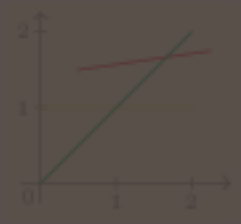
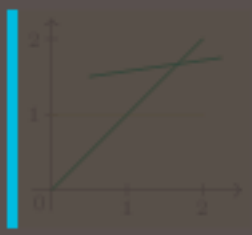
much nicer if we could name any way we want. To achieve this, we use the special rule for style sheets that says that if there is a subkey of an attribute whose name is the same name as the value, then the value of this key is used instead. This slightly intimidating definition is much easier to understand when we have a look at an example:




```
% Definition of traffic light keys as above
\begin{tikzpicture}
  \datavisualization data group {lines} = {
    data point {x=0, y=0,       set=normal}
    data point {x=2, y=2,       set=normal}
    data point {x=0, y=1,       set=heated}
    data point {x=2, y=1,       set=heated}
    data point {x=0.5, y=1.5,   set=critical}
    data point {x=2.25, y=1.75, set=critical}
  };
  \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    /data point/set/normal/.initial=1,
    /data point/set/heated/.initial=2,
    /data point/set/critical/.initial=3,
    style sheet=traffic light]
  data group {lines};
\end{tikzpicture}
```

Now, it is a bit bothersome that we have to set all these `/data point/set/...` keys by hand. It turns out that this is not necessary: Each time a visualizer is created, a subkey of `/data point/set` with the name of the visualizer is created automatically and a number is stored that is increased for each new visualizer in a data visualization. This means that the three lines starting with `/data point` are inserted automatically for you, so they can be left out. However, you would need them for instance when you would like several different data sets to use the same styling:




```
% Definition of traffic light keys as above
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  /data point/set/critical/.initial=1, % same styling as first set
  style sheet=traffic light]
data group {lines};
```

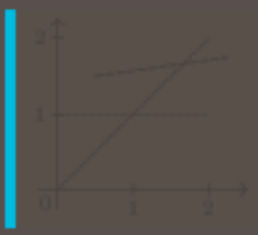
We can a command that slightly simplifies the definition of style sheets:

`\pgfdvdeclarestylesheet{`*name*`}{`*keys*`}`

This command executes the *keys* with the path prefix `/pgf/data visualization/style sheets/`*name*. The above definition of the traffic light style sheet could be rewritten as follows:

```
\pgfdvdeclarestylesheet{traffic light}{
  1/.style={green!60!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
}
```

As a final example, let us create a style sheet that changes the dashing pattern according to the value of the attribute. We do not need to define an large number of styles in this case, but can use the `default style` key to "calculate" the correct dashing.



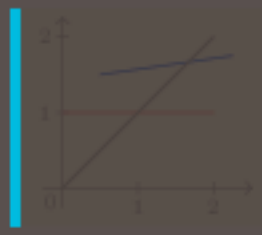

```
\pgfdvdeclarestylesheet{my dashings}{
  default style/.style={dash pattern={on #1pt off 1pt}}
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my dashings]
data group {lines};
```

84.4.3 Creating a New Color Style Sheet

Creating a style sheet that varies colors according to an attribute works the same way as creating a normal style sheet: Subkeys like 1, 2, and so on use the `style` attribute to setup a color. However, instead of using the `color` attribute to set the color, you should use the `visualizer color` key to set the color:

`/tikz/visualizer color=`*(color)* (no default)

This key is used to set the color `visualizer color` to *(color)*. This color is used by visualizers to color the data they visualize, rather than the current "standard color". The reason for not using the normal current color is simply that it makes many internals of the data visualization engine a bit simpler.

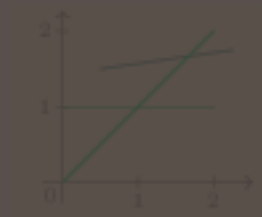



```
\pgfdvdeclarestylesheet{my colors}
{
  default style/.style={visualizer color=black},
  1/.style={visualizer color=black},
  2/.style={visualizer color=red!80!black},
  3/.style={visualizer color=blue!80!black},
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my colors]
data group {lines};
```

There is an additional command that makes it easy to define a style sheet based on a *color series*. Color series are a concept from the `xcolor` package: The idea is that we start with a certain color for the first data set and then add a certain "color offset" for each next data point. Please consult the documentation of the `xcolor` package for details.

