JupyterViz

Visualization Tools for Jupyter and the GAP REPL

1.5.1

28 March 2019

Nathan Carter

Nathan Carter
Email: ncarter@bentley.edu
Homepage: http://nathancarter.github.io
Address: 175 Forest St.
Waltham, MA 02452
USA

Contents

Chapter 1

Introduction

1.1 Purpose

Since 2017, it has been possible to use GAP in Jupyter through the JupyterKernel package. Output was limited to the ordinary text output GAP produces; charts and graphs were not possible.

In 2018, Martins and Pfeiffer released francy (repository, article), which lets users create graphs of a few types (vertices and edges, line chart, bar chart, scatter chart) in a Jupyter notebook. It also allows the user to attach actions to the elements of these charts, which result in callbacks to GAP that can update the visualization.

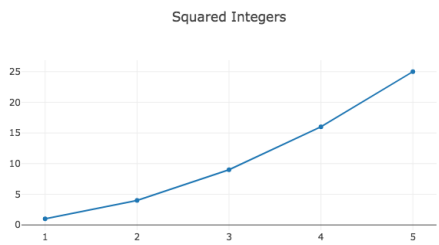
This visualization package has different aims in three ways. First, it can function either in a Jupyter notebook or directly from the GAP REPL on the command line. Second, it aims to make a wider variety of visualizations accessible to GAP users. Third, it does not provide tools for conveniently making such visualizations interactive. Where the francy package excels at interactive visualizations, this package instead gives a broader scope of visualization tools and does not require Jupyter.

These goals are achieved by importing several existing JavaScript visualization toolkits and exposing them to GAP code, as described later in this manual.

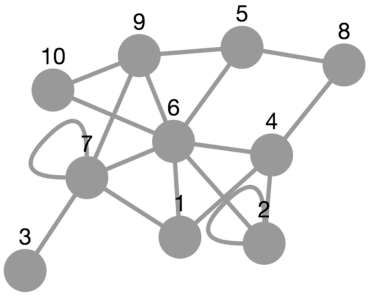
1.2 Terminology (What is a Graph?)

There is an unfortunate ambiguity about the word "graph" in mathematics. It is used to mean both "the graph of a function drawn on coordinate axes" and "a collection of vertices with edges connecting them." This is particularly troublesome in a package like this one, where we will provide tools for drawing both of these things! Consequently, we remove the ambiguity as follows.

We will say "charts and plots" to refer to the first concept (lines, curves, bars, dots, etc. on coordinate axes) and "graphs" (or sometimes "graph drawing") to refer only to the second concept (vertices and edges). This convention holds throughout this entire document.



A plot or chart



A graph

To support both of these types of visualizations, this package imports a breadth of JavaScript visualization libraries (and you can extend it with more, as in Chapter 5). We split them into the categories defined above.

1.2.1 Toolkits for drawing charts and plots

- AnyChart
- CanvasJS
- ChartJS
- Plotly (the default tool used when you call `Plot` (7.1.1))

1.2.2 Toolkits for drawing graphs

- Cytoscape (the default tool used when you call `PlotGraph` (7.1.3))

1.2.3 General purpose tools with which you can define custom visualizations

- D3
- Native HTML `canvas` element
- Plain HTML

1.3 The high-level API and the low-level API

This package exposes the JavaScript tools to the GAP user in two ways.

Foundationally, a low-level API gives direct access to the JSON passed to those tools and to JavaScript code for manipulating the visualizations the tools create. This is powerful but not convenient to use.

More conveniently, a high-level API gives two functions, one for creating plots and charts (`Plot` (7.1.1)) and one for creating graphs (`PlotGraph` (7.1.3)). The high-level API should handle the vast majority of use cases, but if an option you need is not supported by it, there is still the low-level API on which you can fall back.

1.4 Loading the package (in Jupyter or otherwise)

To import this package, use the following GAP command from the command line or from a cell in a Jupyter notebook running a GAP kernel.

Example

```
LoadPackage( "jupyterviz" );
```

To see how to use the package, we recommend next reading Chapter 2 on the high-level API, and if you find it necessary, also Chapter 3 on the low-level API. Each chapter contains numerous examples of how to use the package.

Chapter 2

Using the high-level API

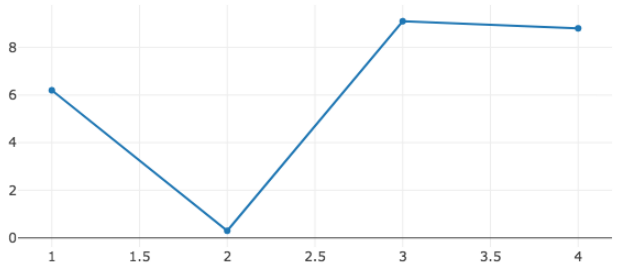
2.1 Charts and Plots

This section covers the `Plot` (7.1.1) function in the high-level API, which is used for showing charts and plots. If invoked in a Jupyter Notebook, it will show the resulting visualization in the appropriate output cell of the notebook. If invoked from the GAP command line, it will use the system default web browser to show the resulting visualization, from which the user can save a permanent copy, print it, etc. This section covers that function through a series of examples, but you can see full details in the function reference in Chapter 7.

To plot a list of numbers as a single data series, just pass the list to `Plot` (7.1.1).

Example

```
Plot( [ 6.2, 0.3, 9.1, 8.8 ] );
```



Notice that the default x values for the data are the sequence `[1..n]`, where n is the length of the data. You can specify the x values manually, like so:

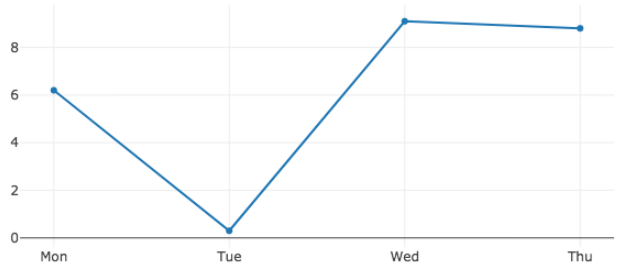
Example

```
Plot( [ 1 .. 4 ], [ 6.2, 0.3, 9.1, 8.8 ] );
```

This is useful if you have a scatter plot of data to make, or if your x values are not numbers at all.

Example

```
Plot( [ "Mon", "Tue", "Wed", "Thu" ], [ 6.2, 0.3, 9.1, 8.8 ] );
```



It is also permissible to send in a list of (x, y) pairs rather than placing the xs and ys in separate lists. This would do the same as the previous:

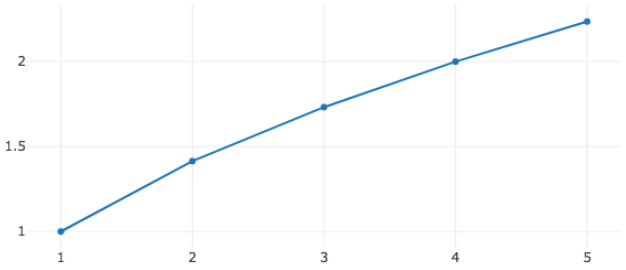
Example

```
Plot( [ [ "Mon", 6.2 ], [ "Tue", 0.3 ], [ "Wed", 9.1 ], [ "Thu", 8.8 ] ] );
```

You can also pass a single-variable numeric function to have it plotted.

Example

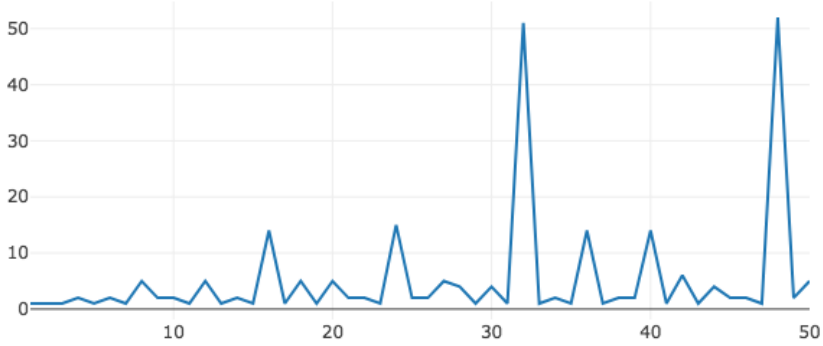
```
Plot( x -> x^0.5 );
```



It assumes a small domain of positive integers, which you can customize as follows. Note that the x values are passed just as before, but in place of the y values, we pass the function that computes them.

Example

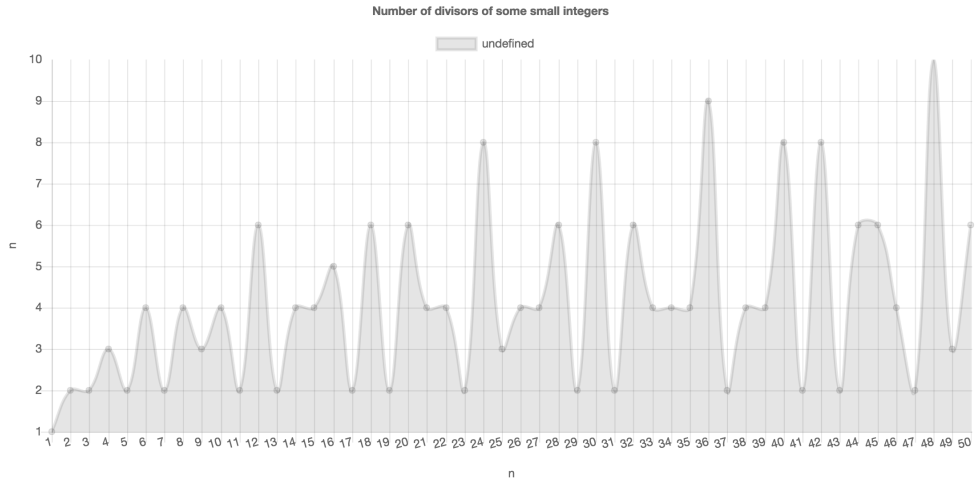
```
Plot( [1..50], NrSmallGroups );
```



You can append a final parameter to the `Plot` (7.1.1) command, a record containing a set of options. Here is an example of using that options record to choose the visualization tool, title, and axis labels. Section 2.2 covers options in detail; this is only a preview.

