recreating-plot-with-tlf

Pierre Chelle

September 9, 2021

Setting up tlf and loading the data

```
require(tlf)
#> Loading required package: tlf
require(ospsuite)
#> Loading required package: ospsuite
#> Warning: package 'ospsuite' was built under R version 4.0.5
#> Loading required package: rClr
#> Warning: package 'rClr' was built under R version 4.0.3
#> Loading the dynamic library for Microsoft .NET runtime...
#> Loaded Common Language Runtime version 4.0.30319.42000
```

Properties of plot aesthetics are managed by `PlotConfiguration` objects and derived sub-classes (see vignette “plot-configuration” for more details). Default properties of `PlotConfiguration` objects are managed by `Theme` objects that can be exported as json files. When loading the `tlf` package, the current theme uses basic properties which lead to basic aesthetics (e.g. labels all use the same angle, leading to *ylabel* not well oriented). Consequently, a first step in using `tlf` is to load a json file that includes desired default aesthetic properties with the function `loadThemeFromJson` and to use its properties as the current default with `useTheme` (as shown in the example below).

```
useTheme(loadThemeFromJson("esqlabs-theme.json"))
```

To create your own theme, you can use the `shiny` UI called by the function `runThemeMaker()` (see vignette “theme-maker” for more details). Another way is to copy and edit the template theme available from `system.file("extdata", "template-theme.json", package = "tlf")`.

Note that some default properties can also be changed after the theme is loaded using functions of the form `setDefaultXX`. This is the case for the legend position (`setDefaultLegendPosition`) and the watermark (`setDefaultWatermark`) as illustrated below. Briefly, the enum `LegendPositions` provides all the available legend positions. Besides, *watermark* and plot labels (*title*, *xlabel*, *ylabel*…) are `Label` class objects whose properties include fonts. However, users can provide either `Label` or `character` variables as input argument. If the input is of class `character`, the `tlf` package will use the fonts of the theme for the label or watermark.

```
setDefaultLegendPosition(LegendPositions$insideTopRight)
setDefaultWatermark(Label$new(text = "esqLabs", size = 20, angle = 30))
```

The code chunks below aim at loading the same data that was used in the esqLabs example.

Observed data is stored in an Excel sheet and is read using a function from package `readxl`. However, the function returns a tibble variable which is re-converted as a classic data.frame (subclass of `data.frame` whose properties are sometime screwing the `tlf` plots).

In the esqLabs example, group names were associated to the observed data *Group Id* variable. An additional column, named *Group*, was added to the observed data to capture that feature. For each value of *Group Id*, a group name was associated.

```
# Load observed dataset
observedData <- as.data.frame(readxl::read_xlsx(
  path = "data/CompiledDataSet.xlsx",
  sheet = "Stevens_2012_placebo"
  ))

# Observed data to plot is a subset of the dataset
observedData <- observedData[observedData$`Group Id` %in% c("Placebo_distal", "Placebo_proximal", "Placebo_total"
), ]

# Associate a name for each group
observedData$Group <- as.character(sapply(
  observedData$`Group Id`,
  FUN = function(group){
    switch(group,
      Placebo_distal = "Stevens 2012 solid distal",
      Placebo_proximal = "Stevens 2012 solid proximal",
      Placebo_total = "Stevens 2012 solid total"
    )
  }))
```

Simulation data are imported using the `ospsuite` package. However, they need to be formatted in a similar fashion as the observed data because the `tlf` plots will require a *x*, *y* and grouping variables. Consequently, the results for different paths need to be concatenated with a grouping variable. Below such a formatting is performed, note that the package `reshape2` might be a more optimal solution to do the same process on wider datasets.

```
# Load simulation dataset
sim <- loadSimulation("data/Stevens_2012_placebo_indiv_results.pkml")
simResults <- importResultsFromCSV(simulation = sim, filePaths = "data/Stevens_2012_placebo_indiv_results.csv")
outputValues <- getOutputValues(simulationResults = simResults,
                                quantitiesOrPaths = list("Organism|Lumen|Stomach|Metformin|Gastric retention",
                                                         "Organism|Lumen|Stomach|Metformin|Gastric retention dist
al",
                                                         "Organism|Lumen|Stomach|Metformin|Gastric retention prox
imal"))

# Data needs to be re-ordered as a data.frame with x, y and groups
outputData <- rbind.data.frame(
  data.frame(
    IndividudalId = outputValues$data$IndividualId,
    Time = outputValues$data$Time,
    Simulations = 100*outputValues$data$`Organism|Lumen|Stomach|Metformin|Gastric retention`,
    GroupId = "Stevens_2012_placebo solid total sim",
    GroupLabel = "Stevens 2012 solid total"
    ),
  data.frame(
    IndividudalId = outputValues$data$IndividualId,
    Time = outputValues$data$Time,
    Simulations = 100*outputValues$data$`Organism|Lumen|Stomach|Metformin|Gastric retention distal`,
    GroupId = "Stevens_2021_placebo solid distal sim",
    GroupLabel = "Stevens 2012 solid distal"
  ),
  data.frame(
    IndividudalId = outputValues$data$IndividualId,
    Time = outputValues$data$Time,
    Simulations = 100*outputValues$data$`Organism|Lumen|Stomach|Metformin|Gastric retention proximal`,
    GroupId = "Stevens_2021_placebo solid proximal sim",
    GroupLabel = "Stevens 2012 solid proximal"
  )
)
```

For observation vs prediction or residual plots, simulated values matching the observed values need to be calculted. Below suggests a code to get such data.

```
# Initialize variable with NAs
observedData$Simulations <- NA
# In each group, get closest simulation time points to observed time points
for(group in unique(observedData$Group)){
  obsSelectedRows <- observedData$Group %in% group
  simSelectedRows <- outputData$GroupLabel %in% group
  obsTimeMatrix <- matrix(observedData[obsSelectedRows, "Time [min]"],
                          sum(simSelectedRows), sum(obsSelectedRows), byrow = TRUE)
  simTimeMatrix <- matrix(outputData[simSelectedRows, "Time"],
                          sum(simSelectedRows), sum(obsSelectedRows))
  timeMatchedData <- as.numeric(sapply(as.data.frame(abs(obsTimeMatrix - simTimeMatrix)), which.min))
  observedData$Simulations[obsSelectedRows] <- outputData$Simulations[simSelectedRows][timeMatchedData]
}

# Calculate corresponding residuals
observedData$Residuals <- observedData$Simulations - observedData$`Fraction [%]`
```

The first lines of the observed dataset are printed below:

Observed Dataset

Time [min]	Fraction [%]	Error [%]	Route	Group Id	Group	Simulations	Residuals
0.00000	100.00000	NA	NA	Placebo_total	Stevens 2012 solid total	0.00000	-100.0000000
13.17227	93.75000	NA	NA	Placebo_total	Stevens 2012 solid total	92.94097	-0.8090317
29.40336	84.16667	NA	NA	Placebo_total	Stevens 2012 solid total	82.19714	-1.9695250
44.64706	72.50000	NA	NA	Placebo_total	Stevens 2012 solid total	70.83093	-1.6690707
73.07983	52.91667	NA	NA	Placebo_total	Stevens 2012 solid total	52.25585	-0.6608156
88.27311	46.25000	NA	NA	Placebo_total	Stevens 2012 solid total	43.65582	-2.5941813

Plots

The `tlf` plots usually require 4 input arguments (see vignette “tlf-workflow” for more details):

- *data*: a data.frame of the data to plot

- *metaData* (optional): a list of meta data on the input *data*. This argument can be used for labelling the *data* in the plot.
- *dataMapping*: an object of class *DataMapping* dedicated to describe which variables are plotted. *dataMapping* usually includes *x* and *y*, as well as grouping variables such as *color*, *shape*, *linetype* and/or *fill*.
- *plotConfiguration* (optional): an object of class *PlotConfiguration* dedicated to describe plot properties such as its labels, background, legend and/or axes.

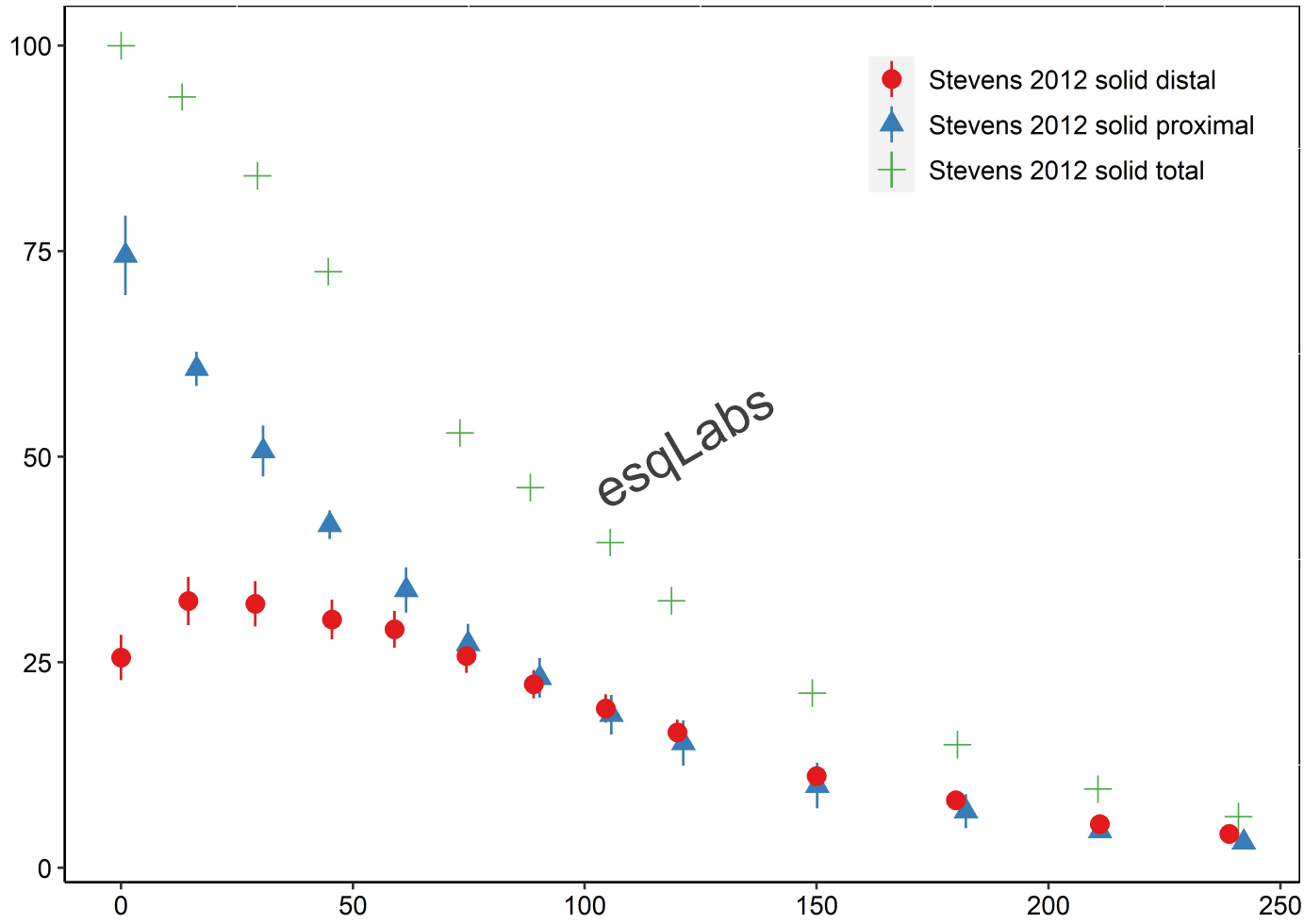
Time Profile Plot

For time profile plots, the `tlf` function *plotTimeProfile* is likely the more appropriate. Its input argument are slightly different from the regular `tlf` plot functions as observed and/or simulated data can be used. As a consequence, 2 other optional arguments are added:

- *observedData*: a data.frame of the observed data to plot. By construction, scatter points and error bars will be plotted for *observedData* while lines and ribbons will be plotted for *data*.
- *observedDataMapping*: an object of class *ObservedDataMapping* describing which variables are plotted. Note that 2 optional inputs are available for such objects: *uncertainty* and *lloq*. (Caution: *uncertainty* variable will be deprecated and replaced by *error* in later versions of `tlf`)

The first example only plotted observed data including error bars. Because `ggplot2` (and consequently `tlf`) uses the grouping variables for defining the legend, missing data for an entire group will cause `ggplot2` to crash as it cannot associate the legend. For this reason, *NA*s in the error variable were replaced by 0 before plotting them.

