PharmaSUG 2026 - Paper DV-105

A Standardized R Graph Library for Production-Ready Analysis Figures

Chunting Zheng, Syneos Health;
Xindai Hu, Vertex Pharmaceuticals, Inc.;
Margaret Huang, Vertex Pharmaceuticals, Inc.

ABSTRACT

Production of Clinical Study Report (CSR) figures requires strict adherence to internal standards, including exact specifications for fonts, layouts, and aesthetics. Traditionally, programmers implement similar code blocks for each new figure, leading to redundancy, inefficiency, and an increased risk of inconsistency. To address this challenge, we developed a comprehensive R graph library that automates the creation of actual analysis figures using standardized templates for common graph types such as scatter, bar, box, swimmer, line, spaghetti, forest and Kaplan-Meier plots.

Each template encapsulates a uniform structure and theme, ensuring consistency across projects while minimizing the need to manually reproduce code. At the same time, flexibility is preserved through high-level parameters (e.g., point size, opacity, text size, model type, formula) and customized arguments that allow additional layers for plot-specific customization. This design is built on the R *ggplot2* and *forestploter* packages and enables users to produce production-ready CSR figures with concise, readable code while maintaining alignment with industry practices.

To demonstrate the utility of this approach, we present practical CSR examples using scatter, bar, and forest plots generated with the proposed R graph library. In selected cases, corresponding SAS implementations are also shown to illustrate differences in code structure and figure generation. Our R graph library reduces code complexity, improves reproducibility, and enhances readability across a range of CSR graph scenarios. Our templated, automation-ready system balances efficiency, flexibility, and regulatory compliance, ultimately streamlining the generation of high-quality CSR graphics.

INTRODUCTION

Figures included in Clinical Study Reports (CSRs) must follow strict internal standards and regulatory guidance, such as ICH E3. In practice, statistical programmers may often generate figures using general-purpose visualization frameworks such as *ggplot2*, where each figure requires constructing a full plotting pipeline and repeatedly specifying graphical elements such as geoms, themes, and formatting options.

As clinical development programs often require the generation of large numbers of figures across multiple analyses and studies, maintaining consistent graphical standards while minimizing repetitive programming becomes an important practical challenge.

To address this issue, we developed a standardized R graph library that generates commonly used analysis figures through reusable plotting templates. Unlike traditional workflows where programmers assemble each plot from individual *ggplot2* layers, the template approach allows users to create figures by providing structured inputs such as the dataset and analysis variables (e.g., `Data`, `X_var`, `Y_var`). The underlying functions automatically apply predefined styling, layout conventions, and graphical behavior aligned with internal reporting standards. This design reduces repetitive formatting work while ensuring the figures produced across studies follow consistent conventions.

The library also incorporates several architectural features that support reliable and consistent figure generation. Plot parameters are organized according to the graphical component they control, for example, legend-related arguments use the `legend_` prefix, while other parameters control point, bar, or line elements. In addition, extension parameters allow users to add additional plotting layouts and further adjust theme elements when needed. The plotting dataset can also be exported for quality control (QC), regardless of whether the input data are pre-summarized or derived from raw observations.

This paper presents the design and implementation of the proposed R graph library and its architecture for standardized figure generation. Through practical examples, we demonstrate how the library reduces coding complexity while maintaining alignment with reporting standards and facilitating efficient quality-

control workflows. Compared with traditional workflows, this approach allows programmers to generate figures with less code and reduced manual formatting. As a result, quality control can focus more on validating analytical results rather than verifying repetitive presentation details.