Example

```
Plot( [1..50], n -> Length( DivisorsInt( n ) ),
      rec(
          tool := "chartjs",
          title := "Number of divisors of some small integers",
          xaxis := "n",
          yaxis := "number of divisors of n"
      )
);
```

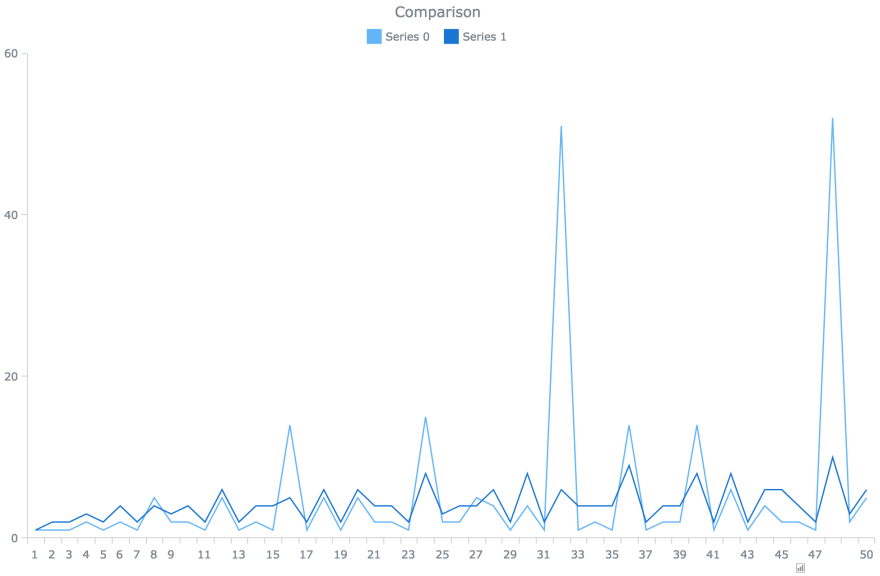


You can also put multiple data series (or functions) on the same plot. Let's pretend you wanted to compare the number of divisors of n with the number of groups of order n for the first 50 positive integers n.

To do so, take each call you would make to `Plot` (7.1.1) to make the separate plots, and place those arguments in a list. Pass both lists to `Plot` (7.1.1) to combine the plots, as shown below. You can put the options record in either list. Options specified earlier take precedence if there's a conflict.

Example

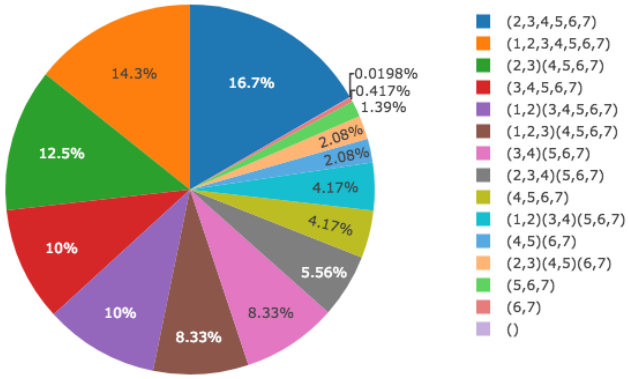
```
# We're combining Plot( [1..50], NrSmallGroups );
# with Plot( [1..50], n -> Length( DivisorsInt( n ) ) );
# and adding some options:
Plot(
    [ [1..50], NrSmallGroups,
      rec( title := "Comparison", tool := "anychart" ) ],
    [ [1..50], n -> Length( DivisorsInt( n ) ) ]
);
```



The default plot type is "line", as you've been seeing in the preceding examples. You can also choose "bar", "column", "pie", and others. Let's use a pie chart to show the relative sizes of the conjugacy classes in a group.

Example

```
G := Group( (1,2,3,4,5,6,7), (1,2) );;
CCs := List( ConjugacyClasses( G ), Set );;
Plot(
    # x values are class labels; we'll use the first element in the class
    List( CCs, C -> PrintString( C[1] ) ),
    # y values are class sizes; these determine the size of pie slices
    List( CCs, Length ),
    # ask for a pie chart with enough height that we can read the legend
    rec( type := "pie", height := 500 )
);
```



2.2 Options for charts and plots

The options record passed as the final parameter to `Plot` (7.1.1) (or as the final element in each list passed to `Plot` (7.1.1), if you are plotting multiple series on the same plot) can have the following entries.

- `tool` - the visualization tool to use to make the plot, as a string. The default is "plotly". The full list of tools is available in Section 1.2.
- `type` - the type of chart, as a string, the default for which is "line". Which types are available depends on which tool you are using, though it is safe to assume that most common chart types (line, bar, pie) are supported by all tools. Section 3.4 contains links to the documentation for each tool, so that you might see its full list of capabilities.
- `height` - the height in pixels of the visualization to produce. A sensible default is provided, which varies by tool.
- `width` - the width in pixels of the visualization to produce. If omitted, the tool usually fills the width available. In a Jupyter Notebook output cell, this is the width of the cell. A visualization shown outside of a Jupyter notebook will take up the entire width of the web page in which it is displayed.
- `title` - the title to place at the top of the chart, as a string. Can be omitted.
- `xaxis` - the text to write below the *x* axis, as a string. Can be omitted.
- `yaxis` - the text to write to the left of the *y* axis, as a string. Can be omitted.
- `name` - this option should be specified in the options object for each separate data series, as opposed to just once for the entire plot. It assigns a string name to that data series, typically included in the chart's legend.

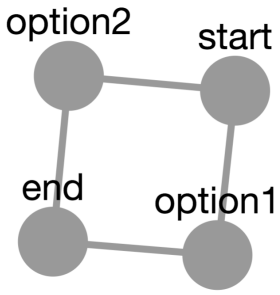
2.3 Graphs

This section covers the `PlotGraph` (7.1.3) function in the high-level API, which is used for drawing graphs. If invoked in a Jupyter Notebook, it will show the resulting visualization in the appropriate output cell of the notebook. If invoked from the GAP command line, it will use the system default web browser to show the resulting visualization. This section covers that function through a series of examples, but you can see full details in the function reference in Chapter 7.

You can make a graph by calling `PlotGraph` (7.1.3) on a list of edges, each of which is a pair (a list of length two).

Example

```
PlotGraph( [ [ "start", "option1" ], [ "start", "option2" ],
             [ "option1", "end" ], [ "option2", "end" ] ] );
```

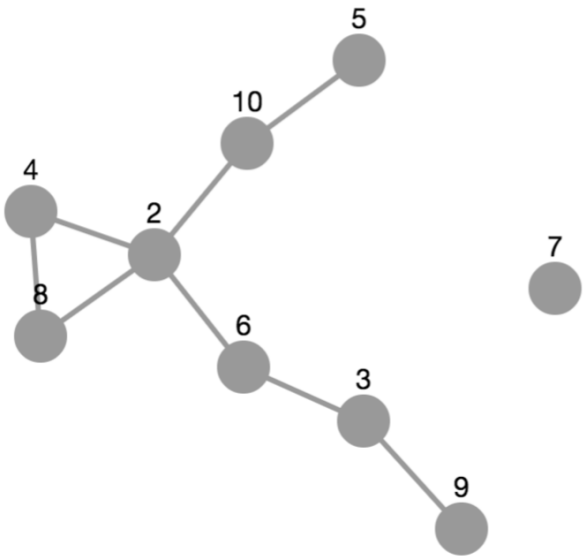


Vertex names can be strings, as shown above, or any GAP data; they will be converted to strings using `PrintString`. As you can see, the set of vertices is assumed to be the set of things mentioned in the edges. But you can specify the vertex set separately.

For example, if you wanted to graph the divisibility relation on a set of integers, some elements may not be included in any edge.

Example

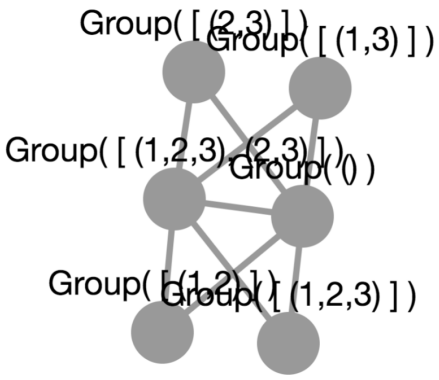
```
PlotGraph( [ 2 .. 10 ],
           [ [ 2, 4 ], [ 2, 6 ], [ 2, 8 ], [ 2, 10 ],
             [ 3, 6 ], [ 3, 9 ], [ 4, 8 ], [ 5, 10 ] ] );
```



But for anything other than a small graph, specifying the vertex or edge set manually may be inconvenient. Thus if you have a vertex set, you can create the edge set by passing a binary relation as a GAP function. Here is an example to create a subgroup lattice.

Example

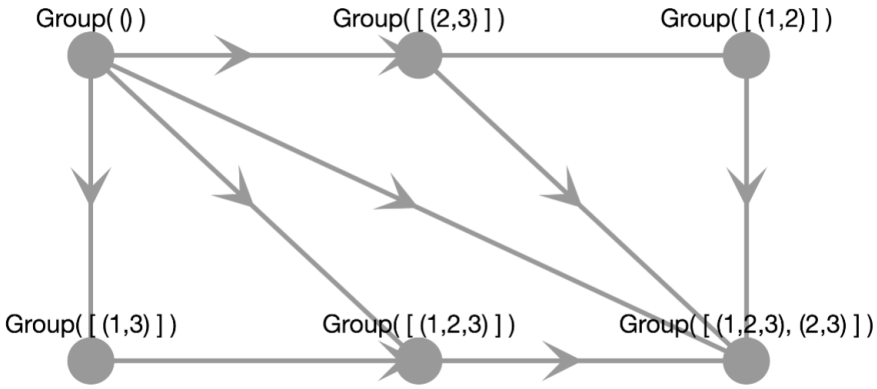
```
G := Group( (1,2,3), (1,2) );
S := function ( H, G )
    return IsSubgroup( G, H ) and H <> G;
end;
PlotGraph( AllSubgroups( G ), S );
```



But all three of the graphs just shown would be better if they had directed edges. And we might want to organize them differently in the view, perhaps even with some colors, etc. To this end, you can pass an options parameter as the final parameter to `PlotGraph` (7.1.3), just as you could for `Plot` (7.1.1).

Example

```
G := Group( (1,2,3), (1,2) );
S := function ( H, G )
    return IsSubgroup( G, H ) and H <> G;
end;
PlotGraph( AllSubgroups( G ), S,
    rec( directed := true, layout := "grid", arrowscale := 3 ) );
```



The next section covers all options in detail.

2.4 Options for graphs

The options record passed as the final parameter to `PlotGraph` (7.1.3) can have the following entries.

- `tool` - the visualization tool to use to make the plot, as a string. The default is "cytoscape". The full list of tools is available in Section 1.2.
- `layout` - the name of the layout algorithm to use, as a string. Permitted values vary by tool. Currently cytoscape supports "preset" (meaning you must have specified the nodes' positions manually), "cose" (virtual-spring-based automatic layout), "random", "grid", "circle", "concentric" (multiple concentric circles), and "breadthfirst" (a hierarchy).
- `vertexwidth` and `vertexheight` - the dimensions of each vertex.
- `vertexcolor` - the color of the vertices in the graph. This should be a string representing an HTML color, such as "#ccc" or "red".
- `edgewidth` - the thickness of each edge.
- `edgecolor` - the color of each edge and its corresponding arrow. This should be a string representing an HTML color, such as "#ccc" or "red".
- `directed` - a boolean defaulting to false, whether to draw arrows to visually indicate that the graph is a directed graph
- `arrowscale` - a multiplier to increase or decrease the size of arrows in a directed graph.
- `height` - the height in pixels of the visualization to produce. A sensible default is provided, which varies by tool.
- `width` - the width in pixels of the visualization to produce. If omitted, the tool usually fills the width available. In a Jupyter Notebook output cell, this is the width of the cell. A visualization shown outside of a Jupyter notebook will take up the entire width of the web page in which it is displayed.

