

Modul:Graph

- **basemap:** sets the base map. The map definitions must follow the [TopoJSON](#) format and if saved in Wikipedia are available for this module. Maps in the default directory [Special:Prefixindex/Template:Graph:Map/Inner/](#) like [Worldmap2c-json](#) should only be referenced by their name while omitting the Module:Graph/ prefix to allow better portability. The parameter also accepts URLs, e.g. maps from other Wikipedia versions (the link should follow the scheme of [//en.wikipedia.org/w/index.php?title=mapname&action=raw](https://en.wikipedia.org/w/index.php?title=mapname&action=raw), i.e. protocol-relative without leading http/s and a trailing action=raw to fetch the raw content only). URLs to maps on external sites should be avoided for the sake of link stability, performance, security, and should be assumed to be blocked by the software or browser anyway.
- **scale:** the scaling factor of the map (default: 100)
- **projection:** the [map projection](#) to use. Supported values are listed at <https://github.com/d3/d3-geo-projection>. The default value is [equiangular projection](#).
- **center:** map center (corresponds in the map data to both comma-separated values of the scale field)
- **feature:** which geographic objects should be displayed (corresponds in the map data to the name of the field under the objects field). The default is value countries.
- **ids of geographic entities:** The actual parameter names depend on the base map and the selected feature. For example, for the above mentioned world map the ids are [ISO country codes](#). The values can be either colors or numbers in case the geographic entities should be associated with numeric data: DE=lightblue marks Germany in light blue color, and DE=80.6 assigns Germany the value 80.6 (population in millions). In the latter case, the actual color depends on the following parameters.
- **colorScale:** the color palette to use for the color scale. The palette must be provided as a comma-separated list of color values. The color values must be given either as #rgb/#rrggbb or by a [CSS color name](#). Instead of a list, the built-in color palettes [category10](#) and [category20](#) can also be used.
- **scaleType:** supported values are linear for a linear mapping between the data values and the color scale, log for a log mapping, pow for a power mapping (the exponent can be provided as pow 0.5), sqrt for a square-root mapping, and quantize for a quantized scale, i.e. the data is grouped in as many classes as the color palette has colors.
- **domainMin:** lower boundary of the data values, i.e. smaller data values are mapped to the lower boundary
- **domainMax:** upper boundary of the data values, i.e. larger data values are mapped to the upper boundary
- **legend:** show color legend (does not work with quantize)
- **defaultValue:** default value for unused geographic entities. In case the id values are colors the default value is silver, in case of numbers it is 0.
- **formatjson:** format JSON object for better legibility

chart

Creates a JSON object for <graph> to display charts. In the article namespace the template [Template:Graph:Chart](#) should be used instead. See its page for use cases.

Parameters:

- **width**: width of the chart
- **height**: height of the chart
- **type**: type of the chart: line for **line charts**, area for **area charts**, and rect for (column) **bar charts**, and pie for **pie charts**. Multiple series can stacked using the stacked prefix, e.g. stackedarea.
- **interpolate**: **interpolation** method for line and area charts. It is recommended to use monotone for a **monotone cubic interpolation** – further supported values are listed at <https://github.com/nyurik/vega/wiki/Marks#line>.
- **colors**: color palette of the chart as a comma-separated list of colors. The color values must be given either as #rgb/#rrggbb/#aarrggbb or by a **CSS color name**. For #aarrggbb the aa component denotes the **alpha channel**, i.e. FF=100% opacity, 80=50% opacity/transparency, etc. (The default color palette if n <= 10 is Category10: **Lua-Fehler: bad argument #1 to "get" (not a valid title)** else is Category20: **Lua-Fehler: bad argument #1 to "get" (not a valid title)**). See [Template:ChartColors](#) for details.
- **xAxisTitle** and **yAxisTitle**: captions of the x and y axes
- **xAxisMin**, **xAxisMax**, **yAxisMin**, and **yAxisMax**: minimum and maximum values of the x and y axes (not yet supported for bar charts). These parameters can be used to invert the scale of a numeric axis by setting the lowest value to the Max and highest value to the Min.
- **xAxisFormat** and **yAxisFormat**: changes the formatting of the axis labels. Supported values are listed at <https://github.com/d3/d3-3.x-api-reference/blob/master/Formatting.md#numbers> for numbers. For example, the format % can be used to output percentages. For date/time specification of supported values is <https://github.com/d3/d3-3.x-api-reference/blob/master/Time-Formatting.md> , e.g. xAxisFormat=%d-%m-%Y for result 13-01-1977.
- **xAxisAngle**: rotates the x axis labels by the specified angle. Recommended values are: -45, +45, -90, +90
- **xType** and **yType**: data types of the values, e.g. integer for integers, number for real numbers, date for dates (e.g. YYYY-MM-DD), and string for ordinal values (use string to prevent axis values from being repeated when there are only a few values). Remarks: Date type doesn't work for bar graphs. For date data input please use ISO date format (e.g. YYYY-MM-DD) acc. to [date and time formats used in HTML](#). Other date formats may work but not in all browsers. Date is unfortunately displayed only in en-US format for all Wikipedia languages. Workaround is to use **xAxisFormat** and **yAxisFormat** with numerical dates format.
- **xScaleType** and **yScaleType**: scale types of the x and y axes, e.g. linear for linear scale (default), log for logarithmic scale and sqrt for square root scale.
 - A logarithmic chart allows only positive values to be plotted. A square root scale chart cannot show negative values.
- **x**: the x-values as a comma-separated list, for dates and time see remark in **xType** and **yType**

