FLOT Documentation

**About**

flot is a JavaScript plotting library for engineering and scientific applications derived from Flot: <http://www.flotcharts.org/>

Take a look at the the examples in examples/index.html; they should give a good impression of what flot can do, and the source code of the examples is probably the fastest way to learn how to use flot.

**Installation**

Just include the JavaScript file after you've included jQuery.

Generally, all modern browsers are supported.

You need at least jQuery 1.2.6, but try at least 1.3.2 for interactive charts because of performance improvements in event handling.

**Basic usage**

Create a placeholder div to put the graph in:

<div id="placeholder"></div>

You need to set the width and height of this div, otherwise the plot library doesn't know how to scale the graph. You can do it inline like this:

<div id="placeholder" style="width:600px;height:300px"></div>

You can also do it with an external stylesheet. Make sure that the placeholder isn't within something with a display:none CSS property - in that case, Flot has trouble measuring label dimensions which results in garbled looks and might have trouble measuring the placeholder dimensions which is fatal (it'll throw an exception).

Then when the div is ready in the DOM, which is usually on document ready, run the plot function:

$.plot($("#placeholder"), data, options);

Here, data is an array of data series and options is an object with settings if you want to customize the plot. Take a look at the examples for some ideas of what to put in or look at the [API reference](https://github.com/flot/flot/blob/master/API.md). Here's a quick example that'll draw a line from (0, 0) to (1, 1):

$.plot($("#placeholder"), [ [[0, 0], [1, 1]] ], { yaxis: { max: 1 } });

The plot function immediately draws the chart and then returns a plot object with a couple of methods.

## Introduction

Consider a call to the plot function:

var plot = $.plot(placeholder, data, options)

The placeholder is a jQuery object or DOM element or jQuery expression that the plot will be put into. This placeholder needs to have its width and height set as explained in the [README](https://github.com/flot/flot/blob/master/README.md) (go read that now if you haven't, it's short). The plot will modify some properties of the placeholder so it's recommended you simply pass in a div that you don't use for anything else. Make sure you check any fancy styling you apply to the div, e.g. background images have been reported to be a problem on IE 7.

The plot function can also be used as a jQuery chainable property. This form naturally can't return the plot object directly, but you can still access it via the 'plot' data key, like this:

var plot = $("#placeholder").plot(data, options).data("plot");

The format of the data is documented below, as is the available options. The plot object returned from the call has some methods you can call. These are documented separately below.

Note that in general Flot gives no guarantees if you change any of the objects you pass in to the plot function or get out of it since they're not necessarily deep-copied.

## Data Format

The data is an array of data series:

[ series1, series2, ... ]

A series can either be raw data or an object with properties. The raw data format is an array of points:

[ [x1, y1], [x2, y2], ... ]

E.g.

[ [1, 3], [2, 14.01], [3.5, 3.14] ]

Note that to simplify the internal logic in Flot both the x and y values must be numbers (even if specifying time series, see below for how to do this). This is a common problem because you might retrieve data from the database and serialize them directly to JSON without noticing the wrong type. If you're getting mysterious errors, double check that you're inputting numbers and not strings.

If a null is specified as a point or if one of the coordinates is null or couldn't be converted to a number, the point is ignored when drawing. As a special case, a null value for lines is interpreted as a line segment end, i.e. the points before and after the null value are not connected.

Lines and points take two coordinates. For filled lines and bars, you can specify a third coordinate which is the bottom of the filled area/bar (defaults to 0).

The format of a single series object is as follows:

{

color: color or number

data: rawdata

label: string

lines: specific lines options

bars: specific bars options

points: specific points options

xaxis: number

yaxis: number

clickable: boolean

hoverable: boolean

shadowSize: number

highlightColor: color or number

}

You don't have to specify any of them except the data, the rest are options that will get default values. Typically you'd only specify label and data, like this:

{

label: "y = 3",

data: [[0, 3], [10, 3]]

}

The label is used for the legend, if you don't specify one, the series will not show up in the legend.

If you don't specify color, the series will get a color from the auto-generated colors. The color is either a CSS color specification (like "rgb(255, 100, 123)") or an integer that specifies which of auto-generated colors to select, e.g. 0 will get color no. 0, etc.

The latter is mostly useful if you let the user add and remove series, in which case you can hard-code the color index to prevent the colors from jumping around between the series.

The "xaxis" and "yaxis" options specify which axis to use. The axes are numbered from 1 (default), so { yaxis: 2} means that the series should be plotted against the second y axis.

"clickable" and "hoverable" can be set to false to disable interactivity for specific series if interactivity is turned on in the plot, see below.

The rest of the options are all documented below as they are the same as the default options passed in via the options parameter in the plot command. When you specify them for a specific data series, they will override the default options for the plot for that data series.

Here's a complete example of a simple data specification:

[ { label: "Foo", data: [ [10, 1], [17, -14], [30, 5] ] },

{ label: "Bar", data: [ [11, 13], [19, 11], [30, -7] ] }

]

## Plot Options

All options are completely optional. They are documented individually below, to change them you just specify them in an object, e.g.

var options = {

series: {

lines: { show: true },

points: { show: true }

}

};

$.plot(placeholder, data, options);

## Customizing the legend

legend: {

show: boolean

labelFormatter: null or (fn: string, series object -> string)

labelBoxBorderColor: color

noColumns: number

position: "ne" or "nw" or "se" or "sw"

margin: number of pixels or [x margin, y margin]

backgroundColor: null or color

backgroundOpacity: number between 0 and 1

container: null or jQuery object/DOM element/jQuery expression

sorted: null/false, true, "ascending", "descending", "reverse", or a comparator

}

The legend is generated as a table with the data series labels and small label boxes with the color of the series. If you want to format the labels in some way, e.g. make them to links, you can pass in a function for "labelFormatter". Here's an example that makes them clickable:

labelFormatter: function(label, series) {

// series is the series object for the label

return '<a href="#' + label + '">' + label + '</a>';

}

To prevent a series from showing up in the legend, simply have the function return null.

