TIBCO Spotfire® Clinical Graphics

GOM User's Guide

Software Release 2.2
August 2012

two-second advantage™

Important Information

The correct bibliographical reference for this document is as follows:

TIBCO Spotfire ® Clinical Graphics 2.2 GOM User's Guide, TIBCO Software Inc.

For technical support, please visit http://spotfire.tibco.com/support and register for a support account.

CONTENTS

TIBCO SPOTFIRE CLINICAL GRAPHICS INTRODUCTION

1

OVERVIEW

This document provides details about using the Spotfire S+ Graphics Object Model (the GOM). The examples and code contained in this PDF provide a starting point for developing your own scripts, or for interpreting scripts produced by the point-and-click TIBCO Spotfire Clinical Graphics (TSCG) Client.

For Help with the GOM functions, see the CHM (on Windows) or the HTML help files for the functions.

Note
When we refer to the object model, we refer to it as the GOM; however, the package, and the object are lower case (that is, `gom`), and the fuction is lower-case (that is, `gom`).

GETTING STARTED

The Graphics Object Model library (`gom`) is provided as a Spotfire S+ package. To load and use it, from Spotfire S+, type the following:

```
library(gom)
```

The `gom` library always positions itself at the top of the search list.

```
search()

[1] "C:\\\MYDOCU~1\\SPOTFI~2\\Project1"
[2] "gom"
. . .
```

Use the `gom()` function to create all graphs. Although all code produces graphs in any device, it was designed for the `graphlet()` device supplied with TIBCO Spotfire Clinical Graphics.

To get help with the gom package, at the Spotfire S+ command prompt, type:

```
?gom
```

INTRODUCTION TO THE GOM

The `gom()` function is a model interface to the underlying Graphic Object Model. The `gom()` design centers around the following concepts:

- *Model statement*: a variable formula for positions, conditioning and grouping.
- *Data statement*: the data frame that holds the values to be graphed.
- *Panel statement*: the graphic elements such as points, lines or bars to be used.

The `gom()` function creates graphs using expressions such as:

```
gom(conc ~ time, data = Indometh)
```

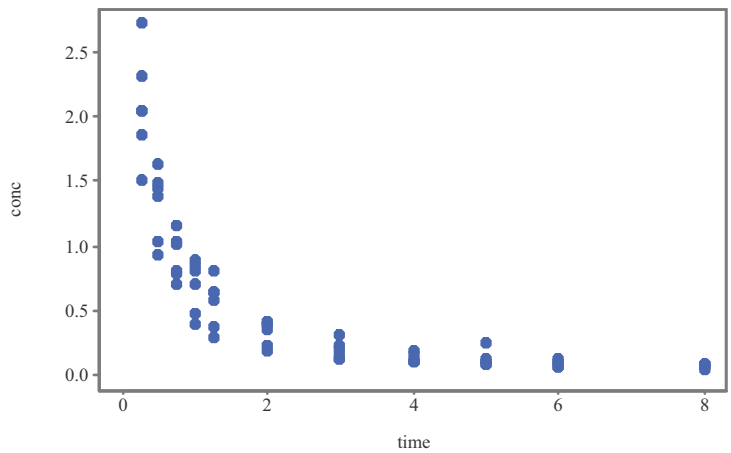


Figure 1.1: *Scatter plot example.*

In this expression, `Indometh` is the data that holds the variables `conc` and `time`, and the model formula `conc ~ time` is the expression that specifies plotting `conc` as a model of `time`.

Spotfire S+ Trellis and R lattice users are familiar with this type of formula via the plot-specific interfaces `xyplot(conc ~ time, data = Indometh)`. By plot-specific, we mean that the two frameworks contain functions indicating specific plots (for example, `dotplot()` creates a dot plot and `histogram()` produces a histogram). By contrast, the `gom()` function is plot-type agnostic.

Graphics Elements

Using `gom()`, you can draw a particular plot type by providing a single graphic element or a combination of graphic elements. For example, you can draw a scatter plot by specifying in `gom()` the default `ge.points` graphic element. For example:

```
gom(conc ~ round(time), data = Indometh, panel = ge.points)
```

You can draw a box plot by specifying the `ge.boxplot` element:

```
gom( conc ~ round(time), data = Indometh,
  panel = ge.boxplot)
```

Combining `ge.points` and `ge.boxplots` in a `list()` draws a boxplot with points superposed:

```
gom(conc ~ round(time),
  data = Indometh,
  panel = list(ge.boxplot, ge.points)
)
```

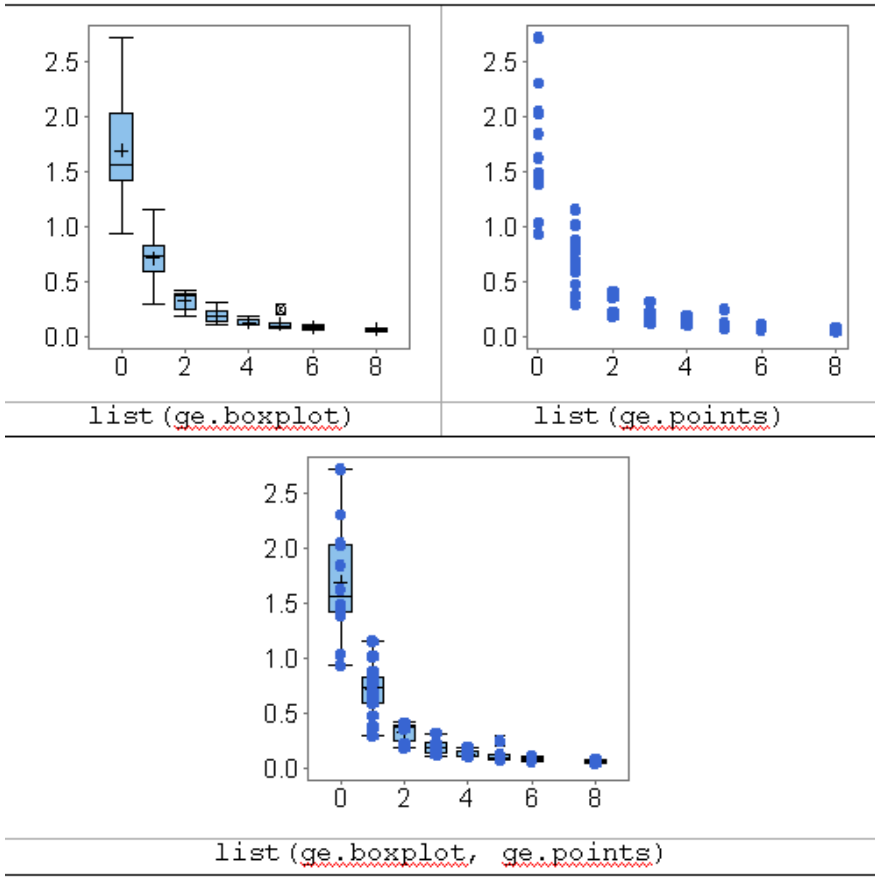


Figure 1.2: *Combining graphics elements.*

As `gom()` building blocks, the graphics elements contain much of the logic and behavior. You can reuse, combine, and develop new graph types using graphics elements.

Groups

Using groups, you can stratify within the plot region. For example, you can separate points by applying a new style for each group level:

```
gom( conc ~ time,
  groups = ~ Subject,
```

```
  data = Indometh,
  page=list(
      legend = list(numberOfColumns = 3))
)
```

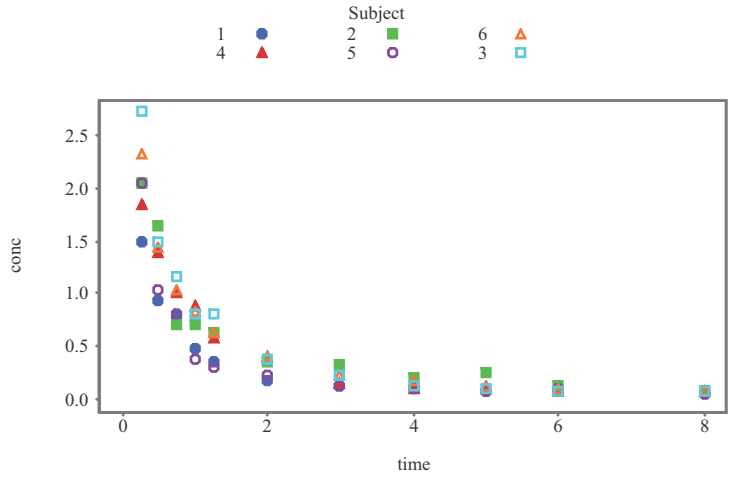


Figure 1.3: *Grouped scatter plot example.*

Note
The legend is displayed in three columns to preserve space. For more information about `legend`, see the section The Page Object on page 15.

Trellis

Formulas of the type `y ~ x | g` repeat the graph in panels. Each panel corresponds to a unique value of `g` and the values plotted to the subset corresponding to `g`. You can interpret this as *conditioning*. That is, `y` is a model of `x` given `g`.