- **y** or **y1, y2, ...**: the y-values for one or several data series, respectively. For pie charts y2 denotes the radius of the corresponding sectors. For dates and time see remark in **xType** and **yType**
- **legend**: show legend (only works in case of multiple data series)
- **y1Title, y2Title, ...**: defines the label of the respective data series in the legend
- **linewidth**: line width for line charts or distance between the pie segments for pie charts. Setting to 0 with type=line creates a scatter plot.
- **linewidths**: different line widths may be defined for each series of data with csv, if set to 0 with "showSymbols" results with points graph, eg.: linewidths=1, 0, 5, 0.2
- **showSymbols**: show symbol on data point for line graphs, if number is provided it's size of symbol, default 2.5. may be defined for each series of data with csv, eg.: showSymbols=1, 2, 3, 4
- **symbolsShape**: custom shape for symbol: circle, x, square, cross, diamond, triangle_up, triangle_down, triangle_right, triangle_left. May be defined for each series of data with csv, eg.: symbolsShape= circle, cross, square
- **symbolsNoFill**: if true symbol will be without fill (only stroke),
- **symbolsStroke**: if "x" symbol is used or option "symbolsNoFill" symbol stroke width, default 2.5
- **showValues**: Additionally, output the y values as text. (Currently, only (non-stacked) bar and pie charts are supported.) The output can be configured used the following parameters provided as name1:value1, name2:value2 (e.g. **Vorlage:Para**).
 - **format**: Format the output according to <https://github.com/d3/d3-3.x-api-reference/blob/master/Formatting.md#numbers> for numbers and <https://github.com/d3/d3-3.x-api-reference/blob/master/Time-Formatting.md> for date/time.
 - **fontcolor**: text color
 - **fontsize**: text size
 - **offset**: move text by the given offset. For bar charts and pie charts with midangle this also defines if the text is inside or outside the chart.
 - **angle** (pie charts only): text angle in degrees or midangle (default) for dynamic angles based on the mid-angle of the pie sector.
- **innerRadius**: For pie charts: defines the inner radius to create a *doughnut chart*.
- **xGrid** and **yGrid**: display grid lines on the x and y axes.
- **Annotations**
 - **vAnnotationsLine** and **hAnnotationsLine**: display vertical or horizontal annotation lines on specific values e.g. hAnnotationsLine=4, 5, 6
 - **vAnnotationsLabel** and **hAnnotationsLabel**: display vertical or horizontal annotation labels for lines e.g. hAnnotationLabel = label1, label2, label3
- **formatjson**: format JSON object for better legibility

Template wrappers

The functions mapWrapper and chartWrapper are wrappers to pass all parameters of the calling template to the respective map and chart functions.

Note: In the editor preview the graph extension creates a **canvas element** with vector graphics. However, when saving the page a **PNG** raster graphics is generated instead. `{{#invoke:Graph|function_wrapper_name}}`

```
-- ATTENTION:      Please edit this code at https://de.wikipedia.org/wiki/Modul
--                  This way all wiki languages can stay in sync. Thank you!
--
--      BUGS:      X-Axis label format bug? (xAxisFormat =) https://en.wikiped
--                  linewidths - doesnt work for two values (eg 0, 1) but v
--                  clamp - "clamp" used to avoid marks outside marks area
--      TODO:
--                  marks:
--                      - line strokeDash + serialization,
--                      - symStroke serialization
--                      - symbolsNoFill serialization
--                      - arbitrary SVG path symbol shape as symbolsSha
--                      - annotations
--                      - vertical / horizontal line at specifi
--                      - rectangle shape for x,y data range
--                      - graph type serialization (deep rebuild require
--      - second axis (deep rebuild required - assignment of series to one

-- Version History ( PLEASE UPDATE when modifying anything_):
-- 2020-09-01 Vertical and horizontal line annotations
-- 2020-08-08 New logic for "nice" for x axis (problem with scale when xType =
-- 2020-06-21 Serializes symbol size
--             transparent symbols (from line colour) - buggy (incorrect opacit
--             Linewidth serialized with "linewidths"
--             Variable symbol size and shape of symbols on line charts, default
--             p.chartDebugger(frame) for easy debug and JSON output
-- 2020-06-07 Allow lowercase variables for use with [[Template:Wikidata list]]
-- 2020-05-27 Map: allow specification which feature to display and changing th
-- 2020-04-08 Change default showValues.fontcolor from black to persistentGrey
-- 2020-04-06 Logarithmic scale outputs wrong axis labels when "nice"=true
-- 2020-03-11 Allow user-defined scale types, e.g. logarithmic scale
-- 2019-11-08 Apply color-inversion-friendliness to legend title, labels, and x
-- 2019-01-24 Allow comma-separated lists to contain values with commas
-- 2018-10-13 Fix browser color-inversion issues via #54595d per [[mw:Template:
-- 2018-09-16 Allow disabling the legend for templates
-- 2018-09-10 Allow grid lines
-- 2018-08-26 Use user-defined order for stacked charts
-- 2018-02-11 Force usage of explicitly provided x minimum and/or maximum valu
-- 2017-08-08 Added showSymbols param to show symbols on line charts
-- 2016-05-16 Added encodeTitleForPath() to help all path-based APIs graphs lik
-- 2016-03-20 Allow omitted data for charts, labels for line charts with string
-- 2016-01-28 For maps, always use wikiraw:// protocol. https:// will be disabl

local p = {}

--add debug text to this string with eg.          debuglog = debuglog .. "" .. "\n
--invoke chartDebugger() to get graph JSON and this string
debuglog = "Debug " .. "\n\n"

local baseMapDirectory = "Module:Graph/"
local persistentGrey = "#54595d"

local shapes = {}
shapes = {
    circle = "circle", x= "M-.5, -.5L.5, .5M.5, -.5L-.5, .5" , square = "square",
    cross = "cross", diamond = "diamond", triangle_up = "triangle-up",
```

```

        triangle_down = "triangle-down", triangle_right = "triangle-right",
        triangle_left = "triangle-left",
        banana = "m -0.5281,0.2880 0.0020,0.0192 m 0,0 c 0.1253,0.0543 0.2118,0.0
    }

local function numericArray(csv)
    if not csv then return end

    local list = mw.text.split(csv, "%s*,%s*")
    local result = {}
    local isInteger = true
    for i = 1, #list do
        if list[i] == "" then
            result[i] = nil
        else
            result[i] = tonumber(list[i])
            if not result[i] then return end
            if isInteger then
                local int, frac = math.modf(result[i])
                isInteger = frac == 0.0
            end
        end
    end

    return result, isInteger
end

local function stringArray(text)
    if not text then return end

    local list = mw.text.split(mw.usttring.gsub(tostring(text), "\\,", "<COMMA")
    for i = 1, #list do
        list[i] = mw.usttring.gsub(mw.text.trim(list[i]), "<COMMA>", ",")
    end
    return list
end

local function isTable(t) return type(t) == "table" end

local function copy(x)
    if type(x) == "table" then
        local result = {}
        for key, value in pairs(x) do result[key] = copy(value) end
        return result
    else
        return x
    end
end

function p.map(frame)
    -- map path data for geographic objects
    local basemap = frame.args.basemap or "Template:Graph:Map/Inner/Worldmap2
    -- scaling factor
    local scale = tonumber(frame.args.scale) or 100
    -- map projection, see https://github.com/mbostock/d3/wiki/Geo-Projection
    local projection = frame.args.projection or "equiarectangular"
    -- default value for geographic objects without data
    local defaultValue = frame.args.defaultValue or frame.args.defaultvalue
    local scaleType = frame.args.scaleType or frame.args.scaletype or "linear
    -- minimaler Wertebereich (nur für numerische Daten)
    local domainMin = tonumber(frame.args.domainMin or frame.args.domainmin)
    -- maximaler Wertebereich (nur für numerische Daten)
    local domainMax = tonumber(frame.args.domainMax or frame.args.domainmax)