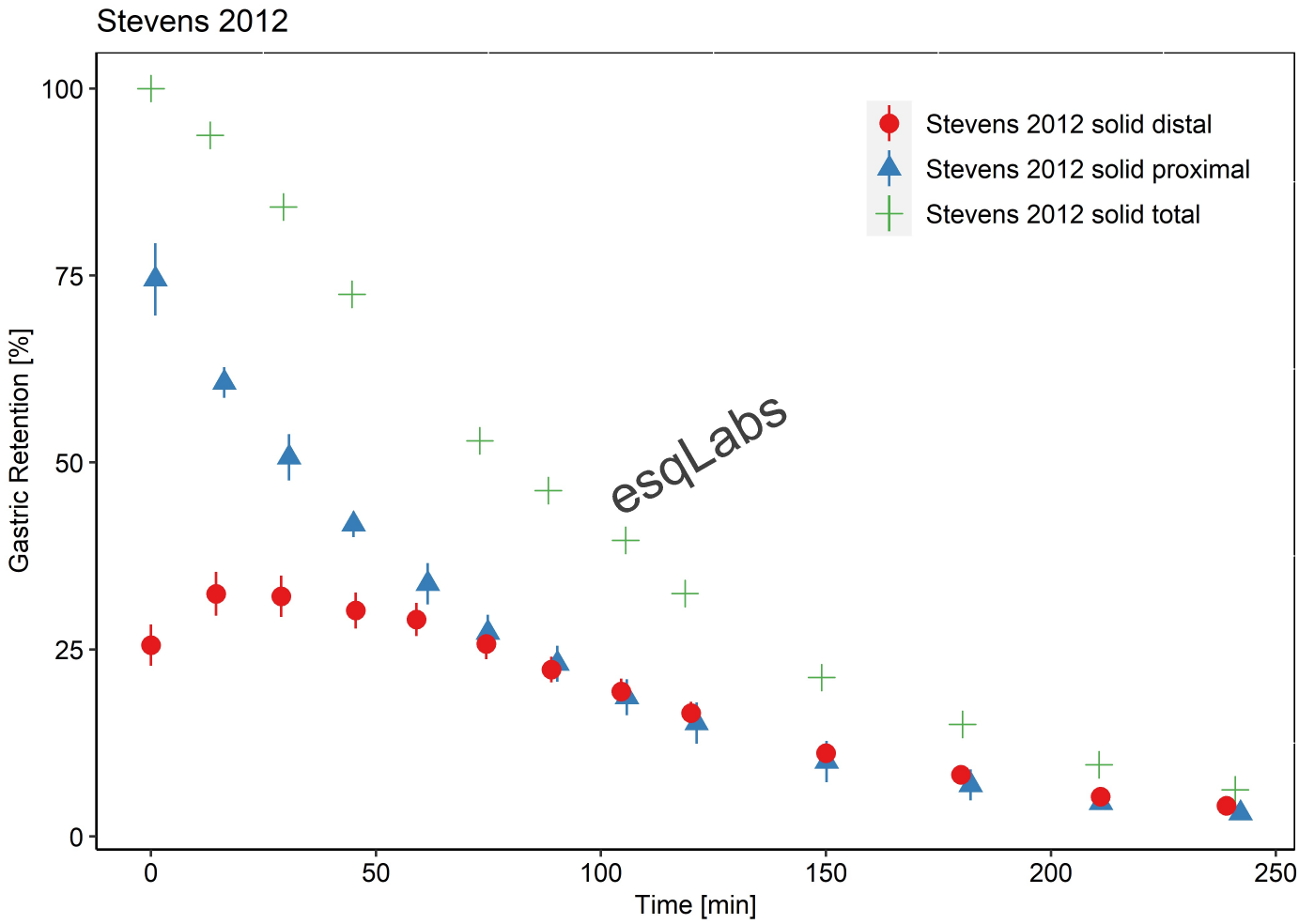
```
observedData$`Error [%]`[is.na(observedData$`Error [%]`)] <- 0

plotTimeProfile(
  observedData = observedData,
  observedDataMapping = ObservedDataMapping$new(
    x = "Time [min]",
    y = "Fraction [%]",
    color = "Group",
    shape = "Group",
    uncertainty = "Error [%]" # uncertainty will be deprecated and replaced by error
    )
  )
```



In this plot no label for x and y axes are printed. Currently, time profile plot only uses the names from the input *data* to get default names for these axes.

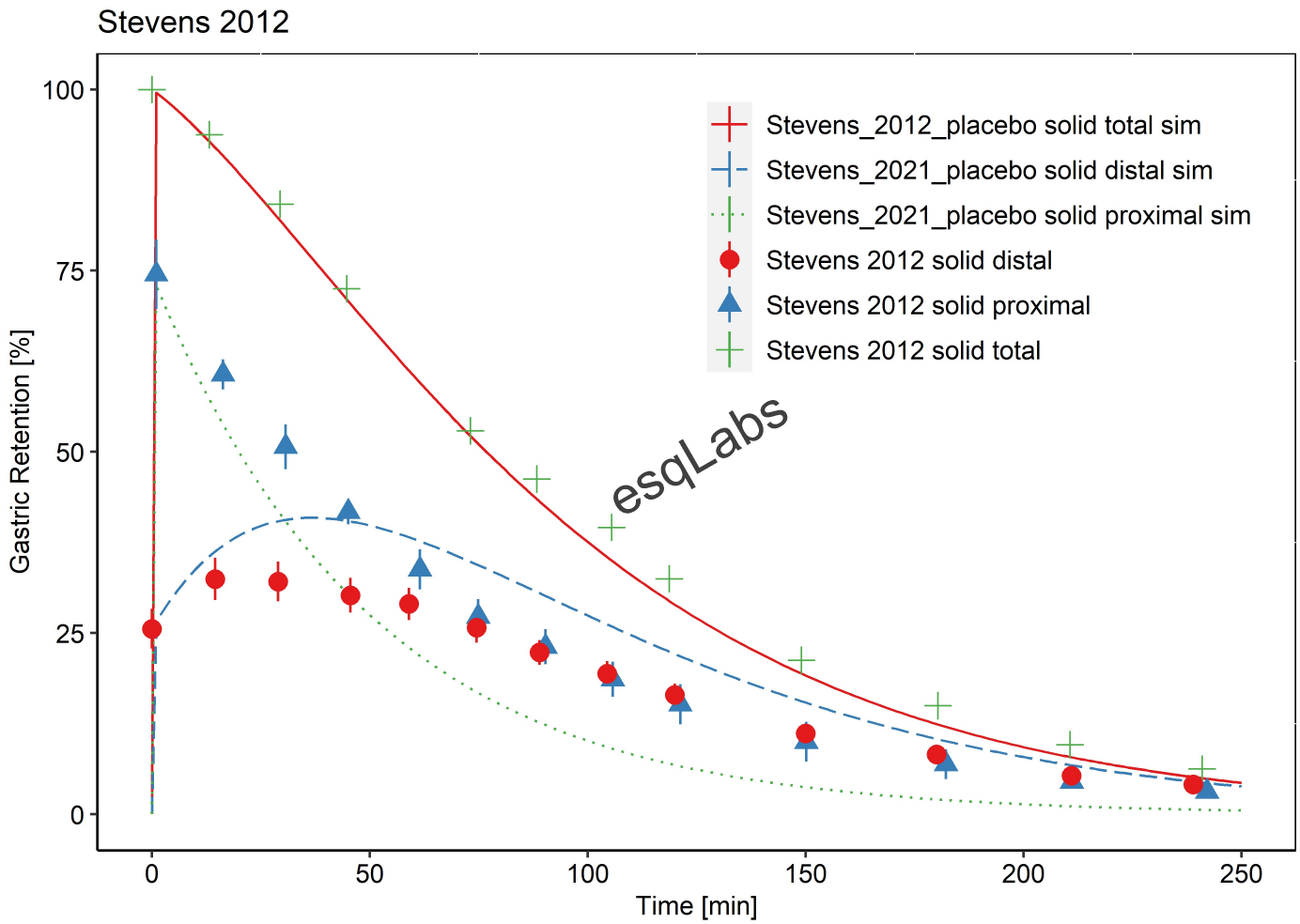
It is possible to tune the plot labels easily before or after creating the plot. To tune labels or other plot properties before creating the plot, plotConfiguration is required. In the example below, the names for title, xlabel and ylabel is updated. As described at the beginning, you can also input `Label` objects if you want to also update the fonts. It is also possible to first create the *PlotConfiguration* object and then update its properties.

```
plotTimeProfile(
  observedData = observedData,
  observedDataMapping = ObservedDataMapping$new(
    x = "Time [min]",
    y = "Fraction [%]",
    color = "Group",
    shape = "Group",
    uncertainty = "Error [%]"
    ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
  )
```