Chapter 3

Using the low-level API

3.1 The `CreateVisualization` function

The low-level API is accessed through just one function, `CreateVisualization` (7.2.5). You can view its complete documentation in the function reference in Chapter 7, but examples are given in this chapter.

Nearly all visualizations in this package are created by passing to the `CreateVisualization` (7.2.5) function records describing what to draw. Even visualizations created by the high-level API documented in Chapter 2 call the `CreateVisualization` (7.2.5) function under the hood. Those records are converted into JSON form by the json package, and handed to whichever JavaScript toolkit you have chosen to use for creating the visualization (or the default tool if you use a high-level function and do not specify).

The sections below describe how to communicate with each of the visualization tools this package makes available, using `CreateVisualization` (7.2.5).

3.2 Looking beneath the high-level API

There are a few techniques for taking a call to the high-level API (either to `Plot` (7.1.1) or `PlotGraph` (7.1.3)) and computing what data it eventally passes to `CreateVisualization` (7.2.5). This is a great starting point for learning the data formats that `CreateVisualization` (7.2.5) expects, in preparation for either tweaking them or creating them from scratch. We cover two examples here.

3.2.1 Looking beneath `Plot`

Assume that you have a plot that you're creating with the high-level API, like the following example.

Example

```
Plot( x -> x^0.5, rec( tool := "canvasjs" ) );
```

You can find out what kind of data is being passed, under the hood, to `CreateVisualization` (7.2.5) by running the following code.

Example

```
dataSeries := JUPVIZMakePlotDataSeries( x -> x^0.5 );;
big := ConvertDataSeriesForTool.canvasjs( [ dataSeries ] );
# The result is the following GAP record:
```

```
# rec(
#     animationEnabled := true,
#     data := [
#         rec(
#             dataPoints := [
#                 rec( x := 1, y := 1 ),
#                 rec( x := 2, y := 1.4142135623730951 ),
#                 rec( x := 3, y := 1.7320508075688772 ),
#                 rec( x := 4, y := 2. ),
#                 rec( x := 5, y := 2.2360679774997898 )
#             ],
#             type := "line"
#         )
#     ],
#     height := 400
# )
```

That record is passed to `CreateVisualization` (7.2.5) with a call like the following.

Example

```
CreateVisualization( rec( tool := "canvasjs", data := big ) );
```

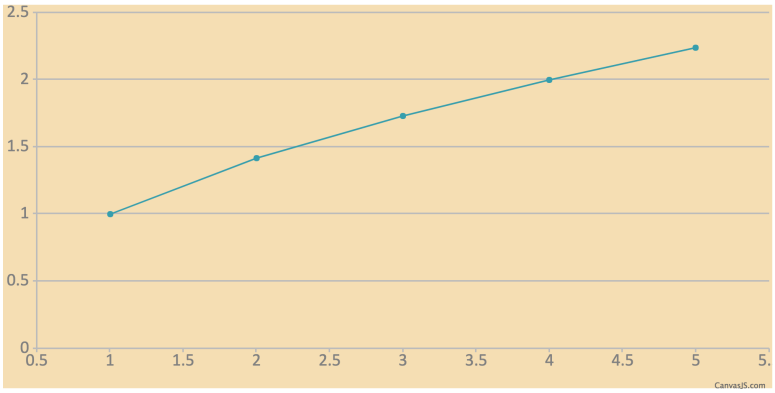
If you wanted to change any of the internal options, such as the default `animationEnabled := true` or the default `height := 400`, you could alter the record yourself before passing it on to `CreateVisualization` (7.2.5).

Such options may be specific to the tool you've chosen, and are not guaranteed to work with other tools. For example, you can't change `"canvasjs"` to `"anychart"` and expect the `animationEnabled` setting to work, because it is specific to CanvasJS. See Section 3.4 for links to each tool's documentation, which give detailed data formats.

If you had researched other options about CanvasJS and wanted to include those, you could do so as well, as shown below.

Example

```
big.animationEnabled := false;;     # changing an option
big.height := 500;;                 # changing an option
big.backgroundColor := "#F5DEB3";;  # adding an option
CreateVisualization( rec( tool := "canvasjs", data := big ) );
```



3.2.2 Looking beneath `PlotGraph`

In the previous section, we saw how you could take a call to `Plot` (7.1.1) and find out what data that call would pass to `CreateVisualization` (7.2.5). You can do the same with `PlotGraph` (7.1.3), but it takes a few more steps.

First, we you must have a list of your graph's vertices. Here we will assume it is in a variable called `vertices`. Second, you must have a list of your graph's edges. Similarly, we will assume it is in a variable called `edges`.

Example

```
vertices := [ 1, 2, 3, 4 ];
edges := [ [ 1, 2 ], [ 2, 3 ], [ 2, 4 ] ];
```

You can then convert your graph into the format passed to `CreateVisualization` (7.2.5) as follows. Note that at the time of this writing, there is only one graph visualization tool, cytoscape, so we use that one directly.

Example

```
big := ConvertGraphForTool.cytoscape( rec(
    vertices := vertices,
    edges := edges,
    options := rec() # or any options you like here
) );
# The result is the following GAP record:
# rec(
#     elements := [
#         rec( data := rec( id := "1" ) ),
#         rec( data := rec( id := "2" ) ),
#         rec( data := rec( id := "3" ) ),
#         rec( data := rec( id := "4" ) ),
#         rec( data := rec( source := "1", target := "2" ) ),
#         rec( data := rec( source := "2", target := "3" ) ),
#         rec( data := rec( source := "2", target := "4" ) )
#     ],
#     layout := rec( name := "cose" ),
#     style := [
#         rec(
#             selector := "node",
#             style := rec( content := "data(id)" )
#         )
#     ]
# )
```

That record is passed to `CreateVisualization` (7.2.5) with a call like the following. Note the inclusion of a default height, if you don't provide one.

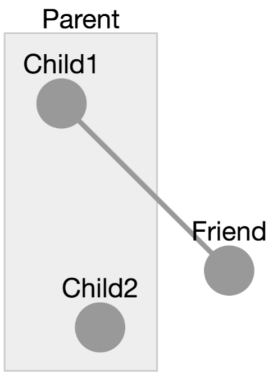
Example

```
CreateVisualization( rec(
    tool := "cytoscape", data := big, height := 400
) );
```

If you wanted to change any of the internal options, including creating elements not supported by the simple high-level API, you could alter or recreate the contents of the `big` record. We do so here, using features we could learn from the cytoscape JSON format reference, linked to in Section 3.4.

Example

```
CreateVisualization( rec(
    tool := "cytoscape",
    height := 400,
    data := rec(
        elements := [
            rec( # node 1
                group := "nodes",
                data := rec( id := "Child1", parent := "Parent" ),
                position := rec( x := 100, y := 100 ),
                selected := false,
                selectable := true,
                locked := false,
                grabbable := true
            ),
            rec( # node 2
                data := rec( id := "Friend" ),
                renderedPosition := rec( x := 200, y := 200 )
            ),
            rec( # node 3
                data := rec( id := "Child2", parent := "Parent" ),
                position := rec( x := 123, y := 234 )
            ),
            rec( # node parent
                data := rec(
                    id := "Parent",
                    position := rec( x := 200, y := 100 )
                )
            ),
            rec( # edge 1
                data := rec(
                    id := "Edge1",
                    source := "Child1",
                    target := "Friend"
                )
            )
        ],
        layout := rec( name := "preset" ),
        style := [
            rec(
                selector := "node",
                style := rec( content := "data(id)" )
            )
        ]
    )
) );
```



3.3 Using JSON from a file

As the documentation cited in Section 3.4 states, all of the underlying visualization tools used by this package accept input in JSON form. You might have some data in that form generated by another source or downloaded from the web. For example, in this package's directory, the file `example/EV Charge Points.json` contains JSON data from one of the Plotly project's blog posts.

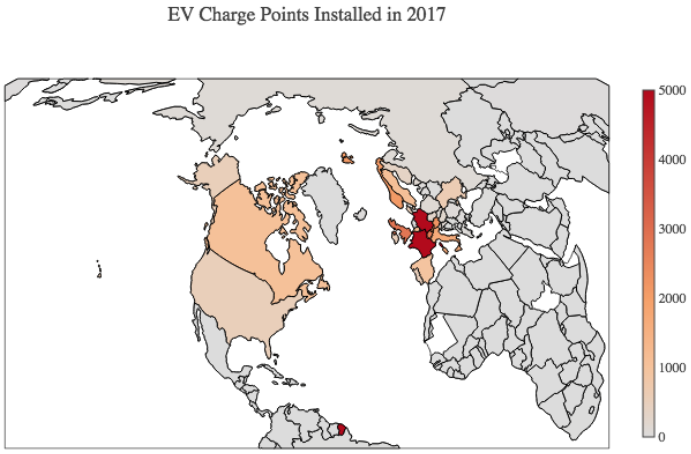
You can load it and use it in a visualization as follows.

Example

```
# read file and convert JSON into a GAP record
jsonText := ReadAll( InputTextFile( "EV Charge Points.json" ) );;
gapRecord := JsonStringToGap( jsonText );;

# ensure it's big enough to be visible:
if IsBound( gapRecord.layout ) then
    gapRecord.layout.height := 500;;
else
    gapRecord.layout := rec( height := 500 );;
fi;

# show it
CreateVisualization( rec( tool := "plotly", data := gapRecord ) );
```



3.4 Documentation for each visualization tool

This section provides links to documentation on the web for each JavaScript visualization tool supported by this package. When possible, we link directly to that portion of the tool's documentation that covers its JSON data format requirements.

3.4.1 Charting and plotting tools

- `anychart`'s JSON data format is given here:
  `https://docs.anychart.com/Working_with_Data/Data_From_JSON`
- `canvasjs`'s JSON data format is given here:
  `https://canvasjs.com/docs/charts/chart-types/`
- `chartjs`'s JSON data format is given here:
  `http://www.chartjs.org/docs/latest/getting-started/usage.html`
- `plotly`'s JSON data format is given here:
  `https://plot.ly/javascript/plotlyjs-function-reference/#plotlynewplot`

3.4.2 Graph drawing tools

- `cytoscape`'s JSON data format is given here:
  `http://js.cytoscape.org/#notation/elements-json`

3.4.3 General-purpose tools for custom visualizations

- `canvas` is a regular HTML canvas element, on which you can draw using arbitrary JavaScript included in the *`code`* parameter

- `d3` is loaded into an SVG element in the notebook's output cell, and the caller can call any D3 methods on that element thereafter, using arbitrary JavaScript included in the *`code`* parameter. It does not support creating charts from JSON input only, but its full documentation appears here: https://github.com/d3/d3/wiki
- `html` fills the output element with arbitrary HTML, which the caller should provide as a string in the `html` field of *`data`*, as documented below

3.5 Example uses of the low-level API

3.5.1 Example: AnyChart

Following the conventions in the AnyChart documentation linked to in the previous section, we could give AnyChart the following JSON to produce a pie chart.