```

```
-- Farbwerte der Farbskala (nur für numerische Daten)
local colorScale = frame.args.colorScale or frame.args.colorscales or "cat"
-- show legend
local legend = frame.args.legend
-- the map feature to display
local feature = frame.args.feature or "countries"
-- map center
local center = numericArray(frame.args.center)
-- format JSON output
local formatJson = frame.args.formatjson

-- map data are key-value pairs: keys are non-lowercase strings (ideally)
local values = {}
local isNumbers = nil
for name, value in pairs(frame.args) do
    if mw.ustr.find(name, "[^%l]+$") and value and value ~= "" then
        if isNumbers == nil then isNumbers = tonumber(value) end
        local data = { id = name, v = value }
        if isNumbers then data.v = tonumber(data.v) end
        table.insert(values, data)
    end
end

if not defaultValue then
    if isNumbers then defaultValue = 0 else defaultValue = "silver" end
end

-- create highlight scale
local scales
if isNumbers then
    if colorScale then colorScale = string.lower(colorScale) end
    if colorScale == "category10" or colorScale == "category20" then
        scales = {
            {
                name = "color",
                type = scaleType,
                domain = { data = "highlights", field = "v" },
                range = colorScale,
                nice = true,
                zero = false
            }
        }
        if domainMin then scales[1].domainMin = domainMin end
        if domainMax then scales[1].domainMax = domainMax end

        local exponent = string.match(scaleType, "pow%s+(%d+%.?%d+)") --
        if exponent then
            scales[1].type = "pow"
            scales[1].exponent = exponent
        end
    end
end

-- create legend
if legend then
    legend = {
        {
            fill = "color",
            offset = 120,
            properties = {
                title = { fontSize = { value = 14 } },
                labels = { fontSize = { value = 12 } },
                legend =

```

```

        {
            stroke = { value = "silver" },
            strokeWidth = { value = 1.5 }
        }
    }
}

end

-- get map url
local basemapUrl
if (string.sub(basemap, 1, 10) == "wikiraw://") then
    basemapUrl = basemap
else
    -- if not a (supported) url look for a colon as namespace separator
    if not string.find(basemap, ":") then basemap = baseMapDirectory
    basemapUrl = "wikiraw://" .. mw.uri.encode(mw.title.new(basemap))
end

local output =
{
    version = 2,
    width = 1, -- generic value as output size depends solely on map
    height = 1, -- ditto
    data =
    {
        {
            -- data source for the highlights
            name = "highlights",
            values = values
        },
        {
            -- data source for map paths data
            name = feature,
            url = basemapUrl,
            format = { type = "topojson", feature = feature }
            transform =
            {
                {
                    -- geographic transformation ("geographic")
                    type = "geopath",
                    value = "data",
                    scale = scale,
                },
                {
                    -- join ("zip") of multiple data sources
                    type = "lookup",
                    keys = { "id" }, -- key for lookup
                    on = "highlights", -- name of source
                    onKey = "id", -- key for source
                    as = { "zipped" }, -- name of result
                    default = { v = defaultValue } -- default value
                }
            }
        }
    },
    marks =
    {
        -- output markings (map paths and highlights)
        {
            type = "path",

```



```

        from = { data = feature },
        properties =
        {
            enter = { path = { field = "layout_path"
            update = { fill = { field = "zipped.v" }
            hover = { fill = { value = "darkgrey" } }
        }
    },
    legends = legend
}
if (scales) then
    output.scales = scales
    output.marks[1].properties.update.fill.scale = "color"
end

local flags
if formatJson then flags = mw.text.JSON_PRETTY end
return mw.text.jsonEncode(output, flags)
end

local function deserializeXData(serializedX, xType, xMin, xMax)
    local x

    if not xType or xType == "integer" or xType == "number" then
        local isInteger
        x, isInteger = numericArray(serializedX)
        if x then
            xMin = tonumber(xMin)
            xMax = tonumber(xMax)
            if not xType then
                if isInteger then xType = "integer" else xType = "number"
            end
        end
        if xType then error("Numbers expected for parameter 'x'")
        end
    end
    if not x then
        x = stringArray(serializedX)
        if not xType then xType = "string" end
    end
    return x, xType, xMin, xMax
end

local function deserializeYData(serializedYs, yType, yMin, yMax)
    local y = {}
    local areAllInteger = true

    for yNum, value in pairs(serializedYs) do
        local yValues
        if not yType or yType == "integer" or yType == "number" then
            local isInteger
            yValues, isInteger = numericArray(value)
            if yValues then
                areAllInteger = areAllInteger and isInteger
            end
            if yType then
                error("Numbers expected for parameter 'y'")
            end
            return deserializeYData(serializedYs, "string")
        end
        if not yValues then yValues = stringArray(value) end
    end
end