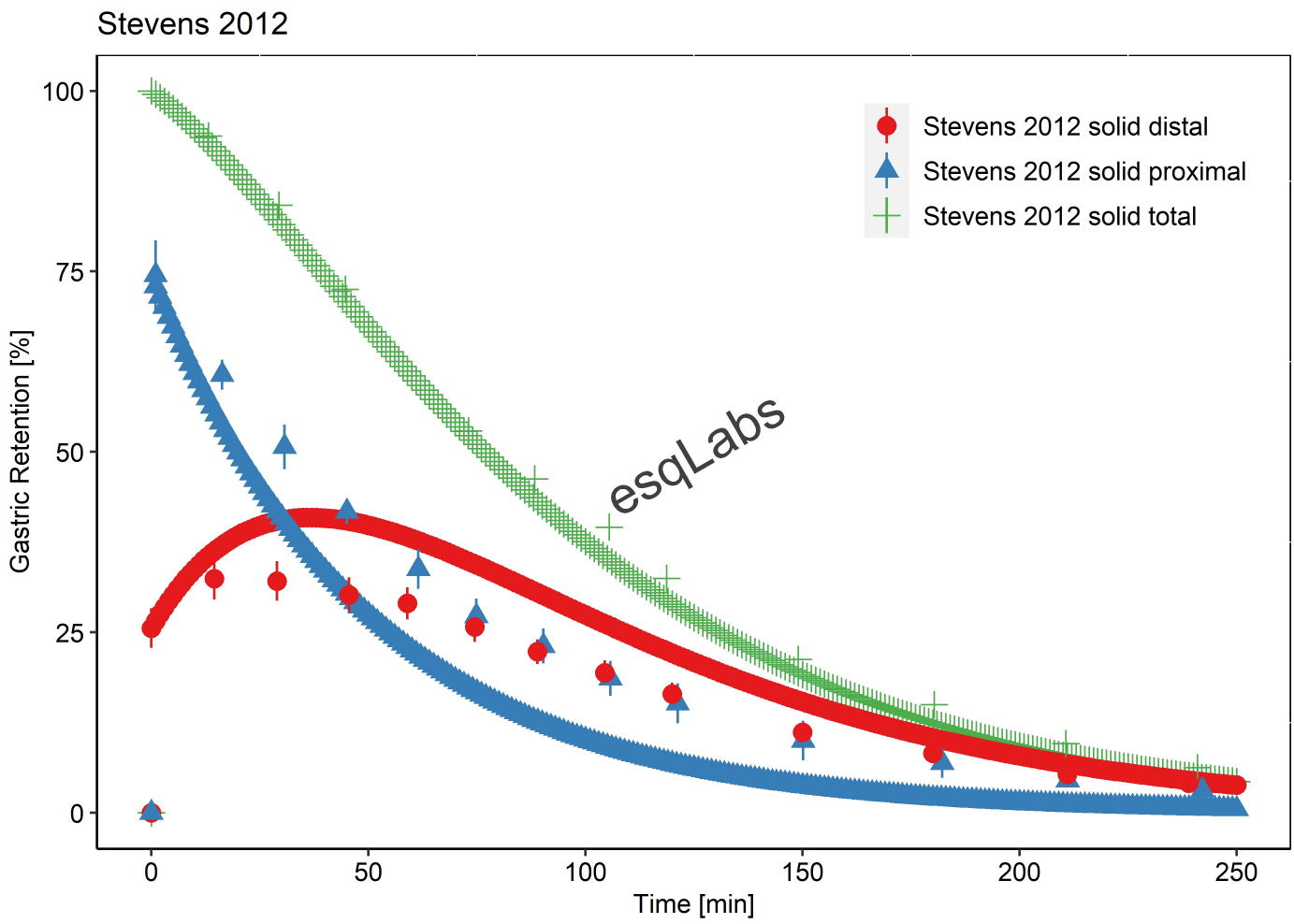
Adding simulations to the time profile plot can be done by including *data* and *dataMapping* arguments. If no *xlabel* and *ylabel* was provided in the plot configuration, the labels would have been “Time” and “Simulations”.

```
plotTimeProfile(
  data = outputData,
  dataMapping = TimeProfileDataMapping$new(
    x = "Time",
    y = "Simulations",
    color = "GroupId"
    ),
  observedData = observedData,
  observedDataMapping = ObservedDataMapping$new(
    x = "Time [min]",
    y = "Fraction [%]",
    color = "Group",
    shape = "Group",
    uncertainty = "Error [%]"
  ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
)
```



As stated in the esqLabs example, if no mapping between observed and simulated data is performed, their color pattern may not match. Mapping the same group names does not work because the same legending will be applied to observed and simulated data by `ggplot2` as illustrated below (leading simulations to print points instead of lines).

```
# ggplot2 issue when using same factor levels
plotTimeProfile(
  data = outputData,
  dataMapping = TimeProfileDataMapping$new(
    x = "Time",
    y = "Simulations",
    color = "GroupLabel"
  ),
  observedData = observedData,
  observedDataMapping = ObservedDataMapping$new(
    x = "Time [min]",
    y = "Fraction [%]",
    color = "Group",
    shape = "Group",
    uncertainty = "Error [%]"
  ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
)
```



The best way to get the appropriate colors and legend is to update the legend field from the plot configuration as illustrated below:

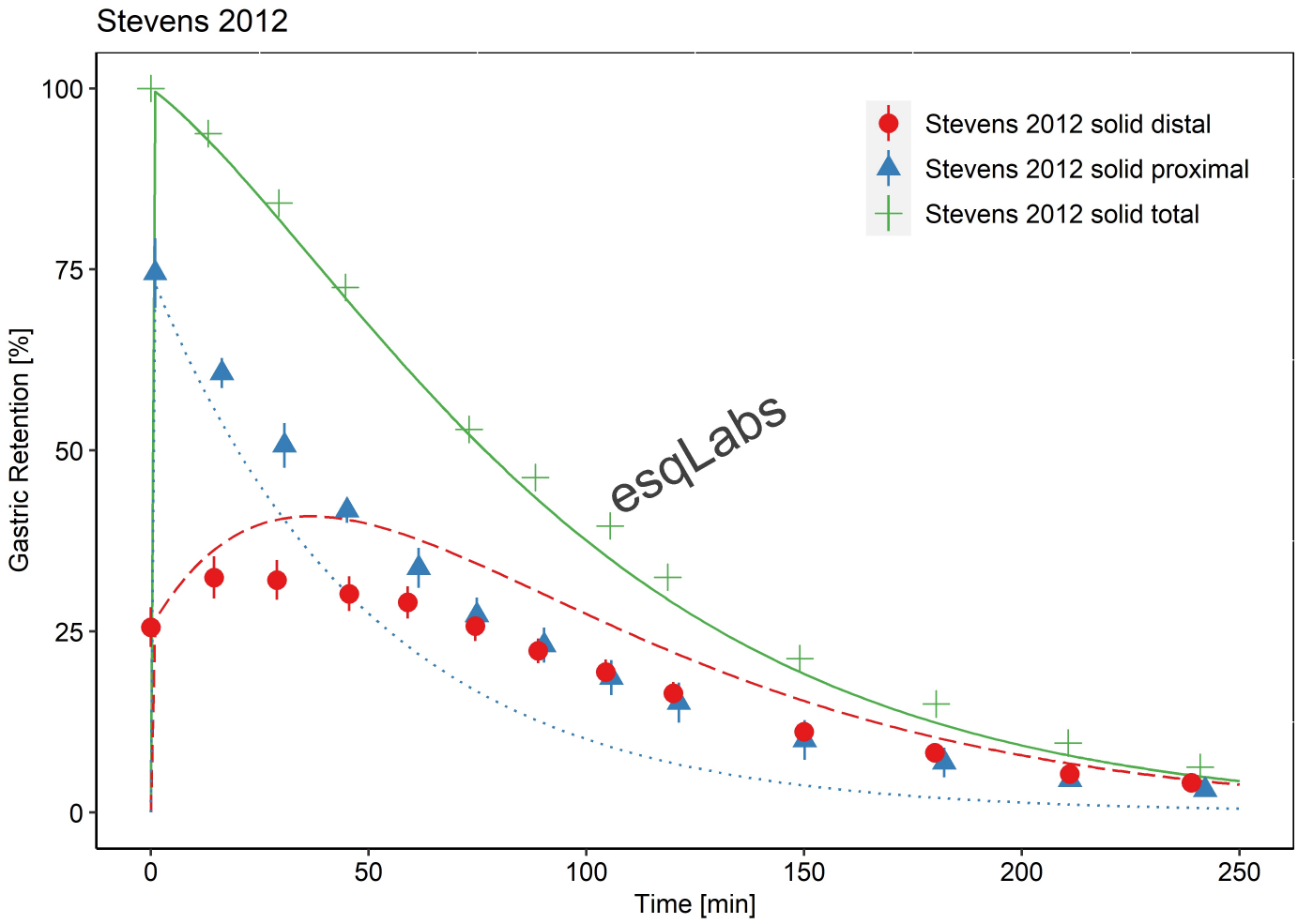
```
timeProfilePlot <- plotTimeProfile(
  data = outputData,
  dataMapping = TimeProfileDataMapping$new(
    x = "Time",
    y = "Simulations",
    color = "GroupId"
  ),
  observedData = observedData,
  observedDataMapping = ObservedDataMapping$new(
    x = "Time [min]",
    y = "Fraction [%]",
    color = "Group",
    shape = "Group",
    uncertainty = "Error [%]"
  ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
)

legend <- getLegendCaption(timeProfilePlot)
knitr::kable(legend)
```

name	label	visibility	order	color	shape	size	linetype	fill
Stevens_2012_placebo solid total sim	Stevens_2012_placebo solid total sim	TRUE	1	#E41A1C		0.5	solid	NA
Stevens_2021_placebo solid distal sim	Stevens_2021_placebo solid distal sim	TRUE	2	#377EB8		0.5	longdash	NA
Stevens_2021_placebo solid proximal sim	Stevens_2021_placebo solid proximal sim	TRUE	3	#4DAF4A		0.5	dotted	NA
Stevens 2012 solid distal	Stevens 2012 solid distal	TRUE	4	#E41A1C	circle	3.0	blank	NA
Stevens 2012 solid proximal	Stevens 2012 solid proximal	TRUE	5	#377EB8	triangle	3.0	blank	NA
Stevens 2012 solid total	Stevens 2012 solid total	TRUE	6	#4DAF4A	plus	3.0	blank	NA

The legend caption is a data.frame indicating the name of the legend entry, its printed name (*label*) and other properties that can be updated using functions such as *setLegendCaption* or *setCaptionColor*, *setCaptionLabels*…

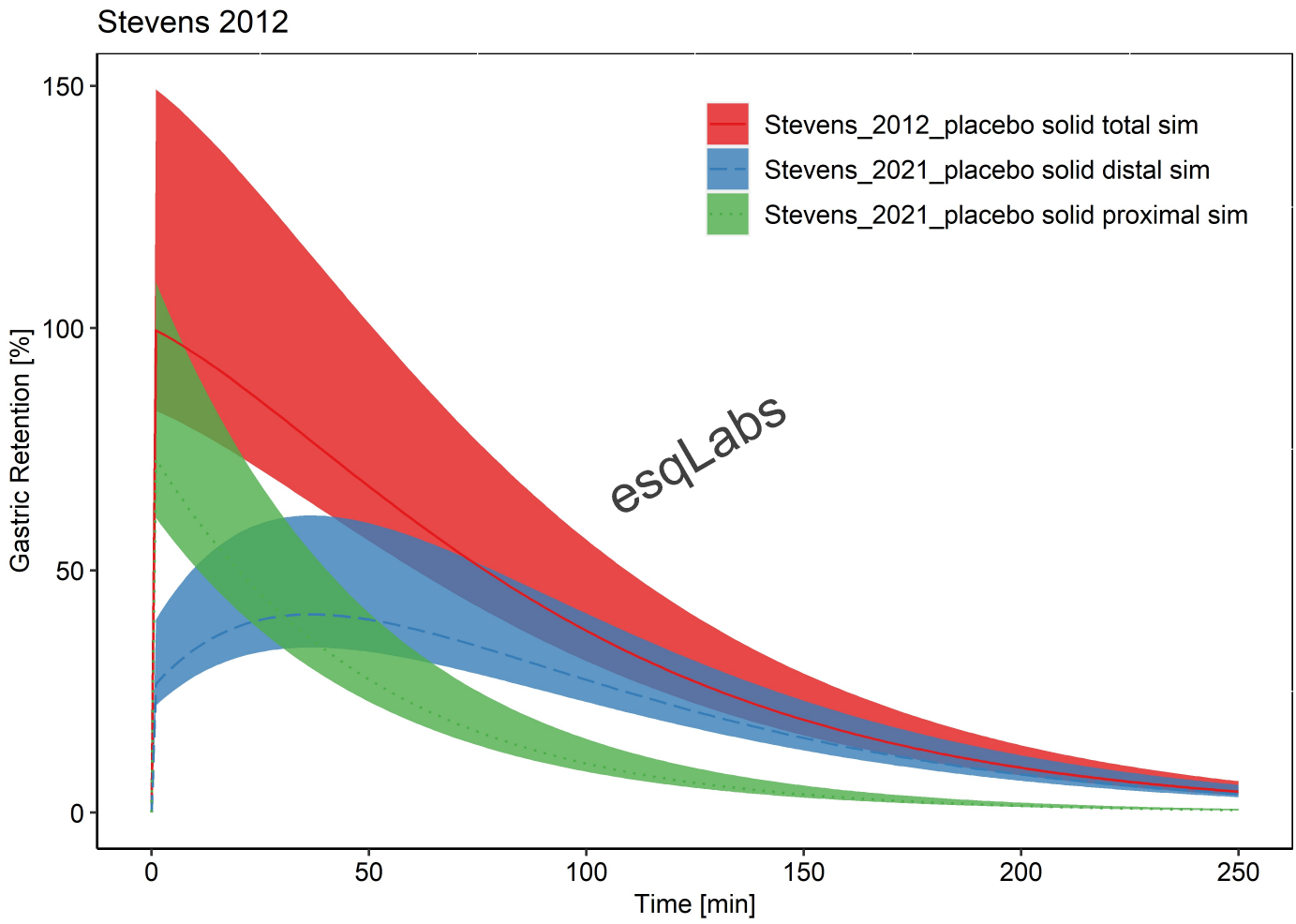
In this example, legends entries 2-4, 3-5 and 1-6 should linked to map observed and simulated data. The column *name* can be used for such mapping instead of the row values. To prevent duplicated legend, the *visibility* column can also be updated.

```
# Simulations get same color as mapped observed data
legend$color[c(2,3,1)] <- legend$color[c(4,5,6)]
# Simulations are set invisible to prevent duplicate legends
legend$visibility <- c(rep(FALSE,3), rep(TRUE,3))

setLegendCaption(timeProfilePlot, caption = legend)
```