`\tikzdvdeclarestylesheetcolorseries{`*(name)*`}{`*(color model)*`}{`*(initial color)*`}{`*(step)*`}`

This command creates a new style sheet using `\pgfdvdeclarestylesheet`. This style sheet will only have a default style setup that maps numbers to the color in the color series starting with *(initial color)* and having a stepping of *(step)*. Note that when the value of the attribute is 1, which it is the first data set, the *second* color in the color series is used (since counting starts at 0 for color series). Thus, in general, you need to start the *(initial color)* "one early".




```
\tikzdvdeclarestylesheetcolorseries{greens}{hsb}{0.3,1,0.8}{0,-0.4,-1}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=greens]
data group {lines};
```

84.5 Reference: Style Sheets for Lines

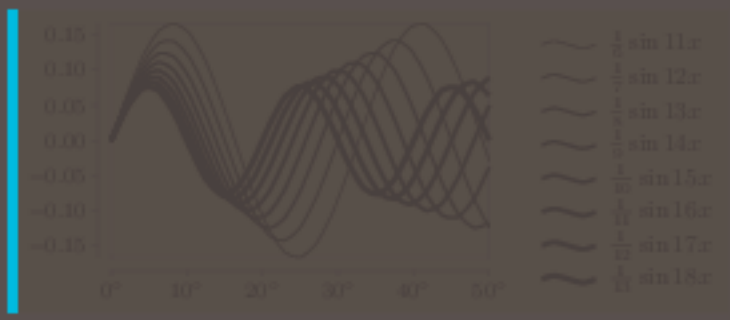
The following style sheets can be applied to visualizations that use the `visualize as line` and related keys. For the examples, the following style and data set are used:

```
\tikzdatavisualizationset {
  example visualization/.style={
    scientific axes=clean,
    y axis={ticks={style={
          /pgf/number format/fixed,
          /pgf/number format/fixed zerofill,
          /pgf/number format/precision=2}},
    x axis={ticks={tick suffix={$}^\circ$}},
    1={label in legend={text={$\frac{1}{6}\sin 11x$}},
    2={label in legend={text={$\frac{1}{7}\sin 12x$}},
    3={label in legend={text={$\frac{1}{8}\sin 13x$}},
    4={label in legend={text={$\frac{1}{9}\sin 14x$}},
    5={label in legend={text={$\frac{1}{10}\sin 15x$}},
    6={label in legend={text={$\frac{1}{11}\sin 16x$}},
    7={label in legend={text={$\frac{1}{12}\sin 17x$}},
    8={label in legend={text={$\frac{1}{13}\sin 18x$}}
  }
}
```

```
\tikz \datavisualization data group {sin functions} = {
  data [format=function] {
    var set : {1,...,8};
    var x : interval  0:50;
    func y = sin(\value x * (\value{set}+10))/(\value{set}+5);
  }
};
```

Style sheet `vary thickness`

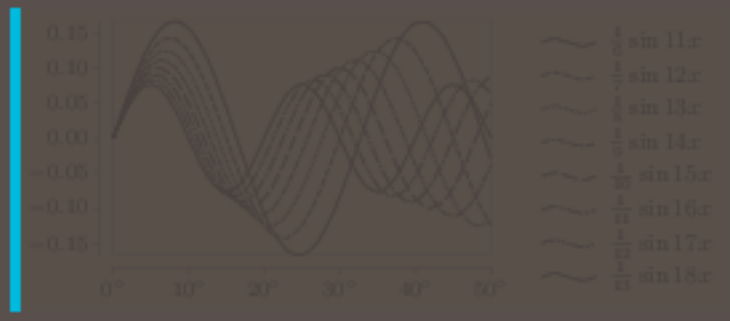
This style varies the thickness of lines. It should be used only when there are only two or three lines, and even then it is not particularly pleasing visually.




```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness]
data group {sin functions};
```

Style sheet `vary dashing`

This style varies the dashing of lines. Although it is not particularly pleasing visually and although visualizations using this style sheet tend to look "excited" (but not necessarily "exciting"), this style sheet is often the best choice when the visualization is to be printed in black and white.

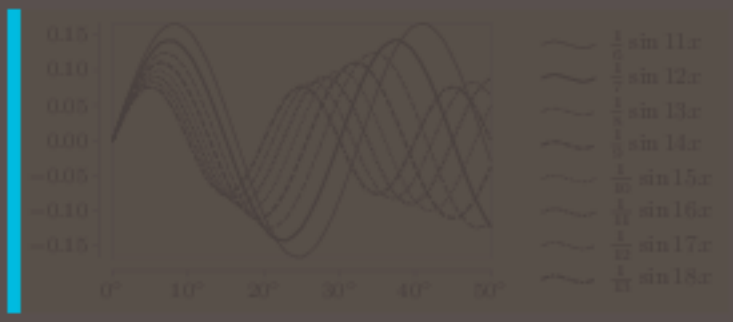



```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary dashing]
data group {sin functions};
```

As can be seen, there are only seven distinct dashing patterns. The eighth and further lines will use a solid line once more. You will then have to specify the dashing "by hand" using the `style` option together with the visualizer.