"noColumns" is the number of columns to divide the legend table into. "position" specifies the overall placement of the legend within the plot (top-right, top-left, etc.) and margin the distance to the plot edge (this can be either a number or an array of two numbers like [x, y]). "backgroundColor" and "backgroundOpacity" specifies the background. The default is a partly transparent auto-detected background.

If you want the legend to appear somewhere else in the DOM, you can specify "container" as a jQuery object/expression to put the legend table into. The "position" and "margin" etc. options will then be ignored. Note that Flot will overwrite the contents of the container.

Legend entries appear in the same order as their series by default. If "sorted" is "reverse" then they appear in the opposite order from their series. To sort them alphabetically, you can specify true, "ascending" or "descending", where true and "ascending" are equivalent.

You can also provide your own comparator function that accepts two objects with "label" and "color" properties, and returns zero if they are equal, a positive value if the first is greater than the second, and a negative value if the first is less than the second.

sorted: function(a, b) {

// sort alphabetically in ascending order

return a.label == b.label ? 0 : (

a.label > b.label ? 1 : -1

)

}

## Customizing the axes

xaxis, yaxis: {

show: null or true/false

position: "bottom" or "top" or "left" or "right"

mode: null or "time" ("time" requires jquery.flot.time.js plugin)

timezone: null, "browser" or timezone (only makes sense for mode: "time")

color: null or color spec

tickColor: null or color spec

font: null or font spec object

min: null or number

max: null or number

autoscaleMargin: null or number

transform: null or fn: number -> number

inverseTransform: null or fn: number -> number

ticks: null or number or ticks array or (fn: axis -> ticks array)

tickSize: number or array

minTickSize: number or array

tickFormatter: (fn: number, object -> string) or string

tickDecimals: null or number

labelWidth: null or number

labelHeight: null or number

reserveSpace: null or true

tickLength: null or number

alignTicksWithAxis: null or number

}

All axes have the same kind of options. The following describes how to configure one axis, see below for what to do if you've got more than one x axis or y axis.

If you don't set the "show" option (i.e. it is null), visibility is auto-detected, i.e. the axis will show up if there's data associated with it. You can override this by setting the "show" option to true or false.

The "position" option specifies where the axis is placed, bottom or top for x axes, left or right for y axes. The "mode" option determines how the data is interpreted, the default of null means as decimal numbers. Use "time" for time series data; see the time series data section. The time plugin (jquery.flot.time.js) is required for time series support.

The "color" option determines the color of the line and ticks for the axis, and defaults to the grid color with transparency. For more fine-grained control you can also set the color of the ticks separately with "tickColor".

You can customize the font and color used to draw the axis tick labels with CSS or directly via the "font" option. When "font" is null - the default - each tick label is given the 'flot-tick-label' class. For compatibility with Flot 0.7 and earlier the labels are also given the 'tickLabel' class, but this is deprecated and scheduled to be removed with the release of version 1.0.0.

To enable more granular control over styles, labels are divided between a set of text containers, with each holding the labels for one axis. These containers are given the classes 'flot-[x|y]-axis', and 'flot-[x|y]#-axis', where '#' is the number of the axis when there are multiple axes. For example, the x-axis labels for a simple plot with only a single x-axis might look like this:

<div class='flot-x-axis flot-x1-axis'>

<div class='flot-tick-label'>January 2013</div>

...

</div>

For direct control over label styles you can also provide "font" as an object with this format:

{

size: 11,

lineHeight: 13,

style: "italic",

weight: "bold",

family: "sans-serif",

variant: "small-caps",

color: "#545454"

}

The size and lineHeight must be expressed in pixels; CSS units such as 'em' or 'smaller' are not allowed.

The options "min"/"max" are the precise minimum/maximum value on the scale. If you don't specify either of them, a value will automatically be chosen based on the minimum/maximum data values. Note that Flot always examines all the data values you feed to it, even if a restriction on another axis may make some of them invisible (this makes interactive use more stable).

The "autoscaleMargin" is a bit esoteric: it's the fraction of margin that the scaling algorithm will add to avoid that the outermost points ends up on the grid border. Note that this margin is only applied when a min or max value is not explicitly set. If a margin is specified, the plot will furthermore extend the axis end-point to the nearest whole tick. The default value is "null" for the x axes and 0.02 for y axes which seems appropriate for most cases.

"transform" and "inverseTransform" are callbacks you can put in to change the way the data is drawn. You can design a function to compress or expand certain parts of the axis non-linearly, e.g. suppress weekends or compress far away points with a logarithm or some other means. When Flot draws the plot, each value is first put through the transform function. Here's an example, the x axis can be turned into a natural logarithm axis with the following code:

xaxis: {

transform: function (v) { return Math.log(v); },

inverseTransform: function (v) { return Math.exp(v); }

}

Similarly, for reversing the y axis so the values appear in inverse order:

yaxis: {

transform: function (v) { return -v; },

inverseTransform: function (v) { return -v; }

}

Note that for finding extrema, Flot assumes that the transform function does not reorder values (it should be monotone).

The inverseTransform is simply the inverse of the transform function (so v == inverseTransform(transform(v)) for all relevant v). It is required for converting from canvas coordinates to data coordinates, e.g. for a mouse interaction where a certain pixel is clicked. If you don't use any interactive features of Flot, you may not need it.

The rest of the options deal with the ticks.

If you don't specify any ticks, a tick generator algorithm will make some for you. The algorithm has two passes. It first estimates how many ticks would be reasonable and uses this number to compute a nice round tick interval size. Then it generates the ticks.

You can specify how many ticks the algorithm aims for by setting "ticks" to a number. The algorithm always tries to generate reasonably round tick values so even if you ask for three ticks, you might get five if that fits better with the rounding. If you don't want any ticks at all, set "ticks" to 0 or an empty array.

Another option is to skip the rounding part and directly set the tick interval size with "tickSize". If you set it to 2, you'll get ticks at 2, 4, 6, etc. Alternatively, you can specify that you just don't want ticks at a size less than a specific tick size with "minTickSize". Note that for time series, the format is an array like [2, "month"], see the next section.