For example, you can condition a time concentration profile on subjects, obtaining a concentration profile for each individual:

```
gom( conc ~ time | Subject,data = Indometh)
```

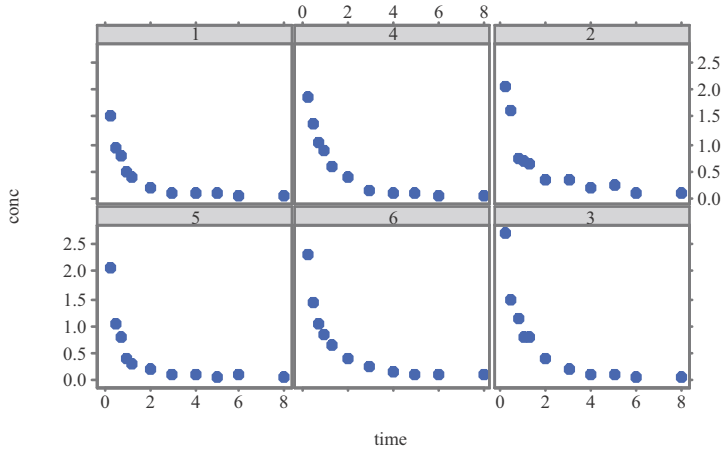


Figure 1.4: *Trellis scatter plot example.*

Note that the rows and columns have identical scales and are exactly identical to the unconditioned graph. This design is one of the key principles in the Trellis methodology.

Row, Column, and Matrix Plots

You can bind panels by variable with `gom()`. This feature is similar to Trellis, but the panels display different information. For example, using Row plots, you can stack plots. The following example demonstrates stock quotes on top of volume, using a common time x-axis, or measurements with a common x-axis:

```
gom( glyco + conc ~ time, data = Quinidine)
```

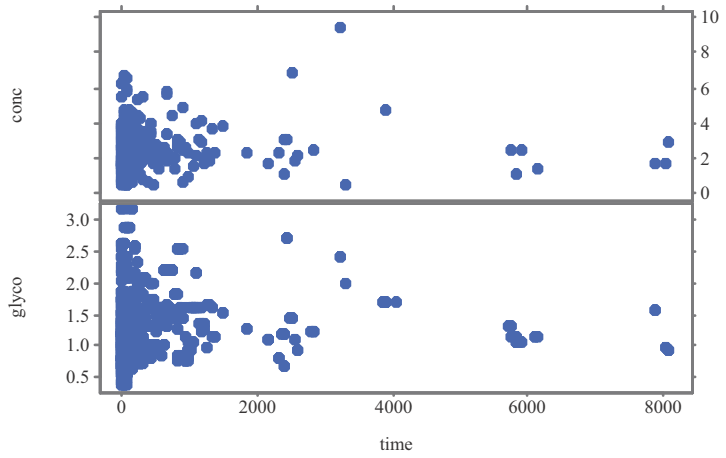


Figure 1.5: *Row plot example.*

Column plots, or side-by-side plots, are useful when the y-axis is categorical, and you want to display different summaries in the panels. The Adverse Event Double Dot Plot is a good example of this plot type, where you can display the incidence rate in the first panel and the relative risk with confidence intervals in the next panel. Column plots are created using expressions such as:

```
gom(Type ~ Fuel + Weight + Disp., data = fuel.frame,
  scales = list(
      x = list( majorTickLabelSrt=90),
      y = list(enableMajorGrid = TRUE)
  )
)
```

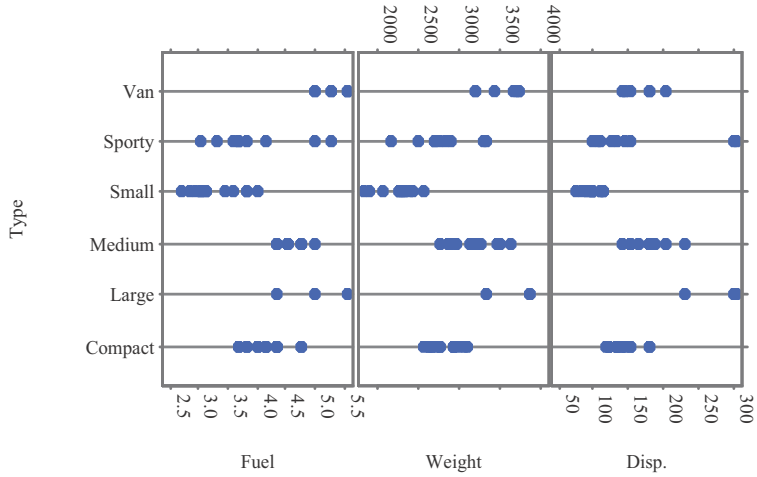


Figure 1.6: *Column plot example.*

While combining Row and Column plots is a rarer scenario, you can use `gom()` to do so. For example:

```
gom(Mileage + Disp. ~ Fuel + Weight , data = fuel.frame)
```

You can create matrix plots using expressions such as:

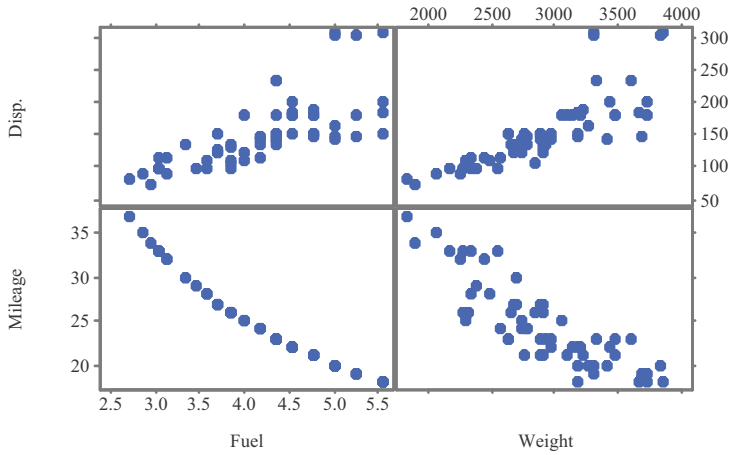


Figure 1.7: *Row-column plot.*

```
gom( ~ Weight + Mileage + Disp., data = fuel.frame)
```

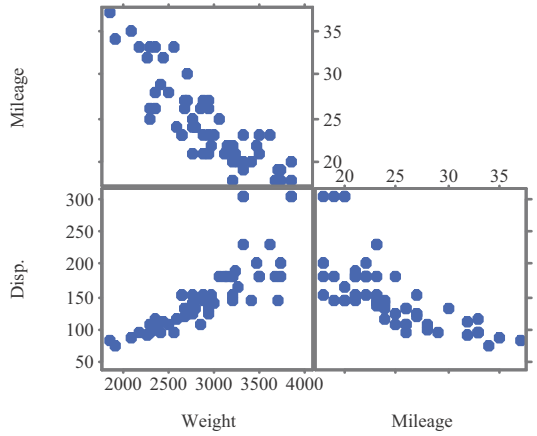


Figure 1.8: *Matrix plot example.*

Graphics Elements by Panel

You can use different graphic elements in the panels. For example:

```
gom( Disp. ~ Mileage + "TOTAL", data = fuel.frame,
  panel = list ( list(ge.points) , list(ge.boxplot) ),
  graphTable = list( columnRatio = c(.9,.1) ),
  scales = list(alternating = 1 )
)
```

This produces a column plot where the first panel is equivalent to `Disp. ~ Time` and the second equivalent to plotting `Disp. ~ "TOTAL"`.

The panel statement

```
panel = list ( list(ge.points) , list(ge.boxplot) ),
```

specifies including points in the first panel and a boxplot in the next.

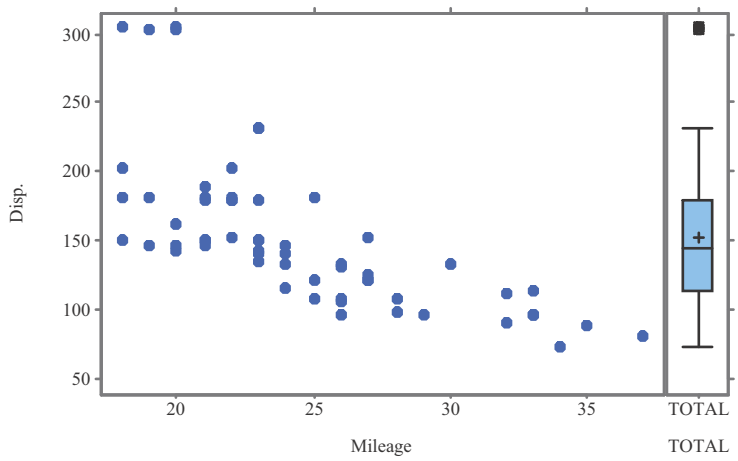


Figure 1.9: *Column plot with varying plot elements.*

Note that `"TOTAL"` is just a shortcut to include a constant (intercept). The `columnRatio = c(.9,.1)` is used to allocate 90% space to the first column and 10% space to the 2nd column, obtaining a wide panel for `Disp. ~ Time` points panel, and a skinny boxplot summary `Disp. ~ "TOTAL"` panel.

THE GOM

2

THE GRAPHICS OBJECT MODEL

This chapter describes the objects and their parameters that comprise the TIBCO Spotfire® Graphics Object Model (GOM). The following provides an illustration how these objects are presented in a graphic.