Example

```
{
    "chart" : {
        "type" : "pie",
        "data" : [
            { "x" : "Subgroups of order 6", "value" : 1 },
            { "x" : "Subgroups of order 3", "value" : 1 },
            { "x" : "Subgroups of order 2", "value" : 3 },
            { "x" : "Subgroups of order 1", "value" : 1 }
        ]
    }
}
```

In GAP, the same data would be represented with a record, as follows.

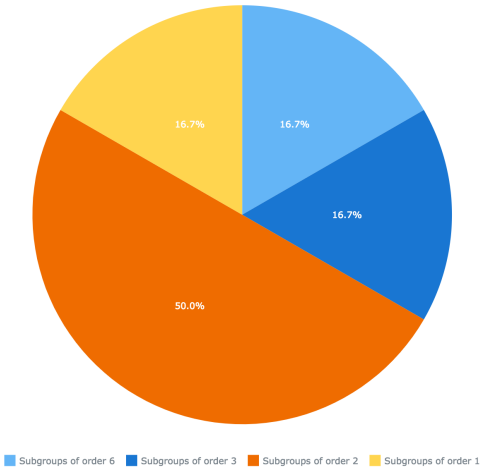
Example

```
myChartData := rec(
    chart := rec(
        type := "pie",
        data := [
            rec( x := "Subgroups of order 6", value := 1 ),
            rec( x := "Subgroups of order 3", value := 1 ),
            rec( x := "Subgroups of order 2", value := 3 ),
            rec( x := "Subgroups of order 1", value := 1 )
        ]
    )
);
```

We can pass that data directly to `CreateVisualization` (7.2.5). We wrap it in a record that specifies the tool to use and optionally other details not used in this example.

Example

```
CreateVisualization( rec( tool := "anychart", data := myChartData ) );
```



3.5.2 Example: Cytoscape

Unlike AnyChart, which is for charts and plots, Cytoscape is for graph drawing. A tiny Cytoscape graph (just $A \to B$) is represented by the following JSON.

Example

```
{
    elements : [
        { data : { id : "A" } },
        { data : { id : "B" } },
        { data : { id : "edge", source : "A", target : "B" } }
    ],
    layout : { name : "grid", rows : 1 }
}
```

Cytoscape graphs can also have style attributes not shown here. Refer to its documentation, linked to in the previous section.

Rather than copy this data directly into GAP, let's generate graph data in the same format using GAP code. Here we make a graph of the first 50 positive integers, with $n \to m$ iff $n \mid m$ (ordinary integer divisibility).

Example

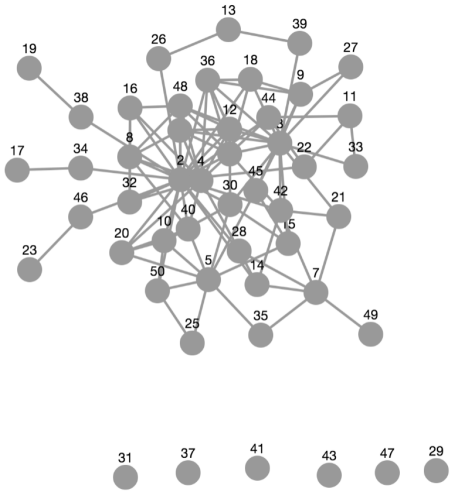
```
N := 50;
elements := [ ];
for i in [2..N] do
    Add( elements, rec( data := rec( id := String( i ) ) ) );
    for j in [2..i-1] do
        if i mod j = 0 then
            Add( elements, rec( data := rec(
                source := String( j ),
                target := String( i ) ) ) );
        fi;
    od;
od;
```

We then need to choose a layout algorithm. The Cytoscape documentation suggests that the "cose" layout works well as a force-directed layout. Here, we do choose a height (in pixels) for the result,

because Cytoscape does not automaticlly resize visualizations to fit their containing HTML element. We also set the style for each node to display its ID (which is the integer associated with it).

Example

```
CreateVisualization( rec(
    tool := "cytoscape",
    height := 600,
    data := rec(
        elements := elements, # computed in the code above
        layout := rec( name := "cose" ),
        style := [
            rec( selector := "node", style := rec( content := "data(id)" ) )
        ]
    )
) );
```



Chapter 4

Using general tools (HTML, canvas, D3)

4.1 Why these tools are present

These general tools can be used as building blocks to create other custom visualization tools. As a first example, the canvas tool installs an HTML canvas element and then lets you draw arbitrary shapes on it with JavaScript code. As a second example, some of the high-level tools this package imports were built on top of D3, a foundational visualization toolkit, which you can access directly.

First, we cover an as-yet-unmentioned feature of `CreateVisualization` (7.2.5) that lets us make use of such general tools.

4.2 Post-processing visualizations

The `CreateVisualization` (7.2.5) function takes an optional second parameter, a string of JavaScript code to be run once the visualization has been rendered. For example, if the visualization library you were using did not support adding borders around a visualization, but you wanted to add one, you could add it by writing one line of JavaScript code to run after the visualization was rendered.

Example

```
CreateVisualization(
    rec(
        # put your data here, as in previous sections
    ),
    "visualization.style.border = '5px solid black'"
)
```

This holds for any visualization tool, not just AnyChart. In the code given in the second parameter, two variables will be defined for your use: `element` refers to the HTML element inside of which the visualization was built and `visualization` refers to the HTML element of the visualization itself, as produced by the toolkit you chose. When used in a Jupyter Notebook, `element` is the output cell itself.

Now that we know that we can run arbitrary JavaScript code on a visualization once it's been produced, we can call `CreateVisualization` (7.2.5) to produce rather empty results, then fill them using our own JavaScript code. The next section explains how this could be done to create custom visualizations.

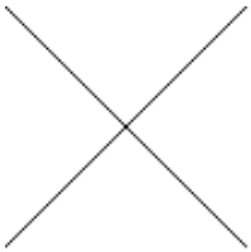
4.3 Injecting JavaScript into general tools

4.3.1 Example: Native HTML Canvas

You can create a blank canvas, then use the existing JavaScript canvas API to draw on it. This example is trivial, but more complex examples are possible.

Example

```
CreateVisualization(
    rec( tool := "canvas", height := 300 ),
    """
    // visualization is the canvas element
    var context = visualization.getContext( '2d' );
    // draw an X
    context.beginPath();
    context.moveTo( 0, 0 );
    context.lineTo( 100, 100 );
    context.moveTo( 100, 0 );
    context.lineTo( 0, 100 );
    context.stroke();
    """
);
```



4.3.2 Example: Plain HTML

This is the degenerate example of a visualization. It does not create any visualization, but lets you specify arbitrary HTML content instead. It is provided here merely as a convenient way to insert HTML into the notebook.

Example

```
CreateVisualiation(
    rec(
        tool := "html",
        data := rec(
            html := "<i>Any</i> HTML can go here.  Tables, buttons, whatever."
        )
    ),
    """
    // Here you could install event handlers on tools created above.
    // For example, if you had created a button with id="myButton":
    var button = document.getElementById( "myButton" );
    button.addEventListener( "click", function () {
        alert( "My button was clicked." );
    } );
```

```
    """
);
```

When writing such JavaScript code, note that the Jupyter Notebook has access to a useful function that this package has installed, `runGAP`. Its signature is `runGAP(stringToEvaluate,callback)` and the following code shows an example of how you could call it from JavaScript in the notebook.

Example

```
runGAP( "2^100;", function ( result, error ) {
    if ( result )
        alert( "2^100 = " + result );
    else
        alert( "GAP gave this error: " + error );
} );
```

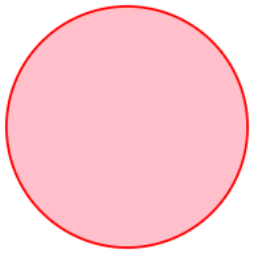
This function is not available if running this package outside of a Jupyter Notebook.

4.3.3 Example: D3

While D3 is one of the most famous and powerful JavaScript visualization libraries, it does not have a JSON interface. Consequently, we can interact with D3 only through the JavaScript code passed in the second parameter to `CreateVisualization` (7.2.5). This makes it much less convenient, but we include it in this package for those who need it.

Example

```
CreateVisualization(
    rec( tool := "d3" ),
    """
    // arbitrary JavaScript code can go here to interact with D3, like so:
    d3.select( visualization ).append( "circle" )
        .attr( "r", 50 ).attr( "cx", 55 ).attr( "cy", 55 )
        .style( "stroke", "red" ).style( "fill", "pink" );
    """
);
```



Chapter 5

Adding new visualization tools

5.1 Why you might want to do this

The visualization tools made available by this package (Plotly, D3, CanvasJS, etc.) provide many visualization options. However, you may come across a situation that they do not cover. Or a new and better tool may be invented after this package is created, and you wish to add it to the package.

There are two supported way to do this. First, for tools that you wish to be available to all users of this package, you can alter the package code itself to include the tool. (Then please create a pull request so that your work might be shared with other GAP users in a subsequent release of this package.) Second, for tools that you need for just one project or just one other package, there is support for installing such tools at runtime. This chapter documents both approaches, each in its own section. But first, we begin with the list of what you will need to have on hand before you begin, which is the same for both approaches.

5.2 What you will need

Begin by gathering the following information.

- A URL on the internet that serves the JavaScript code defining the new visualization tool you wish to add. For instance, the ChartJS library is imported from CloudFlare, at `https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.7.2/Chart.bundle.min.js`. It is best if you have this URL from a Content Delivery Network (CDN) to ensure very high availability. This URL may not be necessary in all cases. For instance, perhaps the new visualization tool you wish to install can be defined using the basic JavaScript features in all browsers, or is imported via an `iframe` rather than as a script in the page itself. If you choose to use such a URL, it will be imported using RequireJS, which expects you to omit the final `.js` suffix at the end.

- Knowledge of how to write a short JavaScript function that can embed the given tool into any given DOM `Element`. For many tools, this is just a single call to the main class's contructor or the library's initialization function. Or, if you haven't imported any library that constructs the visualization for you, then this function may be more extensive, as you construct the visualization yourself.

- While not necessary, it makes things easy if you know a function to call in your chosen library that converts JSON data into a visualization. This makes it easier for users to pass all the required data and options from GAP code, which is the typical user's preferred way of defining a visualization.

With this information available, proceed to either of the next two sections, depending on whether you intend to upgrade this package itself with a new visualization, or just install one into it at runtime.

5.3 Extending this package with a new tool

This section explains how to enhance this package itself. If you follow these instructions, you should submit a pull request to have your work added to the main repository for the package, and thus eventually included in the next release of GAP.

If instead you wish to install a new visualization at runtime for just your own use in a particular project (or in a package that depends on this one), refer to the instructions in the Section 5.4 instead.

Throughout these steps, I will assume that the name of the new tool you wish to install is `NEWTOOL`. I choose all capital letters to make it stand out, so that you can tell where you need to fill in blanks in the examples, but you should probably use lower-case letters, to match the convention used by all of the built-in tools.