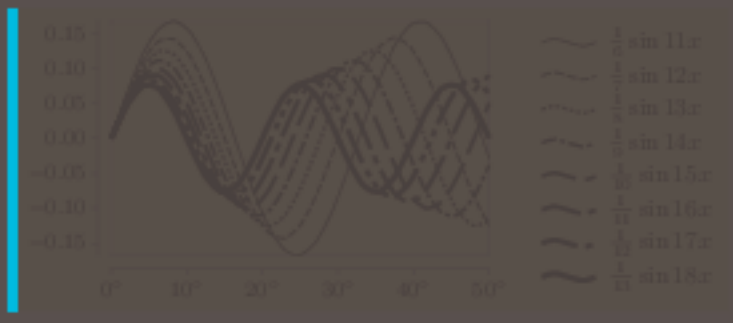
Style sheet `vary dashing and thickness`

This style alternates between varying the thickness and the dashing of lines. The difference to just using both the `vary thickness` and `vary dashing` is that too thick lines are avoided. Instead, this style creates clearly distinguishable line styles for many lines (up to 14) with a minimum of visual clutter. This style is the most useful for visualizations when many different lines (ten or more) should be printed in black and white.




```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness
              and dashing]
data group (six functions);
```

For comparison, here is the must-less-than-satisfactory result of combining the two independent style sheets:




```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness,
  style sheet=vary dashing]
data group (six functions);
```

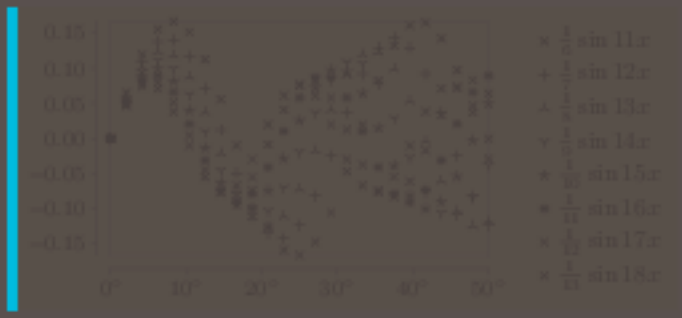
84.6 Reference: Style Sheets for Scatter Plots

The following style sheets can be used both for scatter plots and also with lines. In the latter case, the marks are added to the lines.

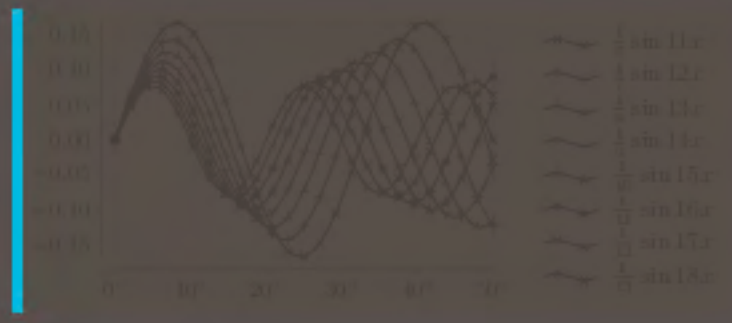
Style sheet `cross marks`

This style uses different crosses to distinguish between the data points of different data sets. The crosses were chosen in such a way that when two different cross marks lie at the same coordinate, their overall shape allows one to still uniquely determine which marks are on top of each other.

This style supports only up to six different data sets and requires the `plotmarks` library.




```
\tikz \datavisualization [
  visualize as scatter/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=cross marks]
data group (six functions);
```



84.7 Reference: Color Style Sheets

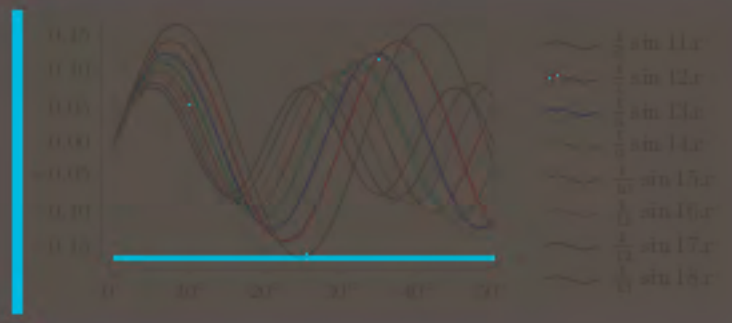
Color style sheets are very useful for creating visually pleasing data visualizations that contain multiple data sets. However, there are two things to keep in mind:

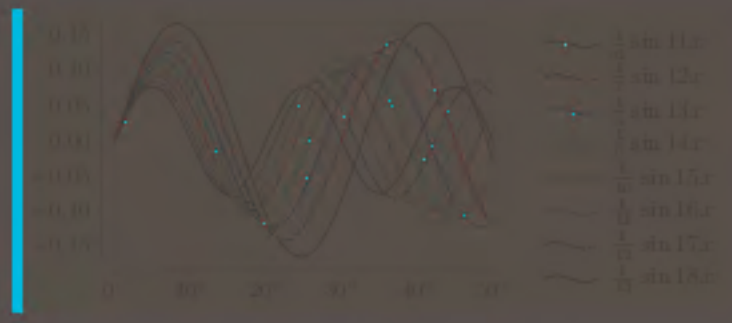
- At some point, every data visualization is printed or photocopied in black and white by someone. In this case, data sets can often no longer be distinguished.
- A few people are color blind. They will not be able to distinguish between red and green lines (and some people are not even able to distinguish colors at all).

