: https://jsxgraph.org/share
- JSXGraph API documentation: https://jsxgraph.org/docs

All examples include source code.

2.2 Include JSXGraph - skeleton page

How to include JSXGraph:

- Local copy of `jsxgraphcore.js` and `jsxgraph.css` (download or npm)
- https://cdn.jsdelivr.net/npm/jsxgraph/distrib/jsxgraphcore.js (or another CDN)

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>JSXGraph template</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
    <link href="https://cdn.jsdelivr.net/npm/jsxgraph/distrib/jsxgraph.css" rel="
        stylesheet" type="text/css" />
    <script src="https://cdn.jsdelivr.net/npm/jsxgraph/distrib/jsxgraphcore.js"
        type="text/javascript" charset="UTF-8"></script>

    <!-- The next line is optional: MathJax -->
    <script src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-chtml.js" id="
        MathJax-script" async></script>
  </head>
  <body>

  <div id="jxgbox" class="jxgbox" style="width:500px; height:200px;"></div>

  <script>
    var board = JXG.JSXGraph.initBoard('jxgbox', {boundingbox: [-5, 2, 5, -2]});
  </script>

  </body>
</html>
```

3 Themes

With v1.7.0 the possibility to use color themes has been introduced. As a template theme, the `mono_thin` theme is included in `jsxgraphcore.js`. A theme can be enabled by using the board attribute `theme`. At the time being, the theme can not be changed during the run time of a board.

```
JXG.Options.text.fontSize = 12;

const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-5, 5, 5, -5], axis: true,
    theme: 'mono_thin'
});

var a = board.create('slider', [[1, 4], [3, 4], [-10, 1, 10]]);
var p1 = board.create('point', [1, 2]);
var ci1 = board.create('circle', [p1, 0.7]);
var cu = board.create('functiongraph', ['x^2']);
var l1 = board.create('line', [2, 3, -1]);
var l2 = board.create('line', [-5, -3, -1], { dash: 2 });
var i1 = board.create('intersection', [l1, l2]);
var pol = board.create('polygon', [[1, 0], [4, 0], [3.5, 1]]);
var an = board.create('angle', [pol.vertices[1], pol.vertices[0], pol.vertices[2]])
    ;
var se = board.create('sector', [pol.vertices[1], pol.vertices[2], pol.vertices
    [0]]);
var ci1 = board.create('circle', [[-3, -3], 0.7], { center: { visible: true } });
```

- See it live at https://jsfiddle.net/nqxL9ejf/

3.1 Create your own theme

If you intend to create your own theme, take the `mono_thin` as a starting point, see source code of `mono_thin.js` on github.

- Create a file like `mono_thin.js`

```
JXG.themes['theme_name'] = {

    board: {
        showInfobox: false,
        showCopyright: true,
        defaultAxes: {
            x: {
                ticks: {
                    minorTicks: 0,
                    majorHeight: 10,
                    majorTickEndings: [1, 0]
                }
            },
            y: {
```

```
                ticks: {
                    minorTicks: 0,
                    majorHeight: 10,
                    majorTickEndings: [0, 1]
                }
            }
        }
    },
    ...
};
```

- Load this file in your HTML file *after* loading `jsxgraphcore.js`
- Set the board attribute `theme: 'theme_name'`

4 Plotting of implicit curves

Finally: JSXGraph has implicit curves!

```
onst board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-5, 5, 5, -5], axis:true
});

var a, b, c, f;
a = board.create('slider', [[-3, 4.5], [3, 4.5], [-3, 1, 3]], {
    name: 'a', stepWidth: 0.1,
    snapValues: [0.0],
    snapValueDistance: 0.4
});
b = board.create('slider', [[-3, 4], [3, 4], [-3, 1, 3]], {
    name: 'b', stepWidth: 0.1,
    snapValues: [0.0],
    snapValueDistance: 0.4
});

// Circle
f = 'x**2 + y**2 - 4';

// Ellipse
// f = '4 * x**2 + 4 * x * y + 2 * y**2 - 2 * y';