1. Clone the repository for this package.
2. Enter the `lib/js/` folder in the repository.
3. Duplicate the file `viz-tool-chartjs.js` and rename it suitably for the tool you wish to import, such as `viz-tool-NEWTOOL.js`. It *must* begin with `viz-tool-`.
4. Edit that file. At the top, you will notice the installation of the CDN URL mentioned in the previous section. Replace it with the URL for your toolkit, and replace the identifier `chartjs` with `NEWTOOL`.

Example

```
window.requirejs.config( {
    paths : {
        NEWTOOL : 'https://cdn.example.com/NEWTOOL.min.js'
    }
} );
```

5. In the middle of the same file, feel free to update the comments to reflect your toolkit rather than ChartJS.
6. At the end of the same file, you will notice code that installs `chartjs` as a new function in the `window.VisualizationTools` object. Replace it with code that installs your tool instead. See the comments below for some guidance.

Example

```
window.VisualizationTools.NEWTOOL = function ( element, json, callback ) {
    // The variable "element" is the HTML element in the page into
    // which you should place your visualization.  For example, perhaps
    // your new toolkit does its work in an SVG element, so you need one:
    var result = document.createElement( 'SVG' );
    element.append( result );
```

```
    // The variable "json" is all the data, in JSON form, passed from
    // GAP to tell you how to create a visualization.  The data format
    // convention is up to you to explain and document with your new
    // tool.  Two attributes in particular are important here, "width"
    // and "height" -- if you ignore everything else, at least respect
    // those in whatever way makes sense for your visualization.  Here
    // is an example for an SVG:
    if ( json.width ) result.width = json.width;
    if ( json.height ) result.width = json.height;
    // Then use RequireJS to import your toolkit (which will use the CDN
    // URL you registered above) and use it to fill the element with the
    // desired visualization.  You may or may not need to modify "json"
    // before passing it to your toolkit; this is up to the conventions
    // you choose to establish.
    require( [ 'NEWTOOL' ], function ( NEWTOOL ) {
        // Use your library to set up a visualization.  Example:
        var viz = NEWTOOL.setUpVisualizationInElement( result );
        // Tell your library what to draw.  Example:
        viz.buildVisualizationFromJSON( json );
        // Call the callback when you're done.  Pass the element you were
        // given, plus the visualization you created.
        callback( element, result );
    } );
};
```

7. Optionally, in the `lib/js/` folder, run the `minify-all-scripts.sh` script, which compresses your JavaScript code to save on data transfer, memory allocation, and parsing time. Rerun that script each time you change your file as well.

8. You should now be able to use your new visualization tool in GAP. Verify that your changes worked, and debug as necessary. If you are testing in a Jupyter Notebook, you may be able to notice the change only if you refresh in your browser the page containing notebook and also restart the GAP kernel in that same page. Then try code like the following to test what you've done.

Example

```
CreateVisualization( rec(
    tool := "NEWTOOL",
    # any other data you need goes here as a GAP record,
    # which the GAP json package will convert into JSON
) );
```

At this point, you have added support in `CreateVisualization` (7.2.5) for the new tool but have not extended that support to include the high-level functions `Plot` (7.1.1) or `PlotGraph` (7.1.3). If possible, you should add that support as well, by following the steps below.

1. Read the documentation for either `ConvertDataSeriesForTool` (7.1.2) or `ConvertGraphForTool` (7.1.4), depending on whether the new tool you have installed supports plots or graphs. If it supports both, read both. That documentation explains the new function you would need to install in one or both of those records in order to convert the type of data users provide to `Plot` (7.1.1) or `PlotGraph` (7.1.3) into the type of data used by `CreateVisualization` (7.2.5).

2. Edit the `main.gi` file in this package. Find the section in which new elements are added to the `ConvertDataSeriesForTool` (7.1.2) or `ConvertGraphForTool` (7.1.4) records. Add a new section of code that installs a new field for your tool. It will look like one of the following two blocks (or both if your tool supports both types of visualization).

Example

```
ConvertDataSeriesForTool.NEWTOOL := function ( series )
  local result;
  # Write the code here that builds the components of the
  # GAP record you need, stored in result.
  # You can leverage series.x, series.y, and series.options.
  return result;
end;
ConvertGraphForTool.NEWTOOL := function ( graph )
  local result;
  # Write the code here that builds the components of the
  # GAP record you need, stored in result.
  # You can leverage graph.vertices, graph.edges, and graph.options.
  return result;
end;
```

3. Test your work by loading the updated package into GAP and making a call to `Plot` (7.1.1) or `PlotGraph` (7.1.3) that specifically requests the use of your newly-supported visualization tool.

Example

```
# for plots:
Plot( x -> x^2, rec( tool := "NEWTOOL" ) );
# or for graphs:
PlotGraph( RandomMat( 5, 5 ), rec( tool := "NEWTOOL" ) );
```

   Verify that it produces the desired results.

4. Once your changes work, commit them to the repository and submit a pull request back to the original repository, to have your work included in the default distribution.

A complete and working (but silly) example follows. It is a tiny enough visualization tool that it cannot support either plotting data nor drawing graphs, so we don't have to install high-level API support. This portion would go in `lib/js/viz-tool-color.js`:

Example

```
// No need to import any library from a CDN for this little example.
window.VisualizationTools.color = function ( element, json, callback ) {
    // just writes json.text in json.color, that's all
    var span = document.createElement( 'span' );
    span.textContent = json.text;
    span.style.color = json.color;
    callback( element, span );
};
```

This is an example usage of that simple tool from GAP in a Jupyter notebook:

Example

```
CreateVisualization( rec(
    tool := "color",
    text := "Happy St. Patrick's Day.",
```

```
    color := "green"
) );
```

5.4 Installing a new tool at runtime

This section explains how to add a new visualization tool to this package at runtime, by calling functions built into the package. This is most useful when the visualization tool you wish to install is useful in only a narrow context, such as one of your projects or packages.

If you have a visualization tool that might be of use to anyone who uses this package, consider instead adding it to the package itself and submitting a pull request to have it included in the next release. The previous section explains how to do that.

To install a new visualization tool at runtime, you have two methods available. You can either provide all the JavaScript code yourself or you can provide the necessary ingredients that will be automatically filled into a pre-existing JavaScript code template. We will examine both methods in this section.

The previous section thoroughly documents the two types of code that are likely to show up in the definition of a new tool: the installation into RequireJS of the tool's CDN URL and the installation into `window.VisualizationTool` of a function that uses that tool to create a visualization from a given JSON object.

If you have all of this JavaScript code already stored in a single GAP string (or in a file that you can load into a string), call it `S`, then you can install it into this package with a single function call, like so:

Example

```
InstallVisualizationTool( "TOOL_NAME_HERE", S );
```

Here is a trivial working example. It is sufficiently small that it does not install any new JavaScript libraries into RequireJS.

Example

```
# GAP code to install a new visualization tool:
InstallVisualizationTool( "smallExample",
"""
window.VisualizationTool.smallExample =
function ( element, json, callback ) {
    element.innerHTML = '<span color=red>' + json.text + '</span>';
    callback( element, element.childNodes[0] );
}
"""
) );

# GAP code to use that new visualization tool:
CreateVisualization( rec(
    tool := "smallExample",
    text := "This text will show up red."
) );
```

Because the assignment of a function to create visualizations from JSON is the essential component of installing a new visualization, we have made that step easier by creating a template into which you can just fill in the function body. So the above call to `InstallVisualizationTool` (7.2.3) is equivalent to the following call to `InstallVisualizationToolFromTemplate` (7.2.4).

Example

```
InstallVisualizationToolFromTemplate( "smallExample",
"""
    element.innerHTML = '<span color=red>' + json.text + '</span>';
    callback( element, element.childNodes[0] );
"""
) );
```

If you provide a third parametr to `InstallVisualizationToolFromTemplate` (7.2.4), it is treated as the CDN URL for an external library, and code is automatically inserted that installs that external library into RequireJS and wraps the tool's function body in a `require` call. For instance, the CanvasJS library (which is built into this package) could have been installed with code like the following.

Example

```
InstallVisualizationToolFromTemplate( "canvasjs",
"""
    ( new window.CanvasJS.Chart( element, json.data ) ).render();
    window.resizeToShowContents( element );
    callback( element, element.childNodes[0] );
""",
"https://cdnjs.cloudflare.com/ajax/libs/canvasjs/1.7.0/canvasjs.min.js"
) );
```

While RequireJS demands that you omit the `.js` suffix from such an URL, `InstallVisualizationToolFromTemplate` (7.2.4) will automatically remove it for you if you forget to remove it.

After using either of those two methods, if the new visualization tool is capable of drawing either plots or graphs, and you wish to expose it to the high-level API, you should follow the steps for doing so documented in the second half of Section 5.3.

Chapter 6

Limitations

When this package is being used in a Jupyter Notebook, it has the following limitations.

- If this package is used in `PlotDisplayMethod_Jupyter` mode in a Jupyter notebook, and visualizations are created by this package, then the notebook is saved and later reloaded, the visualizations will not persist. They will be replaced by an error message instructing the user to re-run the cell that created the visualization. You can get around this by setting `PlotDisplayMethod := PlotDisplayMethod_JupyterSimple`, but this increases the size of your notebook by embedding all the JavaScript needed by the visualizations in the notebook itself. Note that `PlotDisplayMethod_Jupyter` is the default mode in the notebook.
- The `nbconvert` tool, which converts `.ipynb` files into other formats, will not include the visualizations, because `nbconvert` is not a browser that can evaluate the JavaScript code that generates the visualizations.
- When using the `PlotDisplayMethod_Jupyter` mode, most visualizations load a JavaScript library from a CDN, which thus requires a working Internet connection to function.

When it is being used from the command line, it has the following limitations.

- The JavaScript function `runGAP` introduced in Section 4.3 is not available. That function depends upon the ability to ask the Jupyter Kernel to run GAP code, and thus when there is no Jupyter Kernel, that function cannot work.
- Each new call to `Plot` (7.1.1), `PlotGraph` (7.1.3), or `CreateVisualization` (7.2.5) will be stored in a new temporary file on the user's filesystem and thus shown in a new tab or window in the user's browser. That is, one does not iteratively improve a single visualization, but is forced to open a new window or tab for each call.

Chapter 7

Function reference

7.1 High-Level Public API

7.1.1 Plot

▷ `Plot(`*various*`)` (function)

Returns: one of two things, documented below

If evaluated in a Jupyter Notebook, the result of this function, when rendered by that notebook, will run JavaScript code that generates and shows a plot in the output cell, which could be any of a wide variety of data visualizations, including bar charts, pie charts, scatterplots, etc. (To draw a vertex-and-edge graph, see `PlotGraph` (7.1.3) instead.)

If evaluated outside of a Jupyter Notebook, the result of this function is the name of a temporary file stored on disk in which HTML code for such a visualization has been written, and on which GAP has already invoked the user's default web browser. The user should see the visualization appear in the browser immediately before the return value is shown.