For these reasons, if there is any chance that the data visualization will be printed in black and white at some point, consider combining color style sheets with style sheets like `vary dashing` to make data sets distinguishable in all situations.

Style sheet `strong colors`

This style sheets uses pure primary colors that can very easily be distinguished. Although not as visually pleasing as the `vary hue` style sheet, the visualizations are easier to read when this style sheet is used. Up to six different data sets are supported.

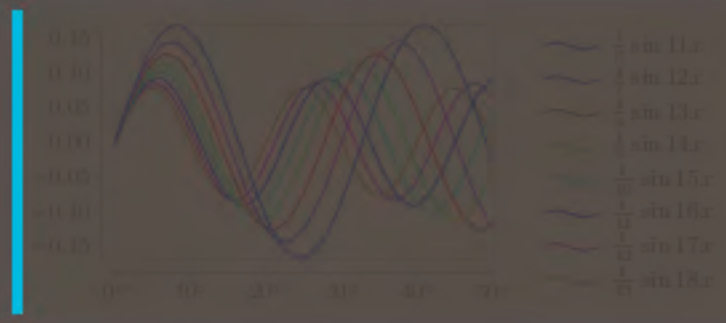




Unlike `strong colors`, the following style sheets support, in principle, an unlimited number of data sets. In practice, as always, more than four or five data sets lead to nearly indistinguishable data sets.

Style sheet `very hue`

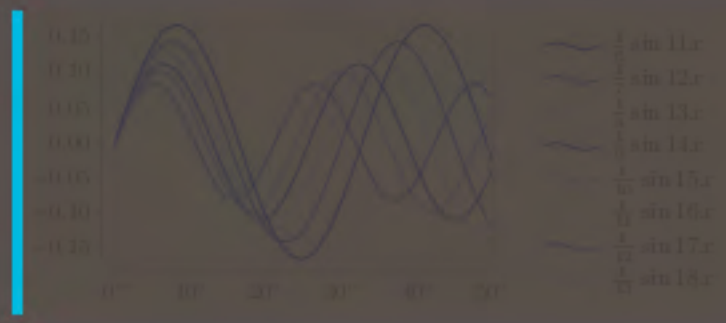
This style uses a different hue for each data set.



```
\tikz \datavisualization
  [scientific axes=clean, visualize as
  data group (six functions);
```

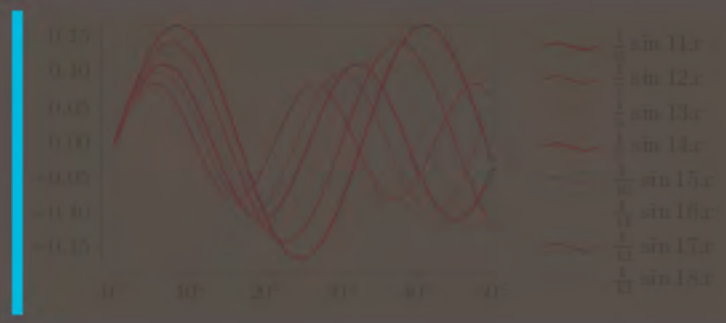
Style sheet `shades of blue`

As the name suggests, different shades of blue are used for different data sets.



```
\tikz \datavisualization
  [scientific axes=clean, visualize as
  data group (six functions);
```

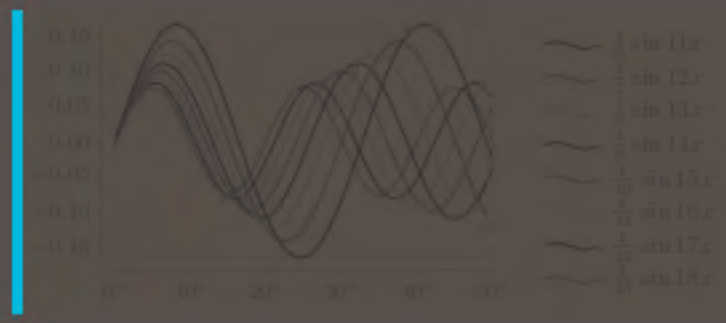
Style sheet `shades of red`



```
\tikz \datavisualization
  [scientific axes=clean, visualize as
  data group (six functions);
```

Style sheet `gray scale`

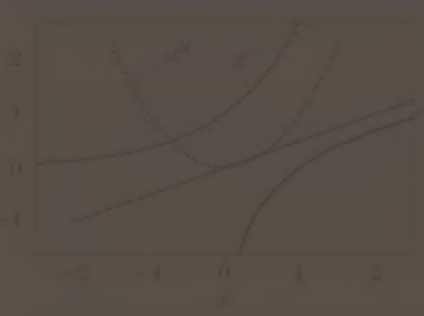
For once, this style sheet can also be used when the visualization is printed in black and white.



```
\tikz \datavisualization
  [scientific axes=clean, visualize as
  data group (six functions);
```

`/tikz/data visualization/legend entry options/text=`*(text)* (no default)

Use this key to setup the *(text)* that is shown as the label of the data set.




```
\tikz \datavisualization
  data group {section class};
```

In addition to the two keys described above, there are further keys that are described in Section 84.9.6.