```



```

        y[yNum] = yValues
    end
    if not yType then
        if areAllInteger then yType = "integer" else yType = "number" end
    end
    if yType == "integer" or yType == "number" then
        yMin = tonumber(yMin)
        yMax = tonumber(yMax)
    end

    return y, yType, yMin, yMax
end

local function convertXYToManySeries(x, y, xType, yType, seriesTitles)
    local data =
    {
        name = "chart",
        format =
        {
            type = "json",
            parse = { x = xType, y = yType }
        },
        values = {}
    }
    for i = 1, #y do
        local yLen = table.maxn(y[i])
        for j = 1, #x do
            if j <= yLen and y[i][j] then table.insert(data.values, {
            end
        end
        return data
    end

local function convertXYToSingleSeries(x, y, xType, yType, yNames)
    local data = { name = "chart", format = { type = "json", parse = { x = xType, y = yType } } }
    for j = 1, #y do data.format.parse[yNames[j]] = yType end

    for i = 1, #x do
        local item = { x = x[i] }
        for j = 1, #y do item[yNames[j]] = y[j][i] end

        table.insert(data.values, item)
    end
    return data
end

local function getXScale(chartType, stacked, xMin, xMax, xType, xScaleType)
    if chartType == "pie" then return end

    local xscale =
    {
        name = "x",
        range = "width",
        zero = false, -- do not include zero value
        domain = { data = "chart", field = "x" }
    }
    if xScaleType then xscale.type = xScaleType else xscale.type = "linear" end
    if xMin then xscale.domainMin = xMin end
    if xMax then xscale.domainMax = xMax end
    if xMin or xMax then
        xscale.clamp = true
        xscale.nice = false
    end
end

```

```

end
if chartType == "rect" then
    xscale.type = "ordinal"
    if not stacked then xscale.padding = 0.2 end -- pad each bar group
else
    if xType == "date" then
        xscale.type = "time"
    elseif xType == "string" then
        xscale.type = "ordinal"
        xscale.points = true
    end
end
if xType and xType ~= "date" and xScaleType ~= "log" then xscale.nice = true end
return xscale
end

local function getYScale(chartType, stacked, yMin, yMax, yType, yScaleType)
    if chartType == "pie" then return end

    local yscale = {
        name = "y",
        --type = yScaleType or "linear",
        range = "height",
        -- area charts have the lower boundary of their filling at y=0 (so zero = chartType ~= "line",
        nice = yScaleType ~= "log" -- force round numbers for y scale, but
    }
    if yScaleType then yscale.type = yScaleType else yscale.type = "linear" end
    if yMin then yscale.domainMin = yMin end
    if yMax then yscale.domainMax = yMax end
    if yMin or yMax then yscale.clamp = true end
    if yType == "date" then yscale.type = "time"
    elseif yType == "string" then yscale.type = "ordinal" end
    if stacked then
        yscale.domain = { data = "stats", field = "sum_y" }
    else
        yscale.domain = { data = "chart", field = "y" }
    end
end
return yscale
end

local function getColorScale(colors, chartType, xCount, yCount)
    if not colors then
        if (chartType == "pie" and xCount > 10) or yCount > 10 then colors = {} end
    end

    local colorScale = {
        {
            name = "color",
            type = "ordinal",
            range = colors,
            domain = { data = "chart", field = "series" }
        }
    }
    if chartType == "pie" then colorScale.domain.field = "x" end
    return colorScale
end

local function getAlphaColorScale(colors, y)
    local alphaScale
    -- if there is at least one color in the format "#aarrggbb", create a table
    if isTable(colors) then
        local alphas = {}
    end
end

```

```

        local hasAlpha = false
        for i = 1, #colors do
            local a, rgb = string.match(colors[i], "#(%X%X)(%X%X%X%X%X%X%X%X)"
            if a then
                hasAlpha = true
                alphas[i] = tostring(tonumber(a, 16) / 255.0)
                colors[i] = "#" .. rgb
            else
                alphas[i] = "1"
            end
        end
        for i = #colors + 1, #y do alphas[i] = "1" end
        if hasAlpha then alphaScale = { name = "transparency", type = "ordinal"
    end
    return alphaScale
end

local function getLineScale(linewidths, chartType)
    local lineScale = {}

    lineScale =
    {
        name = "line",
        type = "ordinal",
        range = linewidths,
        domain = { data = "chart", field = "series" }
    }

    return lineScale
end

local function getSymSizeScale(symSize)
    local SymSizeScale = {}
    SymSizeScale =
    {
        name = "symSize",
        type = "ordinal",
        range = symSize,
        domain = { data = "chart", field = "series" }
    }

    return SymSizeScale
end

local function getSymShapeScale(symShape)
    local SymShapeScale = {}
    SymShapeScale =
    {
        name = "symShape",
        type = "ordinal",
        range = symShape,
        domain = { data = "chart", field = "series" }
    }

    return SymShapeScale
end

local function getValueScale(fieldName, min, max, type)
    local valueScale =
    {
        name = fieldName,
        type = type or "linear",
        domain = { data = "chart", field = fieldName },
        range = { min, max }
    }

```

```

    }
    return valueScale
end

local function addInteractionToChartVisualisation(plotMarks, colorField, dataField)
    -- initial setup
    if not plotMarks.properties.enter then plotMarks.properties.enter = {} end
    plotMarks.properties.enter[colorField] = { scale = "color", field = dataField }

    -- action when cursor is over plot mark: highlight
    if not plotMarks.properties.hover then plotMarks.properties.hover = {} end
    plotMarks.properties.hover[colorField] = { value = "red" }

    -- action when cursor leaves plot mark: reset to initial setup
    if not plotMarks.properties.update then plotMarks.properties.update = {} end
    plotMarks.properties.update[colorField] = { scale = "color", field = dataField }
end

local function getPieChartVisualisation(yCount, innerRadius, outerRadius, linewidth)
    local chartvis =
    {
        type = "arc",
        from = { data = "chart", transform = { { field = "y", type = "pie" } } },
        properties =
        {
            enter = {
                innerRadius = { value = innerRadius },
                outerRadius = { },
                startAngle = { field = "layout_start" },
                endAngle = { field = "layout_end" },
                stroke = { value = "white" },
                strokeWidth = { value = linewidth or 1 }
            }
        }
    }

    if radiusScale then
        chartvis.properties.enter.outerRadius.scale = radiusScale.name
        chartvis.properties.enter.outerRadius.field = radiusScale.domain
    else
        chartvis.properties.enter.outerRadius.value = outerRadius
    end

    addInteractionToChartVisualisation(chartvis, "fill", "x")

    return chartvis
end

local function getChartVisualisation(chartType, stacked, colorField, yCount, innerRadius)
    if chartType == "pie" then return getPieChartVisualisation(yCount, innerRadius) end

    local chartvis =
    {
        type = chartType,
        properties =
        {
            -- chart creation event handler
            enter =
            {
                x = { scale = "x", field = "x" },
                y = { scale = "y", field = "y" }
            }
        }
    }