If you want to completely override the tick algorithm, you can specify an array for "ticks", either like this:

ticks: [0, 1.2, 2.4]

Or like this where the labels are also customized:

ticks: [[0, "zero"], [1.2, "one mark"], [2.4, "two marks"]]

You can mix the two if you like.

For extra flexibility you can specify a function as the "ticks" parameter. The function will be called with an object with the axis min and max and should return a ticks array. Here's a simplistic tick generator that spits out intervals of pi, suitable for use on the x axis for trigonometric functions:

function piTickGenerator(axis) {

var res = [], i = Math.floor(axis.min / Math.PI);

do {

var v = i \* Math.PI;

res.push([v, i + "\u03c0"]);

++i;

} while (v < axis.max);

return res;

}

You can control how the ticks look like with "tickDecimals", the number of decimals to display (default is auto-detected).

Alternatively, for ultimate control over how ticks are formatted you can provide a function to "tickFormatter". The function is passed two parameters, the tick value and an axis object with information, and should return a string. The default formatter looks like this:

function formatter(val, axis) {

return val.toFixed(axis.tickDecimals);

}

The axis object has "min" and "max" with the range of the axis, "tickDecimals" with the number of decimals to round the value to and "tickSize" with the size of the interval between ticks as calculated by the automatic axis scaling algorithm (or specified by you). Here's an example of a custom formatter:

function suffixFormatter(val, axis) {

if (val > 1000000)

return (val / 1000000).toFixed(axis.tickDecimals) + " MB";

else if (val > 1000)

return (val / 1000).toFixed(axis.tickDecimals) + " kB";

else

return val.toFixed(axis.tickDecimals) + " B";

}

"labelWidth" and "labelHeight" specifies a fixed size of the tick labels in pixels. They're useful in case you need to align several plots. "reserveSpace" means that even if an axis isn't shown, Flot should reserve space for it - it is useful in combination with labelWidth and labelHeight for aligning multi-axis charts.

"tickLength" is the length of the tick lines in pixels. By default, the innermost axes will have ticks that extend all across the plot, while any extra axes use small ticks. A value of null means use the default, while a number means small ticks of that length - set it to 0 to hide the lines completely.

If you set "alignTicksWithAxis" to the number of another axis, e.g. alignTicksWithAxis: 1, Flot will ensure that the autogenerated ticks of this axis are aligned with the ticks of the other axis. This may improve the looks, e.g. if you have one y axis to the left and one to the right, because the grid lines will then match the ticks in both ends. The trade-off is that the forced ticks won't necessarily be at natural places.

## Multiple axes

If you need more than one x axis or y axis, you need to specify for each data series which axis they are to use, as described under the format of the data series, e.g. { data: [...], yaxis: 2 } specifies that a series should be plotted against the second y axis.

To actually configure that axis, you can't use the xaxis/yaxis options directly - instead there are two arrays in the options:

xaxes: []

yaxes: []

Here's an example of configuring a single x axis and two y axes (we can leave options of the first y axis empty as the defaults are fine):

{

xaxes: [ { position: "top" } ],

yaxes: [ { }, { position: "right", min: 20 } ]

}

The arrays get their default values from the xaxis/yaxis settings, so say you want to have all y axes start at zero, you can simply specify yaxis: { min: 0 } instead of adding a min parameter to all the axes.

Generally, the various interfaces in Flot dealing with data points either accept an xaxis/yaxis parameter to specify which axis number to use (starting from 1), or lets you specify the coordinate directly as x2/x3/... or x2axis/x3axis/... instead of "x" or "xaxis".

## Time series data

Please note that it is now required to include the time plugin, jquery.flot.time.js, for time series support.

Time series are a bit more difficult than scalar data because calendars don't follow a simple base 10 system. For many cases, Flot abstracts most of this away, but it can still be a bit difficult to get the data into Flot. So we'll first discuss the data format.

The time series support in Flot is based on Epoch timestamps, i.e., everywhere a time value is expected or handed over, a number is used. This is a number, not a Date object. Flot supports three different time bases in which the timestamps can be given: seconds, milliseconds and microseconds. The timestamp is therefore the number of microseconds, milliseconds or seconds since January 1, 1970 00:00:00 UTC.

The time base in which the timestamps are given to Flot can be selected by setting the "timeBase" option to "microseconds", "milliseconds" or "seconds" in axis options. If not set, it defaults to "seconds".

You can see a native Javascript timestamp (in milliseconds) like this

alert((new Date()).getTime())

There are different schools of thought when it comes to display of timestamps. Many will want the timestamps to be displayed according to a certain time zone, usually the time zone in which the data has been produced. Some want the localized experience, where the timestamps are displayed according to the local time of the visitor. Flot supports both. Optionally you can include a third-party library to get additional timezone support.

Default behavior is that Flot always displays timestamps according to UTC. The reason being that the core Javascript Date object does not support other fixed time zones. Often your data is at another time zone, so it may take a little bit of tweaking to work around this limitation.

The easiest way to think about it is to pretend that the data production time zone is UTC, even if it isn't. So if you have a datapoint at 2002-02-20 08:00, you can generate a timestamp for eight o'clock UTC even if it really happened eight o'clock UTC+0200.

In PHP you can get an appropriate timestamp with:

strtotime("2002-02-20 UTC") \* 1000

In Python you can get it with something like:

calendar.timegm(datetime\_object.timetuple()) \* 1000

In Ruby you can get it using the #to\_i method on the [Time](http://apidock.com/ruby/Time/to_i) object. If you're using the active\_support gem (default for Ruby on Rails applications) #to\_i is also available on the DateTime and ActiveSupport::TimeWithZone objects. You simply need to multiply the result by 1000:

Time.now.to\_i \* 1000 # => 1383582043000

# ActiveSupport examples:

DateTime.now.to\_i \* 1000 # => 1383582043000

ActiveSupport::TimeZone.new('Asia/Shanghai').now.to\_i \* 1000

# => 1383582043000

In .NET you can get it with something like:

public static int GetJavascriptTimestamp(System.DateTime input)

{

System.TimeSpan span = new System.TimeSpan(System.DateTime.Parse("1/1/1970").Ticks);

System.DateTime time = input.Subtract(span);

return (long)(time.Ticks / 10000);

}

Javascript also has some support for parsing date strings, so it is possible to generate the timestamps manually client-side.