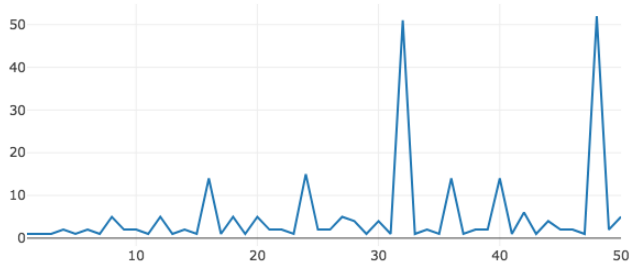
This function can take data in a wide variety of input formats. Here is the current list of acceptable formats:

- If `X` is a list of x values and `Y` is a list of y values then `Plot(X,Y)` plots them as ordered pairs.
- If `X` is a list of x values and `f` is a GAP function that can be applied to each x to yield a corresponding y, then `Plot(X,f)` computes those corresponding y values and plots everything as ordered pairs.
- If `P` is a list of (x,y) pairs then `Plot(P)` plots those ordered pairs.
- If `Y` is a list of y values then `Plot(Y)` assumes the corresponding x values are 1, 2, 3, and so on up to the length of `Y`. It then plots the corresponding set of ordered pairs.
- If `f` is a GAP function then `Plot(f)` assumes that `f` requiers integer inputs and evaluates it on a small domain (1 through 5) of x values and plots the resulting (x,y) pairs.
- In any of the cases above, a new, last argument may be added that is a GAP record (call it `R`) containing options for how to draw the plot, including the plot type, title, axes options, and more. Thus the forms `Plot(X,Y,R)`, `Plot(X,f,R)`, `Plot(P,R)`, `Plot(Y,R)`, and `Plot(f,R)` are all acceptable. (For details, see `ConvertDataSeriesForTool` (7.1.2).)

- If `A1` is a list of arguments fitting any of the cases documented above (such as `[X,f]`) and `A2` is as well, and so on through `An`, then `Plot(A1,A2,...,An)` creates a combination plot with all of the data from each of the arguments treated as a separate data series. If the arguments contain conflicting plot options (e.g., the first requests a line plot and the second a bar chart) then the earliest option specified takes precedence.

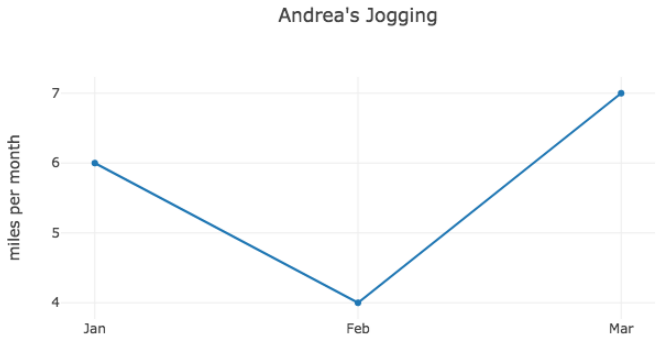
Example

```
# Plot the number of small groups of order n, from n=1 to n=50:
Plot( [1..50], NrSmallGroups );
```



Example

```
# Plot how much Andrea has been jogging lately:
Plot( ["Jan","Feb","Mar"], [46,59,61],
      rec( title := "Andrea's Jogging", yaxis := "miles per month" ) );
```



7.1.2 ConvertDataSeriesForTool

▷ `ConvertDataSeriesForTool` (global variable)

The JupyterViz Package has a high-level API and a low-level API. The high-level API involves functions like `Plot`, which take data in a variety of convenient formats, and produce visualizations from them. The low-level API can be used to pass JSON data structures to JavaScript visualization

tools in their own native formats for rendering. The high-level API is built on the low-level API, using key functions to do the conversion.

The conversion functions for plots are stored in a global dictionary in this variable. It is a GAP record mapping visualization tool names (such as plotly, etc., a complete list of which appears in Section 1.1) to conversion functions. Only those tools that support plotting data in the form of (x,y) pairs should be included. (For example, tools that specialize in drawing vertex-and-edge graphs are not relevant here.)

Each conversion function must behave as follows. It expects its input object to be a single data series, which will be a GAP record with three fields:

- `x` - a list of x values for the plot
- `y` - the corresponding list of y values for the same plot
- `options` - another (inner) GAP record containing any of the options documented in Section 2.2.

The output of the conversion function should be a GAP record amenable to conversion (using `GapToJsonString` from the json package) into JSON. The format of the JSON is governed entirely by the tool that will be used to visualize it, each of which has a different data format it expects.

Those who wish to install new visualization tools for plots (as discussed in Chapter 5) will want to install a new function in this object corresponding to the new tool. If you plan to do so, consider the source code for the existing conversion functions, which makes use of two useful convenince methods, `JUPVIZFetchWithDefault` (7.3.12) and `JUPVIZFetchIfPresent` (7.3.13). Following those examples will help keep your code consistent with existing code and as concise as possible.

7.1.3 PlotGraph

▷ `PlotGraph(`*various*`)` (function)

Returns: one of two things, documented below

If evaluated in a Jupyter Notebook, the result of this function, when rendered by that notebook, will run JavaScript code that generates and shows a graph in the output cell, not in the sense of coordinate axes, but in the sense of vertices and edges. (To graph a function or data set on coordinate axes, use `Plot` (7.1.1) instead.)

If evaluated outside of a Jupyter Notebook, the result of this function is the name of a temporary file stored on disk in which HTML code for such a visualization has been written, and on which GAP has already invoked the user's default web browser. The user should see the visualization appear in the browser immediately before the return value is shown.

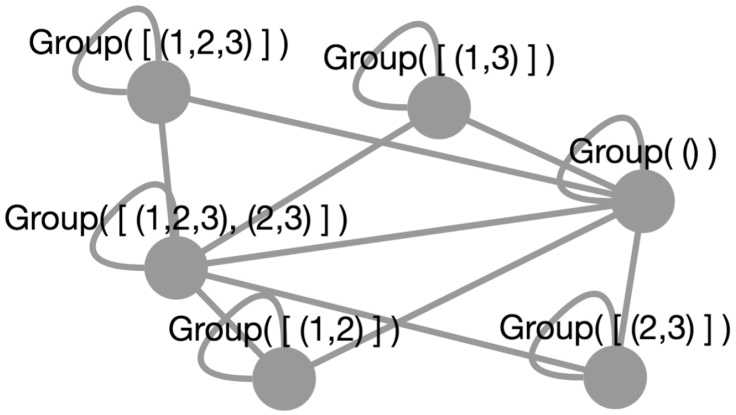
This function can take data in a wide variety of input formats. Here is the current list of acceptable formats:

- If `V` is a list and `E` is a list of pairs of items from `V` then `PlotGraph(V,E)` treats them as vertex and edge sets, respectively.
- If `V` is a list and `R` is a GAP function then `PlotGraph(V,R)` treats `V` as the vertex set and calls `R(v1,v2)` for every pair of vertices (in both orders) to test whether there is an edge between them. It exepcts `R` to return boolean values.
- If `E` is a list of pairs then `PlotGraph(E)` treats `E` as a list of edges, inferring the vertex set to be any vertex mentioned in any of the edges.

- If M is a square matrix then PlotGraph(M) treats M as an adjacency matrix whose vertices are the integers 1 through n (the height of the matrix) and where two vertices are connected by an edge if and only if that matrix entry is positive.
- In any of the cases above, a new, last argument may be added that is a GAP record containing options for how to draw the graph, such as the tool to use. For details on the supported options, see ConvertGraphForTool (7.1.4).

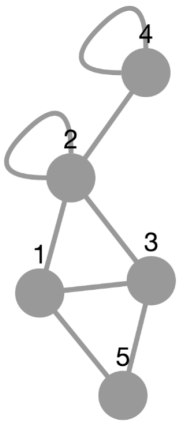
Example

```
# Plot the subgroup lattice for a small group:
G := Group((1,2),(2,3));
PlotGraph( AllSubgroups(G), IsSubgroup );
```



Example

```
# Plot a random graph on 5 vertices:
# (The results change from one run to the next, of course.)
PlotGraph( RandomMat(5,5) );
```



7.1.4 ConvertGraphForTool

▷ ConvertGraphForTool (global variable)

The JupyterViz Package has a high-level API and a low-level API. The high-level API involves functions like `PlotGraph`, which take data in a variety of convenient formats, and produce visualizations from them. The low-level API can be used to pass JSON data structures to JavaScript visualization tools in their own native formats for rendering. The high-level API is built on the low-level API, using key functions to do the conversion.

The conversion functions for graphs are stored in a global dictionary in this variable. It is a GAP record mapping visualization tool names (such as cytoscape, a complete list of which appears in Section 1.1) to conversion functions. Only those tools that support graphing vertex and edge sets should be included. (For example, tools that specialize in drawing plots of data stored as (x,y) pairs are not relevant here.)

Each conversion function must behave as follows. It expects its input object to be a single graph, which will be a GAP record with three fields:

- `vertices` - a list of vertex names for the graph. These can be any GAP data structure and they will be converted to strings with `PrintString`. The one exception is that you can give each vertex a position by making it a record with three entries: `name`, `x`, and `y`. In this way, you can manually lay out a graph.
- `edges` - a list of pairs from the `vertices` list, each of which represents an edge
- `options` - a GAP record containing any of the options documented in Section 2.4.

The output of the conversion function should be a GAP record amenable to conversion (using `GapToJsonString` from the json package) into JSON. The format of the JSON is governed entirely by the tool that will be used to visualize it, each of which has a different data format it expects.

Those who wish to install new visualization tools for graphs (as discussed in Chapter 5) will want to install a new function in this object corresponding to the new tool. If you plan to do so, consider the source code for the existing conversion functions, which makes use of two useful convenince methods, `JUPVIZFetchWithDefault` (7.3.12) and `JUPVIZFetchIfPresent` (7.3.13). Following those examples will help keep your code consistent with existing code and as concise as possible.

7.1.5 PlotDisplayMethod

▷ `PlotDisplayMethod` (global variable)

The JupyterViz Package can display visualizations in three different ways, and this global variable is used to switch among those ways.

Example

```
PlotDisplayMethod := PlotDisplayMethod_HTML;
```

Users of this package almost never need to alter the value of this variable because a sensible default is chosen at package loading time. If the JupyterViz Package is loaded after the JupyterKernel Package, it notices the presence of that package and leverage its tools to set up support for plotting in a Jupyter environment. Furthermore, it will initialize `PlotDisplayMethod` to `PlotDisplayMethod_Jupyter` (7.1.6), which is probably what the user wants. Note that if one calls `LoadPackage("JupyterViz");` from a cell in a Jupyter notebook, this is the case that applies, because clearly in such a case, the JupyterKernel Package was already loaded.

If the JupyterViz Package is loaded without the JupyterKernel Package already loaded, then it will initialize `PlotDisplayMethod` to `PlotDisplayMethod_HTML` (7.1.8), which is what

the user probably wants if using GAP from a terminal, for example. You may later assign `PlotDisplayMethod` to another value, but doing so has little purpose from the REPL. You would need to first load the JupyterKernel Package, and even then, all that would be produced by this package would be data structures that would, if evaluated in a Jupyter notebook, produce visualizations.