84.9.2 Rows and Columns of Legend Entries

In a legend, the different legend entries are arranged in a matrix, which typically has only one row or one column. For the impatient reader: Say `rows=1` to get everything in a row, say `columns=1` to get everything in a single column, and skip the rest of this section.

The more patient reader will appreciate that when there are very many different data sets in a single visualization, it may be necessary to use more than one row or column beside the legend. TikZ comes with a rather powerful mechanism for distributing the multiple legend entries over the matrix.

The first thing to decide is in which "direction" the entries should be inserted into the matrix. Suppose we have a 3×3 matrix and our entries are a, b, c, and so on. Then, one might place the a in the upper left corner of the matrix, b in the upper middle position, c in the upper right position, and d in the middle left position. This is a "first right, then down" strategy. A different strategy might be to place the a in the upper left corner, but b in the middle left position, c in the lower left position, and d then in the upper middle position. This is a "first down, then right" strategy. In certain situations it might even make sense to place a in the lower right corner and then go "first up, then left".

All of these strategies are supported by the `legend` command. You can configure which strategy is used using the following keys:

`/tikz/data visualization/legend options/down then right` (no value)

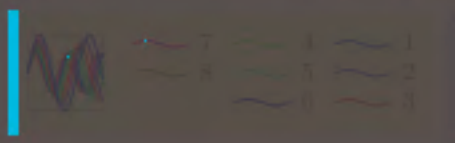
Causes the legend entries to fill the legend matrix first downward and, once a column is full, the next column is begun to the right of the previous one. This is the default.


```
\tikz \datavisualization
  data group {sin function};
```

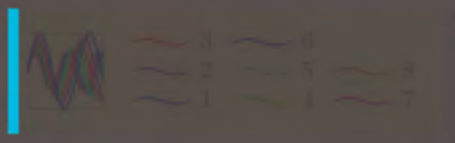
In the example, the `legend` example is the following style:

```
\tikzdatavisualizationset{
  legend example/.style={
    scientific axes, all axes=legend=section, ticks=minor,
    1={label in legend={text=1}},
    2={label in legend={text=2}},
    3={label in legend={text=3}},
    4={label in legend={text=4}},
    5={label in legend={text=5}},
    6={label in legend={text=6}},
    7={label in legend={text=7}},
    8={label in legend={text=8}},
```

/tikz/data visualization/legend options/down then left (no value)



/tikz/data visualization/legend options/up then right (no value)



/tikz/data visualization/legend options/up then left (no value)

/tikz/data visualization/legend options/left then up (no value)

/tikz/data visualization/legend options/left then down (no value)

/tikz/data visualization/legend options/right then up (no value)

/tikz/data visualization/legend options/right then down (no value)

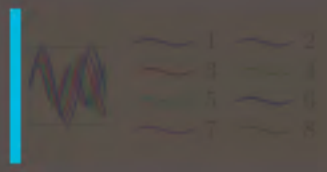
Having configured the directions in which the matrix is being filled, you must next setup the number of rows or columns that are to be shown. There are actually two different ways of doing so. The first way is to specify a maximum number of rows or columns. For instance, you might specify that there should be at most ten rows to a column and when there are more, a new column should be begun. This is achieved using the following keys:

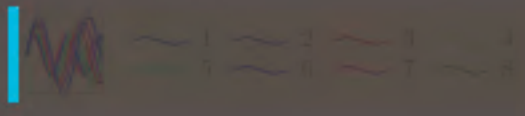
/tikz/data visualization/legend options/max rows=⟨number⟩ (no default)

As the legend matrix is being filled, whenever the number of rows in the current column would exceed ⟨number⟩, a new column is started.

/tikz/data visualization/legend options/max columns=⟨number⟩ (no default)

This key works like max rows, only now the number of columns is monitored. Note that this strategy only really makes sense when the when you use this key with a strategy that first goes left or right and then up or down.