```

```

}
addInteractionToChartVisualisation(chartvis, colorField, "series")
if colorField == "stroke" then
  chartvis.properties.enter.strokeWidth = { value = linewidth or 2
  if type(lineScale) == "table" then
    chartvis.properties.enter.strokeWidth.value = nil
    chartvis.properties.enter.strokeWidth =
    {
      scale = "line",
      field= "series"
    }
  end
end

if interpolate then chartvis.properties.enter.interpolate = { value = int

if alphaScale then chartvis.properties.update[colorField .. "Opacity"] =
-- for bars and area charts set the lower bound of their areas
if chartType == "rect" or chartType == "area" then
  if stacked then
    -- for stacked charts this lower bound is the end of the
    chartvis.properties.enter.y2 = { scale = "y", field = "l

  else
    --[[
    for non-stacking charts the lower bound is y=0
    TODO: "yscale.zero" is currently set to "true" for this c
    For the similar behavior "y2" should actually be set to v
    if there are only positive or negative values in the data
    chartvis.properties.enter.y2 = { scale = "y", value = 0 }
  end
end

-- for bar charts ...
if chartType == "rect" then
  -- set 1 pixel width between the bars
  chartvis.properties.enter.width = { scale = "x", band = true, off
  -- for multiple series the bar marking needs to use the "inner" s
  if not stacked and yCount > 1 then
    chartvis.properties.enter.x.scale = "series"
    chartvis.properties.enter.x.field = "series"
    chartvis.properties.enter.width.scale = "series"
  end
end

-- stacked charts have their own (stacked) y values
if stacked then chartvis.properties.enter.y.field = "layout_start" end

-- if there are multiple series group these together
if yCount == 1 then
  chartvis.from = { data = "chart" }
else
  -- if there are multiple series, connect colors to series
  chartvis.properties.update[colorField].field = "series"
  if alphaScale then chartvis.properties.update[colorField .. "Opa

  -- if there are multiple series, connect linewidths to series
  if charttype == "line" then
    chartvis.properties.update["strokeWidth"].field = "series
  end

  -- apply a grouping (facetting) transformation
  chartvis =
  {
    type = "group",
    marks = { chartvis },
  }

```

```

        from =
        {
            data = "chart",
            transform =
            {
                {
                    type = "facet",
                    groupby = { "series" }
                }
            }
        }
    }
    -- for stacked charts apply a stacking transformation
    if stacked then
        table.insert(chartvis.from.transform, 1, { type = "stack" })
    else
        -- for bar charts the series are side-by-side grouped by
        if chartType == "rect" then
            -- for bar charts with multiple series: each series has its own scale
            local groupScale =
            {
                name = "series",
                type = "ordinal",
                range = "width",
                domain = { field = "series" }
            }

            chartvis.from.transform[1].groupby = "x"
            chartvis.scales = { groupScale }
            chartvis.properties = { enter = { x = { field = "series" } } }
        end
    end
end

return chartvis
end

local function getTextMarks(chartvis, chartType, outerRadius, scales, radiusScale)
    local properties
    if chartType == "rect" then
        properties =
        {
            x = { scale = chartvis.properties.enter.x.scale, field = "x" },
            y = { scale = chartvis.properties.enter.y.scale, field = "y" },
            dx = { scale = chartvis.properties.enter.x.scale, band = "width", value = 0 },
            dy = { scale = chartvis.properties.enter.x.scale, band = "height", value = 0 },
            align = { },
            baseline = { value = "middle" },
            fill = { },
            angle = { value = -90 },
            fontSize = { value = tonumber(showValues.fontSize) or 11 }
        }
        if properties.y.offset >= 0 then
            properties.align.value = "right"
            properties.fill.value = showValues.fontcolor or "white"
        else
            properties.align.value = "left"
            properties.fill.value = showValues.fontcolor or persistenter.fontcolor
        end
    elseif chartType == "pie" then
        properties =
        {
            x = { group = "width", mult = 0.5 },
            y = { group = "height", mult = 0.5 },
        }
    end
end

```

```

        radius = { offset = tonumber(showValues.offset) or -4 },
        theta = { field = "layout_mid" },
        fill = { value = showValues.fontcolor or persistentGrey },
        baseline = { },
        angle = { },
        fontSize = { value = tonumber(showValues.fontsize) or mat
    }
    if (showValues.angle or "midangle") == "midangle" then
        properties.align = { value = "center" }
        properties.angle = { field = "layout_mid", mult = 180.0 }

        if properties.radius.offset >= 0 then
            properties.baseline.value = "bottom"
        else
            if not showValues.fontcolor then properties.fill
            properties.baseline.value = "top"
        end
    elseif tonumber(showValues.angle) then
        -- quantize scale for aligning text left on right half-cir
        local alignScale = { name = "align", type = "quantize", c
        table.insert(scales, alignScale)

        properties.align = { scale = alignScale.name, field = "la
        properties.angle = { value = tonumber(showValues.angle) }
        properties.baseline.value = "middle"
        if not tonumber(showValues.offset) then properties.radius
    end

    if radiusScale then
        properties.radius.scale = radiusScale.name
        properties.radius.field = radiusScale.domain.field
    else
        properties.radius.value = outerRadius
    end
end

if properties then
    if showValues.format then
        local template = "datum.y"
        if yType == "integer" or yType == "number" then template
        elseif yType == "date" then template = template .. "|time
        end
        properties.text = { template = "{" .. template .. "}" }
    else
        properties.text = { field = "y" }
    end

    local textmarks =
    {
        {
            type = "text",
            properties =
            {
                enter = properties
            }
        }
    }
    if chartvis.from then textmarks.from = copy(chartvis.from) end
    return textmarks
end
end

local function getSymbolMarks(chartvis, symSize, symShape, symStroke, noFill, alp
    local symbolmarks