7.1.6 PlotDisplayMethod_Jupyter

▷ `PlotDisplayMethod_Jupyter` (global variable)

This global constant can be assigned to the global variable `PlotDisplayMethod` (7.1.5) as documented above. Doing so produces the following results.

- Functions such as `Plot` (7.1.1), `PlotGraph` (7.1.3), and `CreateVisualization` (7.2.5) will return objects of type `JupyterRenderable`, which is defined in the JupyterKernel Package.
- Such objects, when rendered in a Jupyter cell, will run a block of JavaScript contained within them, which will create the desired visualization.
- Such scripts tend to request additional information from GAP as they are running, by using calls to the JavaScript function `Jupyter.kernel.execute` defined in the notebook. Such calls are typically to fetch JavaScript libraries needed to create the requested visualization.
- Visualizations produced this way will not be visible if one later closes and then reopens the Jupyter notebook in which they are stored. To see the visualizations again, one must re-evaluate the cells that created them, so that the required libraries are re-fetched from the GAP Jupyter kernel.

7.1.7 PlotDisplayMethod_JupyterSimple

▷ `PlotDisplayMethod_JupyterSimple` (global variable)

This global constant can be assigned to the global variable `PlotDisplayMethod` (7.1.5) as documented above. Doing so produces the following results.

- Functions such as `Plot` (7.1.1), `PlotGraph` (7.1.3), and `CreateVisualization` (7.2.5) will return objects of type `JupyterRenderable`, which is defined in the JupyterKernel Package.
- Such objects, when rendered in a Jupyter cell, will run a block of JavaScript contained within them, which will create the desired visualization.
- Such scripts will be entirely self-contained, and thus will not make any additional requests from the GAP Jupyter kernel. This makes such objects larger because they must contain all the required JavaScript visualization libraries, rather than being able to request them as needed later.
- Visualizations produced this way will be visible even if one later closes and then reopens the Jupyter notebook in which they are stored, because all the code needed to create them is included in the output cell itself, and is re-run upon re-opening the notebook.

7.1.8 PlotDisplayMethod_HTML

▷ `PlotDisplayMethod_HTML` (global variable)

This global constant can be assigned to the global variable `PlotDisplayMethod` (7.1.5) as documented above. Doing so produces the following results.

- Functions such as `Plot` (7.1.1), `PlotGraph` (7.1.3), and `CreateVisualization` (7.2.5) will return no value, but will instead store HTML (and JavaScript) code for the visualization in a temporary file on the filesystem, then launch the operating system's default web browser to view that file.
- Such files are entirely self-contained, and require no GAP session to be running to continue viewing them. They can be saved anywhere the user likes for later viewing, printing, or sharing without GAP.
- Visualizations produced this way will not be visible if one later closes and then reopens the Jupyter notebook in which they are stored. To see the visualizations again, one must re-evaluate the cells that created them, so that the required libraries are re-fetched from the GAP Jupyter kernel.

7.2 Low-Level Public API

7.2.1 RunJavaScript

▷ `RunJavaScript(`*`script[, returnHTML]`*`)` (function)

Returns: one of two things, documented below

If run in a Jupyter Notebook, this function returns an object that, when rendered by that notebook, will run the JavaScript code given in *`script`*.

If run outside of a Jupyter Notebook, this function creates an HTML page containing the given *`script`*, an HTML element on which that script can act, and the RequireJS library for importing other script tools. It then opens the page in the system default web browser (thus running the script) and returns the path to the temporary file in which the script is stored.

In this second case only, the optional second parameter (which defaults to false) can be set to true if the caller does not wish the function to open a web browser, but just wants the HTML content that would have been displayed in such a browser returned as a string instead.

When the given code is run, the varible `element` will be defined in its environment, and will contain either the output element in the Jupyter notebook corresponding to the code that was just evaluated or, in the case outside of Jupyter, the HTML element mentioned above. The script is free to write to that element in both cases.

7.2.2 LoadJavaScriptFile

▷ `LoadJavaScriptFile(`*`filename`*`)` (function)

Returns: the string contents of the file whose name is given

Interprets the given *`filename`* relative to the `lib/js/` path in the JupyterViz package's installation folder, because that is where this package stores its JavaScript libraries. A `.js` extension will be added to *`filename`* iff needed. A `.min.js` extension will be added iff such a file exists, to prioritize minified versions of files.

If the file has been loaded before in this GAP session, it will not be reloaded, but will be returned from a cache in memory, for efficiency.

If no such file exists, returns `fail` and caches nothing.

7.2.3 InstallVisualizationTool

▷ `InstallVisualizationTool(`*`toolName, script`*`)` (function)

Returns: boolean indicating success (true) or failure (false)

This function permits extending, at runtime, the set of JavaScript visualization tools beyond those that are built into the JupyterViz package.

The first argument must be the name of the visualization tool (a string, which you will later use in the `tool` field when calling `CreateVisualization` (7.2.5)). The second must be a string of JavaScript code that installs into `window.VisualizationTools.TOOL_NAME_HERE` the function for creating visualizations using that tool. It can also define other helper functions or make calls to `window.requirejs.config`. For examples of how to write such JavaScript code, see the chapter on extending this package in its manual.

This function returns false and does nothing if a tool of that name has already been installed. Otherwise, it installs the tool and returns true.

There is also a convenience method that calls this one on your behalf; see `InstallVisualizationToolFromTemplate` (7.2.4).

7.2.4 InstallVisualizationToolFromTemplate

▷ `InstallVisualizationToolFromTemplate(`*`toolName, functionBody[, CDNURL]`*`)` (function)

Returns: boolean indicating success (true) or failure (false)

This function is a convenience function that makes it easier to use `InstallVisualizationTool` (7.2.3); see the documentation for that function, then read on below for how this function makes it easier.

Most visualization tools do two things: First, they install a CDN URL into `window.requirejs.config` for some external JavaScript library that must be loaded in the client to support the given type of visualization. Second, they install a function as `window.VisualizationTools.TOOL_NAME_HERE` accepting parameters `element`, `json`, and `callback`, and building the desired visualization inside the given DOM element. Such code often begins with a call to `require(['...'],function(library}{/*...*/}))`, but not always.

This function will write for you the boiler plate code for calling `window.requirejs.config` and the declaration and installation of a function into `window.VisualizationTools.TOOL_NAME_HERE`. You provide the function body and optionally the CDN URL. (If you provide no CDN URL, then no external CDN will be installed into `requirejs`.)

7.2.5 CreateVisualization

▷ `CreateVisualization(`*`data[, code]`*`)` (function)

Returns: one of two things, documented below

If run in a Jupyter Notebook, this function returns an object that, when rendered by that notebook, will produce the visualization specified by *`data`* in the corresponding output cell, and will also run any given *`code`* on that visualization.

If run outside of a Jupyter Notebook, this function creates an HTML page containing the visualization specified by *data* and then opens the page in the system default web browser. It will also run any given *code* as soon as the page opens. The *data* must be a record that will be converted to JSON using GAP's json package.

The second argument is optional, a string containing JavaScript *code* to run once the visualization has been created. When that code is run, the variables `element` and `visualization` will be in its environment, the former holding the output element in the notebook containing the visualization, and the latter holding the visualization element itself.

The *data* should have the following attributes.

- `tool` (required) - the name of the visualization tool to use. Currently supported tools are listed in Section 1.2 and links to their documentation are given in Section 3.4.
- `data` (required) - subobject containing all options specific to the content of the visualization, often passed intact to the external JavaScript visualization library. You should prepare this data in the format required by the library specified in the `tool` field, following the documentation for that library, linked to in Section 3.4.
- `width` (optional) - width to set on the output element being created
- `height` (optional) - similar, but height

Example

```
CreateVisualization( rec(
    tool := "html",
    data := rec( html := "I am <i>SO</i> excited about this." )
), "console.log( 'Visualization created.' );" );
```

7.3 Internal methods

Using the convention common to GAP packages, we prefix all methods not intended for public use with a sequence of characters that indicate our particular package. In this case, we use the `JUPVIZ` prefix. This is a sort of "poor man's namespacing."

None of these methods should need to be called by a client of this package. We provide this documentation here for completeness, not out of necessity.

7.3.1 JUPVIZAbsoluteJavaScriptFilename

▷ `JUPVIZAbsoluteJavaScriptFilename(`*`filename`*`)` (function)

Returns: a JavaScript filename to an absolute path in the package dir

Given a relative *filename*, convert it into an absolute filename by prepending the path to the `lib/js/` folder within the JupyterViz package's installation folder. This is used by functions that need to find JavaScript files stored there.

A `.js` extension is appended if none is included in the given *filename*.

7.3.2 JUPVIZLoadedJavaScriptCache

▷ `JUPVIZLoadedJavaScriptCache` (global variable)

A cache of the contents of any JavaScript files that have been loaded from this package's folder. The existence of this cache means needing to go to the filesystem for these files only once per GAP session. This cache is used by `LoadJavaScriptFile` (7.2.2).

7.3.3 JUPVIZFillInJavaScriptTemplate

▷ `JUPVIZFillInJavaScriptTemplate(`*`filename, dictionary`*`)` (function)

Returns: a string containing the contents of the given template file, filled in using the given dictionary

A template file is one containing identifiers that begin with a dollar sign (`$`). For example, `$one` and `$two` are both identifiers. One "fills in" the template by replacing such identifiers with whatever text the caller associates with them.

This function loads the file specified by *`filename`* by passing that argument directly to `LoadJavaScriptFile` (7.2.2). If no such file exists, returns `fail`. Otherwise, it proceed as follows.

For each key-value pair in the given *`dictionary`*, prefix a `$` onto the key, suffix a newline character onto the value, and then replace all occurrences of the new key with the new value. The resulting string is the result.

The newline character is included so that if any of the values in the *`dictionary`* contains single-line JavaScript comment characters (`//`) then they will not inadvertently affect later code in the template.

7.3.4 JUPVIZRunJavaScriptFromTemplate

▷ `JUPVIZRunJavaScriptFromTemplate(`*`filename, dictionary[, returnHTML]`*`)` (function)

Returns: the composition of `RunJavaScript` (7.2.1) with `JUPVIZFillInJavaScriptTemplate` (7.3.3)

This function is quite simple, and is just a convenience function. The optional third argument is passed on to RunJavaScript internally.

7.3.5 JUPVIZRunJavaScriptUsingRunGAP

▷ `JUPVIZRunJavaScriptUsingRunGAP(`*`jsCode[, returnHTML]`*`)` (function)

Returns: an object that, if rendered in a Jupyter notebook, will run *`jsCode`* as JavaScript after `runGAP` has been defined

There is a JavaScript function called `runGAP`, defined in the `using-runGAP.js` file distributed with this package. That function makes it easy to make callbacks from JavaScript in a Jupyter notebook to the GAP kernel underneath that notebook. This GAP function runs the given *`jsCode`* in the notebook, but only after ensuring that `runGAP` is defined globally in that notebook, so that *`jsCode`* can call `runGAP` as needed.