// "Dynamic" conic section
//f = (x, y) => x ** 2 - 2 * x * y - 2 * x + (a.Value() + 1) * y ** 2 + (4 * a.
    Value() + 2) * y + 4 * a.Value() - 3;

// Difficult curves
// f = 'abs(x) - abs(y) - a.Value()';

// Loop
// f = 'x**3 - 2 * x * y + y**3';

// Cusp
// f = '-(y**2) + x**3';

// Watt's curve
//f = (x, y) => (x ** 2 + y ** 2) * (x ** 2 + y ** 2 - (a.Value() ** 2 + b.Value()
    ** 2 - 1)) ** 2 + 4 * a.Value() ** 2 * y ** 2 * (x ** 2 + y ** 2 - b.Value() **
     2);

c = board.create('implicitcurve', [f], {
    strokeWidth: 2,
    strokeColor: JXG.palette.red,
    strokeOpacity: 0.8,
    resolution_outer: 1,
    resolution_inner: 1
});
```

- See it live at https://jsfiddle.net/vys87dtj/28/
- Note that here we use the new slider attributes `snapValues` and `snapValueDistance` to easily

choose the slider value 0.

- The plotting algorithm for the element `implicitcurve` comes with lot of options, see the API docs. Most important for the detection of isolated curve components are `resolution_outer` and `resolution_inner`. Small values given there increase the chance that isolated components are detected, but also increase the run time of the algorithm.

If performance is an issue and the implicit curve is static, the attribute `needsRegularUpdate:false` for the curve might help.

4.1 Plot of contour lines

Here is a simple example how to plot contour lines using implicit plots

```
let contours = [0.5, 1, 1.5, 2];

for (let i = 0; i < contours.length; i++) {
    board.create("implicitcurve", [
        (x, y) => x ** .5 * y ** .5 - contours[i],
        [0.25, 3], [0.5, 4]   // Domain
    ], {
        strokeWidth: 2,
        strokeColor: JXG.palette.red,
        strokeOpacity: (1 + i) / contours.length,
        needsRegularUpdate: false
    });
}
```

- See it live at https://jsfiddle.net/hkwq2905/

4.2 Example: Lagrange multiplier

- Extended example by Wigand Rathmann to visualize the *method of Lagrange multipliers* https://jsfiddle.net/1tsnxfp9/3/

4.3 Improved `otherintersection` for implicit curves (and other elements)

This element computes the “other” intersection point of e.g. the intersection of a line and a circle. This is helpful if one intersection point is already determined by the construction. For example, two circles may be defined both through a common point.

```
let C = board.create('point', [0, 0,], {name: 'C'});
let A = board.create('point', [-2, -1], {
    name: 'A',
    label: {anchorX: 'right', offset: [-10, 0]}
});
let D = board.create('point', [-2, -3], {name: 'D'});

let circ1 = board.create('circle', [C, A]);
let circ2 = board.create('circle', [D, A]);

let I1 = board.create('intersection', [circ1, circ2, 1], {name: 'I_1'});
let I2 = board.create('otherintersection', [circ1, circ2, A], {
    name: 'I_2',
    label: {anchorX: 'right', offset: [-10, 0]}
});
```

- See it live at https://jsfiddle.net/3coh2rj4/1/

New is that instead of supplying the intersection point that is to be avoided, a list of intersection points can be supplied.

Example: visualize group law on elliptic curves

```
var curve, A, B, line, C;

curve = board.create('implicitcurve', ['-(y**2) + x**3 - 2 * x + 1']);
A = board.create('glider', [-1.5, 0, curve]);
B = board.create('glider', [0, 1, curve]);
line = board.create('line', [A, B]);
C = board.create('otherintersection', [line, curve, [A, B]]);
```

- See example.

5 Complex numbers

Since many years JSXGraph contains arithmetics for complex numbers. This seems to be a lesser used area. Recently, we got the request to supply an algorithm to compute *all* roots of a polynomial with real coefficients.

Multiplication of numbers in the complex plane

```
var p_z1 = board.create('point', [0, 1], {name: 'z_1'});
var p_z2 = board.create('point', [0, -1], {name: 'z_2'});

var p_z3 = board.create('point', [
  () => {
    let z1 = new JXG.Complex(p_z1.X(), p_z1.Y());
    let z2 = new JXG.Complex(p_z2.X(), p_z2.Y());
    let z = JXG.C.mult(z1, z2);
```

```
    return [z.real, z.imaginary];
  }
], {name: 'z_3', color: 'blue'});
```

- See https://jsfiddle.net/L7cmn3q4/3/

Complex roots of polynomial with real coefficients

- Fascinating algorithm due to Weierstraß, Durand, Kerner, Aberth, and Ehrlich.