```



```

symbolmarks =
{
    type = "symbol",
    properties =
    {
        enter =
        {
            x = { scale = "x", field = "x" },
            y = { scale = "y", field = "y" },
            strokeWidth = { value = symStroke },
            stroke = { scale = "color", field = "series" },
            fill = { scale = "color", field = "series" },
        }
    }
}
if type(symShape) == "string" then
    symbolmarks.properties.enter.shape = { value = symShape }
end
if type(symShape) == "table" then
    symbolmarks.properties.enter.shape = { scale = "symShape", field =
end
if type(symSize) == "number" then
    symbolmarks.properties.enter.size = { value = symSize }
end
if type(symSize) == "table" then
    symbolmarks.properties.enter.size = { scale = "symSize", field =
end
if noFill then
    symbolmarks.properties.enter.fill = nil
end
if alphaScale then
    symbolmarks.properties.enter.fillOpacity =
    { scale = "transparency", field = "series" }
    symbolmarks.properties.enter.strokeOpacity =
    { scale = "transparency", field = "series" }
end
if chartvis.from then symbolmarks.from = copy(chartvis.from) end

return symbolmarks
end

local function getAnnoMarks(chartvis, stroke, fill, opacity)
    local vannolines, hannolines, vannolabels, vannolabels
    vannolines =
    {
        type = "rule",
        from = { data = "v_anno" },
        properties =
        {
            update =
            {
                x = { scale = "x", field = "x" },
                y = { value = 0 },
                y2 = { field = { group = "height" } },
                strokeWidth = { value = stroke },
                stroke = { value = persistentGrey },
                opacity = { value = opacity }
            }
        }
    }
    vannolabels =
    {
        type = "text",
    }

```

```

        from = { data = "v_anno" },
        properties =
        {
            update =
            {
                x = { scale = "x", field = "x", offset = 3 },
                y = { field = { group = "height" }, offset = -3 },
                text = { field = "label" },
                baseline = { value = "top" },
                angle = { value = -90 },
                fill = { value = persistentGrey },
                opacity = { value = opacity }
            }
        }
    }
    hannolines =
    {
        type = "rule",
        from = { data = "h_anno" },
        properties =
        {
            update =
            {
                y = { scale = "y", field = "y" },
                x = { value = 0 },
                x2 = { field = { group = "width" } },
                strokeWidth = { value = stroke },
                stroke = { value = persistentGrey },
                opacity = { value = opacity }
            }
        }
    }
    hannolabels =
    {
        type = "text",
        from = { data = "h_anno" },
        properties =
        {
            update =
            {
                y = { scale = "y", field = "y", offset = 3 },
                x = { value = 0 , offset = 3 },
                text = { field = "label" },
                baseline = { value = "top" },
                angle = { value = 0 },
                fill = { value = persistentGrey },
                opacity = { value = opacity }
            }
        }
    }
    return vannolines, vannolabels, hannolines, hannolabels
end

local function getAxes(xTitle, xAxisFormat, xAxisAngle, xType, xGrid, yTitle, yAxisFormat, yType, yGrid)
    local xAxis, yAxis
    if chartType ~= "pie" then
        if xType == "integer" and not xAxisFormat then xAxisFormat = "d"
        xAxis =
        {
            type = "x",
            scale = "x",
            title = xTitle,
            format = xAxisFormat,
            grid = xGrid
        }
    end
    if yType == "integer" and not yAxisFormat then yAxisFormat = "d"
    yAxis =
    {
        type = "y",
        scale = "y",
        title = yTitle,
        format = yAxisFormat,
        grid = yGrid
    }
end

```

```

}
if xAxisAngle then
    local xAxisAlign
    if xAxisAngle < 0 then xAxisAlign = "right" else xAxisAlign = "left"
    xAxis.properties =
    {
        title =
        {
            fill = { value = persistentGrey }
        },
        labels =
        {
            angle = { value = xAxisAngle },
            align = { value = xAxisAlign },
            fill = { value = persistentGrey }
        },
        ticks =
        {
            stroke = { value = persistentGrey }
        },
        axis =
        {
            stroke = { value = persistentGrey },
            strokeWidth = { value = 2 }
        },
        grid =
        {
            stroke = { value = persistentGrey }
        }
    }
else
    xAxis.properties =
    {
        title =
        {
            fill = { value = persistentGrey }
        },
        labels =
        {
            fill = { value = persistentGrey }
        },
        ticks =
        {
            stroke = { value = persistentGrey }
        },
        axis =
        {
            stroke = { value = persistentGrey },
            strokeWidth = { value = 2 }
        },
        grid =
        {
            stroke = { value = persistentGrey }
        }
    }
end

if yType == "integer" and not yAxisFormat then yAxisFormat = "d"
yAxis =
{
    type = "y",
    scale = "y",
    title = yTitle,
    format = yAxisFormat,

```

```

        grid = yGrid
      }
      yAxis.properties =
      {
        title =
        {
          fill = { value = persistentGrey }
        },
        labels =
        {
          fill = { value = persistentGrey }
        },
        ticks =
        {
          stroke = { value = persistentGrey }
        },
        axis =
        {
          stroke = { value = persistentGrey },
          strokeWidth = { value = 2 }
        },
        grid =
        {
          stroke = { value = persistentGrey }
        }
      }

    end

    return xAxis, yAxis
  end

  local function getLegend(legendTitle, chartType, outerRadius)
    local legend =
    {
      fill = "color",
      stroke = "color",
      title = legendTitle,
    }
    legend.properties = {
      title = {
        fill = { value = persistentGrey },
      },
      labels = {
        fill = { value = persistentGrey },
      },
    }
    if chartType == "pie" then
      legend.properties = {
        -- move legend from center position to top
        legend = {
          y = { value = -outerRadius },
        },
        title = {
          fill = { value = persistentGrey }
        },
        labels = {
          fill = { value = persistentGrey },
        },
      }
    end
    return legend
  end
end