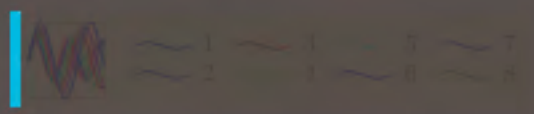




The second way of specifying the number of entries in a row or column is to specify an "ideal number of rows or columns". The idea is as follows: Suppose that we use the standard strategy and would like to have everything in two columns. Then if there are eight entries, the first four should go to the first column, while the next four should go to the second column. If we have 20 entries, the first ten should go the first column and the next ten to the second, and so on. So, in general, the objective is to distribute the entries evenly so the this "ideal number of columns" is reached. Only when there are too few entries to achieve this or when the number of entries per column would exceed the max rows value, will the number of columns deviate from this ideal value.

/tikz/data visualization/legend options/ideal number of columns=*⟨number⟩* (no default)

Specifies that the entries should be split into *⟨number⟩* different columns, whenever possible. However, when there would be more than the `max rows` value of rows per column, more columns than the ideal number are created.

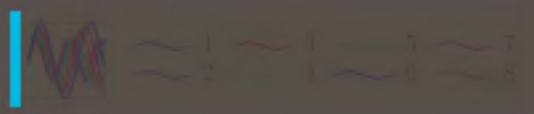


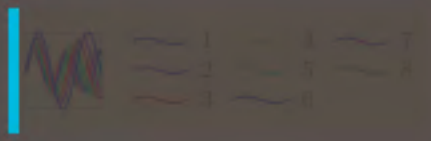
/tikz/data visualization/legend options/rows=*⟨number⟩* (no default)

Shortland for `ideal number of rows=`*⟨number⟩*.

/tikz/data visualization/legend options/ideal number of rows=*⟨number⟩* (no default)

Works like `ideal number of columns`.





/tikz/data visualization/legend options/columns=*⟨number⟩* (no default)

Shortland for `ideal number of columns=`*⟨number⟩*.

84.9.3 Legend Placement: The General Mechanism

A legend can either be placed next to the data visualization or inside the data visualization at some place where there are no data entries. Both approaches have advantages: Placing the legend next to the visualization minimizes the "cluttering" by keeping all the extra information apart from the actual data, while placing the legend inside the visualization minimizes the distance between the data sets and their explanations, making it easier for the eye to connect them.

For both approaches there are options that make the placement easier, see Sections 84.9.4 and 84.9.5, but these options internally just map to the following two options:

\pgfarrowsaddtolateoptions{*code*}

This command works like \pgfarrowsaddtooptions, only the *code* will be executed "later" than the code added by the normal version of the command. This is useful for keys that depend on the length of an arrow: Keys like `width'` want to define the arrow width as a multiple of the arrow length, but when the `width'` key is given, the length may not yet have been specified. By making the computation of the width a "late" option, we ensure that \pgfarrowslength will have been setup correctly.

If you define a new option that sets a dimensions and if that dimension should change in accordance to the setting of either `scale length` or `scale width`, you need to make PGF "aware" of this using the following key:

\pgfarrowsaddtolengthscalelist{*dimension register*}

Each time an arrow tip is used, the given *dimension register* will be multiplied by the `scale length` factor prior to the actual drawing. You call this command only once in the preamble somewhere.

\pgfarrowsaddtowidthscalelist{*dimension register*}

Works like \pgfarrowsaddtolengthscalelist, only for width parameters.

\pgfarrowsthreeparameters{*line-width dependent size specification*}

This command is useful for parsing the values given to keys like `length` or `width` the expect a dimension followed optionally for some numbers. This command converts the *line-width dependent size specification*, which may consist of one, two, or three numbers, into a triple of three numbers in curly braces, which gets stored in the macro \pgfarrowstheparameters. Here is an example, where \showvalueofmacro is used in this example to show the value stored in a macro:

```
{2.0pt}{1}{0}
```

```
\pgfarrowsthreeparameters{2pt 1}
\showvalueofmacro\pgfarrowstheparameters
```

\pgfarrowslinewidthdependent{*dimension*}{*line width factor*}{*outer factor*}

This command take three parameters and does the "line width dependent computation" described on page 190 for the `length` key. The result is returned in \pgf@x.

The idea is that you can setup line-width dependent keys like `length` or `width` using code like the following:

```
\pgfkeys{/pgf/arrow keys/depth/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions\expandafter{%
    \expandafter\pgfarrowslinewidthdependent\pgfarrowstheparameters% compute...
    \pgfarrowsdepth\pgf@x% ... and store.
  }%
}
```

\pgfarrowslengthdependent{*dimension*}{*length factor*}{*dummy*}

This command take three parameters, of which the last one is ignored, and does the "length dependent computation" described for the `width'` and `inset'` keys. The result is returned in \pgf@x.