```
const board = JXG.JSXGraph.initBoard(BOARDID, {
    boundingbox: [-9, 9, 9, -9],
    axis: true
});

// Coefficients [a_0, ..., a_n] of a polynomial,
// starting with a_0, a_1, ...
var coefficients = [-1, 3, -9, 1, 0, 0, -8, 9, -9, 1];
var roots = JXG.Math.Numerics.polzeros(coefficients);

for (let i = 0; i < roots.length; i++) {
    board.create('point', [roots[i].real, roots[i].imaginary], {
        withLabel: false,
        fixed: true
    });
}

board.create('text', [-8, 3, JXG.Math.Numerics.generatePolynomialTerm(coefficients,
     coefficients.length - 1, 'x', 0)]);
```

- See it live at Complex roots of polynomial with real coefficients.

6 Statistics methods

- Gamma function: `JXG.Math.gamma(z)`
- Random numbers for the following distributions:
  - uniform,
  - normal,
  - exponential,
  - beta,
  - gamma,
  - chi-square,
  - F,
  - T,
  - binomial,

- geometric,
- hypergeometric,
- Poisson,
- Pareto

6.1 Example: random numbers in the binomial distribution

- Visualize the binomial distribution, see Wikipedia
- Example by Tom Berendt, see API docs

```
const board = JXG.JSXGraph.initBoard('jxgbox',
    { boundingbox: [-1.7, .5, 30, -.03], axis: true });

// We do 3 experiments
let runs = [
    [0.5, 20, 'blue'],
    [0.7, 20, 'green'],
    [0.5, 40, 'red'],
];

let labelY = .4;
runs.forEach((run, i) => {
    // Create a legend
    board.create('segment', [[7, labelY - (i / 50)], [9, labelY - (i / 50)]], {
        strokeColor: run[2] });
    board.create('text', [10, labelY - (i / 50), `p=${run[0]}, n=${run[1]}`]);

    // In each experiment we create 50000 random numbers according to the binomial
        distribution
    let x = Array(50000).fill(0).map(() => JXG.Math.Statistics.randomBinomial(run
        [1], run[0]));

    // Analyse the data as histogram
    let res = JXG.Math.Statistics.histogram(x, {
        bins: 40,
        density: true,
        cumulative: false,
        range: [0, 40]
    });

    // Plot the histogram
    board.create('curve', [res[1], res[0]], { strokeColor: run[2] });
});
```

- See it live at https://jsfiddle.net/yu5jrzgo/

6.2 New function `JXG.Math.Statistics.histogram`

- Puts data into bins which then can be plotted.

- Another example by Tom Berendt, see API docs, for the gamma distribution, using two boards.

```
const board = JXG.JSXGraph.initBoard('jxgbox',
    { boundingbox: [-1.7, .5, 20, -.03], axis: true });
const board2 = JXG.JSXGraph.initBoard('jxgbox2',
    { boundingbox: [-1.6, 1.1, 20, -.06], axis: true });

// We do 7 experiments
let runs = [
    [0.5, 1.0, 'brown'],
    [1.0, 2.0, 'red'],
    [2.0, 2.0, 'orange'],
    [3.0, 2.0, 'yellow'],
    [5.0, 1.0, 'green'],
    [9.0, 0.5, 'black'],
    [7.5, 1.0, 'purple'],
];

let labelY = .4
runs.forEach((run, i) => {
    // Create a legend
    board.create('segment', [[7, labelY - (i / 50)], [9, labelY - (i / 50)]], {
        strokeColor: run[2] });
    board.create('text', [10, labelY - (i / 50), `k=${run[0]}, theta=${run[1]}`]);