```

```

function p.chart(frame)
  -- chart width and height
  local graphwidth = tonumber(frame.args.width) or 200
  local graphheight = tonumber(frame.args.height) or 200
  -- chart type
  local chartType = frame.args.type or "line"
  -- interpolation mode for line and area charts: linear, step-before, step
  local interpolate = frame.args.interpolate
  -- mark colors (if no colors are given, the default 10 color palette is used)
  local colorString = frame.args.colors
  if colorString then colorString = string.lower(colorString) end
  local colors = stringArray(colorString)
  -- for line charts, the thickness of the line; for pie charts the gap between slices
  local linewidth = tonumber(frame.args.linewidth)
  local linewidthsString = frame.args.linewidths
  local linewidths
  if linewidthsString and linewidthsString ~= "" then linewidths = numericArray(linewidthsString)
  -- x and y axis caption
  local xTitle = frame.args.xAxisTitle or frame.args.xaxistitle
  local yTitle = frame.args.yAxisTitle or frame.args.yaxistitle
  -- x and y value types
  local xType = frame.args.xType or frame.args.xtype
  local yType = frame.args.yType or frame.args.ytype
  -- override x and y axis minimum and maximum
  local xMin = frame.args.xAxisMin or frame.args.xaxismin
  local xMax = frame.args.xAxisMax or frame.args.xaxismax
  local yMin = frame.args.yAxisMin or frame.args.yaxismin
  local yMax = frame.args.yAxisMax or frame.args.yaxismax
  -- override x and y axis label formatting
  local xAxisFormat = frame.args.xAxisFormat or frame.args.xaxisformat
  local yAxisFormat = frame.args.yAxisFormat or frame.args.yaxisformat
  local xAxisAngle = tonumber(frame.args.xAxisAngle) or tonumber(frame.args.xaxisangle)
  -- x and y scale types
  local xScaleType = frame.args.xScaleType or frame.args.xscaletype
  local yScaleType = frame.args.yScaleType or frame.args.yscaletype
  -- log scale require minimum > 0, for now it's not possible to plot negative values
  -- if xScaleType == "log" then
  --   if (not xMin or tonumber(xMin) <= 0) then xMin = 0.1 end
  --   if not xType then xType = "number" end
  -- end
  -- if yScaleType == "log" then
  --   if (not yMin or tonumber(yMin) <= 0) then yMin = 0.1 end
  --   if not yType then yType = "number" end
  -- end

  -- show grid
  local xGrid = frame.args.xGrid or frame.args.xgrid or false
  local yGrid = frame.args.yGrid or frame.args.ygrid or false
  -- for line chart, show a symbol at each data point
  local showSymbols = frame.args.showSymbols or frame.args.showsymbols
  local symbolsShape = frame.args.symbolsShape or frame.args.symbolsshape
  local symbolsNoFill = frame.args.symbolsNoFill or frame.args.symbolsnofill
  local symbolsStroke = tonumber(frame.args.symbolsStroke) or frame.args.symbolsstroke
  -- show legend with given title
  local legendTitle = frame.args.legend
  -- show values as text
  local showValues = frame.args.showValues or frame.args.showvalues
  -- show v- and h-line annotations
  local v_annoLineString = frame.args.vAnnotatonsLine or frame.args.vannotatonsline
  local h_annoLineString = frame.args.hAnnotatonsLine or frame.args.hannotatonsline
  local v_annoLabelString = frame.args.vAnnotatonsLabel or frame.args.vannotatonslabel
  local h_annoLabelString = frame.args.hAnnotatonsLabel or frame.args.hannotatonslabel

```

```
-- decode annotations cvs
local v_annoLine, v_annoLabel, h_annoLine, h_annoLabel
if v_annoLineString and v_annoLineString ~= "" then

    if xType == "number" or xType == "integer" then
        v_annoLine = numericArray(v_annoLineString)

    else
        v_annoLine = stringArray(v_annoLineString)

    end
    v_annoLabel = stringArray(v_annoLabelString)
end
if h_annoLineString and h_annoLineString ~= "" then

    if yType == "number" or yType == "integer" then
        h_annoLine = numericArray(h_annoLineString)

    else
        h_annoLine = stringArray(h_annoLineString)

    end
    h_annoLabel = stringArray(h_annoLabelString)
end

-- pie chart radiuses
local innerRadius = tonumber(frame.args.innerRadius) or tonumber(frame.args.radius)
local outerRadius = math.min(graphwidth, graphheight)
-- format JSON output
local formatJson = frame.args.formatjson

-- get x values
local x
x, xType, xMin, xMax = deserializeXData(frame.args.x, xType, xMin, xMax)

-- get y values (series)
local yValues = {}
local seriesTitles = {}
for name, value in pairs(frame.args) do
    local yNum
    if name == "y" then yNum = 1 else yNum = tonumber(string.match(name, "y(%d+)"))
    if yNum then
        yValues[yNum] = value
        -- name the series: default is "y<number>". Can be overwritten
        seriesTitles[yNum] = frame.args["y" .. yNum .. "Title"] or "y" .. yNum
    end
end
local y
y, yType, yMin, yMax = deserializeYData(yValues, yType, yMin, yMax)

-- create data tuples, consisting of series index, x value, y value
local data
if chartType == "pie" then
    -- for pie charts the second series is merged into the first
    data = convertXYToSingleSeries(x, y, xType, yType, { "y", "r" })
else
    data = convertXYToManySeries(x, y, xType, yType, seriesTitles)
end
```

```

end

-- configure stacked charts
local stacked = false
local stats
if string.sub(chartType, 1, 7) == "stacked" then
    chartType = string.sub(chartType, 8)
    if #y > 1 then -- ignore stacked charts if there is only one series
        stacked = true
        -- aggregate data by cumulative y values
        stats =
        {
            {
                name = "stats", source = "chart", transform =
                {
                    {
                        type = "aggregate",
                        groupby = { "x" },
                        summarize = { y = "sum" }
                    }
                }
            }
        }
    end
end

-- add annotations to data
local vannoData, hannoData

if v_annoLine then
    vannoData = { name = "v_anno", format = { type = "json", parse =
    for i = 1, #v_annoLine do
        local item = { x = v_annoLine[i], label = v_annoLabel[i]
        table.insert(vannoData.values, item)
    end
end
if h_annoLine then
    hannoData = { name = "h_anno", format = { type = "json", parse =
    for i = 1, #h_annoLine do
        local item = { y = h_annoLine[i], label = h_annoLabel[i]
        table.insert(hannoData.values, item)
    end
end

-- create scales
local scales = {}

local xscale = getXScale(chartType, stacked, xMin, xMax, xType, xScaleType)
table.insert(scales, xscale)
local yscale = getYScale(chartType, stacked, yMin, yMax, yType, yScaleType)
table.insert(scales, yscale)

local colorScale = getColorScale(colors, chartType, #x, #y)
table.insert(scales, colorScale)

local alphaScale = getAlphaColorScale(colors, y)
table.insert(scales, alphaScale)

local lineScale
if (linewidths) and (chartType == "line") then
    lineScale = getLineScale(linewidths, chartType)
    table.insert(scales, lineScale)
end

local radiusScale