If you've already got the real UTC timestamp, it's too late to use the pretend trick described above. But you can fix up the timestamps by adding the time zone offset, e.g. for UTC+0200 you would add 2 hours to the UTC timestamp you got. Then it'll look right on the plot. Most programming environments have some means of getting the timezone offset for a specific date (note that you need to get the offset for each individual timestamp to account for daylight savings).

The alternative with core Javascript is to interpret the timestamps according to the time zone that the visitor is in, which means that the ticks will shift with the time zone and daylight savings of each visitor. This behavior is enabled by setting the axis option "timezone" to the value "browser".

If you need more time zone functionality than this, there is still another option. If you include the "timezone-js" library <https://github.com/mde/timezone-js> in the page and set axis.timezone to a value recognized by said library, Flot will use timezone-js to interpret the timestamps according to that time zone.

Once you've gotten the timestamps into the data and specified "time" as the axis mode, Flot will automatically generate relevant ticks and format them. As always, you can tweak the ticks via the "ticks" option

* just remember that the values should be timestamps (numbers), not Date objects.

Tick generation and formatting can also be controlled separately through the following axis options:

minTickSize: array

timeformat: null or format string

monthNames: null or array of size 12 of strings

dayNames: null or array of size 7 of strings

twelveHourClock: boolean

Here "timeformat" is a format string to use. You might use it like this:

xaxis: {

mode: "time",

timeBase: "milliseconds",

timeformat: "%Y/%m/%d"

}

This will result in tick labels like "2000/12/24". A subset of the standard strftime specifiers are supported (plus the nonstandard %q):

%a: weekday name (customizable)

%b: month name (customizable)

%d: day of month, zero-padded (01-31)

%e: day of month, space-padded ( 1-31)

%H: hours, 24-hour time, zero-padded (00-23)

%I: hours, 12-hour time, zero-padded (01-12)

%m: month, zero-padded (01-12)

%M: minutes, zero-padded (00-59)

%q: quarter (1-4)

%S: seconds, zero-padded (00-59)

%s: sub-seconds, accuracy can be denoted with a number (e.g., %3s)

%y: year (two digits)

%Y: year (four digits)

%p: am/pm

%P: AM/PM (uppercase version of %p)

%w: weekday as number (0-6, 0 being Sunday)

Flot 0.8 switched from %h to the standard %H hours specifier. The %h specifier is still available, for backwards-compatibility, but is deprecated and scheduled to be removed permanently with the release of version 1.0.

You can customize the month names with the "monthNames" option. For instance, for Danish you might specify:

monthNames: ["jan", "feb", "mar", "apr", "maj", "jun", "jul", "aug", "sep", "okt", "nov", "dec"]

Similarly you can customize the weekday names with the "dayNames" option. An example in French:

dayNames: ["dim", "lun", "mar", "mer", "jeu", "ven", "sam"]

If you set "twelveHourClock" to true, the autogenerated timestamps will use 12 hour AM/PM timestamps instead of 24 hour. This only applies if you have not set "timeformat". Use the "%I" and "%p" or "%P" options if you want to build your own format string with 12-hour times.

If you want to have ticks with a fixed sub-second accuracy, you can add a number in the subsecond specifier (e.g., "%3s" for millisecond accuracy) in the "timeformat" string". If the accuracy for the sub-second timestamps is not given, Flot will automatically determine the accuracy depending on the timespan of the axis and the number of ticks, to create.

If the Date object has a strftime property (and it is a function), it will be used instead of the built-in formatter. Thus you can include a strftime library such as <http://hacks.bluesmoon.info/strftime/> for more powerful date/time formatting.

If everything else fails, you can control the formatting by specifying a custom tick formatter function as usual. Here's a simple example which will format December 24 as 24/12:

tickFormatter: function (val, axis) {

var d = new Date(val);

return d.getUTCDate() + "/" + (d.getUTCMonth() + 1);

}

Note that for the time mode "tickSize" and "minTickSize" are a bit special in that they are arrays on the form "[value, unit]" where unit is one of "second", "minute", "hour", "day", "month" and "year". So you can specify

minTickSize: [1, "month"]

to get a tick interval size of at least 1 month and correspondingly, if axis.tickSize is [2, "day"] in the tick formatter, the ticks have been produced with two days in-between.

## Customizing the data series

series: {

lines, points, bars: {

show: boolean

lineWidth: number

fill: boolean or number

fillColor: null or color/gradient

}

lines, bars: {

zero: boolean

}

points: {

radius: number

symbol: "circle" or function

}

bars: {

barWidth: number

align: "left", "right" or "center"

horizontal: boolean

}

lines: {

steps: boolean

}

shadowSize: number

highlightColor: color or number

}

colors: [ color1, color2, ... ]

The options inside "series: {}" are copied to each of the series. So you can specify that all series should have bars by putting it in the global options, or override it for individual series by specifying bars in a particular the series object in the array of data.

The most important options are "lines", "points" and "bars" that specify whether and how lines, points and bars should be shown for each data series. In case you don't specify anything at all, Flot will default to showing lines (you can turn this off with lines: { show: false }). You can specify the various types independently of each other, and Flot will happily draw each of them in turn (this is probably only useful for lines and points), e.g.

var options = {

series: {

lines: { show: true, fill: true, fillColor: "rgba(255, 255, 255, 0.8)" },

points: { show: true, fill: false }

}

};

"lineWidth" is the thickness of the line or outline in pixels. You can set it to 0 to prevent a line or outline from being drawn; this will also hide the shadow.

"fill" is whether the shape should be filled. For lines, this produces area graphs. You can use "fillColor" to specify the color of the fill. If "fillColor" evaluates to false (default for everything except points which are filled with white), the fill color is auto-set to the color of the data series. You can adjust the opacity of the fill by setting fill to a number between 0 (fully transparent) and 1 (fully opaque).