For population time profiles, *dataMapping* can also include *ymin* and *ymax* variables indicating the limits of the ribbons to be plotted.

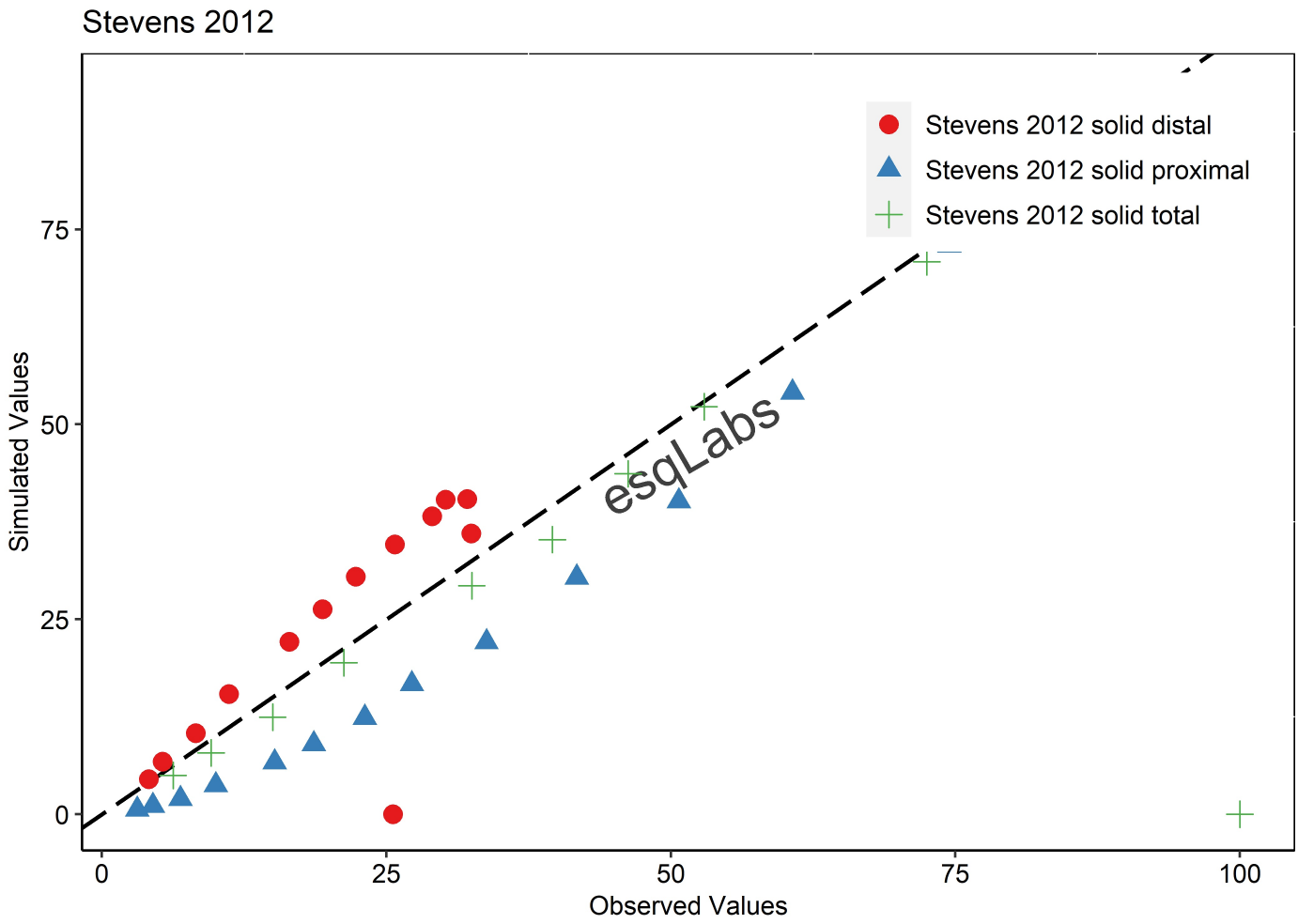
```
outputData$ymin <- outputData$Simulations/1.2
outputData$ymax <- outputData$Simulations*1.5

plotTimeProfile(
  data = outputData,
  dataMapping = TimeProfileDataMapping$new(
    x = "Time",
    y = "Simulations",
    ymin = "ymin",
    ymax = "ymax",
    color = "GroupId"
  ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
)
```



Observations vs Predictions Plots

For obs vs pred plots, the `tlf` function is *plotObsVsPred* can be used as shown below. Note that if the grouping variable were different between *color* and *shape*, the legend would be split between a color legend and a shape legend.

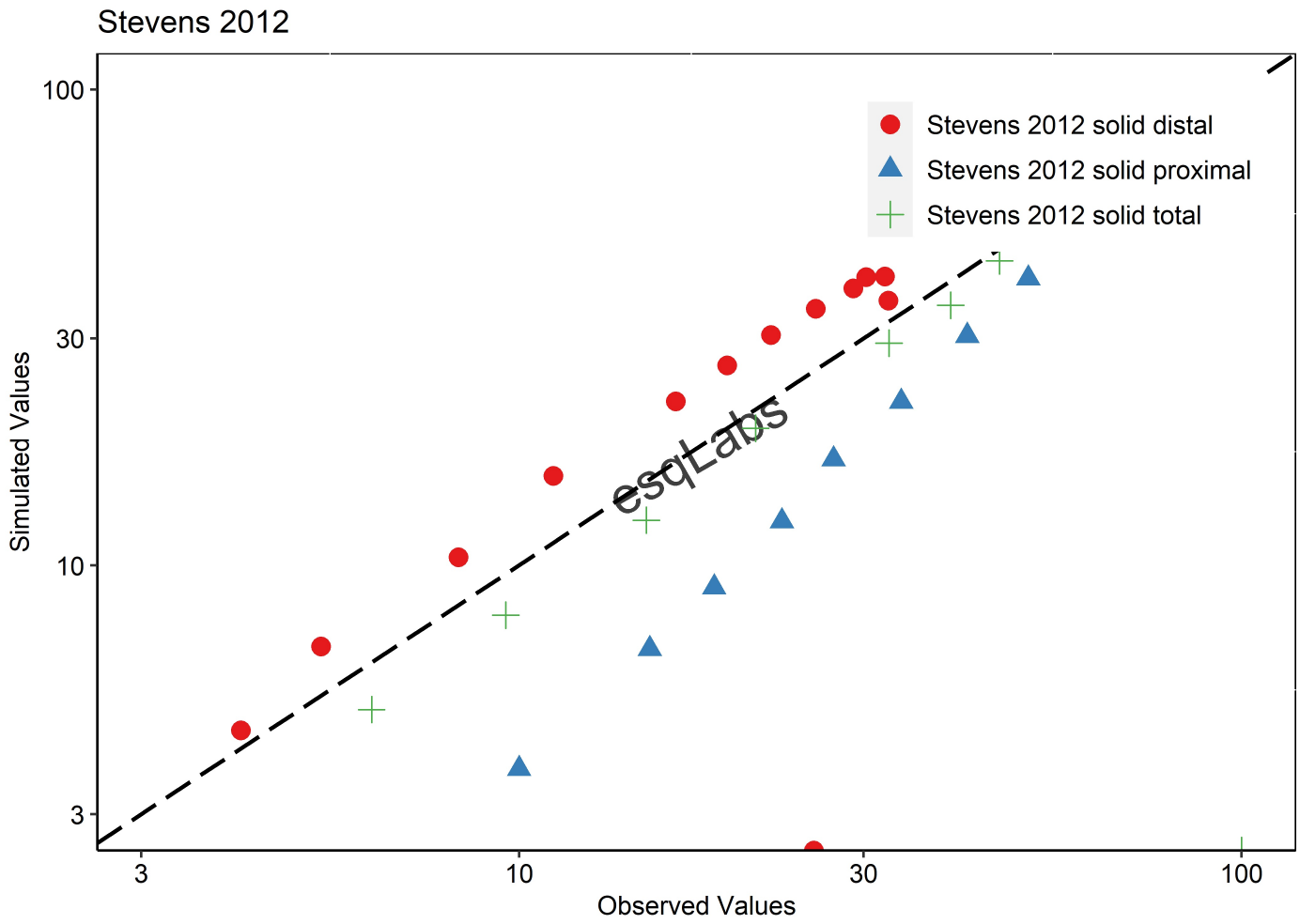
```
plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group"
  ),
  plotConfiguration = ObsVsPredPlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Observed Values",
    ylabel = "Simulated Values"
  )
)
```



The default plot properties provide a plot in linear scale with only the identity line plotted. Many optional parameters can be added to tune the plot.