    // In each experiment we create 50000 random numbers according to the gamma
        distribution
    let x = Array(50000).fill(0).map(() => JXG.Math.Statistics.randomGamma(run[0],
        run[1]));

    // Plot density
    let res = JXG.Math.Statistics.histogram(x, { bins: 50, density: true,
        cumulative: false, range: [0, 20] });
    board.create('curve', [res[1], res[0]], { strokeColor: run[2], strokeWidth: 2
        });

    // Plot cumulative density
    res = JXG.Math.Statistics.histogram(x, { bins: 50, density: true, cumulative:
        true, range: [0, 20] });
    res[0].unshift(0);  // add zero to front so cumulative starts at zero
    res[1].unshift(0);
    board2.create('curve', [res[1], res[0]], { strokeColor: run[2], strokeWidth: 2
        });
});
```

- See it live at https://jsfiddle.net/tk6e7b1n/

6.3 Visualization for distributions and densities

See the excellent page by Wigand Rathmann. These statistics examples will be copied to https://jsxgraph.org/share soon.

7 Label positioning for lines, circles, and curves

The positioning of labels on 1-dimensional element like lines, circles, curves has been simplified. The general syntax is `position: 'len side'`, where `len` is a number (optionally with unit). Each 1-dimensional element has a direction and `side` determines on which side of the element (when looking in the given direction) the label is positioned. The value of `side` is either 'left' or 'right'.

Possible units for `len` are:

- xfr, denoting a fraction of the whole. x is expected to be a number between 0 and 1.
- x%, a percentage. x is expected to be a number between 0 and 100.
- x, a number: only possible for line elements and circles. For lines, the label is positioned x user units from the starting point. For circles, the number is interpreted as degree, e.g. 45°. For everything else, 0 is taken instead.
- xpx, a pixel value: only possible for line elements. The label is positioned x pixels from the starting point. For non-lines, 0% is taken instead.

`distance:number` determines the distance of the label from the element, the number given will be multiplied by the size of the labels' text box.

Examples

```
var l1 = board.create('line', [[-3, 2], [3, 2]], {
    name: 'line_1',
    straightFirst: false,
    straightLast: false,
    withLabel: true,
    point1: { visible: true, name: 'A'},
    point2: { visible: true, name: 'B'},
    label: {
        anchorX: 'left',
        anchorY: 'middle',
        offset: [0, 0],
        fontSize: 16,
        fixed: true,
        distance: 1.5,
        position: '1 right'
    }
});
```

- See https://jsfiddle.net/5mLuvqye/

```
var c1 = board.create('circle', [[0, 0], 3], {
    name: 'c_1',
    withLabel: true,
    label: {
        anchorX: 'middle',
        anchorY: 'middle',
```

```
        offset: [0, 0],
        fontSize: 32,
        distance: 2.5,
        position: '45 right'
    }
});
```

- See https://jsfiddle.net/0yr8enhj/

```
var cu1 = board.create('functiongraph', ['3 * sin(x)', -3, 3], {
    name: 'cu_1',
    withLabel: true,
    label: {
        anchorX: 'middle',
        anchorY: 'middle',
        offset: [0, 0],
        fontSize: 32,
        distance: 1.5,
        position: '0.4fr right'
    }
});
```

- See https://jsfiddle.net/2gL3f6jb/

8 Major overhaul of elements grid and axis

The grid element is a very old element in JSXGraph, showing grid lines. It was a feature of the grid to *not* adapt to the zoom level of the board. Later, it was more or less superseeded by the default axes which come with *infinite ticks lines*.

Example: https://jsfiddle.net/yrkgL1wa/

Now, we revamped both, the axis element and the grid element, and see the grid element as useful in situations where coordinate axes have not yet been introduced, but the grid lines *suggest the presence of some sort of mathematical graphics*.

8.1 Axis

The main new feature for axes is that it can be controlled what happens if the user shifts the viewport (pans the board). The most important new attribute for handling this is `position`. It can attain the following values:

- 'static': Standard behavior of the axes as know in JSXGraph.

- ‘fixed’: The axis is placed in a fixed position. Depending on the attribute anchor, it is positioned to the right or left of the edge of the board as seen from the axis with a distance defined in distanceBoarder. The axis will stay at the given position, when the user navigates through the board.
- ‘sticky’: This mixes the two settings static and fixed. When the user shifts the viewport, the axis remains in the visible area (taking into account anchor and anchorDist). If the axis itself is in the visible area, the axis can be moved by navigation.
- See it live at API docs

The attribute `anchorDist` controls how far from the edge of the board the axis should stick or be fixed. The units are the same as for the line labels above.

Example for ‘fixed’:

```
var axis1, axis2, circle;

board.create('axis', [[0,0],[1,0]],{
    position: 'fixed',
    anchor: 'right',
    anchorDist: '0.1fr'
});

board.create('axis', [[0,0],[0,1]], {
    position: 'fixed',
    anchor: 'left',
    anchorDist: 1
});

board.create('circle', [[5, 5], 2.5]);
```

Example for ‘sticky’:

```
var axis1, axis2, f;

board.create('axis', [[0,0],[1,0]],{
    position: 'sticky',
    anchor: 'right',
    anchorDist: '0.2fr'
});

board.create('axis', [[0,0],[0,1]], {
    position: 'sticky',
    anchor: 'right left',
    anchorDist: '75px'
});

f = board.create('functiongraph', ['x^2]);
```

In the above examples, we create new axes with these attributes. The same attributes can be set to the default axes like this:

```
const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-9, 17, 9, -17],
    axis: true,
    defaultAxes: {
        x: {
            position: 'sticky',
            anchor: 'right',
        },
        y: {
            position: 'fixed',
            anchor: 'left'
        }
    }
});
```

- For styling aspects of axes see also this example

8.2 Grid

```
const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-9, 17, 9, -17],
    zoom: {
        min: 0.5,
        max: 2
    }
```

Features of the overhauled grid element:

- There are major and minor lines (“ticks”), in analogy to axes.
- There are other shapes than lines available.
- The grid can adapt to zooming, in sync with lines.
- See https://jsxgraph.org/docs/symbols/Grid.html.
- See also the presentation of Carsten Miller and Andreas Walter.

8.3 Tick labels as fractions

```
board.create('axis', [[0,1], [1,1]], {
    ticks: {
        label: {
            toFraction: true,
            useMathjax: true,
            display: 'html',
            anchorX: 'middle',
            offset: [0, -10]
        }
    }
});
```

- See it live at https://jsfiddle.net/bm5pe0w9/
- See also https://jsxgraph.org/docs/symbols/Ticks.html#labels

9 Develop (with) JSXGraph

When developing JSXGraph applications, it is sometimes helpful if the source code of JSXGraph is loaded into a web page. For example, one can get the exact error location in the source code and might get a clue of the problem from the code. Of course, for developing JSXGraph the source code it is indispensible. Unfortunately, starting with v1.5.0, the import of JSXGraph source code as module was not easily possible. Fortunately, this has changed.

To be precise, in v1.9.0 the internal module imports have been refactored. Now, all modules are imported with the file name ending `*.js`. The consequence is that even with very simple web servers it is possible to server JSXGraph as a module.

Here, we will give a very short introduction on how to use the source code of JSXGraph.

9.1 Get the source code

The released source code can be downloaded from npmjs. The latest, possibly unreleased source code is available at github. The very latest code is in the branch `develop`.

The code in branch `develop` might be broken sometimes.

Most users use some sort of IDE and do not use a CLI or terminal. Here, I describe the approach using a terminal.

- First, git has to be installed.
- To get the code, open a terminal, navigate to the folder which should contain the SXGraph code and type:

```
git clone https://github.com/jsxgraph/jsxgraph.git jsxgraph
```

- By default, the branch `main` (containing the stable version) is checked out. In the subdirectory `jsxgraph` you can now switch to the branch `develop` by typing:

```
git checkout develop
```

9.2 Develop locally

Here is my personal workflow to develop JSXGraph:

For JSXGraph development, I always create test files in a folder `jsxdev` outside of the folder `jsxgraph`. Therefore, I'm starting a simple python-based web server in `jsxdev` like this:

```
python3 -m http.server -d ..
```

A typical test page in my folder `jsxdev` looks like this:

```
<!DOCTYPE html>
<html>

<head>
    <title>JSXGraph</title>
    <meta name="robots" content="noindex, nofollow">
    <meta charset="utf-8">

    <link rel="stylesheet" type="text/css" href="../jsxgraph/distrib/jsxgraph.css"
        />
</head>

<body>
    <div id="jxgbox" class="jxgbox" style="width:600px; height:600px"></div>
    <script type="module">
        "use strict";
        import JXG from '../jsxgraph/src/index.js';

        var board = JXG.JSXGraph.initBoard('jxgbox', {
            boundingbox: [-5, 5, 5, -5],
            axis: true
        });
    </script>
</body>

</html>
```

Important are:

- `<script type="module"> ... </script>`
- `import JXG from '../jsxgraph/src/index.js';`
- Open the folder in a web browser using the address `http://localhost:8000/jsxdev`

10 The fine points of text positioning

Above we discussed how to position labels of 1-dimensional objects like lines, circles and curves. Now, we look at more subtle possibilities to position and format texts and labels.

10.1 Text positioning

- See the following examples live at https://jsfiddle.net/4ytp9x3n/1/

Texts

- That is the coordinates of a text element determine the position of the anchor point of the text box. In the example below, the position of the text's anchor point is at $[0, 0]$.
- A text element has 9 possible anchor points, namely all combinations of values of the attributes `anchorX` and `anchorY`.
- Additionally, a text element can be rotated by the attribute `rotate` (values are given in degree).
- Text elements can be styled with a customized CSS class.

```
// Fine tune text position
var txt1 = board.create('text', [0, 0, 'hello'], {
    cssClass: 'myText',
    anchorX: 'left',   // 'left', 'middle', 'right'
    anchorY: 'bottom', // 'top', 'middle', 'bottom'
    // rotate: 90,
    fontSize: 24
});
```

Label

- Labels of elements can be customized by the same attributes as text elements. Additionally, labels have the attributes `position`, `distance` and `offset`. The latter is an array of length 2 and sets an additional offset given in pixels.

```
// Fine tune position of a label
var seg = board.create('segment', [[1, 2], [4, 2]], {
    point1: {visible: true},
    point2: {visible: true},
    name: 'line',
    withLabel: true,
    label: {
        cssClass: 'myText',
        position: '1 left',
        distance: 0,
        offset: [0, 0],
        anchorY: 'bottom',
        anchorX: 'left',
        rotate: 45,
```

```
        fixed: false
    }
});
```

- Note in passing: the difference between `withLabel:false` and `label: {visible: false}` is that in the first case, the label is not created and thus does not exist at all. In the second case, the label exists, but is hidden (`display:none` in CSS). This might make a difference in performance if there are hundreds of labels.

Texts with an anchor

- It is possible to set the attribute `anchor` for a regular text element. Then, the text element acts like sort of an additional label of its anchor element.
- If the anchor element is supplied, the given coordinates determine the relative position (in user coordinates) of the text element from its anchor element.
- In v1.10.0 the default anchor point on an 1-dimensional element of such a text element is in the middle. The new features of the attributes `position` and `distance` are ignored (yet?).

```
// Fine tune position of a label
var seg = board.create('segment', [[1, 2], [4, 2]], {
    // ... as above
});

// Bind text element to an anchor element
var txt2 = board.create('text', [0, 0, 'label'], {
    cssClass: 'myText',
    anchor: seg,
    position: '10% left', // ignored in 1.10.0
    anchorX: 'left',      // 'left', 'middle', 'right'
    anchorY: 'bottom',    // 'top', 'middle', 'bottom'
    rotate: 0,
    fontSize: 24,
    fixed: false,
});
```

10.2 Formatting numbers

- See the following examples live at https://jsfiddle.net/8xmqk0jg/2/

Static texts

- In case a text consists of a number (only), then by default it is printed unformatted.
- Formatting is enabled by setting the attribute `formatNumber:true`.
- Formatting is one of the following:
  - truncating to certain number of fixed digits e.g. `digits:2`.

  - formating with internationalization support, see workshop in 2023 and MDN for possible options.
  - formatting as fraction by setting `toFraction:true`.