For bars, fillColor can be a gradient, see the gradient documentation below. "barWidth" is the width of the bars in units of the x axis (or the y axis if "horizontal" is true), contrary to most other measures that are specified in pixels. For instance, for time series the unit is milliseconds so 24 \* 60 \* 60 \* 1000 produces bars with the width of a day. "align" specifies whether a bar should be left-aligned (default), right-aligned or centered on top of the value it represents. When "horizontal" is on, the bars are drawn horizontally, i.e. from the y axis instead of the x axis; note that the bar end points are still defined in the same way so you'll probably want to swap the coordinates if you've been plotting vertical bars first.

Area and bar charts normally start from zero, regardless of the data's range. This is because they convey information through size, and starting from a different value would distort their meaning. In cases where the fill is purely for decorative purposes, however, "zero" allows you to override this behavior. It defaults to true for filled lines and bars; setting it to false tells the series to use the same automatic scaling as an un-filled line.

For lines, "steps" specifies whether two adjacent data points are connected with a straight (possibly diagonal) line or with first a horizontal and then a vertical line. Note that this transforms the data by adding extra points.

For points, you can specify the radius and the symbol. The only built-in symbol type is circles, for other types you can use a plugin or define them yourself by specifying a callback:

function cross(ctx, x, y, radius, shadow) {

var size = radius \* Math.sqrt(Math.PI) / 2;

ctx.moveTo(x - size, y - size);

ctx.lineTo(x + size, y + size);

ctx.moveTo(x - size, y + size);

ctx.lineTo(x + size, y - size);

}

The parameters are the drawing context, x and y coordinates of the center of the point, a radius which corresponds to what the circle would have used and whether the call is to draw a shadow (due to limited canvas support, shadows are currently faked through extra draws). It's good practice to ensure that the area covered by the symbol is the same as for the circle with the given radius, this ensures that all symbols have approximately the same visual weight.

"shadowSize" is the default size of shadows in pixels. Set it to 0 to remove shadows.

"highlightColor" is the default color of the translucent overlay used to highlight the series when the mouse hovers over it.

The "colors" array specifies a default color theme to get colors for the data series from. You can specify as many colors as you like, like this:

colors: ["#d18b2c", "#dba255", "#919733"]

If there are more data series than colors, Flot will try to generate extra colors by lightening and darkening colors in the theme.

## Customizing the grid

grid: {

show: boolean

aboveData: boolean

color: color

backgroundColor: color/gradient or null

margin: number or margin object

labelMargin: number

axisMargin: number

markings: array of markings or (fn: axes -> array of markings)

borderWidth: number or object with "top", "right", "bottom" and "left" properties with different widths

borderColor: color or null or object with "top", "right", "bottom" and "left" properties with different colors

minBorderMargin: number or null

clickable: boolean

hoverable: boolean

autoHighlight: boolean

mouseActiveRadius: number

}

interaction: {

redrawOverlayInterval: number or -1

}

The grid is the thing with the axes and a number of ticks. Many of the things in the grid are configured under the individual axes, but not all. "color" is the color of the grid itself whereas "backgroundColor" specifies the background color inside the grid area, here null means that the background is transparent. You can also set a gradient, see the gradient documentation below.

You can turn off the whole grid including tick labels by setting "show" to false. "aboveData" determines whether the grid is drawn above the data or below (below is default).

"margin" is the space in pixels between the canvas edge and the grid, which can be either a number or an object with individual margins for each side, in the form:

margin: {

top: top margin in pixels

left: left margin in pixels

bottom: bottom margin in pixels

right: right margin in pixels

}

"labelMargin" is the space in pixels between tick labels and axis line, and "axisMargin" is the space in pixels between axes when there are two next to each other.

"borderWidth" is the width of the border around the plot. Set it to 0 to disable the border. Set it to an object with "top", "right", "bottom" and "left" properties to use different widths. You can also set "borderColor" if you want the border to have a different color than the grid lines. Set it to an object with "top", "right", "bottom" and "left" properties to use different colors. "minBorderMargin" controls the default minimum margin around the border - it's used to make sure that points aren't accidentally clipped by the canvas edge so by default the value is computed from the point radius.

"markings" is used to draw simple lines and rectangular areas in the background of the plot. You can either specify an array of ranges on the form { xaxis: { from, to }, yaxis: { from, to } } (with multiple axes, you can specify coordinates for other axes instead, e.g. as x2axis/x3axis/...) or with a function that returns such an array given the axes for the plot in an object as the first parameter.

You can set the color of markings by specifying "color" in the ranges object. Here's an example array:

markings: [ { xaxis: { from: 0, to: 2 }, yaxis: { from: 10, to: 10 }, color: "#bb0000" }, ... ]

If you leave out one of the values, that value is assumed to go to the border of the plot. So for example if you only specify { xaxis: { from: 0, to: 2 } } it means an area that extends from the top to the bottom of the plot in the x range 0-2.

A line is drawn if from and to are the same, e.g.

markings: [ { yaxis: { from: 1, to: 1 } }, ... ]

would draw a line parallel to the x axis at y = 1. You can control the line width with "lineWidth" in the range object.

An example function that makes vertical stripes might look like this:

markings: function (axes) {

var markings = [];

for (var x = Math.floor(axes.xaxis.min); x < axes.xaxis.max; x += 2)

markings.push({ xaxis: { from: x, to: x + 1 } });

return markings;

}

If you set "clickable" to true, the plot will listen for click events on the plot area and fire a "plotclick" event on the placeholder with a position and a nearby data item object as parameters. The coordinates are available both in the unit of the axes (not in pixels) and in global screen coordinates.

Likewise, if you set "hoverable" to true, the plot will listen for mouse move events on the plot area and fire a "plothover" event with the same parameters as the "plotclick" event. If "autoHighlight" is true (the default), nearby data items are highlighted automatically. If needed, you can disable highlighting and control it yourself with the highlight/unhighlight plot methods described elsewhere.