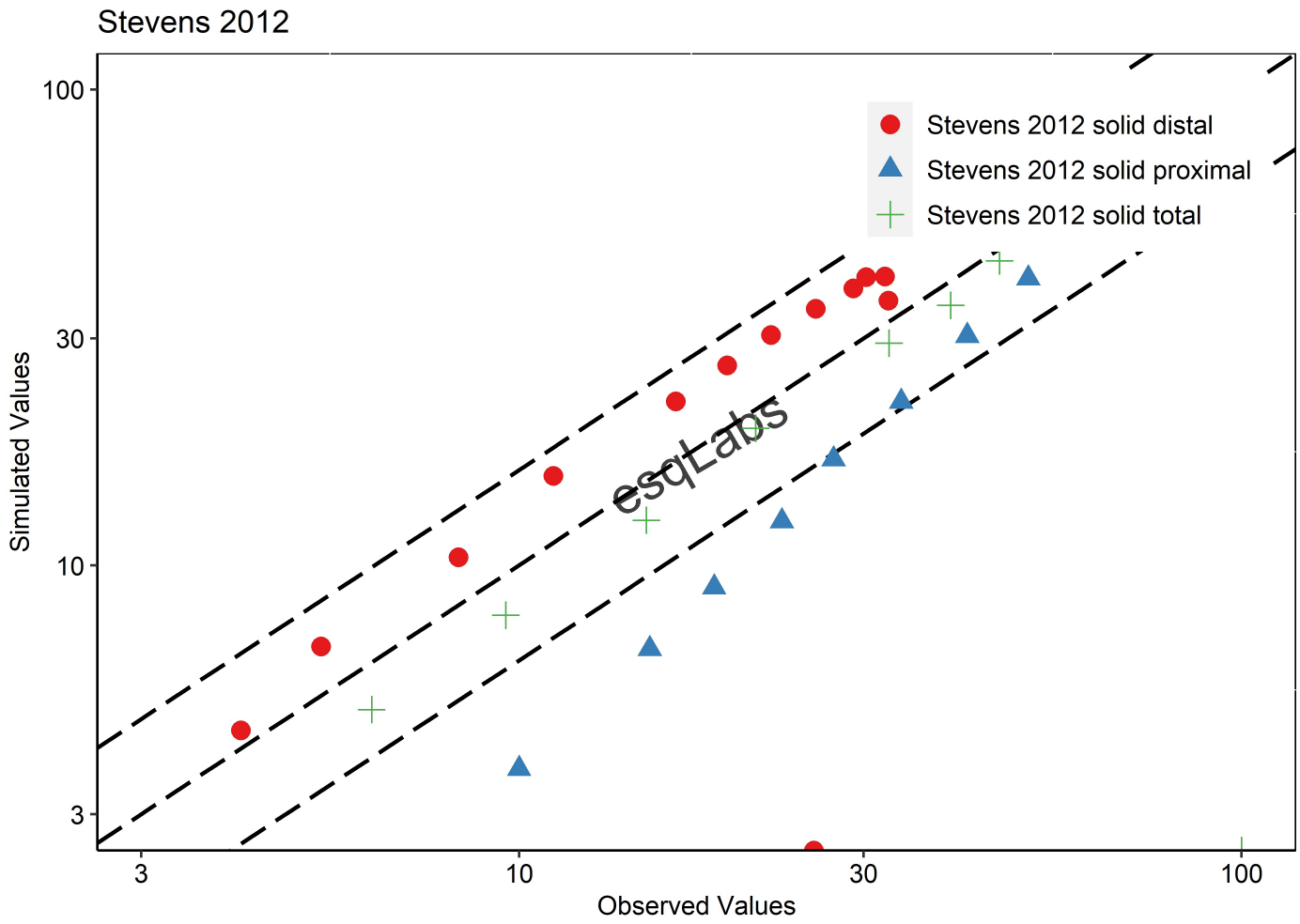
Changing the scales and axes limits can be done before or after creating the plot. The input arguments *xScale*, *yScale*, *xLimits* and * yLimits* can be passed on the initialization of a *PlotConfiguration* object. Or updated afterwards as shown below.

```
obsVsPredPlotConfiguration <- ObsVsPredPlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Observed Values",
    ylabel = "Simulated Values"
  )
obsVsPredPlotConfiguration$xAxis$scale <- Scaling$log
obsVsPredPlotConfiguration$yAxis$scale <- Scaling$log
# Currently, there seems to be a bug in xAxis limits which are not updated
obsVsPredPlotConfiguration$xAxis$limits <- c(3,100)
obsVsPredPlotConfiguration$yAxis$limits <- c(3,100)

plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group"
  ),
  plotConfiguration = obsVsPredPlotConfiguration
)
#> Warning: Transformation introduced infinite values in continuous y-axis
```



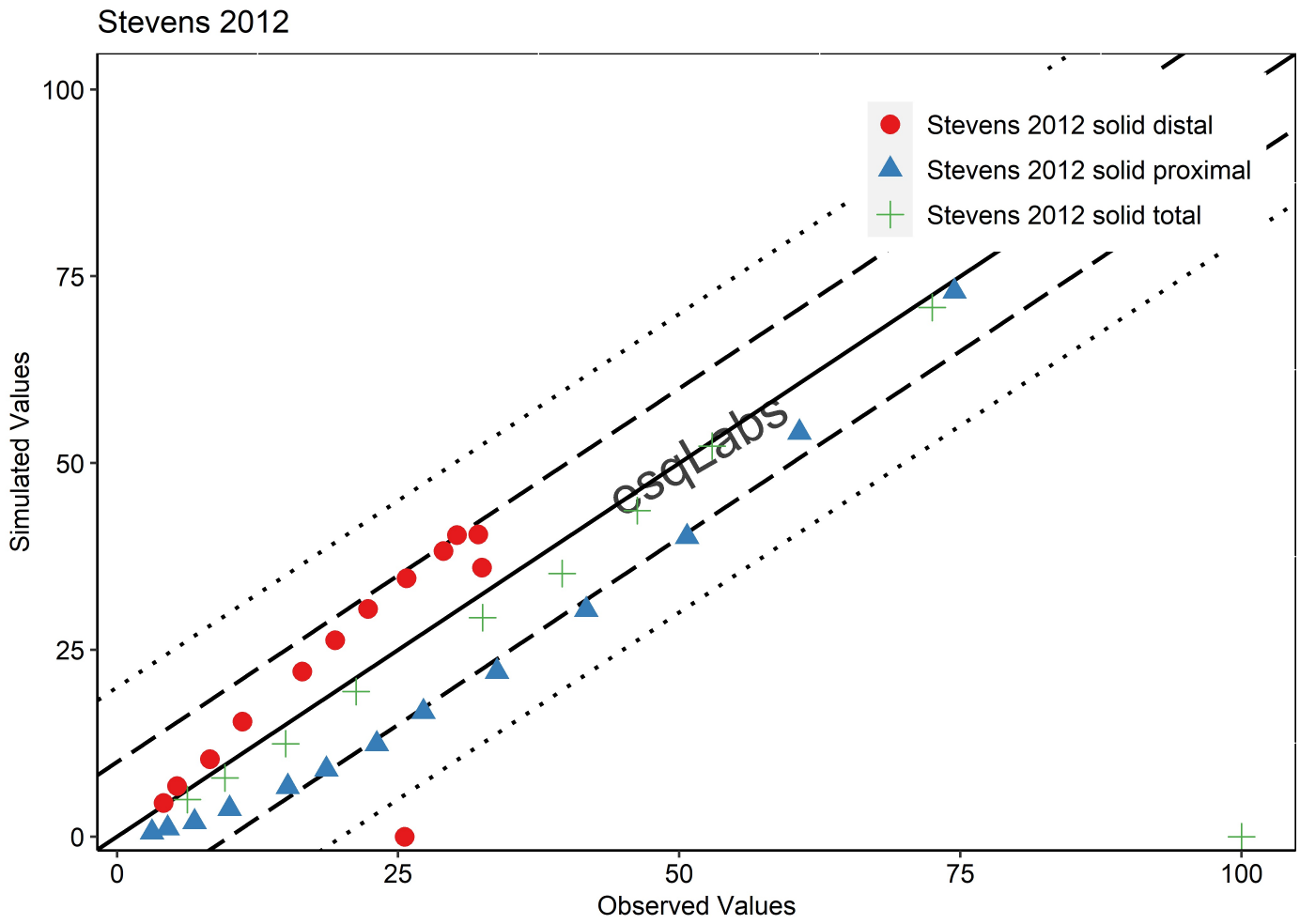
Adding multiple lines is also possible using the the input *lines* of *ObsVsPredDataMapping* objects. The values in lines will lead to plot lines parallel to the identity line with an offset corresponding either to the value in the case of linear scale, and to the log transformed value in the case of log scale. The values in the plot configuration will only define their color, linetype…

```
# shows how the lines are plotted
obsVsPredPlotConfiguration$lines
#> <ThemeAestheticSelections>
#>   Inherits from: <ThemeAestheticMaps>
#>   Public:
#>     alpha: 1
#>     clone: function (deep = FALSE)
#>     color: #000000
#>     fill: NA
#>     initialize: function (color = NULL, fill = NULL, shape = NULL, size = NULL,
#>     linetype: longdash
#>     shape: blank
#>     size: 0.75
#>     toJson: function ()

# Since plot is log-scale
# lines = c(0,0.2,-0.2) correspond to lines of 10^c(0,0.2,-0.2) fold-errors
plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group",
    lines = c(0, 0.2, -0.2)
  ),
  plotConfiguration = obsVsPredPlotConfiguration
)
#> Warning: Transformation introduced infinite values in continuous y-axis
```



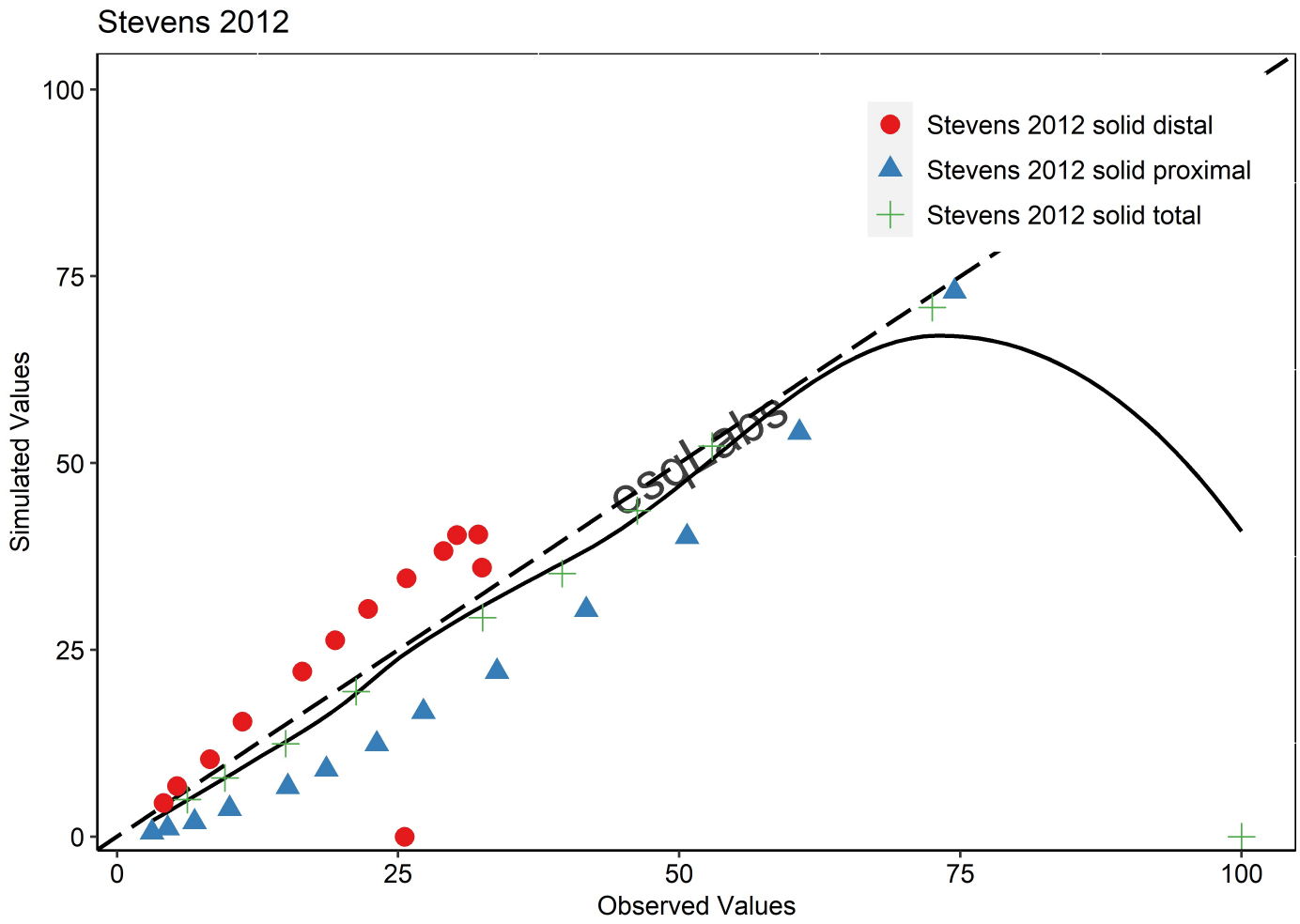
In the example below, *lines* were input as a list which means that they are grouped and each group of new lines will be plotted as defined in the plot configuration field *linetype* of *lines*. In more details, the identity line will be plotted as a solid line while the lines of +/- 10% errors will be plotted as long dashed lines and the lines of +/- 20% errors will be plotted as dotted lines. Since scale is now linear, the lines are not fold errors anymore but the directly error (offsets of 0, 10, -10, 20 and -20%).

```
# Update the way lines are plotted:
obsVsPredPlotConfiguration$lines$linetype <- c(Linetypes$solid, Linetypes$longdash, Linetypes$dotted)
obsVsPredPlotConfiguration$xAxis$scale <- Scaling$lin
obsVsPredPlotConfiguration$yAxis$scale <- Scaling$lin

plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group",
    lines = list(a = 0,
                 b = c(10, -10),
                 c = c(20, -20))
  ),
  plotConfiguration = obsVsPredPlotConfiguration
)
```