```
// Static text
var txt1 = board.create('text', [1, 4, 1.74999999], {
        cssClass: 'myText',
        fontSize: 24,
        digits: 2,
        toFraction: true,
        formatNumber: true
    });
```

Dynamic texts

- In a dynamic text the string, number to be displayed is given by a function. For example, we want to show the slope of the following line.

```
var line = board.create('line', [[1, 2], [4, 3]], {
        point1: {
          visible: true,
          snapToGrid: true,
          snapSizeX: 0.1,
          snapSizeY: 0.1
      },
        point2: {
          visible: true,
          snapToGrid: true,
          snapSizeX: 0.1,
          snapSizeY: 0.1
      },
    });
```

- In such a case, the task of formatting is up to the developer.
- We start with an example using a fixed number of digits.

```
// Dynamic text (with anchor) rounded to 5 digits
var txt2 = board.create('text', [0, -0.5,
    () => `slope=${line.Slope().toFixed(5)}`
], {
    anchor: line,
    cssClass: 'myText',
    fontSize: 24
});
```

- The next example uses internationalization:

```
// Customized internationalization function:
const myFormat = function(value) {
    return Intl.NumberFormat(
      'de-DE', {
```

```
            unitDisplay: 'narrow',
            maximumFractionDigits: 2
        }).format(value);
};

// Use internationalization
var txt2 = board.create('text', [0, -1,
    () => `slope=${myFormat(line.Slope())}`
], {
    anchor: line,
    cssClass: 'myText',
    fontSize: 24
});
```

- In the last example, we convert the slope into a fraction

```
// Same with fractions
var txt3 = board.create('text', [0, -1.5,
    () => `slope=${JXG.toFraction(line.Slope())}`
], {
    anchor: line,
    cssClass: 'myText',
    fontSize: 24
});
```

11 Calculus in 3D

Recommendation: for 3D applications always disable panning or - at least - restrict it to two finger gestures. Otherwise, the 3D view can not be manipulated with a finger on touch devices.

```
const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-8, 8, 8, -8],
    keepaspectratio: false,
    axis: false,
    pan: {
        needTwoFingers: true  // !!!
    }
});
```

11.1 3D axes position

```
var board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-8, 8, 8, -8],
    keepaspectratio: false,
    axis: false,
    pan: {
        needTwoFingers: true
    }
```

```
});

var bound = [-4, 6];
var view = board.create('view3d',
    [[-6, -3], [8, 8],
    [bound, bound, bound]],
    {
        projection: 'central',
        trackball: { enabled: true },

        // Main axes
        axesPosition: 'border',  // 'border', 'center', 'none'

        // Axes at center
        xAxis: { strokeColor: 'blue', strokeWidth: 2 },

        // Axes at border
        xAxisBorder: {
            strokeColor: 'blue',
            // lastArrow: true,
            ticks3d: {
                ticksDistance: 1,
                strokeColor: 'blue',
                drawLabels: true,
                label: {
                }
            }
        },

        // Planes
        xPlaneRear: { visible: true, mesh3d: { visible: false } },

        // Axes on planes
        xPlaneRearYAxis: { visible: false},
        xPlaneRearZAxis: { visible: false},
        yPlaneRearXAxis: { visible: false },
        yPlaneRearZAxis: { visible: false },
        zPlaneRearXAxis: { visible: false },
        zPlaneRearYAxis: { visible: false }
    });

var txt1 = view.create('text3d', [[1, 2, 1], 'hello'], {
    fontSize: 20
});

var p1 = view.create('point3d', [6, 0, -4], { withLabel: false });

var txt2 = view.create('text3d', [
    () => [p1.X(), p1.Y(), p1.Z() + 1],
    'point'], { withLabel: false });
```

- Note that in v1.10.0 3D texts were introduced. These texts are not projected (yet?).
- See it live at https://jsfiddle.net/94vLdr1f/

11.2 3D vector fields