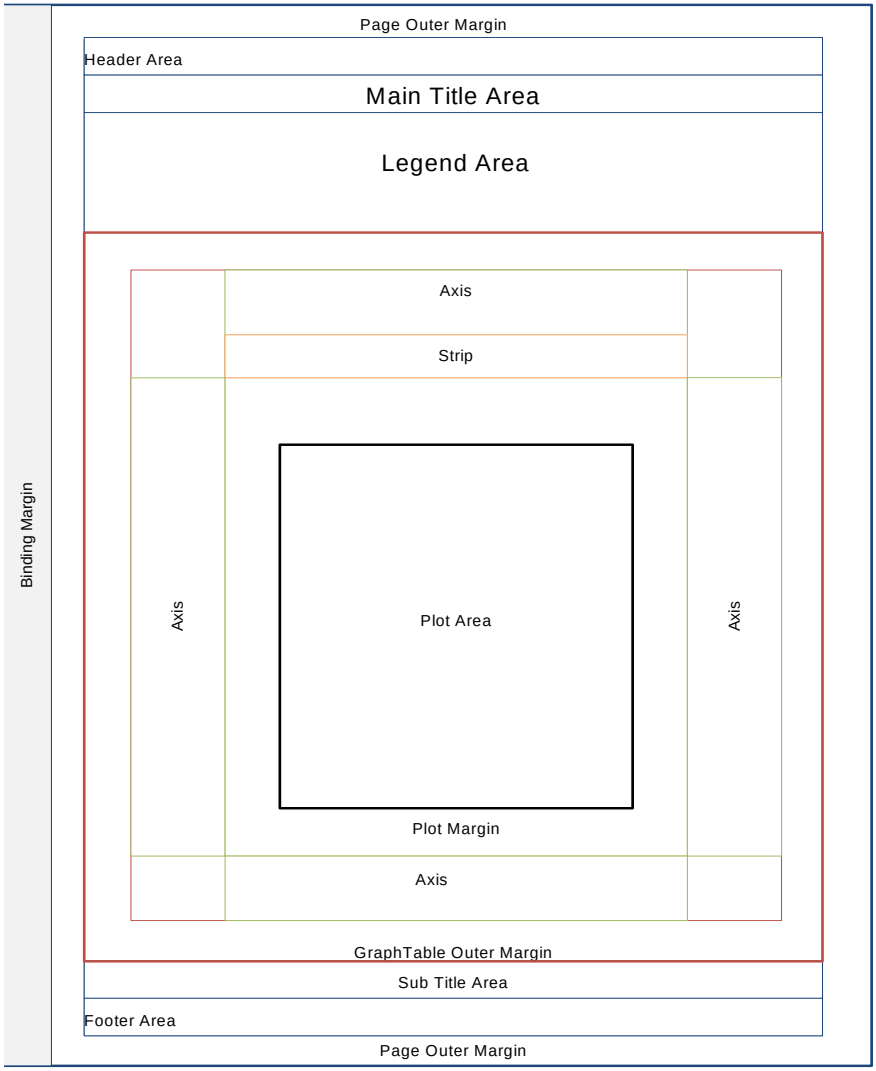


Figure 2.1: *The GOM, illustrated.*

The Page Object

The `page` argument in `gom()` parameterizes the `Page()` creator. The most common settings include the outer margins, headers, footers, and other page annotations.

Table 2.1: *`Page` parameters.*

Parameter	Description
`alternate`	Indicates if binding margin should be alternated between pages. A logical. The default is `FALSE`.
`annotation`	A list of objects. In the GOM Help files, see `gom.segments`, `gom.arrow`, and `MultiLineTextBox`.
`bindingMargin`	Sets the the binding margin to be used with a `Margin` object. The default is: `bindingMargin = Margin(Size(0, "inch"), "left")`
`footer`	Sets the page footers with a `PageTitle` object. For example: `footer = PageTitle(text="Footer", side="bottom")` The `side` default for `footer` is `bottom`.
`header`	Sets the page headers with a `PageTitle` object. The default is `PageTitle(...)`. For example: `header = PageTitle(text="Header", side="top")` The `side` default for `header` is `top`.
Headers and titles	
`height`	Sets the page height with a `Size` object. For example: `height = Size(11, "inch")`
Layout	
`legend`	A list of arguments passed to the `setupGomLegend()` function.
Legend and Annotation	

Table 2.1: *Page parameters. (Continued)*

Parameter	Description
mainTitle	Sets the page headers. The default is PageTitle(...). For example: mainTitle = PageTitle(text="MainTitle", side="top"). The side default for mainTitle is top.
outerMargin	Sets the outer margin that bounds header and footer with a Margin object. By default, the outer margin is specified as follows: outerMargin = Margin(Size(c(0.2,0.2,0.2,0.2), "inch"))
pageStyle	Sets the color style of the page with a list. For more information, see the Help topic for FrameStyle in the GOM reference. For example: pageStyle = FrameStyle(backgroundColor = "lightgrey")
subTitle	Sets the page headers. The default is PageTitle(...). For example: subTitle = PageTitle(text="subtitle", side="top") The side default for subTitle is bottom.
width	Sets the page width with a Size object. For example: width = Size(8.5, "inch")

Example

To illustrate the Page object depicted in Figure 2.2:

1. Provide arguments to the layout and titles.
2. Specify a grey background and a black border.
3. Position the legend inside the graph table area.

```
gom(conc ~ time, groups = ~ Subject, data = Indometh,
  page = list(
```

```
#Set the page margins
outerMargin = Margin( Size(c(.1,0,.1,0),"inch") ),

#Set the binding margin
bindingMargin = Margin( Size(.5,"cm"),
   side = "left"),

#Include a multiline header
header = PageTitle(text = c("Header1",
   "Header2","Header3"),
      side="top"),

#include a main title and set font size
mainTitle = PageTitle(text="Title1", side="top",
   fontSize=Size(18,"pt")),

#include a subtitle at the bottom of page
subTitle = PageTitle(text = "SubTitle",
   side = "bottom"),

#include a footer at the bottom of page
footer = PageTitle(text = "Header1",
   side = "bottom"),

#Change the legends to inside and place it top right
legend = list(numberOfColumns = 3,
         insideLegend = TRUE,
         legendLocation = "top right",
          backgroundColor = "lightgrey",
         border = "grey" ),

#Finally, color the background and include a border
pageStyle = FrameStyle( backgroundColor = "lightgrey",
```

```
        borderColor = "black")
)
)
```

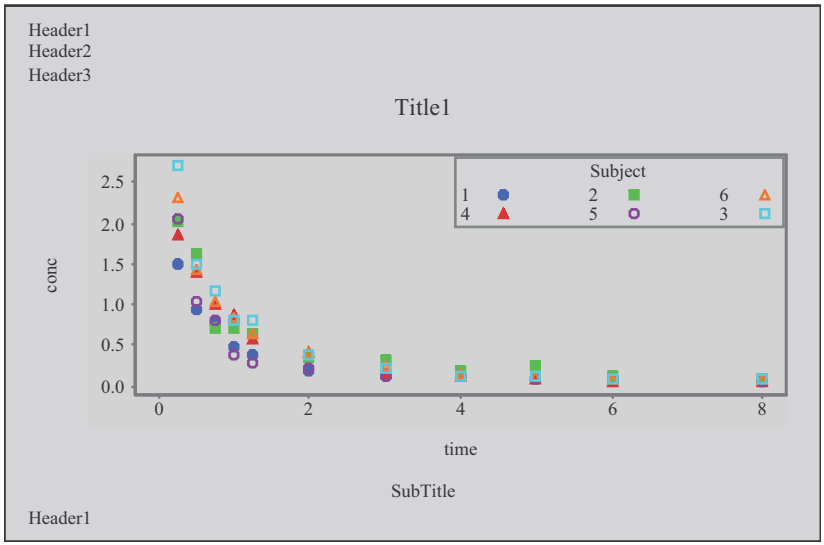


Figure 2.2: *`Page` parameters example.*

The PageTitle Object

The page footers and other titles are handled using the `PageTitle()` creator. A `PageTitle` can include multiple lines. You can position it at the top or at the bottom of the page, and you can specify its style.

Note
Currently, line styling for individual page titles is not supported.

The following table displays the argument available in `PageTitle()`:

Table 2.2: *`PageTitle` parameters.*

Parameter	Description
`fontSize`	Specifies the font size. If `NULL`, `gomOptions("fontSize")` is used. A `Size` object.
`lineSpacing`	Specifies the line spacing. A numeric.
`side`	Specifies the side of the page for the page title. Can be `"top"` or `"bottom"`.
`styles`	Specifies the style of the text. Can be a `list` or `Style` object.
`text`	Specifies the page title's text contents. Can be a character or a `MultiLineText` object.
`titleAdjustment`	Specifies the adjustment. `0` sets left justify, `1` sets right justify, `.5` sets centered text (the default).

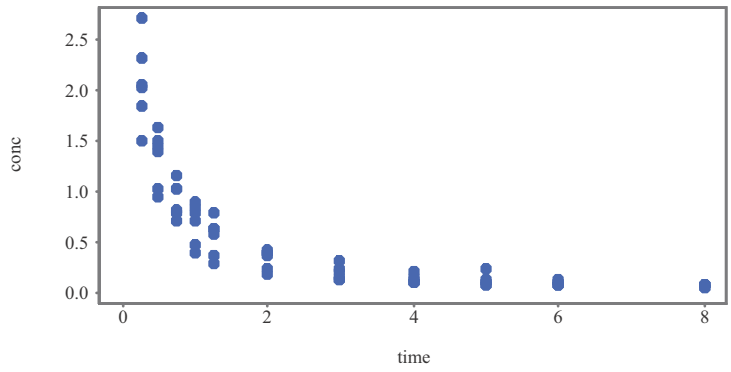
Example

To illustrate the `PageTitle()` creator:

1. Specify the `mainTitle` slot of the `Page` object, but set the `side` argument to `bottom`.
2. Specify titles in red text, with line spacing set to 2, relative to the specified font size of 16 pt.

```
gom( conc ~ time, data = Indometh,
  page =list(
     mainTitle = PageTitle( side = "bottom",
     text=c("header 1","header 2","header 3"),
     lineSpacing = 1.2,
     titleAdjustment = 1,
     fontSize = Size (16, "pt") ,
     styles=list(col=c("red") )
  )
)
```

)



header 1
header 2
header 3

Figure 2.3: *`PageTitle` example.*

GraphTable Object

The *graph table* holds the plot or trellis of panels. It controls the number of columns and rows and the column and row spacing. It also controls the plot aspect ratio. The `GraphTable` object is constructed by passing arguments to it using the `graphTable` argument of `gom()`.