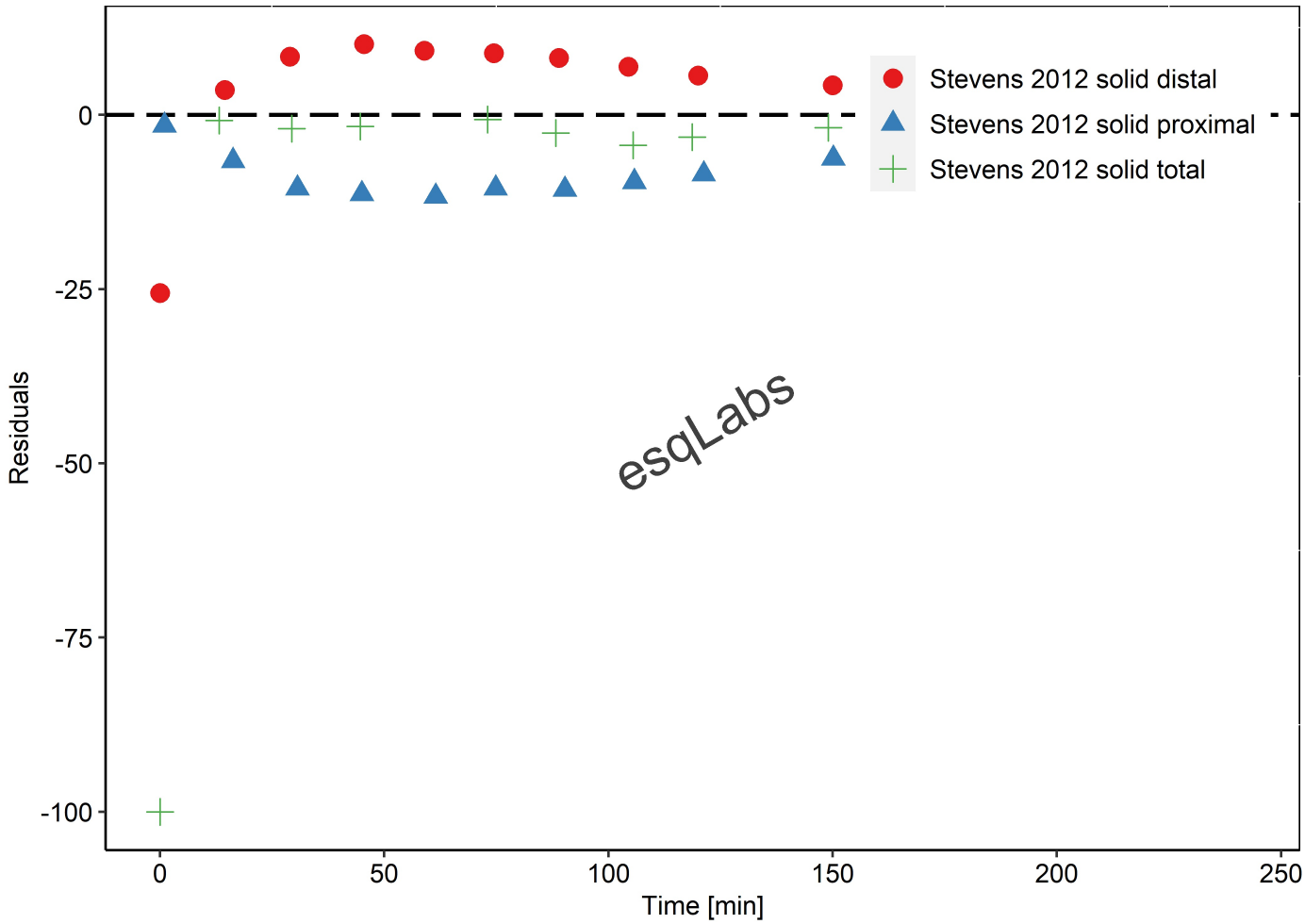
Obs vs Pred plots can also include a regression line. Input of the regression is called *smoother* (same as `ggplot2` input) and uses “lm” (linear model) and “loess” as possible values. The input can either be used in the *dataMapping* or as a separate input of *plotObsVsPred*.

```
# Update the way lines are plotted: identity is longdashes and regression a solid line
obsVsPredPlotConfiguration$lines$linetype <- c(Linetypes$longdash, Linetypes$solid)

plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group"
  ),
  plotConfiguration = obsVsPredPlotConfiguration,
  smoother = "loess"
)
```



The function *plotResVsPred* can also be used to plot residuals as a function of time or predictions. The function works exactly the same way as *plotObsvsPred*, the only difference is that the resulting plot will draw horizontal lines.

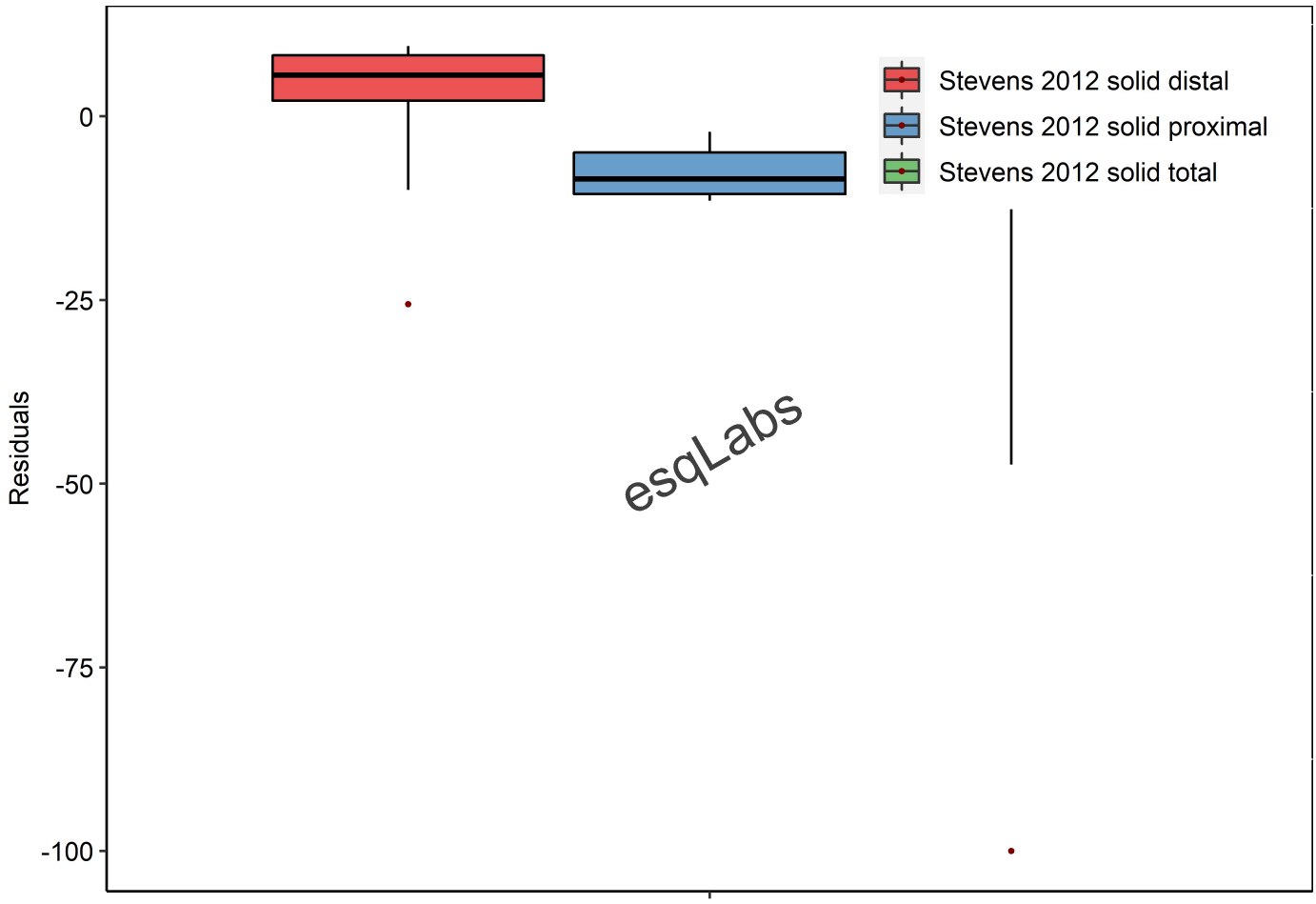
```
plotResVsPred(
  data = observedData,
  dataMapping = ResVsPredDataMapping$new(
    x = "Time [min]",
    y = "Residuals",
    color = "Group",
    shape = "Group"
  )
)
```