You can use "plotclick" and "plothover" events like this:

$.plot($("#placeholder"), [ d ], { grid: { clickable: true } });

$("#placeholder").bind("plotclick", function (event, pos, item) {

alert("You clicked at " + pos.x + ", " + pos.y);

// axis coordinates for other axes, if present, are in pos.x2, pos.x3, ...

// if you need global screen coordinates, they are pos.pageX, pos.pageY

if (item) {

highlight(item.series, item.datapoint);

alert("You clicked a point!");

}

});

The item object in this example is either null or a nearby object on the form:

item: {

datapoint: the point, e.g. [0, 2]

dataIndex: the index of the point in the data array

series: the series object

seriesIndex: the index of the series

pageX, pageY: the global screen coordinates of the point

}

For instance, if you have specified the data like this

$.plot($("#placeholder"), [ { label: "Foo", data: [[0, 10], [7, 3]] } ], ...);

and the mouse is near the point (7, 3), "datapoint" is [7, 3], "dataIndex" will be 1, "series" is a normalized series object with among other things the "Foo" label in series.label and the color in series.color, and "seriesIndex" is 0. Note that plugins and options that transform the data can shift the indexes from what you specified in the original data array.

If you use the above events to update some other information and want to clear out that info in case the mouse goes away, you'll probably also need to listen to "mouseout" events on the placeholder div.

"mouseActiveRadius" specifies how far the mouse can be from an item and still activate it. If there are two or more points within this radius, Flot chooses the closest item. For bars, the top-most bar (from the latest specified data series) is chosen.

If you want to disable interactivity for a specific data series, you can set "hoverable" and "clickable" to false in the options for that series, like this:

{ data: [...], label: "Foo", clickable: false }

"redrawOverlayInterval" specifies the maximum time to delay a redraw of interactive things (this works as a rate limiting device). The default is capped to 60 frames per second. You can set it to -1 to disable the rate limiting.

## Specifying gradients

A gradient is specified like this:

{ colors: [ color1, color2, ... ] }

For instance, you might specify a background on the grid going from black to gray like this:

grid: {

backgroundColor: { colors: ["#000", "#999"] }

}

For the series you can specify the gradient as an object that specifies the scaling of the brightness and the opacity of the series color, e.g.

{ colors: [{ opacity: 0.8 }, { brightness: 0.6, opacity: 0.8 } ] }

where the first color simply has its alpha scaled, whereas the second is also darkened. For instance, for bars the following makes the bars gradually disappear, without outline:

bars: {

show: true,

lineWidth: 0,

fill: true,

fillColor: { colors: [ { opacity: 0.8 }, { opacity: 0.1 } ] }

}

Flot currently only supports vertical gradients drawn from top to bottom because that's what works with IE.

## Plot Methods

The Plot object returned from the plot function has some methods you can call:

* highlight(series, datapoint)

Highlight a specific datapoint in the data series. You can either specify the actual objects, e.g. if you got them from a "plotclick" event, or you can specify the indices, e.g. highlight(1, 3) to highlight the fourth point in the second series (remember, zero-based indexing).

* unhighlight(series, datapoint) or unhighlight()

Remove the highlighting of the point, same parameters as highlight.

If you call unhighlight with no parameters, e.g. as plot.unhighlight(), all current highlights are removed.

* setData(data)

You can use this to reset the data used. Note that axis scaling, ticks, legend etc. will not be recomputed (use setupGrid() to do that). You'll probably want to call draw() afterwards.

You can use this function to speed up redrawing a small plot if you know that the axes won't change. Put in the new data with setData(newdata), call draw(), and you're good to go. Note that for large datasets, almost all the time is consumed in draw() plotting the data so in this case don't bother.

* setupGrid()

Recalculate and set axis scaling, ticks, legend etc.

Note that because of the drawing model of the canvas, this function will immediately redraw (actually reinsert in the DOM) the labels and the legend, but not the actual tick lines because they're drawn on the canvas. You need to call draw() to get the canvas redrawn.

* draw()

Redraws the plot canvas.

* triggerRedrawOverlay()

Schedules an update of an overlay canvas used for drawing interactive things like a selection and point highlights. This is mostly useful for writing plugins. The redraw doesn't happen immediately, instead a timer is set to catch multiple successive redraws (e.g. from a mousemove). You can get to the overlay by setting up a drawOverlay hook.

* width()/height()

Gets the width and height of the plotting area inside the grid. This is smaller than the canvas or placeholder dimensions as some extra space is needed (e.g. for labels).

* offset()

Returns the offset of the plotting area inside the grid relative to the document, useful for instance for calculating mouse positions (event.pageX/Y minus this offset is the pixel position inside the plot).

* pointOffset({ x: xpos, y: ypos })

Returns the calculated offset of the data point at (x, y) in data space within the placeholder div. If you are working with multiple axes, you can specify the x and y axis references, e.g.

o = pointOffset({ x: xpos, y: ypos, xaxis: 2, yaxis: 3 })

// o.left and o.top now contains the offset within the div

* resize()

Tells Flot to resize the drawing canvas to the size of the placeholder. You need to run setupGrid() and draw() afterwards as canvas resizing is a destructive operation. This is used internally by the resize plugin.

* shutdown()

Cleans up any event handlers Flot has currently registered. This is used internally.

There are also some members that let you peek inside the internal workings of Flot which is useful in some cases. Note that if you change something in the objects returned, you're changing the objects used by Flot to keep track of its state, so be careful.

* getData()

Returns an array of the data series currently used in normalized form with missing settings filled in according to the global options. So for instance to find out what color Flot has assigned to the data series, you could do this:

var series = plot.getData();

for (var i = 0; i < series.length; ++i)

alert(series[i].color);

A notable other interesting field besides color is datapoints which has a field "points" with the normalized data points in a flat array (the field "pointsize" is the increment in the flat array to get to the next point so for a dataset consisting only of (x,y) pairs it would be 2).

* getAxes()

Gets an object with the axes. The axes are returned as the attributes of the object, so for instance getAxes().xaxis is the x axis.