The following table lists the available `GraphTable` parameters:

Table 2.3: *`GraphTable` parameters.*

Parameter	Description
`axesMargins`	Specifies the margins for the graph table's axes. Default is `c(0, 0, 0, 0)`.
`columnRatio`	The ratio column width. A numeric. Default is `1`.
`columnSpacing`	Spacing between columns. A `Size` object. Default is `Size(2, "pt")`.

Table 2.3: *`GraphTable` parameters. (Continued)*

Parameter	Description
`height`	Specifies the height of the graph table. Can be a `Size` object or a numeric in inches. The default fills the graph table area.
`labelMargins`	Specifies the margins for the graph table's labels. Default is `c(0, 0, 0, 0)`.
`layoutType`	Sets the `numberOfRows` and `numberOfColumns` calculation method. Applies when `numberOfRows` or `numberOfColumns` are not given. By default, `numberOfColumns` is determined by the number of levels in the first given variable; `numberOfRows` is the number of levels of the second given variable. • `table` (default): The layout is calculated using `good.layout`. If the page ratio is larger than `1`, then `numberOfRows` is set to the first element and `numberOfColumns` to the second. The revese is true if ratio is less than `1`. • `row`: sets `numberOfRows = 1`. • `col`: sets `numberOfColumns = 1`. • `matrix`: computes `numberOfRows = numberOfCells = ceiling(sqrt(numberOfCells))`.
`numberOfColumns`	Number of columns to use. The default is determined by `layoutType`.
`numberOfPages`	The number of pages the graph table spans. You can use `numberOfPages` to plot only a limited number of pages, where the `numberOfRows` and `numberOfColumns` and given values might imply more.

Table 2.3: *GraphTable parameters. (Continued)*

Parameter	Description
numberOfRows	Number of rows to use. The default is determined by layoutType.
outerMargins	Specifies the outer margins of the graph table. Can be vector of a Size objects or a numeric in inches. The default is c(0.2, 0.2, 0.2, 0.2).
plotAreaStyle	Set the style of the plot area. A list or FrameStyle object. For more information, see the Help topic for FrameStyle in the GOM reference.
plotAspectRatio	The aspect ratio of the plot region. A numeric. Default uses the maximum space available in the plot area.
plotInnerMargins	Specifies the inner plot margins of the graph table plot region. Can be be vector of Size objects or a numeric in inches. The default is c(0, 0, 0, 0).
rowRatio	The ratio row heights. A numeric. Default is 1.
rowSpacing	Spacing between rows. A Size object. Default is Size(2, "pt").
width	Specifies the width of the graph table. Can be a Size object or a numeric in inches. The default fills the graph table area.

Examples

```
#Example 1: Set the graph table layout as a row table.
gom( conc ~ time | Subject, data = Indometh,
  graphTable = list(
  layoutType="row", outerMargins=c(0,2.5,0,2.5)
  )
)
```

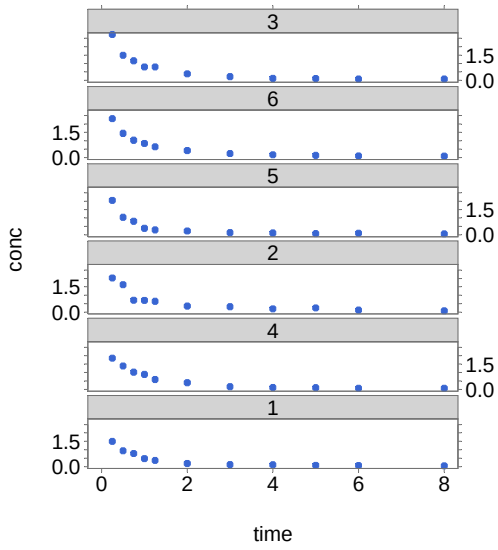


Figure 2.4: *Setting graph table layout.*

```
#Example 2: Set the plot aspect ratio.
gom( conc ~ time | Subject, data = Indometh,
  graphTable=list(plotAspectRatio=2)
)
```

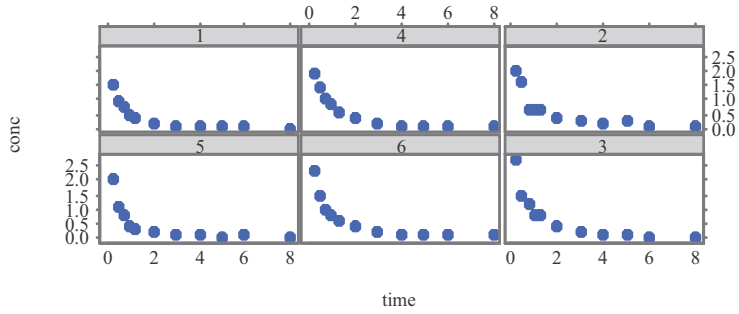


Figure 2.5: *Setting the plot aspect ratio.*

```
#Example 3: Style the graph.
gom(conc ~ time | Subject, data = Indometh,
  graphTable = list(
      rowSpacing =1,
      columnSpacing =1,
      plotAreaStyle =
      FrameStyle(backgroundColor="lightgrey",
         borderColor=gomCol("border"))
      ),
  scales = list( enableMajorGrid=TRUE,
       majorGridStyle=list(col="white")
  )
)
```

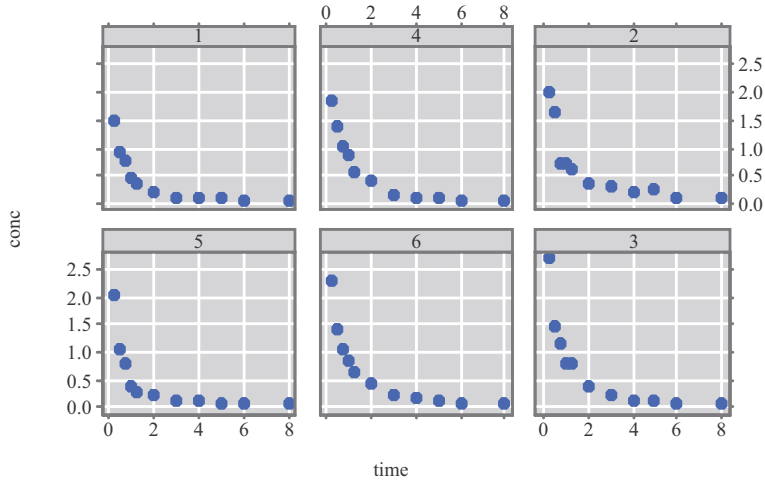


Figure 2.6: *Styling the graph table.*

Strip Object

You can control the panel strip labels using the `strips` argument to `gom()`. The value for this parameter is passed down to the `Strip()` creator. You can control these strip features, among others:

- If the name of the given variable should be displayed.
- If the strip should be on the right-hand side of the panel.
- If the strip should be inside or outside the axis.
- How the strip should be styled.

The following table displays the `Strip` parameters available.

Table 2.4: *`Strip` arguments.*

Parameter	Description
`enabled`	Indicates if the strip should be included. A logical. The default is `TRUE`.
`fontSize`	Specifies the size of the font used in the strip label.
`format`	Specifies parameters in name = value to be passed down to `gom.formatCharacter` with `x` set to `labels`. A list.
`frameStyle`	Specifies the style of the strip frame. For more information, see the Help topic for `FrameStyle` in the GOM reference.
`labelRotation`	Specifies the rotation of the label. The default is determined by `side`: • `top` = `0` • `bottom` = `0` • `left` = `90` • right = `-90`
`labelStyle`	Specifies the style of the strip label. A `list`.

Table 2.4: *`Strip` arguments. (Continued)*

Parameter	Description
`outerStrips`	Indicates that the strip should be rendered outside of the axis. A logical. The default is `FALSE`.
`side`	Indicates the side of the plot for the axis (`bottom`, `left`, `top` (the default), or `right`). A character.
`text`	Indicates that the strip label has optional string field parameters. The default is `[value]`. • To display the label and the value, specify `[label]: [value]`. • To display no text in the strip, specify `text=""`.