You can setup length dependent keys using code like the following:

```
\pgfkeys{/pgf/arrow keys/depth'/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions\expandafter{%
    \expandafter\pgfarrowslengthdependent\pgfarrowstheparameters% compute...
    \pgfarrowsdepth\pgf@x% ... and store.
  }%
}
```

```
\animationexample(my path)(path){
  \pgfsysanimkeytime(0)(1)(1)(0)(0)
  \pgfsysanminvalpath{\pgfsys@moveto{1cm}{0cm}%
                      \pgfsys@lineto{1cm}{1cm}%
                      \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime(2)(1)(1)(0)(0)
  \pgfsysanminvalpath{\pgfsys@moveto{1cm}{1cm}%
                      \pgfsys@lineto{2cm}{1cm}%
                      \pgfsys@lineto{1cm}{0cm}}
  \pgfsysanimate{path}
  \filldraw [ultra thick,draw=blue,fill=blue!20, name=my path]
    (1,0) -- (1,1) -- (2,0); }
```

You can attach arrow tips to paths that are animated and these arrow tips will correctly "rotate and move along" with the path's end points *if* you take the following points into considerations:

- Arrow tips that "rotate and move along" with a path must be specified using a special animation command, see below. The normal arrow tips added to a path would *not* be animated automatically and, indeed, if you add arrow tips to a path using `\pgfsetarrows` and then animate the path, you will get an error message.
- Internally, the arrow tips that "rotate and move along" are drawn using so-called *markers*. These are little graphic objects that can be added to the start and end of paths and that are automatically rotated and move along with the path.

  In principle, the rendering rules used by SVG for markers are the same as for normal arrow tips: The markers are rotated and moved so that the point along a tangent of the path at the start or end of the path. However, when it comes to special cases such as a path with multiple segments, a path with degenerate segments, a closed path, and so on, the rules used by for instance SVG may differ from the placement that PGF will compute.

  Thus, it is best to add arrow tips only to "normal" paths consisting of a single open path segment whose ends can be shortened a bit without causing degeneration.
- When an arrow tip is added to a path, the path must typically be shortened a bit so that the *tip* of the arrow ends where the path would usually end. This shortening is not done by the system layer for to-be-animated paths; you must compute and then animate these shortened paths yourself. (However, the basic layer animation module will do this for you, so you only have to worry about this when you use the system layer directly.)

Let us now have a look at how we add arrow tip markers:

`\pgfsysanimkeytipmarkers`{*⟨start marker⟩*}{*⟨end marker⟩*}

`\pgfsys@animation@tip@markers`{*⟨start marker⟩*}{*⟨end marker⟩*}

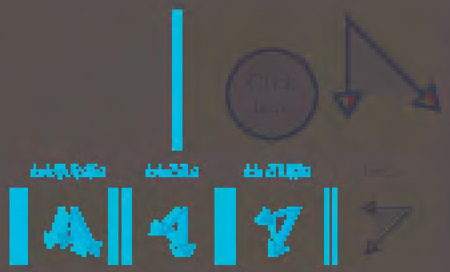
This command specifies that during a path animation the two markers provided as parameters should be added (and rotated and moved along with the path) at the start and end. The *⟨start marker⟩* must either be empty (in which case no marker is added at the start) or it must be a macro storing a value returned by the command `\pgfsys@marker@declare`. In this case, the marker declared symbol will be added to the start during the animation. The same situation applies to the end of the path.

As pointed out earlier, only arrow tips / markers added to paths using this command will be animated along with the path. In particular, you should *not* add arrow tips to to-be-animated paths using `\pgfsetarrow`. However, when you use a base value (`\pgfsys@animation@base`) to set a path, the arrow tips will also be added to this base path.

To sum up, the "correct" way of adding arrow tips to a path that is animated is to proceed as follows:

1. You specify arrow tips for a path using this command.
2. You specify times and values of the to-be-animated path, shortened as necessary to accommodate the length of the arrow tips.
3. You specify the first (or, possibly, some other) value in the time–value sequence as a base value.
4. You create a path animation that applies to a future path.
5. You create this future path as an empty path without arrow tips and draw it. Because of the setting of the base value, instead of the empty path the base path will be used as the "real" path and the animation's arrow tips will be added as arrow tips.

When you have more than one animation for a given path, these different animations may use different arrow tips / markers. This allows you to animate (change) which arrow tip is used on a path over time.



```
% Declare a marker:
\pgfsysmarkerdeclare\mymarker{
  \pgfscope
    \pgfsetcolor{red!75}
    \pgfpathmoveto{\pgfpoint{0pt}{5pt}}\pgfpathlineto{\pgfpoint{2pt}{0pt}}
    \pgfpathlineto{\pgfpoint{0pt}{0pt}}\pgfpathclose
    \pgfusepathqfill
  \endpgfscope
  \pgfpathmoveto{\pgfpoint{0pt}{5pt}}\pgfpathlineto{\pgfpoint{5pt}{0pt}}
  \pgfpathlineto{\pgfpoint{0pt}{-5pt}}\pgfpathclose
  \pgfusepathqstroke
}

\animationexample{path}{path}{
  \pgfsysanimkeytipmarkers{\mymarker}{\mymarker}
  \pgfsysanimkeybase
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{0cm}
                     \pgfsys@lineto{1cm}{1cm}
                     \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{0cm}
                     \pgfsys@lineto{1cm}{1cm}
                     \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{1cm}
                     \pgfsys@lineto{2cm}{1cm}
                     \pgfsys@lineto{1cm}{0cm}}
  \pgfsysanimate{path}
  \filldraw [illegible]
  \path (1,0) (2,1);}
```

\pgfsysanimate{linewidth}

Adds an animation of the line width.