The optional third argument is passed on to RunJavaScript internally.

An example use, from JavaScript, of the `runGAP` function appears at the end of Section 4.3.

7.3.6 JUPVIZRunJavaScriptUsingLibraries

▷ `JUPVIZRunJavaScriptUsingLibraries(`*`libraries, jsCode[, returnHTML]`*`)` (function)

Returns: one of two things, documented below

If run in a Jupyter Notebook, this function returns an object that, when rendered by that notebook, will run *jsCode* as JavaScript after all *libraries* have been loaded (which typically happens asynchronously).

If run outside of a Jupyter Notebook, this function loads all the code for the given *libraries* from disk and concatenates them (with checks to be sure no library is loaded twice) followed by *jsCode*. It then calls RunJavaScript (7.2.1) on the result, to form a web page and display it to the user.

There are a set of JavaScript libraries stored in the lib/js/ subfolder of this package's installation folder. Neither the Jupyter notebook nor the temporary HTML files created from the command line know, by default, about any of those libraries. Thus this function is necessary so that *jsCode* can assume the existence of the tools it needs to do its job.

If the first parameter is given as a string instead of a list of strings, it is treated as a list of just one string.

The optional third argument is passed on to RunJavaScript internally.

Example

```
JUPVIZRunJavaScriptUsingLibraries( [ "mylib.js" ],
    "alert( 'My Lib defines foo to be: ' + window.foo );" );
# Equivalently:
JUPVIZRunJavaScriptUsingLibraries( "mylib.js",
    "alert( 'My Lib defines foo to be: ' + window.foo );" );
```

7.3.7 JUPVIZMakePlotDataSeries

▷ JUPVIZMakePlotDataSeries(*series*) (function)

Returns: a record with the appropriate fields (x, y, options) that can be passed to one of the functions in ConvertDataSeriesForTool (7.1.2)

This function is called by Plot (7.1.1) to convert any of the wide variety of inputs that Plot (7.1.1) might receive into a single internal format. Then that internal format can be converted to the JSON format needed by any of the visualization tools supported by this package.

See the documentation for ConvertDataSeriesForTool (7.1.2) for more information on how that latter conversion takes place, and the format it expects.

7.3.8 JUPVIZMakePlotGraphRecord

▷ JUPVIZMakePlotGraphRecord(*various*) (function)

Returns: a record with the appropriate fields (vertices, edges, options) that can be passed to one of the functions in ConvertGraphForTool (7.1.4)

This function is called by PlotGraph (7.1.3) to convert any of the wide variety of inputs that PlotGraph (7.1.3) might receive into a single internal format. Then that internal format can be converted to the JSON format needed by any of the visualization tools supported by this package.

See the documentation for ConvertGraphForTool (7.1.4) for more information on how that latter conversion takes place, and the format it expects.

7.3.9 JUPVIZPlotDataSeriesList

▷ JUPVIZPlotDataSeriesList(*series1, series2, series3...*) (function)

Returns: a JupyterRenderable object ready to be displayed in the Jupyter Notebook

Because the Plot (7.1.1) function can take a single data series or many data series as input, it detects which it received, then passes the resulting data series (as an array containing one or more series) to this function for collecting into a single plot.

It is not expected that clients of this package will need to call this internal function.

7.3.10 JUPVIZRecordKeychainLookup

▷ JUPVIZRecordKeychainLookup(*record, keychain, default*) (function)

Returns: the result of looking up the chain of keys in the given record

In nested records, such as myRec:=rec(a:=rec(b:=5)), it is common to write code such as myRec.a.b to access the internal values. However when records are passed as parameters, and may not contain every key (as in the case when some default values should be filled in automatically), code like myRec.a.b could cause an error. Thus we wish to first check before indexing a record that the key we're looking up exists. If not, we wish to return the value given as the default instead.

This function accepts a record (which may have other records inside it as values), an array of strings that describe a chain of keys to follow inward (["a","b"] in the example just given), and a default value to return if any of the keys do not exist.

It is not expected that clients of this package will need to call this internal function. It is used primarily to implement the JUPVIZFetchWithDefault (7.3.12) function, which is useful to those who wish to extend the ConvertDataSeriesForTool (7.1.2) and ConvertGraphForTool (7.1.4) objects.

Example

```
myRec := rec( height := 50, width := 50, title := rec(
  text := "GAP", fontSize := 20
) );
JUPVIZRecordKeychainLookup( myRec, [ "height" ], 10 );                # = 50
JUPVIZRecordKeychainLookup( myRec, [ "width" ], 10 );                 # = 50
JUPVIZRecordKeychainLookup( myRec, [ "depth" ], 10 );                 # = 10
JUPVIZRecordKeychainLookup( myRec, [ "title", "text" ], "Title" );    # = "GAP"
JUPVIZRecordKeychainLookup( myRec, [ "title", "color" ], "black" );   # = "black"
JUPVIZRecordKeychainLookup( myRec, [ "one", "two", "three" ], fail ); # = fail
```

7.3.11 JUPVIZRecordsKeychainLookup

▷ JUPVIZRecordsKeychainLookup(*records, keychain, default*) (function)

Returns: the result of looking up the chain of keys in each of the given records until a lookup succeeds

This function is extremely similar to JUPVIZRecordKeychainLookup (7.3.10) with the following difference: The first parameter is a list of records, and JUPVIZRecordKeychainLookup (7.3.10) is called on each in succession with the same keychain. If any of the lookups succeeds, then its value is returned and no further searching through the list is done. If all of the lookups fail, the default is returned.

It is not expected that clients of this package will need to call this internal function. It is used primarily to implement the JUPVIZFetchWithDefault (7.3.12) function, which is useful to those who wish to extend the ConvertDataSeriesForTool (7.1.2) and ConvertGraphForTool (7.1.4) objects.

Example

```
myRecs := [
  rec( height := 50, width := 50, title := rec(
    text := "GAP", fontSize := 20
  ) ),
  rec( width := 10, depth := 10, color := "blue" )
];
JUPVIZRecordsKeychainLookup( myRecs, [ "height" ], 0 );              # = 50
JUPVIZRecordsKeychainLookup( myRecs, [ "width" ], 0 );               # = 50
JUPVIZRecordsKeychainLookup( myRecs, [ "depth" ], 0 );               # = 10
JUPVIZRecordsKeychainLookup( myRecs, [ "title", "text" ], "Title" ); # = "GAP"
JUPVIZRecordsKeychainLookup( myRecs, [ "color" ], "" );              # = "blue"
JUPVIZRecordsKeychainLookup( myRecs, [ "flavor" ], fail );           # = fail
```

7.3.12 JUPVIZFetchWithDefault

▷ `JUPVIZFetchWithDefault(`*`record, others, chain, default, action`*`)` (function)

Returns: nothing

This function is designed to make it easier to write new entries in the `ConvertDataSeriesForTool` (7.1.2) and `ConvertGraphForTool` (7.1.4) functions. Those functions are often processing a list of records (here called `others`) sometimes with one record the most important one (here called `record`) and looking up a `chain` of keys (using `default` just as in `JUPVIZRecordKeychainLookup` (7.3.10)) and then taking some `action` based on the result. This function just allows all of that to be done with a single call.

Specifically, it considers the array of records formed by `Concatenation([record],others)` and calls `JUPVIZRecordsKeychainLookup` (7.3.11) on it with the given `chain` and `default`. (If the `chain` is a string, it is automatically converted to a length-one list with the string inside.) Whatever the result, the function `action` is called on it, even if it is the default.

Example

```
# Trivial examples:
myRec := rec( a := 5 );
myRecs := [ rec( b := 3 ), rec( a := 6 ) ];
f := function ( x ) Print( x, "\n" ); end;
JUPVIZFetchWithDefault( myRec, myRecs, "a", 0, f );       # prints 5
JUPVIZFetchWithDefault( myRec, myRecs, "b", 0, f );       # prints 3
JUPVIZFetchWithDefault( myRec, myRecs, "c", 0, f );       # prints 0
JUPVIZFetchWithDefault( myRec, myRecs, ["a","b"], 0, f ); # prints 0
# Useful example:
JUPVIZFetchWithDefault( primaryRecord, secondaryRecordsList,
  [ "options", "height" ], 400,
  function ( h ) myGraphJSON.height := h; end
);
```

See also `JUPVIZFetchIfPresent` (7.3.13).

7.3.13 JUPVIZFetchIfPresent

▷ `JUPVIZFetchIfPresent(`*`record, others, chain, action`*`)` (function)

Returns: nothing

This function is extremely similar to JUPVIZFetchWithDefault (7.3.12) with the following exception: No default value is provided, and thus if the lookup fails for all the records (including record and everything in others) then the action is not called.

Examples:

Example

```
myRec := rec( a := 5 );
myRecs := [ rec( b := 3 ), rec( a := 6 ) ];
f := function ( x ) Print( x, "\n" ); end;
JUPVIZFetchIfPresent( myRec, myRecs, "a", 0, f );       # prints 5
JUPVIZFetchIfPresent( myRec, myRecs, "b", 0, f );       # prints 3
JUPVIZFetchIfPresent( myRec, myRecs, "c", 0, f );       # does nothing
JUPVIZFetchIfPresent( myRec, myRecs, ["a","b"], 0, f ); # does nothing
```

7.4 Representation wrapper

This code is documented for completeness's sake only. It is not needed for clients of this package. Package maintainers may be interested in it in the future.

The JupyterKernel package defines a method JupyterRender that determines how GAP data will be shown to the user in the Jupyter notebook interface. When there is no method implemented for a specific data type, the fallback method uses the built-in GAP method ViewString.

This presents a problem, because we are often transmitting string data (the contents of JavaScript files) from the GAP kernel to the notebook, and ViewString is not careful about how it escapes characters such as quotation marks, which can seriously mangle code. Thus we must define our own type and JupyterRender method for that type, to prevent the use of ViewString.

The declarations documented below do just that. In the event that ViewString were upgraded to more useful behavior, this workaround could probably be removed. Note that it is used explicitly in the using-library.js file in this package.

If this package is loaded without the JupyterKernel package having already been loaded, then the following functions and tools are not defined, because their definitions rely on global data made available by the JupyterKernel package.

7.4.1 JUPVIZIsFileContents (for IsObject)

▷ JUPVIZIsFileContents(*arg*) (filter)

Returns: true or false

The type we create is called FileContents, because that is our purpose for it (to preserve, unaltered, the contents of a text file).

7.4.2 JUPVIZIsFileContentsRep (for IsComponentObjectRep and JUPVIZIsFileContents)

▷ JUPVIZIsFileContentsRep(*arg*) (filter)

Returns: true or false

The representation for the FileContents type

7.4.3 JUPVIZFileContents (for IsString)

▷ `JUPVIZFileContents(`*arg*`)` (operation)

A constructor for `FileContents` objects

Elsewhere, the JupyterViz package also installs a `JupyterRender` method for `FileContents` objects that just returns their text content untouched.

Index