```
const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-8, 8, 8, -8],
    keepaspectratio: false,
    axis: false,
    pan: {
        needTwoFingers: true
    }
});

const view = board.create('view3d',
    [
        [-6, -3],
        [8, 8],
        [[-3, 3], [-3, 3], [-3, 3]]
    ], {});

var vf = view.create('vectorfield3d', [
    [(x, y, z) => Math.cos(y), (x, y, z) => Math.sin(x), (x, y, z) => z],
    [-2, 5, 2], // x from -2 to 2 in 5 steps
    [-2, 5, 2], // y
    [-2, 5, 2]  // z
], {
    strokeColor: 'red',
    scale: 0.5
});
```

- See it live at https://jsfiddle.net/f9u4a6ez/ and in examples database

11.3 3D contour lines

This is an example to show contour lines in a 3D plot, as well as project the contour lines to the rear z-plane. This code has been suggested by Wigand Rathmann. Eventually, contour lines and their projections will be made available as separate elements in JSXGraph.

```
const board = JXG.JSXGraph.initBoard('jxgbox', {
    boundingbox: [-8, 8, 8, -8],
    keepaspectratio: false,
    axis: false,
    pan: {
        needTwoFingers: true
    }
});

var r = board.create('slider', [[-4, -7], [4, -7], [-5, 2, 5]], {name: 'r'});
var v = board.create('slider', [[-4, -7.5], [4, -7.5], [-5, 0.5, 5]], {
    name: 'v',
    snapValues: [-4, -2, -1, 0, 1, 2, 4],
    snapValueDistance: 0.3
});
```

```
var bound = [-5, 5];
var view = board.create('view3d',
    [[-4, -3], [8, 8],
    [bound, bound, bound]], {
        projection: 'central',
        axesPosition: 'border',
        xPlaneRear: {visible: false},
        yPlaneRear: {visible: false},
        xPlaneRearYAxis: {visible: false},
        xPlaneRearZAxis: {visible: false},
        yPlaneRearXAxis: {visible: false},
        yPlaneRearZAxis: {visible: false},
        zPlaneRearXAxis: {visible: false},
        zPlaneRearYAxis: {visible: false},
        xPlaneFrontYAxis: {visible: false},
        xPlaneFrontZAxis: {visible: false},
        yPlaneFrontXAxis: {visible: false},
        yPlaneFrontZAxis: {visible: false},
        zPlaneFrontXAxis: {visible: false},
        zPlaneFrontYAxis: {visible: false}
});

// 3D surface (function graph)
var F_txt = 'V(r) * sin(x) * cos(y)';
var surf = view.create('functiongraph3d', [F_txt, bound, bound]);

// Invisible 2D implicit curve
var implicit = board.create('implicitcurve', ['V(r) * sin(x) * cos(y) - V(v)', [-5,
     5], [-5, 5]], {
    visible: false
});

var contour = view.create('curve3d', [[], [], []], {strokeWidth: 3});
contour.updateDataArray = function() {
    var i,
        le = implicit.numberPoints;

    // Copy 2D points of the surface to 3D
    this.X = [];
    this.Y = [];
    this.Z = [];
    for (i = 0; i < le; i++) {
        this.X.push(implicit.points[i].usrCoords[1]);
        this.Y.push(implicit.points[i].usrCoords[2]);
        this.Z.push(v.Value());
    }
};

var contour2 = view.create('curve3d', [[], [], []], {strokeWidth: 3, strokeColor: '
    black'});
contour2.updateDataArray = function() {
    var i,
        le = implicit.numberPoints;

    // Copy 2D points of the surface to 3D
```

```
    this.X = [];
    this.Y = [];
    this.Z = [];
    for (i = 0; i < le; i++) {
        this.X.push(implicit.points[i].usrCoords[1]);
        this.Y.push(implicit.points[i].usrCoords[2]);
        this.Z.push(bound[0]);
    }
};

board.update();
```

- See it live at https://jsfiddle.net/65j7xLwq/

11.4 Upcoming elements

- 2D curve `contour`
- 3D curve `contour3d`
- `projectCurve2DTo3D`
- `projectCurve3DTo2D`