Example

```
gom( conc ~ time | Subject, data = Indometh,
  graphTable=list(layoutType="row",
      outerMargins=c(0,1,0,1)),
  scales = list(alternating =1, mirrorTicks = FALSE),
  #include right sided strips,
  #showing the given label on a white background
  strips = list(text="[label]: [value]",
      outerStrip=TRUE,
      side = "right",
      frameStyle=FrameStyle(backgroundColor="white")
  )
)
```

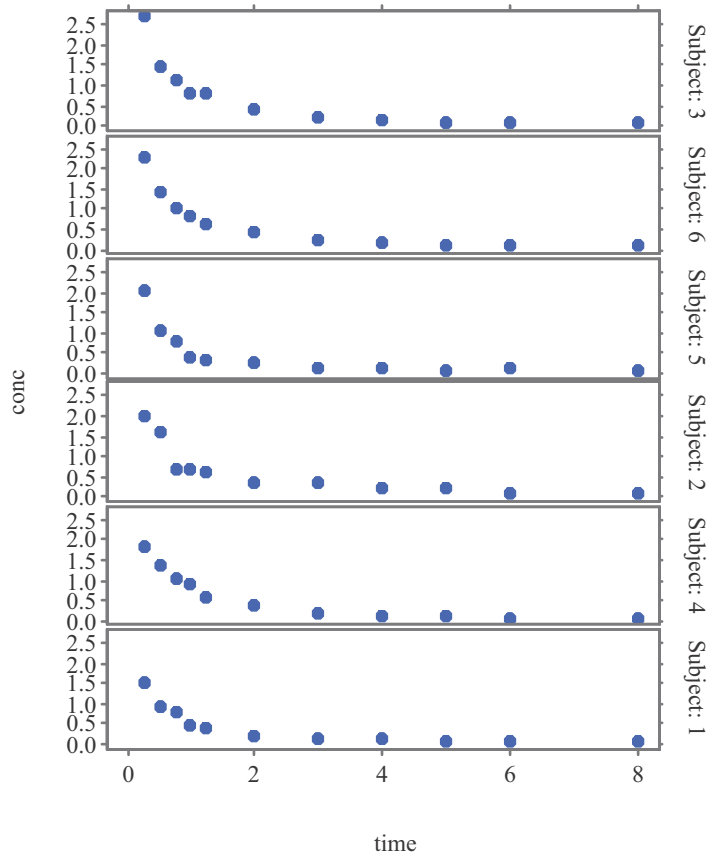


Figure 2.7: *Styling the strip label.*

Scale Object

The GOM attempts to find appropriate scales for the x and y axis. You can specify scale settings providing parameters to the `scales` argument of `gom()` in the form of named lists to the x and y axis. For example:

```
scales = list (
  x = list( from = 0) ,
  y = list( logTransform = 0)
)
```

The `x` list controls the x-axis, and the `y` list controls the y-axis.

The GOM supports fine control of independent axes. For example, for a Trellis plot with free scales, you can get fine control by nesting the `scales` arguments to a deeper level. For example, to set three free x-axis to start from `0`, `1` and `3`, use the following:

```
scales = list (
  x = list(
      list( from = 0 ),
      list( from = 1 ),
      list( from = 3 )
  )
)
```

Table 2.5: *`Scale` parameters.*

Parameter	Description
`alternating`	Determines whether axes alternate from one side of the group of panels to the other. A numeric. Using the default `alternating = 1` can save space if long tick labels are used. For more precise control, `alternating` can be a vector specifying the side of the plot on which each axis is drawn: • `alternating=1` specifies bottom or left only. • `alternating=2` specifies top or right only. • `alternating=0` specifies do not draw. • `alternating=c(1,2)` specifies bottom-top (the default) or left-right alternation. • `alternating=c(2,1)` specifies top-bottom or right-left alternation.
`alternatingTicks`	Indicates whether ticks should be alternated. A logical. The default is `TRUE`.

Table 2.5: *Scale parameters. (Continued)*

Parameter	Description
at	The numeric vector of positions at which the ticks and tick labels are plotted. If at is omitted, gom.prettyValues (if linear) or gom.prettyLogValues (if log scale) is used to calculate at values.
axisLineCol	Specifies the color base line that the tick rests on. A numeric.
clipMajorLabels	Indicates if a major label should be clipped or skipped so the labels fit within the room allocated to the axis. A logical. The default is FALSE.
enabled	Indicates whether the axis should be visible. A logical. The default is TRUE.
enableMajorGrid	Indicates if the grid at major tick marks should be visible. A logical. The default is FALSE.
enableMajorLabels	Indicates if labels are visible. A logical. The default is TRUE.
enableMajorTicks	Indicates if major tick marks are visible. A logical. The default is TRUE.
enableMinorGrid	Indicates if the grid at minor tick marks should be visible. A logical. The default is FALSE.
enableMinorTicks	Indicates if minor tick marks should be visible. A logical. The default is FALSE.

Table 2.5: *`Scale` parameters. (Continued)*

Parameter	Description
`extensionFactor`	Extends the data limits by `extensionFactor` on each end, and then labels the axis internally. A non-negative numeric. The default is `0.04`, or `4%` extension.
`firstMajorTick`	Sets the first tick mark position. A numeric. **Note:** For log scales, `firstMajorTick` is the lower bound of major ticks.
`from`	Specifies the lower scale limit. A numeric. If omitted, `from` is equal to the minimum of `limits`.
`lastMajorTick`	Sets the last tick mark position. A numeric. **Note:** For log scales, `lastMajorTick` is the upper bound of major ticks.
`limits`	A numeric vector of length two, specifying the lower and upper data limits. Defaults to the data range and what is returned by the graphic element prerendering (if anything is returned). **Note**: The `limits` setting is conditioned to the largest of the setting and the data and prerender ranges. To truncate the scale range, see `to` and `from` parameters.
`logTransform`	Indicates if the scale should be log or linear. A logical. The default is `FALSE`, indicating linear.
`logTransformBase`	The base of the log transform. A numeric. Typically, this is `2`,`exp(1)` or `10`. This parameter has an effect only when `logTransform = TRUE`.

Table 2.5: *Scale parameters. (Continued)*

Parameter	Description
majorGridStyle	Specifies the style of the major grid. A list of parameters of the name = value form, where valid parameters are col, lty, and lwd. This parameter has an effect only when enableMajorGrid = TRUE. Default is lightgrey thin solid lines. For example: majorGridStyle=list(col="blue", lty=2, lwd=3)
majorTickLabel	Specifies the tick label. If you set this value, you must specify at. A character.
majorTickLabelAdj	Specifies the tick label justification. A non-negative numeric typically less than 1.
majorTickLabelCol	Specifies the major tick label color. A numeric or character.
majorTickLabelFontSize	Specifies the font size. A Size object.
majorTickLabelFormat	If type is numeric, then the list of arguments is passed to gom.formatNumeric; otherwise, the list of arguments is passed to gom.formatCharacter.
majorTickLabelSrt	Specifies major tick label rotation in degrees measured clockwise (between 90 and -90). A numeric. The default is 0.
majorTickSize	Specifies the size of major tick marks. If the value is positive, tick marks are drawn outside of the plot area. If the value is negative, the tick marks are drawn inside the plot area. A numeric. The default is 1.

Table 2.5: *`Scale` parameters. (Continued)*

Parameter	Description
`majorTickUnit`	Specifies the unit interval for major tick marks. A numeric.
`minorGridStyle`	Specifies the style of the minor grid. A list of parameters of the name = value form, where valid parameters are `col`, `lty`, and `lwd`. This parameter has an effect only when `enableMinorGrid = TRUE`. Default is lightgrey thin dotted lines.
`minorTickSize`	Specifies the size of minor tick marks. If the value is positive, tick marks are drawn outside of the plot area. If the value is negative, the tick marks are drawn inside the plot area. A numeric. The default is `0.5`.
`minorTicksUnit`	Specifies the unit interval for minor tick marks. A numeric.
`mirrorLabels`	Indicates whether tick mark labels should be mirrored. A logical. The default is `FALSE`.
`mirrorTicks`	Indicates whether tick marks (both major and minor) should be mirrored. A logical. The default is `TRUE`.
`numberOfMajorTicks`	Specifies the approximate number of major tick marks desired. An integer.
`numberOfMinorTicks`	Specifies the number of minor tick marks within major tick marks. An integer.

Table 2.5: *`Scale` parameters. (Continued)*

Parameter	Description
`relation`	Controls the relationship between axes on the same side: • `relation = "same"` (default) gives identical vertical or horizontal axes on each panel. The axis is drawn in rows and columns. • `relation = "free"` gives independent vertical or horizontal axes on each panel. Each panel cell has an axis.
`symmetricLimits`	Specifies whether the scale limits should be symmetric around `symmetricLimitsBase`. A logical.
`symmetricLimitsBase`	Specifies the base of symmetry. A numeric. The default is `0` if `logTransform = FALSE`, and `1` if `logTransform = TRUE`. Use this setting only if you set `symmetricLimits = TRUE`.
`tickCol`	Specifies the color for all ticks. A numeric or character.
`title`	Specifies the title placeholder of the axis. A character string.
`to`	Specifies the upper scale limit. If omitted, `to` is equal to maximum of `limits`. A numeric.
`type`	The default is the data-class of the axis variable (numeric or factor).

Sort, Filter and Bin Specifications

The `gom()` function has built in data operations for sorting labels and binning and filtering of data prior to any graphing. Each data operation specification takes the form of the nested lists-of-lists. For example, the arguments are of the following form:

```
sortSpec = list (
  list( targetColumn = "year", sortBy="yield" ),
  list( targetColumn = "site", sortBy="yield" )
)
```

sortSpec

The following table describes the parameters in name-value form that are passed down to the sorting procedure for sorting the levels of factors.

Table 2.6: *Data specifications for* `sortSpec`.