Various things are stuffed inside an axis object, e.g. you could use getAxes().xaxis.ticks to find out what the ticks are for the xaxis. Two other useful attributes are p2c and c2p, functions for transforming from data point space to the canvas plot space and back. Both returns values that are offset with the plot offset. Check the Flot source code for the complete set of attributes (or output an axis with console.log() and inspect it).

With multiple axes, the extra axes are returned as x2axis, x3axis, etc., e.g. getAxes().y2axis is the second y axis. You can check y2axis.used to see whether the axis is associated with any data points and y2axis.show to see if it is currently shown.

* getPlaceholder()

Returns placeholder that the plot was put into. This can be useful for plugins for adding DOM elements or firing events.

* getCanvas()

Returns the canvas used for drawing in case you need to hack on it yourself. You'll probably need to get the plot offset too.

* getPlotOffset()

Gets the offset that the grid has within the canvas as an object with distances from the canvas edges as "left", "right", "top", "bottom". I.e., if you draw a circle on the canvas with the center placed at (left, top), its center will be at the top-most, left corner of the grid.

* getOptions()

Gets the options for the plot, normalized, with default values filled in. You get a reference to actual values used by Flot, so if you modify the values in here, Flot will use the new values. If you change something, you probably have to call draw() or setupGrid() or triggerRedrawOverlay() to see the change.

## Hooks

In addition to the public methods, the Plot object also has some hooks that can be used to modify the plotting process. You can install a callback function at various points in the process, the function then gets access to the internal data structures in Flot.

Here's an overview of the phases Flot goes through:

1. Plugin initialization, parsing options
2. Constructing the canvases used for drawing
3. Set data: parsing data specification, calculating colors, copying raw data points into internal format, normalizing them, finding max/min for axis auto-scaling
4. Grid setup: calculating axis spacing, ticks, inserting tick labels, the legend
5. Draw: drawing the grid, drawing each of the series in turn
6. Setting up event handling for interactive features
7. Responding to events, if any
8. Shutdown: this mostly happens in case a plot is overwritten

Each hook is simply a function which is put in the appropriate array. You can add them through the "hooks" option, and they are also available after the plot is constructed as the "hooks" attribute on the returned plot object, e.g.

// define a simple draw hook

function hellohook(plot, canvascontext) { alert("hello!"); };

// pass it in, in an array since we might want to specify several

var plot = $.plot(placeholder, data, { hooks: { draw: [hellohook] } });

// we can now find it again in plot.hooks.draw[0] unless a plugin

// has added other hooks

The available hooks are described below. All hook callbacks get the plot object as first parameter. You can find some examples of defined hooks in the plugins bundled with Flot.

* processOptions [phase 1]

function(plot, options)

Called after Flot has parsed and merged options. Useful in the instance where customizations beyond simple merging of default values is needed. A plugin might use it to detect that it has been enabled and then turn on or off other options.

* processRawData [phase 3]

function(plot, series, data, datapoints)

Called before Flot copies and normalizes the raw data for the given series. If the function fills in datapoints.points with normalized points and sets datapoints.pointsize to the size of the points, Flot will skip the copying/normalization step for this series.

In any case, you might be interested in setting datapoints.format, an array of objects for specifying how a point is normalized and how it interferes with axis scaling. It accepts the following options:

{

x, y: boolean,

number: boolean,

required: boolean,

defaultValue: value,

autoscale: boolean

}

"x" and "y" specify whether the value is plotted against the x or y axis, and is currently used only to calculate axis min-max ranges. The default format array, for example, looks like this:

[

{ x: true, number: true, required: true },

{ y: true, number: true, required: true }

]

This indicates that a point, i.e. [0, 25], consists of two values, with the first being plotted on the x axis and the second on the y axis.

If "number" is true, then the value must be numeric, and is set to null if it cannot be converted to a number.

"defaultValue" provides a fallback in case the original value is null. This is for instance handy for bars, where one can omit the third coordinate (the bottom of the bar), which then defaults to zero.

If "required" is true, then the value must exist (be non-null) for the point as a whole to be valid. If no value is provided, then the entire point is cleared out with nulls, turning it into a gap in the series.

"autoscale" determines whether the value is considered when calculating an automatic min-max range for the axes that the value is plotted against.

* processDatapoints [phase 3]

function(plot, series, datapoints)

Called after normalization of the given series but before finding min/max of the data points. This hook is useful for implementing data transformations. "datapoints" contains the normalized data points in a flat array as datapoints.points with the size of a single point given in datapoints.pointsize. Here's a simple transform that multiplies all y coordinates by 2:

function multiply(plot, series, datapoints) {

var points = datapoints.points, ps = datapoints.pointsize;

for (var i = 0; i < points.length; i += ps)

points[i + 1] \*= 2;

}

Note that you must leave datapoints in a good condition as Flot doesn't check it or do any normalization on it afterwards.

* processOffset [phase 4]

function(plot, offset)

Called after Flot has initialized the plot's offset, but before it draws any axes or plot elements. This hook is useful for customizing the margins between the grid and the edge of the canvas. "offset" is an object with attributes "top", "bottom", "left" and "right", corresponding to the margins on the four sides of the plot.

* drawBackground [phase 5]

function(plot, canvascontext)

Called before all other drawing operations. Used to draw backgrounds or other custom elements before the plot or axes have been drawn.

* drawSeries [phase 5]

function(plot, canvascontext, series)

Hook for custom drawing of a single series. Called just before the standard drawing routine has been called in the loop that draws each series.

* draw [phase 5]

function(plot, canvascontext)

Hook for drawing on the canvas. Called after the grid is drawn (unless it's disabled or grid.aboveData is set) and the series have been plotted (in case any points, lines or bars have been turned on). For examples of how to draw things, look at the source code.