```

```

    if chartType == "pie" and #y > 1 then
        radiusScale = getValueScale("r", 0, outerRadius)
        table.insert(scales, radiusScale)
    end

    -- decide if lines (strokes) or areas (fills) should be drawn
    local colorField
    if chartType == "line" then colorField = "stroke" else colorField = "fill"

    -- create chart markings
    local chartvis = getChartVisualisation(chartType, stacked, colorField, #y)
    local marks = { chartvis }

    -- text marks
    if showValues then
        if type(showValues) == "string" then -- deserialize as table
            local keyValues = mw.text.split(showValues, "%s*,%s*")
            showValues = {}
            for _, kv in ipairs(keyValues) do
                local key, value = mw.ustring.match(kv, "^%s*(.-)")
                if key then showValues[key] = value end
            end
        end

        local chartmarks = chartvis
        if chartmarks.marks then chartmarks = chartmarks.marks[1] end
        local textmarks = getTextMarks(chartmarks, chartType, outerRadius)
        if chartmarks ~= chartvis then
            table.insert(chartvis.marks, textmarks)
        else
            table.insert(marks, textmarks)
        end
    end

    -- grids
    if xGrid then
        if xGrid == "0" then xGrid = false
        elseif xGrid == 0 then xGrid = false
        elseif xGrid == "false" then xGrid = false
        elseif xGrid == "n" then xGrid = false
        else xGrid = true
        end
    end

    if yGrid then
        if yGrid == "0" then yGrid = false
        elseif yGrid == 0 then yGrid = false
        elseif yGrid == "false" then yGrid = false
        elseif yGrid == "n" then yGrid = false
        else yGrid = true
        end
    end

    -- symbol marks
    if showSymbols and chartType ~= "rect" then
        local chartmarks = chartvis
        if chartmarks.marks then chartmarks = chartmarks.marks[1] end

        if type(showSymbols) == "string" then
            if showSymbols == "" then showSymbols = true
            else showSymbols = numericArray(showSymbols)
            end
        end
    end
end

```



```

        showSymbols = tonumber(showSymbols)
    end

    -- custom size
    local symSize
    if type(showSymbols) == "number" then
        symSize = tonumber(showSymbols*showSymbols*8.5)
    elseif type(showSymbols) == "table" then
        symSize = {}
        for k, v in pairs(showSymbols) do
            symSize[k]=v*v*8.5 -- "size" acc to Vega syntax is area of symbol
        end
    end
else
    symSize = 50
end

-- symSizeScale
local symSizeScale = {}
if type(symSize) == "table" then
    symSizeScale = getSymSizeScale(symSize)
    table.insert(scales, symSizeScale)
end

-- custom shape
if stringArray(symbolsShape) and #stringArray(symbolsShape) > 1 then
    local symShape = " "

    if type(symbolsShape) == "string" and shapes[symbolsShape] then
        symShape = shapes[symbolsShape]
    elseif type(symbolsShape) == "table" then
        symShape = {}
        for k, v in pairs(symbolsShape) do
            if symbolsShape[k] and shapes[symbolsShape[k]] then
                symShape[k]=shapes[symbolsShape[k]]
            else
                symShape[k] = "circle"
            end
        end
    end
else
    symShape = "circle"
end

-- symShapeScale
local symShapeScale = {}
if type(symShape) == "table" then
    symShapeScale = getSymShapeScale(symShape)
    table.insert(scales, symShapeScale)
end

-- custom stroke
local symStroke
if (type(symbolsStroke) == "number") then
    symStroke = tonumber(symbolsStroke)
-- TODO symStroke serialization
elseif type(symbolsStroke) == "table" then
    symStroke = {}
    for k, v in pairs(symbolsStroke) do
        symStroke[k]=symbolsStroke[k]
        --always draw x with stroke
        if symbolsShape[k] == "x" then symStroke[k] = 2
        --always draw x with stroke
        if symbolsNoFill[k] then symStroke[k] = 2.5 end
    end
end
else

```

```

        symStroke = 0
--always draw x with stroke
        if symbolsShape == "x" then symStroke = 2.5 end
--always draw x with stroke
        if symbolsNoFill then symStroke = 2.5 end
    end

--      TODO      -- symStrokeScale
--      local symStrokeScale = {}
--      if type(symStroke) == "table" then
--          symStrokeScale = getSymStrokeScale(symStroke)
--          table.insert(scales, symStrokeScale)
--      end

    local symbolmarks = getSymbolMarks(chartmarks, symSize, symShape)
    if chartmarks ~= chartvis then
        table.insert(chartvis.marks, symbolmarks)
    else
        table.insert(marks, symbolmarks)
    end
end

local vannolines, vannolabels, hannolines, hannolabels = getAnnoMarks(chartmarks)
if vannodata then
    table.insert(marks, vannolines)
    table.insert(marks, vannolabels)
end
if hannodata then
    table.insert(marks, hannolines)
    table.insert(marks, hannolabels)
end

-- axes
local xAxis, yAxis = getAxes(xTitle, xAxisFormat, xAxisAngle, xType, xGrid)

-- legend
local legend
if legendTitle and tonumber(legendTitle) ~= 0 then legend = getLegend(legendTitle)
-- construct final output object
local output =
{
    version = 2,
    width = graphwidth,
    height = graphheight,
    data = { data },
    scales = scales,
    axes = { xAxis, yAxis },
    marks = marks,
    legends = { legend }
}
if vannodata then table.insert(output.data, vannodata) end
if hannodata then table.insert(output.data, hannodata) end
if stats then table.insert(output.data, stats) end

local flags
if formatJson then flags = mw.text.JSON_PRETTY end
return mw.text.jsonEncode(output, flags)
end

function p.mapWrapper(frame)

```

```
        return p.map(frame:getParent())
    end

    function p.chartWrapper(frame)
        return p.chart(frame:getParent())
    end

    function p.chartDebugger(frame)
        return    "\n\nchart JSON\n " .. p.chart(frame) .. " \n\n" .. debuglog
    end

    -- Given an HTML-encoded title as first argument, e.g. one produced with {{ARTICLE}}
    -- convert it into a properly URL path-encoded string
    -- This function is critical for any graph that uses path-based APIs, e.g. PageView
    function p.encodeTitleForPath(frame)
        return mw.uri.encode(mw.text.decode(mw.text.trim(frame.args[1])), 'PATH')
    end

    return p
```