Parameter	Description
`smallToLarge`	The sort direction. Default is `TRUE`. A logical.
`sortByColumn`	Column name. The values to sort the levels of `targetColumn` by. Default is unspecified `NULL`, in which case it is the alphabetic order of `targetColumn` that is used.
`summaryFunction`	A summary function to choose order. Default is `max`. Available are `mean`, `min`, `max`, `median`, and `gom.count`
`targetColumn`	Column name. The factor for which levels are sorted.
`withinColumn`	Column name. The column to sort within. (Default is `NULL`.)
`withinColumnLevel`	The level of `withinColumn` to sort within (required if `withinColumn` is set). A character.

filterSpec

The following table describes the parameters in name-value form to be passed down to the filtering procedure. These are not exposed in the TSCG client.

Table 2.7: *Data specifications for* `filterSpec`.

Parameter	Description
`direction`	Sets the direction to either `"top"` or `"bottom"`. The Default is unspecified `NULL`. A character string.
`directionN`	Sets the number of rows to select (For example, select the top 10). The default is `10`. An integer.
`filterByColumn`	The column name specifying the values to filter by.
`includeLower`	If `FALSE` (the default), it is greater than `lowerValue`. If `TRUE`, greater than or equal to `lowerValue`. A logical.
`includeUpper`	If `FALSE` (the default), less than `upperValue`. If `TRUE`, less than or equal to `upperValue`. A logical.
`lowerValue`	The lower cutoff of `filterByColumn`. The default is unspecified `NULL`. Applies only if `direction` is unspecified `NULL`. A numeric.
`targetColumn`	Factor to select levels to filter by.
`upperValue`	The upper cutoff of `filterByColumn`. The default is unspecified `NULL`. Applies only if direction is unspecified `NULL`.

binSpec

The following table describes the parameters in name-value form to be passed down to the binning procedure. These are not exposed in the TSCG client.

Table 2.8: *Data specifications for* `binSpec`.

Parameter	Description
`binmethod`	Sets the method for determining the number of bins to use. A character string. Options include: • `"scott"` specifies the Scott method. • `"sturges"` specifies the Sturges method. • `"fd"` specifies the Freedman-Diaconis method. • `"pretty"` specifies dividing the numeric in to pretty bins.
`binunit`	Sets the binning unit. A numeric. The default is `NULL` for unspecified. If specified, `nbins` and `binmethod` are ignored.
`breaks`	A vector defining the bin ranges. A numeric.
`equalwidth`	If `TRUE` (the default), bins are defined by equal ranges. If `FALSE`, bins are defined by equal counts. A logical.
`nbins`	Sets the number of bins. An integer. The default is `8`.
`replace`	If `TRUE` (the default), the new bin column replaces the existing column. If `FALSE`, the new bin column is appended to the dataset. A logical.
`suffix`	If `replace` is `FALSE`, `suffix` is appended to the `targetColumn` name. A character string.
`targetColumn`	The numeric column to be binned. A character string.

TIBCO SPOTFIRE CLINICAL GRAPHICS

3

HOW TSCG USES THE GOM

This section discusses the TIBCO Spotfire Clinical Graphics (TSCG) application, the graphical user interface (GUI) designed to provide nonprogrammers the tools to create graphs.

The following workflows help introduce using TSCG:

- *TSCG Graph Definition*: A user defines a graph, and then the end users reuse the graph with the TSCG client.
- *TSCG Production Run*: SAS specifies graph output in a production system, supplying the override parameters at run-time (automated graph production).

Both work flows use a common process to generate graph files on the server, and then render the graphs either in the TSCG client (Graph Definition) or on the server (Production) as part of the production process. See Generating a Graph File from a Graph Definition in the PDF *Working in Production Mode Using TSCG and the Spotfire Statistics Services* for more information about this process.

THE TSCG GRAPH DEFINITION

Users need no knowledge of the S-PLUS programming language to create graphs with TSCG. Using the GUI, users can select the data, the graph type, and the plot elements for graphing.

Creating a Graph Using the TSCG Client

Creating a graph in TSCG involves three steps:

- Defining the graph.
- Generating the graph.
- Rendering the graph.

The three steps of this process are detailed below the figure.

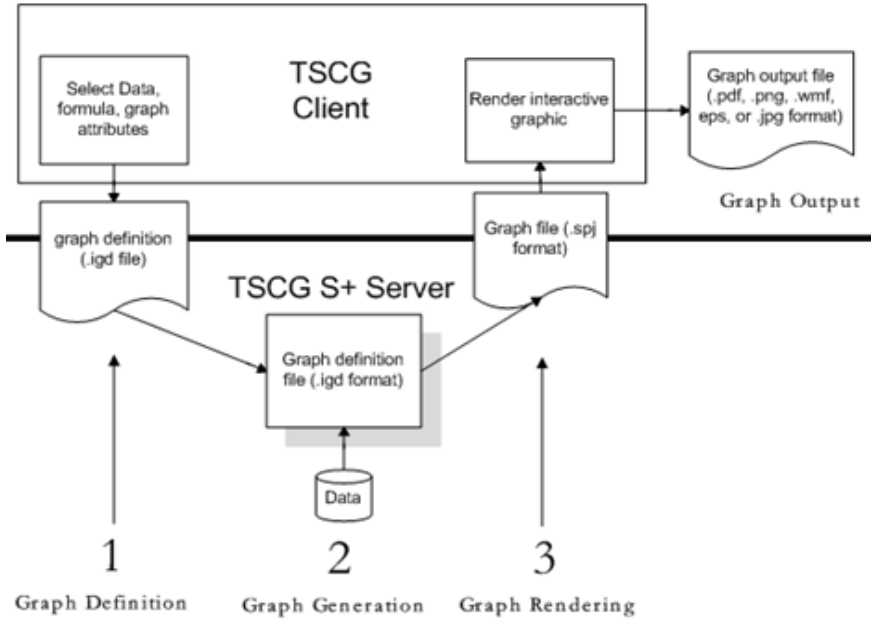


Figure 3.1: *Creating a grasph using the TSCG Client*

1. *Defining a graph*: Using the TSCG client, select the data, the graph type (dot, histogram, Kaplan-Meier, and so on), and the optional and common graphic elements (smoothers, axis scale, line style, color, and so on). After you select these items, you can use TSCG to save the data path and graph

information to a file (with an **.igd** file extension). Use this file for production or open it with TSCG to edit the original selections.

2. *Generating a graph*: The server transforms the **.igd** file into an SPJ file.
3. *Rendering a graph*: The TSCG client reads the SPJ file and renders it in an interactive graphlet viewer in the TSCG user interface. You can continue editing the file, or you can save an output file, save the graph definition, or define a template graph definition.

Automated Graph Creation in SAS Production Mode

Follow the same three steps as described in the section Creating a Graph Using the TSCG Client: Define the workflow, generate the graph, and render the graph in this workflow.

For more information about working in the production mode and generating a graph file from a graph defintion, see the PDF *Working in Production Mode* included in the documentation.

Notes
The PDF Working in Production Mode also contains sample graph definition documents and the Graphlet (SPJ) file .

GRAPHIC ELEMENTS

4

INTRODUCTION

Graphic elements are essentially the same as Trellis panel functions: low level S graphics functions wrapped in a function. The wrapper here is the `Groto` object. (`Groto` is short for "Graphic Prototype", like Proto is often used for prototype object programing.) The object wrapper makes it structured and manageable. We include more information about its behavior, metadata, and templates for default arguments. This is not possible with S functions.

```
class(ge.points)
```

```
[1] "Groto"
```

All `Groto` objects can be listed by using `ls.groto()`:

```
ls.groto()
```

```
                data.class    dataset.date
   ge.areaBars  Groto         2008.12.04  21:44
   ge.areas     Groto         2008.12.04  21:44
   ge.axisLabel Groto         2008.12.04  21:44
   ge.bars      Groto         2008.12.04  21:44
   ge.blank     Groto         2008.12.04  21:44
   ge.boxplot   Groto         2008.12.04  21:44
   ge.bubbles   Groto         2008.12.04  21:44
   ge.cifit     Groto         2008.12.04  21:44
   ge.contour   Groto         2008.12.04  21:44
   ge.delta     Groto         2008.12.04  21:44
   ge.density   Groto         2008.12.04  21:44
   ge.dots      Groto         2008.12.04  21:44
   ge.duration  Groto         2008.12.04  21:44
. . .
```

```
nrow(ls.groto())
```

```
[1] 66
```

You can find help for each groto by typing `?`*`grotoname`*; for example, `?ge.points` in the command line. Arguments and defaults are displayed when you print a `Groto` object:

```
ge.points
```

```
object of class Groto:
call:
function(pointsPointStyle = pointStyle(col =
  gomCol("normal"), cex = 0.8, pch = gomPch("default"),
  type = "p"),
      pointsSortOnX = TRUE,
      pointsJitterX = FALSE,
      pointsJitterY = FALSE,
      pointsDoGroup = TRUE,
      pointsId = NULL,
      pointsSubset = NULL,
      pointsPageAction = NULL)


NULL
```

The `Groto` object extends functions and returns a `Groto` clone with new defaults. You create new `Groto` objects with each `gom()` call, as is shown by the following creations and reuse:

```
#set color
ge.myPoints <- ge.points(pointsPointStyle =
  pointStyle(col= gomCol("spectral")))
#set size
ge.myPoints <- ge.myPoints(pointsPointStyle =
  pointStyle( cex=seq(.2,2,length=8)))
#set symbol
ge.myPoints <- ge.myPoints(pointsPointStyle =
  pointStyle( pch = 16 ))

gom(Fuel ~ Weight, groups = ~Disp., data = fuel.frame,
  panel = ge.myPoints,
  page= list( legend = list(legendLocation="right top"))
)
```