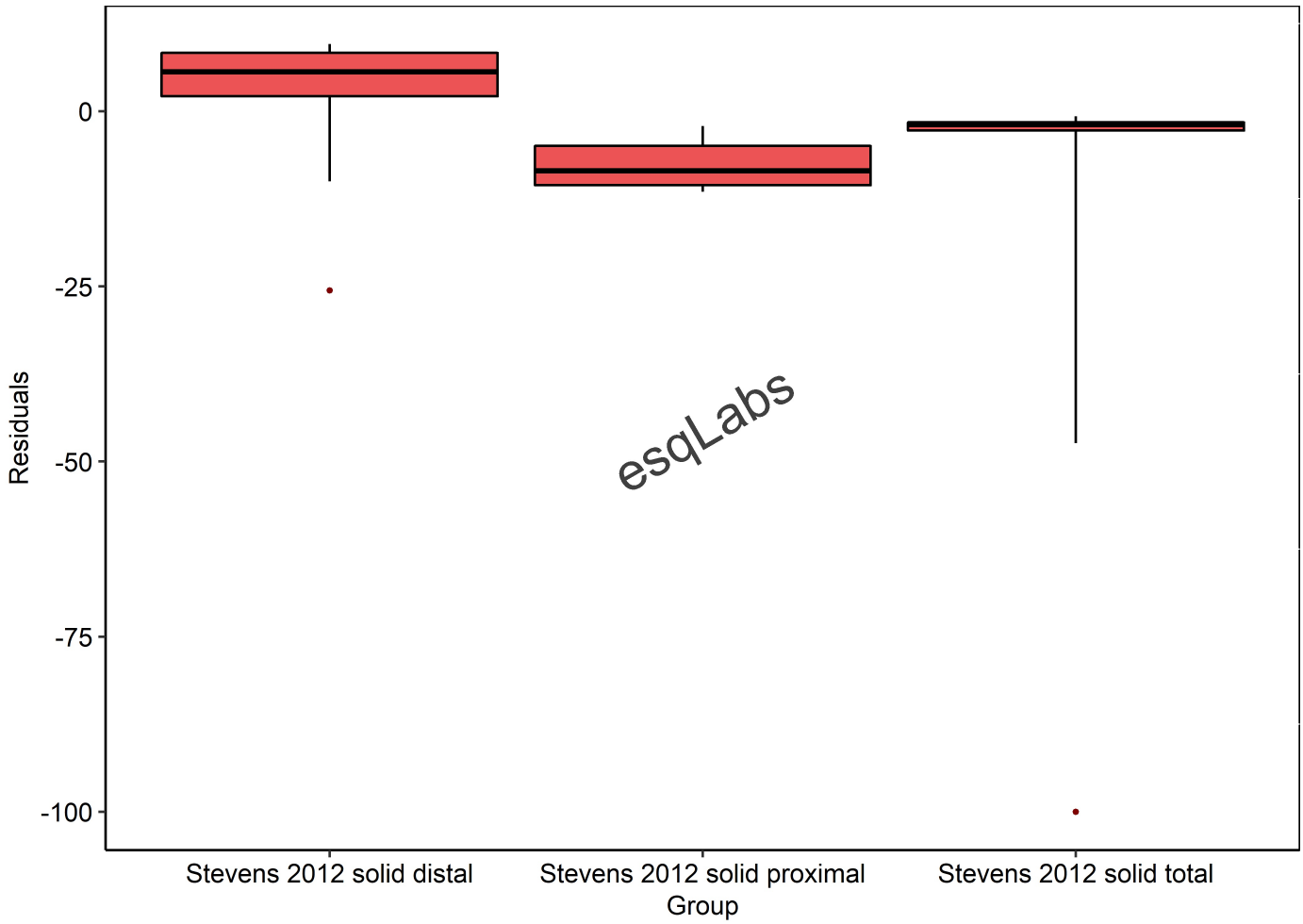
Boxplot

For boxplots (see vignette “box-whisker-vignette” for more details), the `tlf` function is *plotBoxWhisker* can be used as shown below. Grouping variables can be used in *x* variable (either as the same or independent grouping from colors).

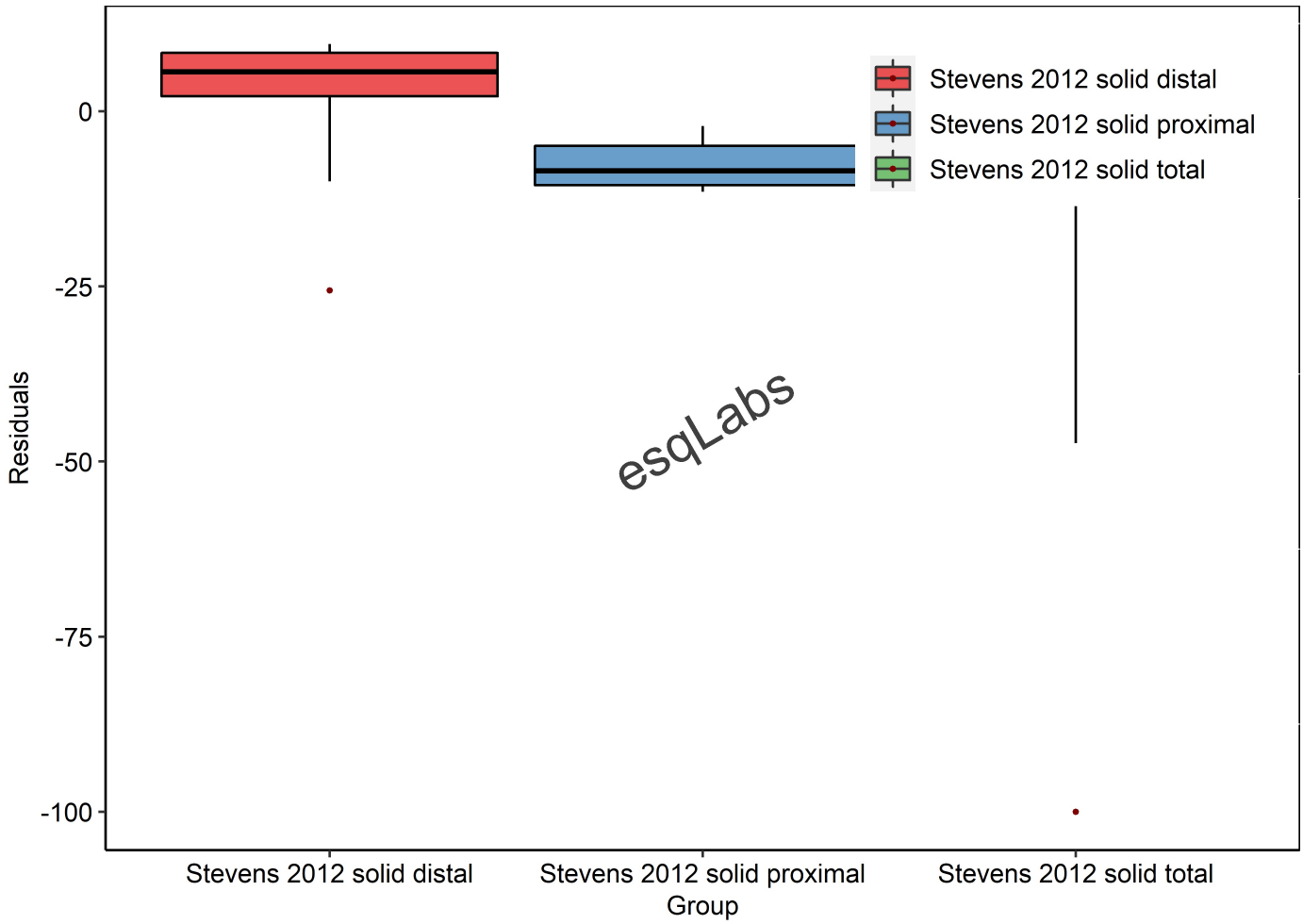
```
plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    y = "Residuals",
    fill = "Group"
  )
)
```




```
plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    x = "Group",
    y = "Residuals"
  )
)
```



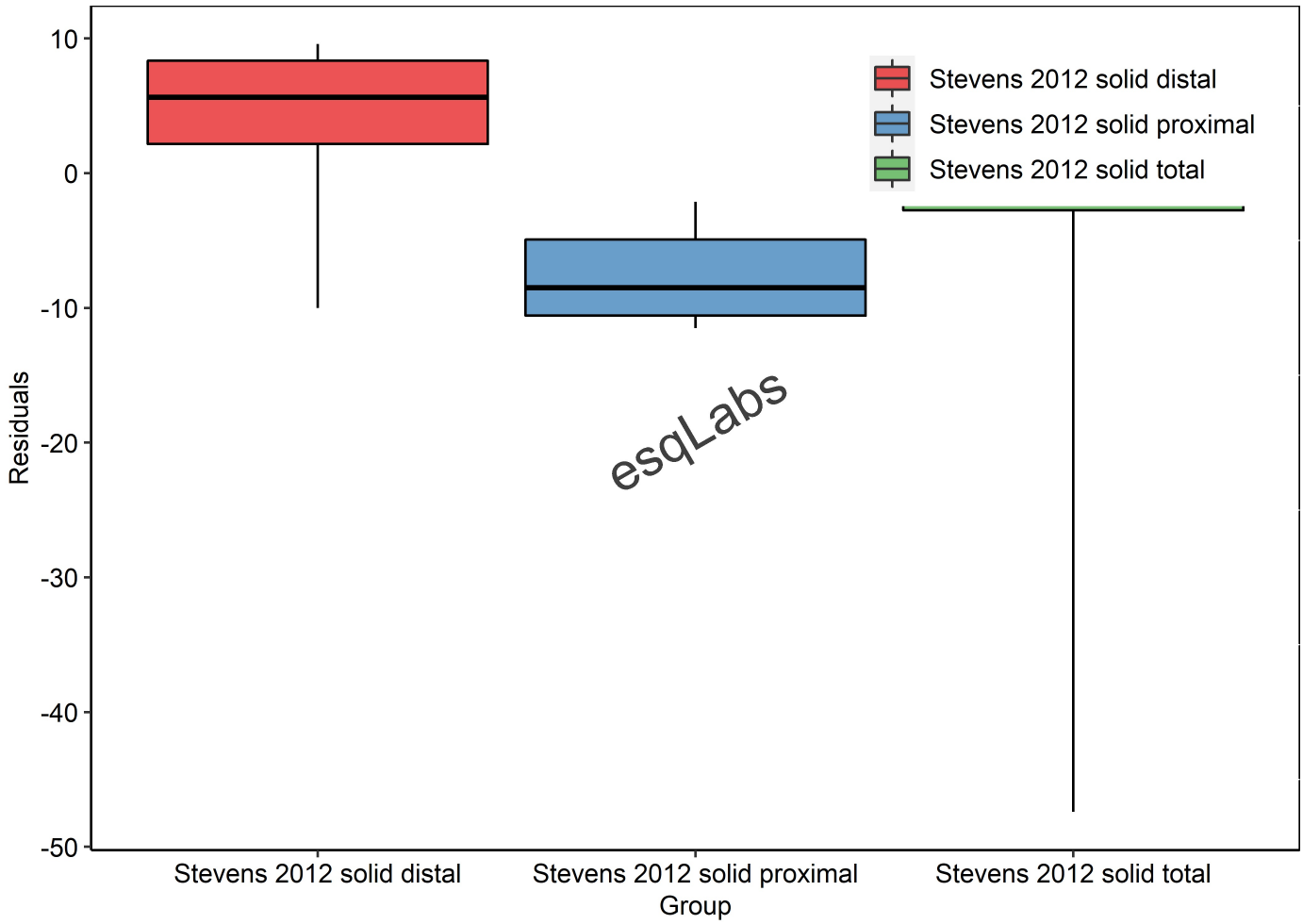

```
plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    x = "Group",
    y = "Residuals",
    fill = "Group"
  )
)
```



Default *dataMapping* plots 5th, 25th, 50th, 75th and 95th percentiles of the data. But they can be changed in *BoxWhiskerDataMapping* inputs using function names for instance available in enum *tlfStatFunctions*.