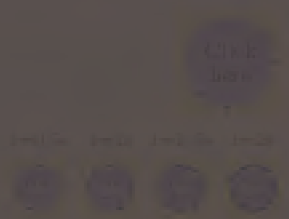
Specify values with \pgfsysanimvaldimension.

```
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldimension{1pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldimension{10pt}
  \pgfsysanimate{linewidth}
}
```

\pgfsysanimate{dash}

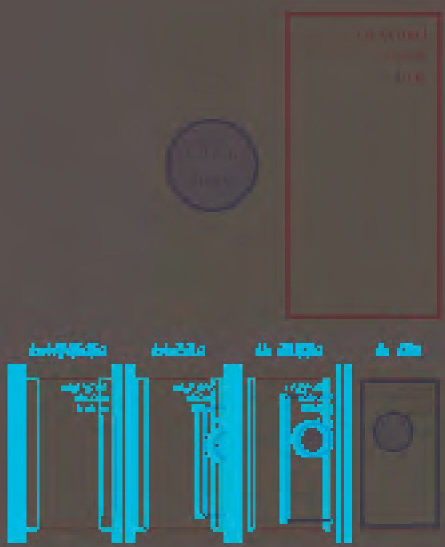
Adds an animation of the dash phase and pattern (like \pgfsyssetdash).

Specify values with \pgfsysanimvaldash.



```
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldash{1pt,10pt}{0pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldash{10pt,3pt}{0pt}
  \pgfsysanimate{dash}
}
```

```
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldash{1cm,1pt}{0pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldash{1cm,1pt}{1cm}
  \pgfsysanimate{dash}
}
```



```
\animationexample{}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalviewbox{-10pt}{-20pt}{10pt}{20pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalviewbox{-5pt}{-15pt}{27pt}{2pt}
  \pgfsysanimate{viewbox}
  \scoped[...]
    \draw ... (0,0) rectangle (1,2) ...
              ...
    \scoped ...
      \draw (-5pt,-15pt) rectangle (27pt,2pt)
        ... \arrow ...
          ...
      \filldraw (0,0) circle ...
```

123.5 Commands for Specifying the Target Object

\pgfsysanimkeywhom{⟨id⟩}{⟨type⟩}

\pgfsys@animation@whom{⟨id⟩}{⟨type⟩}

Sets the target of the animation. The ⟨id⟩ must previously have been created using \pgfsys@new@id, ⟨type⟩ must be a type (the empty type is also allowed). See Section 123.13 for details on ids and types.

123.6 Commands for Specifying Timelines: Specifying Times

Animations are specified using *timelines*, which are functions mapping times to values for these times. The functions are cubic splines, which are specified using time–value pairs plus control points.

In order to specify a time–value pair, you first use the command \pgfsysanimkeytime to specify a time. Next, you use \pgfsysanimval... to specify a value, which adds the time–value pair to the timeline. Note that the times must be given in non-decreasing order. Between time–value pairs, the values are interpolated using a spline.

The first and last times of the timeline are a bit special: The timeline starts on the first time and the duration of the timeline is the difference between the first and last time. "Starting" on the start time actually means that any beginnings (see the commands for specifying beginnings and endings) get an offset the start time; similarly end times are offset by this value.

\pgfsysanimkeytime{⟨time⟩}{⟨entry spline control x⟩}{⟨entry spline control y⟩}{⟨exit spline control x⟩}{⟨exit spline control y⟩}

\pgfsys@animation@time{⟨time⟩}{⟨entry spline control x⟩}{⟨entry spline control y⟩}{⟨exit spline control x⟩}{⟨exit spline control y⟩}

The ⟨time⟩ is a number representing seconds (so 0.5 means 500ms).

The spline between a time–value pair and the next is specified using the four parameters following the time. The first two of these specify the second control point of the interval preceding the time-value pair (called the "entry" control point), the last two parameters specify the first control point of the interval following the pair (called the "exit" control point). Consider, for instance, the following calls:

```
\pgfsysanimkeytime{10}{0.1}{0.2}{0.3}{0.4}
\pgfsysanimvalscalar{100}
\pgfsysanimkeytime{15}{0.5}{0.6}{0.7}{0.8}
\pgfsysanimvalscalar{200}
```

This will create (at least) the time interval [10s, 15s] and the control points for this interval will be (0.3, 0.4) and (0.5, 0.6).

Control points are specified in a different "coordinate" system from the time–value pairs themselves: While the time–value pairs are specified using a number representing seconds and a value using some