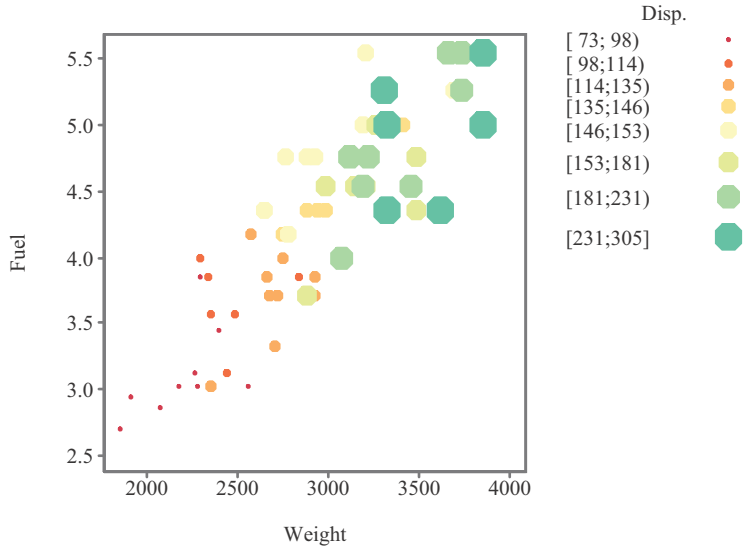


Figure 4.1: *Bubble plot with color scale.*

CREATING GRAPHIC ELEMENTS

You can create new elements using the `GraphicElement()` creator with `name` and `render` as required arguments.

```
ge.helloworld <- GraphicElement(name = "Hello World",
  render = function(...)
  {
      text(3000, 4, "hello world")
}
)

gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.helloworld
)
```

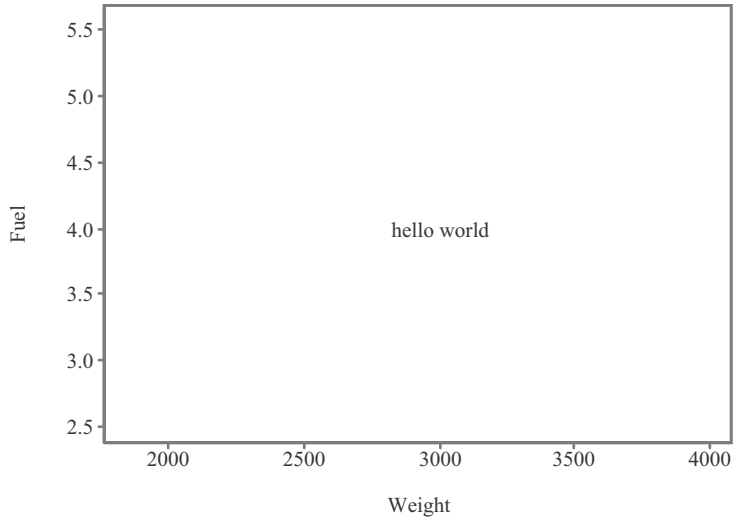


Figure 4.2: *Hello World Example.*

The scales in Figure 4.2 are driven by the input `Weight` and `Fuel` variables. That is the default behavior. You can override the limits by including a `preRender` function. In the following example, set the limits to `0 1` and print text in the center of the graph.

```
ge.helloworld <- GraphicElement(name = "Hello World",
  render = function(...)
  {
     text(.5, .5, "hello world")
  },
  preRender = function(...)
  {
     list(xlim = c(0,1), ylim=c(0,1))
  }
)


gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.helloworld)
```

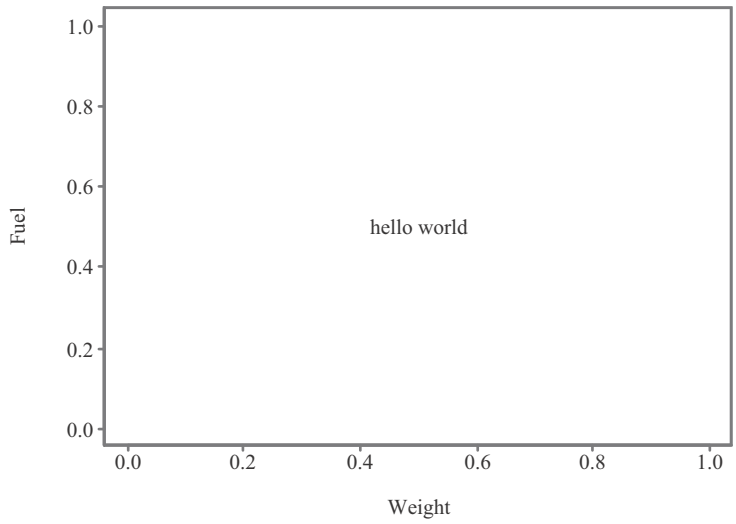


Figure 4.3: *Hello World, rescaled.*

The GOM needs a `preRender` function to calculate scales before it can render the plot. If several graphic elements are combined, and all are setting their own ranges, the range of these ranges is used. This design is illustrated as follows:

```
gom(Fuel ~ Weight, data = fuel.frame,
  panel = list(ge.points, ge.helloworld)
)
```

The above example is trivial and does not use the input data; however, it is general enough to classify as an annotation element, which is useful for including a stamp or a tag on a graph. Usually, you would want data-driven elements. Points, lines, and box-whisker symbols are all driven by input data (x and y). The above `ge.helloworld` example uses the ellipsis argument (...). As with any S function, this ellipsis just means "pass it on." You can print the contents of the ellipsis as follows.

```
ge.printArguments <- GraphicElement(name = "Hello World",
  render = function(...)
  {
      print("RENDER")
      print(names(list(...)))
  }
  , preRender = function(...)
  {
      print("PRERENDER")
      print(names(list(...)))
  }
)
gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.printArguments
)

[1] "PRERENDER"
 [1] "x"
 [2] "y"
 [3] "subscripts"
 [4] "groups"
 [5] "ggp"
 [6] "horizontal"
 [7] "xLogTransform"
 [8] "xLogTransformBase"
 [9] "yLogTransform"
[10] "yLogTransformBase"
[11] "isXSideBlank"
[12] "isYSideBlank"
[13] "inputData"

[1] "RENDER"
 [1] "x"
 [2] "y"
 [3] "subscripts"
```

```
 [4] "groups"
 [5] "ggp"
 [6] "horizontal"
 [7] "xLogTransform"
 [8] "xLogTransformBase"
 [9] "yLogTransform"
[10] "yLogTransformBase"
[11] "isXSideBlank"
[12] "isYSideBlank"
```

In the above example, the most important are the `x` and `y`. These correspond to the formula `y ~ x`. Also, you can call the position vectors, illustrated by the following:

```
ge.helloworld <- GraphicElement(name = "xy Hello World",
  render = function(x,y, ...)
  {
     text(x,y, "Hello World")
  }
)
```

```
gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.helloworld
)
```

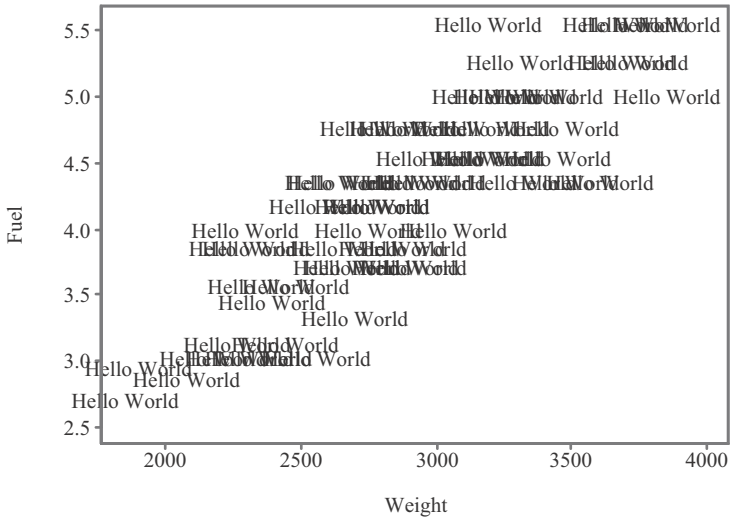


Figure 4.4: *Position driven by data.*

The groups can be used as in the following:

```
ge.helloworld <- GraphicElement(name = "xy Hello World",
  render = function(x,y,groups, ...)
  {
     text(x,y, as.character( groups), col = groups)
  }
)

gom(Fuel ~ Weight, groups = ~Type,  data = fuel.frame,
  panel = ge.helloworld
)
```

Note that the legend is not included automatically. You can specify that option and use a `style` argument. This task is out of the scope for this document. For more information, see the Help file for `GraphicElement`.