* bindEvents [phase 6]

function(plot, eventHolder)

Called after Flot has setup its event handlers. Should set any necessary event handlers on eventHolder, a jQuery object with the canvas, e.g.

function (plot, eventHolder) {

eventHolder.mousedown(function (e) {

alert("You pressed the mouse at " + e.pageX + " " + e.pageY);

});

}

Interesting events include click, mousemove, mouseup/down. You can use all jQuery events. Usually, the event handlers will update the state by drawing something (add a drawOverlay hook and call triggerRedrawOverlay) or firing an externally visible event for user code. See the crosshair plugin for an example.

Currently, eventHolder actually contains both the static canvas used for the plot itself and the overlay canvas used for interactive features because some versions of IE get the stacking order wrong. The hook only gets one event, though (either for the overlay or for the static canvas).

Note that custom plot events generated by Flot are not generated on eventHolder, but on the div placeholder supplied as the first argument to the plot call. You can get that with plot.getPlaceholder() - that's probably also the one you should use if you need to fire a custom event.

* drawOverlay [phase 7]

function (plot, canvascontext)

The drawOverlay hook is used for interactive things that need a canvas to draw on. The model currently used by Flot works the way that an extra overlay canvas is positioned on top of the static canvas. This overlay is cleared and then completely redrawn whenever something interesting happens. This hook is called when the overlay canvas is to be redrawn.

"canvascontext" is the 2D context of the overlay canvas. You can use this to draw things. You'll most likely need some of the metrics computed by Flot, e.g. plot.width()/plot.height(). See the crosshair plugin for an example.

* shutdown [phase 8]

function (plot, eventHolder)

Run when plot.shutdown() is called, which usually only happens in case a plot is overwritten by a new plot. If you're writing a plugin that adds extra DOM elements or event handlers, you should add a callback to clean up after you. Take a look at the section in the [PLUGINS](https://github.com/flot/flot/blob/master/PLUGINS.md) document for more info.

## Plugins

Plugins extend the functionality of Flot. To use a plugin, simply include its Javascript file after Flot in the HTML page.

If you're worried about download size/latency, you can concatenate all the plugins you use, and Flot itself for that matter, into one big file (make sure you get the order right), then optionally run it through a Javascript minifier such as YUI Compressor.

Here's a brief explanation of how the plugin plumbings work:

Each plugin registers itself in the global array $.plot.plugins. When you make a new plot object with $.plot, Flot goes through this array calling the "init" function of each plugin and merging default options from the "option" attribute of the plugin. The init function gets a reference to the plot object created and uses this to register hooks and add new public methods if needed.

See the [PLUGINS](https://github.com/flot/flot/blob/master/PLUGINS.md) document for details on how to write a plugin. As the above description hints, it's actually pretty easy.

## Version number

The version number of Flot is available in $.plot.version.

**Pie Options**

* **option:** *default value* - Description of option
* **show:** *false* - Enable the plugin and draw as a pie.
* **radius:** *'auto'* - Sets the radius of the pie. If value is between 0 and 1 (inclusive) then it will use that as a percentage of the available space (size of the container), otherwise it will use the value as a direct pixel length. If set to 'auto', it will be set to 1 if the legend is enabled and 3/4 if not.
* **innerRadius:** *0* - Sets the radius of the donut hole. If value is between 0 and 1 (inclusive) then it will use that as a percentage of the radius, otherwise it will use the value as a direct pixel length.
* **startAngle:** *3/2* - Factor of PI used for the starting angle (in radians) It can range between 0 and 2 (where 0 and 2 have the same result).
* **tilt:** *1* - Percentage of tilt ranging from 0 and 1, where 1 has no change (fully vertical) and 0 is completely flat (fully horizontal -- in which case nothing actually gets drawn).
* **shadow:**
  + **top:** *5* - Vertical distance in pixel of the tilted pie shadow.
  + **left:** *15* - Horizontal distance in pixel of the tilted pie shadow.
  + **alpha:** *0.02* - Alpha value of the tilted pie shadow.
* **offset:**
  + **top:** *0* - Pixel distance to move the pie up and down (relative to the center).
  + **left:** *'auto'* - Pixel distance to move the pie left and right (relative to the center).
* **stroke:**
  + **color:** *'#FFF'* - Color of the border of each slice. Hexadecimal color definitions are prefered (other formats may or may not work).
  + **width:** *1* - Pixel width of the border of each slice.
* **label:**
  + **show:** *'auto'* - Enable/Disable the labels. This can be set to true, false, or 'auto'. When set to 'auto', it will be set to false if the legend is enabled and true if not.
  + **radius:** *1* - Sets the radius at which to place the labels. If value is between 0 and 1 (inclusive) then it will use that as a percentage of the available space (size of the container), otherwise it will use the value as a direct pixel length.
  + **threshold:** *0* - Hides the labels of any pie slice that is smaller than the specified percentage (ranging from 0 to 1) i.e. a value of '0.03' will hide all slices 3% or less of the total.
  + **formatter:** *[function]* - This function specifies how the positioned labels should be formatted, and is applied after the legend's labelFormatter function. The labels can also still be styled using the class "pieLabel" (i.e. ".pieLabel" or "#graph1 .pieLabel").
  + **radius:** *1* - Sets the radius at which to place the labels. If value is between 0 and 1 (inclusive) then it will use that as a percentage of the available space (size of the container), otherwise it will use the value as a direct pixel length.
  + **background:**
    - **color:** *null* - Backgound color of the positioned labels. If null, the plugin will automatically use the color of the slice.
    - **opacity:** *0* - Opacity of the background for the positioned labels. Acceptable values range from 0 to 1, where 0 is completely transparent and 1 is completely opaque.
* **combine:**
  + **threshold:** *0* - Combines all slices that are smaller than the specified percentage (ranging from 0 to 1) i.e. a value of '0.03' will combine all slices 3% or less into one slice).
  + **color:** *null* - Backgound color of the positioned labels. If null, the plugin will automatically use the color of the first slice to be combined.
  + **label:** *'Other'* - Label text for the combined slice.
* **highlight:**
  + **opacity:** *0.5* - Opacity of the highlight overlay on top of the current pie slice. Currently this just uses a white overlay, but support for changing the color of the overlay will also be added at a later date.