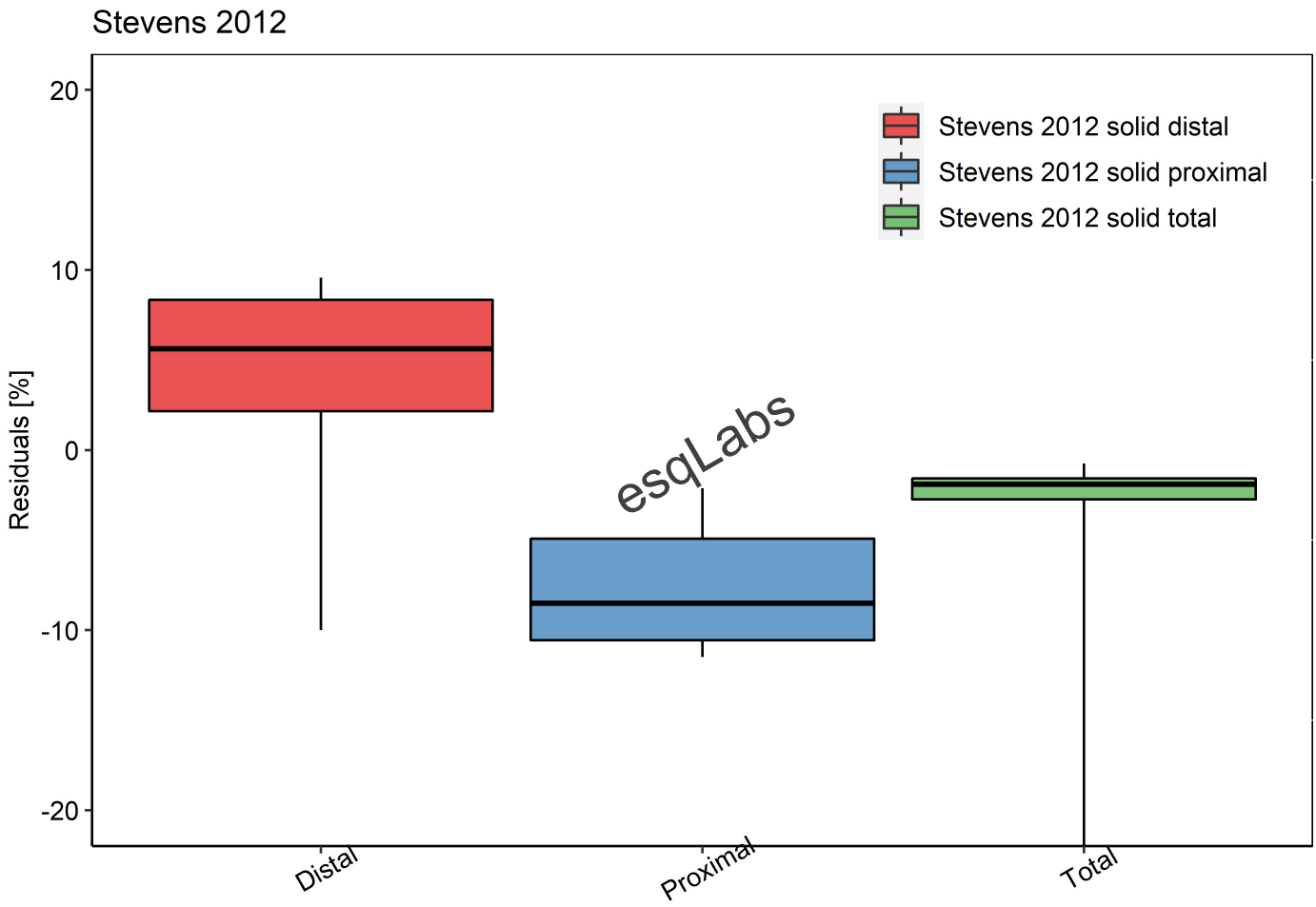
Outliers are flagged by default as being lower than 25th percentile - 1.5IQR and higher than 75th percentile + 1.5IQR. These default can also be changed in *BoxWhiskerDataMapping* inputs. To show or hide outliers in the plot, the logical input *outliers* from *BoxWhiskerPlotConfiguration* can be used or its shortcut in *plotBoxWhisker*.

```
plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    x = "Group",
    y = "Residuals",
    fill = "Group"
  ),
  outliers = FALSE
)
```



A more advanced tuning of the plot configuration properties can also be performed as shown below:

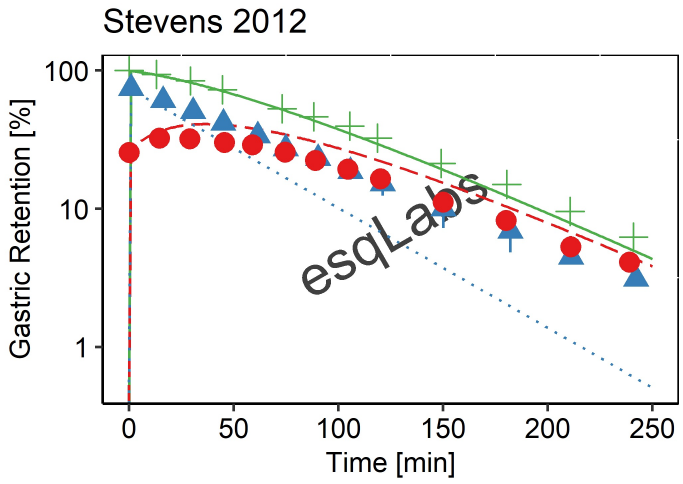
```
boxPlotConfiguration <- BoxWhiskerPlotConfiguration$new()
# Add the title and ylabel
boxPlotConfiguration$labels$title$text <- "Stevens 2012"
boxPlotConfiguration$labels$ylabel$text <- "Residuals [%]"
# Remove xlabel
boxPlotConfiguration$labels$xlabel <- NULL
# Remove outliers
boxPlotConfiguration$outliers <- FALSE
# Center y scale
boxPlotConfiguration$yAxis$limits <- c(-20, 20)
# Rotate the ticklabels
boxPlotConfiguration$xAxis$font$angle <- 30
# Rename the ticklabels
boxPlotConfiguration$xAxis$ticklabels <- c("Distal", "Proximal", "Total")

plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    x = "Group",
    y = "Residuals",
    fill = "Group"
  ),
  plotConfiguration = boxPlotConfiguration
)
```

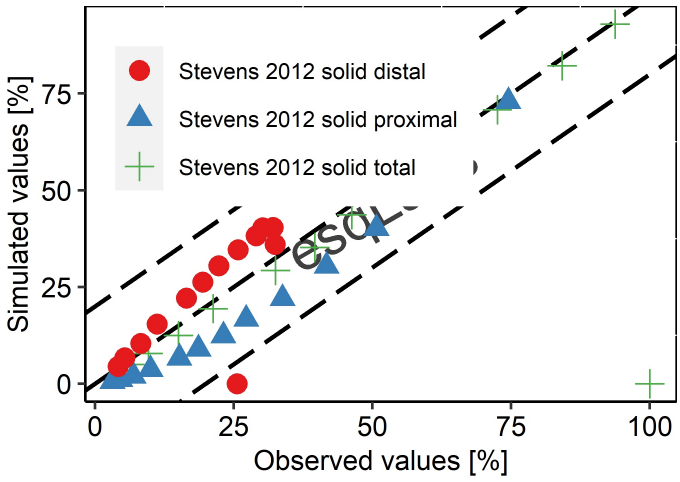


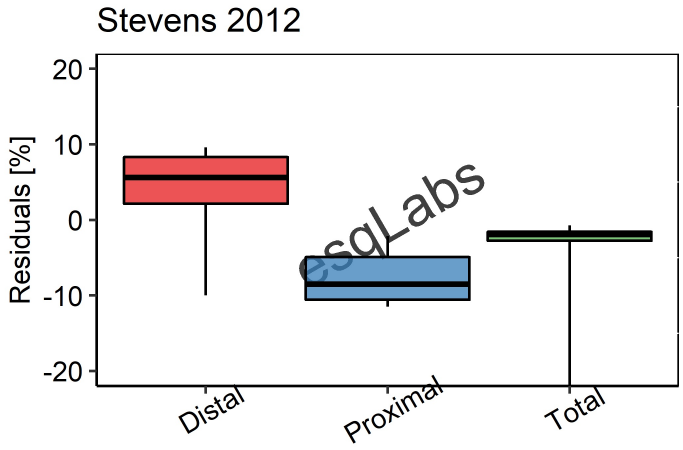
Mutliple Plots

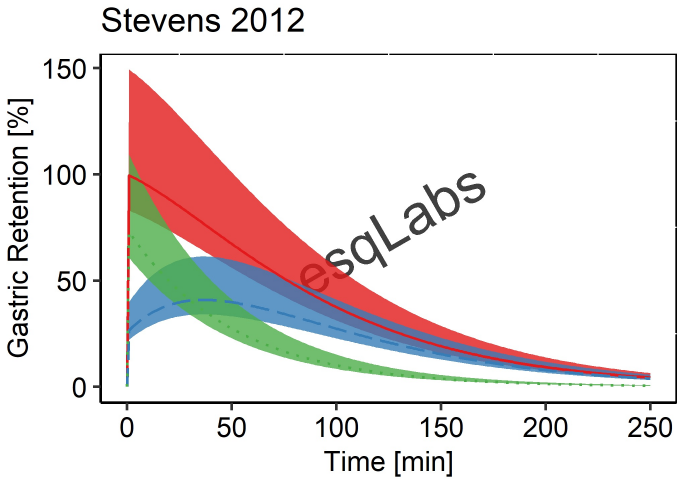
Currently, no function was defined in `tlf` for displaying multiple plots in the same grid as the package `gridExtra` is very convenient to do so.

```
# Using the previous examples:
plotA <- setLegendCaption(timeProfilePlot, caption = legend)
# Plot object updated after its construction
plotA <- setYAxis(plotObject = plotA, scale = Scaling$log)
plotA <- setLegendPosition(plotA, position = LegendPositions$none)

plotB <- plotObsVsPred(
  data = observedData,
  dataMapping = ObsVsPredDataMapping$new(
    x = "Fraction [%]",
    y = "Simulations",
    color = "Group",
    shape = "Group",
    lines = c(0, 20, -20)
  ),
  plotConfiguration = ObsVsPredPlotConfiguration$new(
    xlabel = "Observed values [%]",
    ylabel = "Simulated values [%]"
  )
)
plotB <- setLegendPosition(plotB, position = LegendPositions$insideTopLeft)
plotB <- setLegendFont(plotB, size = 8)

plotC <- plotBoxWhisker(
  data = observedData,
  dataMapping = BoxWhiskerDataMapping$new(
    x = "Group",
    y = "Residuals",
    fill = "Group"
  ),
  plotConfiguration = boxPlotConfiguration
)
plotC <- setLegendPosition(plotC, position = LegendPositions$none)

plotD <- plotTimeProfile(
  data = outputData,
  dataMapping = TimeProfileDataMapping$new(
    x = "Time",
    y = "Simulations",
    ymin = "ymin",
    ymax = "ymax",
    color = "GroupId"
  ),
  plotConfiguration = TimeProfilePlotConfiguration$new(
    title = "Stevens 2012",
    xlabel = "Time [min]",
    ylabel = "Gastric Retention [%]"
  )
)
plotD <- setLegendPosition(plotD, position = LegendPositions$none)

gridExtra::grid.arrange(plotA, plotB, plotC, plotD, ncol = 2)
#> Warning: Transformation introduced infinite values in continuous y-axis

#> Warning: Transformation introduced infinite values in continuous y-axis

#> Warning: Transformation introduced infinite values in continuous y-axis

#> Warning: Transformation introduced infinite values in continuous y-axis
```


Stevens 2012
Gastric Retention [%]
100
10
1
0
50
100
150
200
250
Time [min]
esqLabs


Simulated values [%]
75
50
25
0
0
25
50
75
100
Observed values [%]
Stevens 2012 solid distal
Stevens 2012 solid proximal
Stevens 2012 solid total


Stevens 2012
Residuals [%]
20
10
0
-10
-20
Distal
Proximal
Total
esqLabs


Stevens 2012
Gastric Retention [%]
150
100
50
0
0
50
100
150
200
250
Time [min]
esqLabs