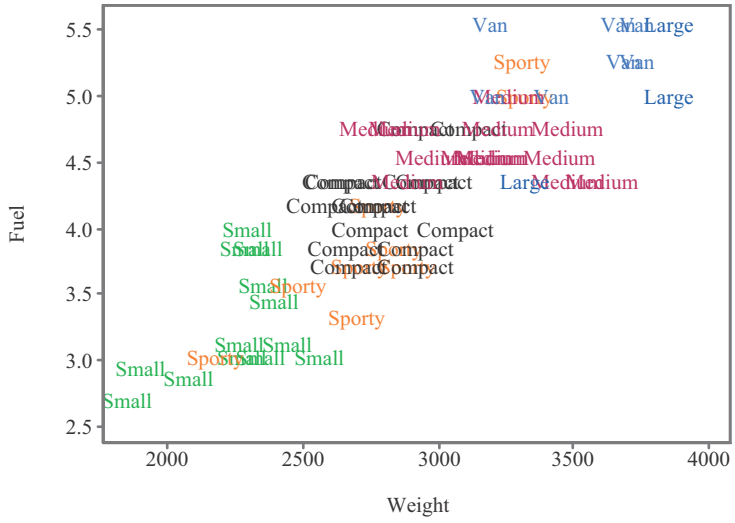


Figure 4.5: *Label and color driven by groups.*

ARGUMENTS

This section illustrates passing arguments to a graphic element.

Start by revisiting the "Hello World" example, described in the section Creating Graphic Elements, except this time, pass in the `label`, `col` and `cex` to `text()`:

```
ge.helloworld <- GraphicElement(name = "Hello World",
  render = function(helloworldText = "Hello World",
     col = "blue", cex = 1, ...)
  {
     text(0.5, 0.5, label = helloworldText, col = col,
        cex = cex)
  },
  preRender = function(...)
  {
     list(xlim = c(0, 1), ylim = c(0, 1))
  }
)
```

Printing `ge.helloworld` displays arguments and their defaults. Note that the ellipsis (…) is not included.

```
ge.helloworld()

object of class Groto:
call:
function(helloworldText = "Hello World", col = "blue",
  cex = 1)
NULL
```

You can use it with defaults or set arguments, as you would with any built-in graphic element:

```
ge.helloworld(helloworldText = "Hello GOM", col = "red",
  cex = 5)

object of class Groto:
call:
function(helloworldText = "Hello GOM", col = "red", cex = 5
  )
NULL
```

```
gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.helloworld(
      helloworldText = "Hello GOM", col = "red", cex = 5)
)
```

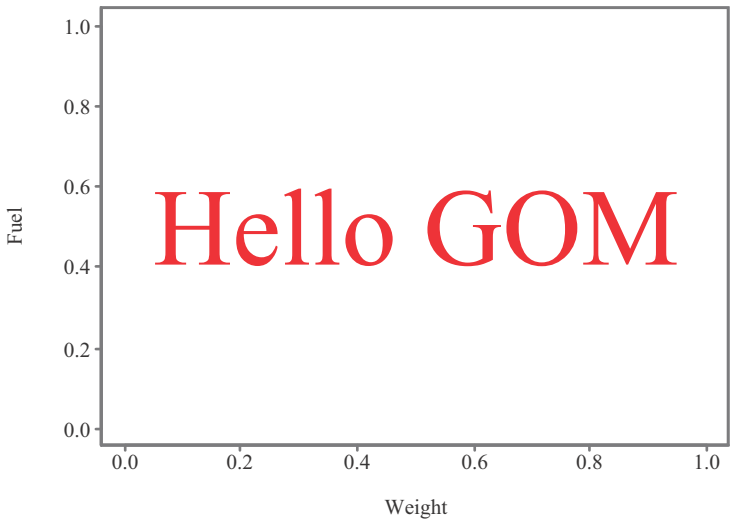


Figure 4.6: *Passing on parameters.*

Arguments must contain a default value. Arguments with missing values are considered "private" and cannot be set by users. For example, `x` and `y` are private, as are subscripts. An argument can always be given the value `NULL`. The next section provides an example.

PASSING DATA AROUND

Using variables in the data set, we can create multi-dimensional plots, include labels, and include other annotation tasks. The data are passed to the `preRender` function using the `inputData` argument. Use the `subscript` argument to slice the data down to the specified panel.

You can use the utility function `preRenderVariableValues()` to extract variables contained in the `inputData` data.frame. For example:

```
ge.helloworld <- GraphicElement(name = "Hello World",
  render = function(x, y, helloworldTextColumn = NULL,
  col = "blue", cex = 1, ...)
  {
     text(x, y, label = helloworldTextColumn, col = col,
     cex = cex)
  },
  preRender = function(x, y, inputData, subscripts,
     helloworldTextColumn = NULL, ...)
  {
     #extract values
     values = preRenderVariableValues(
        variable = helloworldTextColumn,
        inputData = inputData,
        subscripts = subscripts)

     #pass it on
     list(xlim = range(x), ylim = range(y),
        helloworldTextColumn = values)
  }
)
```

```
gom(Fuel ~ Weight, data = fuel.frame,
  panel = ge.helloworld(
      helloworldTextColumn = "Disp.", col = 1:16,
          cex = 0.8)
)
```

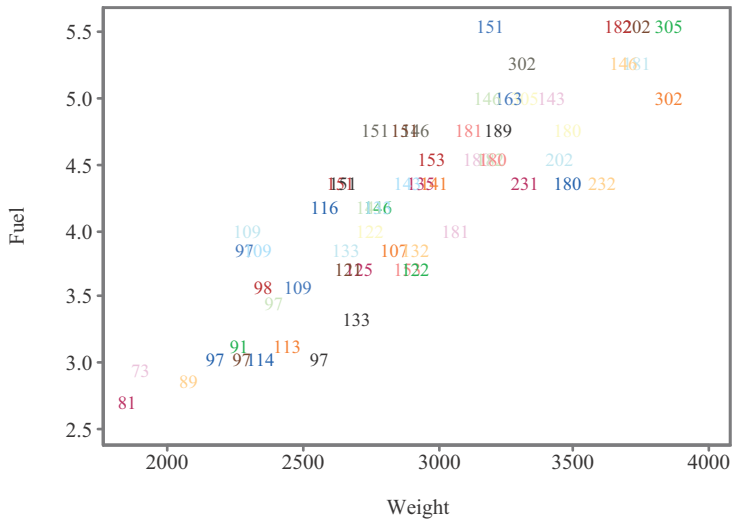


Figure 4.7: *Using input variables as labels.*

In the above example, you extract data from `inputData`, and then pass the extracted data on to the render step. You can set or change all parameters in the render step in the `preRender` step. This practice is used often in the built-in graphic elements, such as `ge.loessfit`. In that element, we compute loess curves that might exceed the data range; therefore, we must compute the loess fit in the `preRender` step. We just pass on the lines to the render step, where we can avoid refitting the curve.

The communication between the `preRender` and render step is global within the panel. That is, graphic element 1 can override (or pass on) arguments in element 2. This design highlights the importance of careful argument naming, but it also introduces flexibility and reuse of data.

```
ge.helloworld1 <- GraphicElement(name = "Hello World",
  render = function(x, y, helloworldTextColumn = NULL,
  col = "blue", cex = 1, ...)
  {
     # Do nothing
  },
  preRender = function(x, y, inputData, subscripts,
     helloworldTextColumn = NULL, ...)
  {
     #Get the data and pass it on
     values = preRenderVariableValues(variable =
        helloworldTextColumn,
        inputData = inputData,
        subscripts = subscripts)
     list(xlim = range(x), ylim = range(y),
     helloworldTextColumn = values)
  }
)

ge.helloworld2 <- GraphicElement(name = "Hello World",
  render = function(x, y, helloworldTextColumn = NULL,
     col = "blue", cex = 1, ...)
  {
     #Render but have no preRender
     text(x, y, label = helloworldTextColumn, col = col,
     cex = cex)
  }
)

gom(Fuel ~ Weight, data = fuel.frame,
  panel = list(
     ge.helloworld1(helloworldTextColumn = "Disp."),
     ge.helloworld2)
)
```

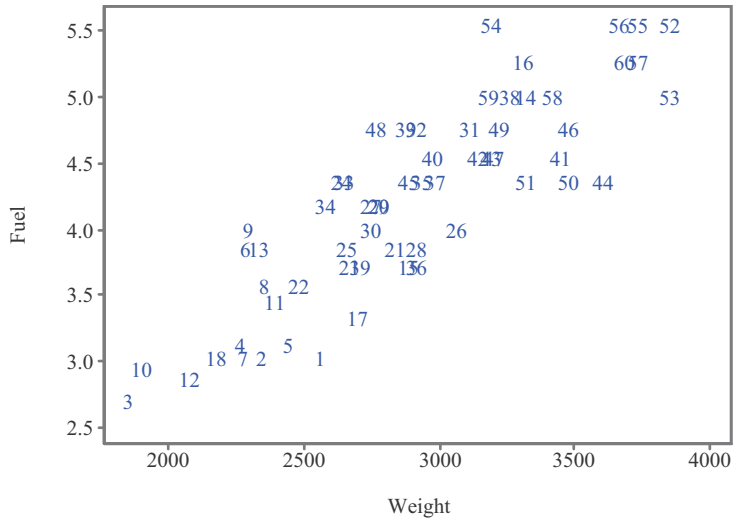


Figure 4.8: *Passing on arguments from one element to another.*

Using the `preRenderVariableValues()` function, you have the advantage of specifying a variable argument as either a character or a formula. If you specify a formula, you can pass on expressions, such as `~ log(Disp.)`. If you specify a character, the name is case insensitive and the match is done after removing non-character symbols. For example, the variable name can be `"DISP." = "dIsP."="DISP_"` . The `preRenderVariableValues()` also includes type checking (for example, factor or numeric) and an argument to stop on error.

INDEX

H

I

L

M

N

O

P

R