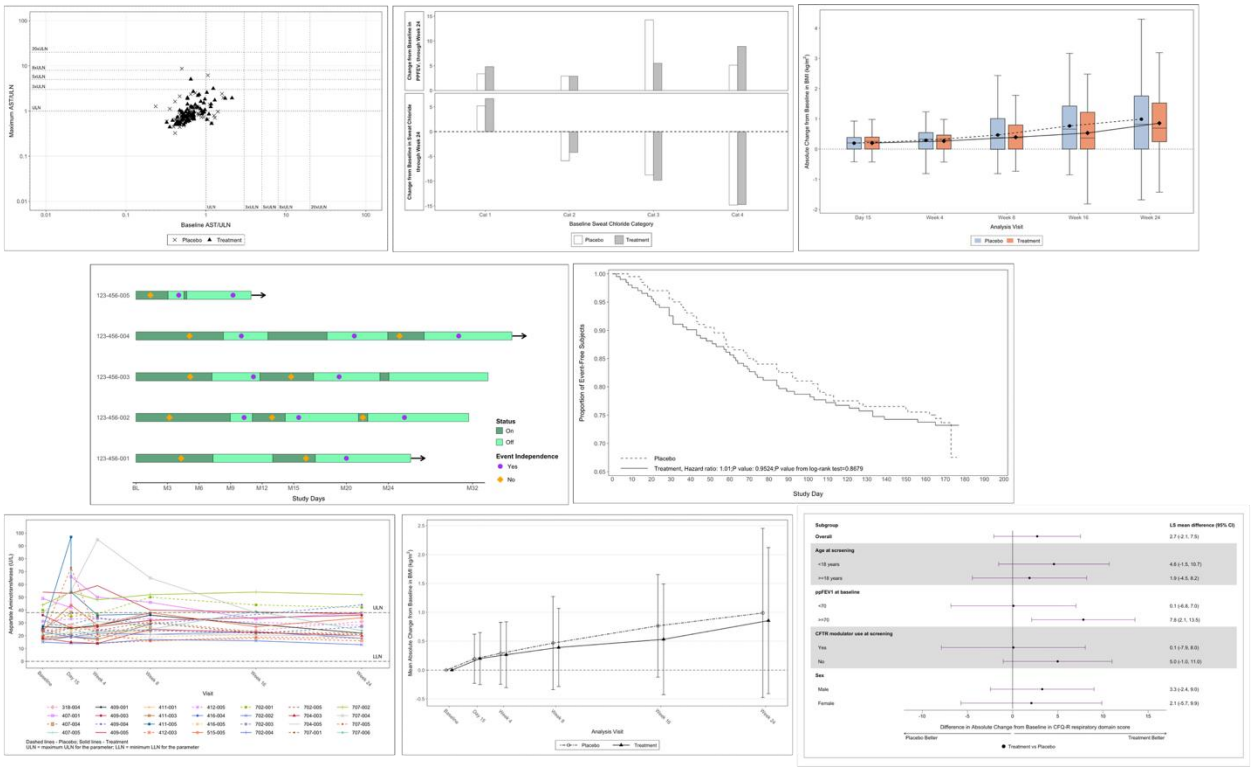

Figure 1. Examples of graphical outputs generated using the proposed graph library, including scatter, bar, box, swimmer, line, spaghetti, forest and Kaplan-Meier plots.

PARAMETER ARCHITECTURE

As summarized in Table 1, the graph library organizes parameters according to the specific component of the plot they control, rather than a large collection of unrelated or narrowly scoped arguments. This architectural principle promotes clarity, predictability, and long-term maintainability across all plotting functions.

With the exception of `r_f_forest`, which is implemented using the *foresploter* framework rather than *ggplot2*, each plotting function begins with three required inputs: the analysis dataset and the x- and y-axis variables. These inputs define the analytical structure of the figure. All subsequent customization options are grouped into logical categories using consistent naming prefixes. For example, legend-related arguments use the `legend_` prefix, point aesthetics use `point_`, bar-specific controls use `bar_`, and error bar settings use `errorBar_`.

Grouping parameters define stratification variables and their display order, influencing both the visual arrangement of data and the sequence of legend entries. Faceting parameters determine how plots are divided into panels, including facet labels and layout direction, thereby supporting multi-panel presentation. Because parameters are grouped semantically, users can infer what a given argument modifies without extensive documentation review.

More specialized graphical elements are similarly isolated into dedicated parameter groups. Reference lines used in forest plots and model overlays used in scatter plots are controlled independently, ensuring that advanced functionality does not complicate routine use cases. Basic plots therefore remain straightforward to construct, while advanced analytical overlays remain optional.

To avoid uncontrolled expansion of function arguments while preserving flexibility, the architecture incorporates two complementary extension mechanisms: `custmz` and the three-dots (…) argument. The `custmz` argument enables users to append native *ggplot2* layers – such as scale adjustment, coordinate transformations, or annotations, without introducing numerous narrowly scoped arguments such as `x_lower` and `x_upper`. For example, axis limits and break definitions can be specified through `scale_*()` or `coord_cartesian()` and passed directly into `custmz`. This approach eliminates parameter explosion while retaining full expressive power.

In contrast, the … argument is reserved exclusively for theme-level styling. Any additional arguments not explicitly defined are automatically captured and passed to the underlying `theme()` function. This allows presentation-level refinements, such as rotating axis tick labels or adjusting legend spacing. By design, `custmz` modifies plot content and analytical layers, whereas … modifies only non-data visual elements. Together, the parameter taxonomy presented in Table 1 establishes a modular and scalable interface for all *ggplot2*-based functions within the library. The architecture reduces unnecessary argument expansion, and allows new plot templates to be incorporated without altering the existing farmwork.

Category	**Purpose**	**Parameters**
Required Inputs	Define the plotting dataset and axes	`Data, X_var, Y_var`
Legend (`legend_*`)	Control legend properties when `show.legend = TRUE`	`legend_title,legend_label, legend_border, legend_location, legend_direction, legend_margin, legend_label_size, legend_fontface`
Point (`point_*`)	Control point appearance (e.g., size, shape)	`point_shape, point_color, point_size`
Bar (`bar_*`)	Control bar appearance	`bar_width, bar_color, bar_opacity, bar_pos`
Error Bar (`errorBar_*`)	Control error bars when `show.meanSE = TRUE`	`errorBar_type, errorBar_width, errorBar_linewidth, errorBar_top_only`
Line (`line_*`)	Control lines or confidence intervals (forest plots)	`line_type, line_color, line_width`
Grouping (`Grouping_*`)	Control data grouping and group order	`Grouping_var, Grouping_order`
Faceting (`facet_*`)	Control paneling and facet labels	`facet_var, facet_label, facet_dir`
Reference Lines (`Xref_line_*`)	Control reference lines in forest plots	`Xref_line_val, Xref_line_type, Xref_line_color, Xref_line_width`
Model Overlay (`model_*`)	Control optional model overlays in scatter plots	`model_formula, model_args`
Output Control	Save the plotting data in `.rds` format for later QC	`output_data`

Category	Purpose	Parameters
Custom Layers	Extend plot content with additional *ggplot2* components (e.g., geoms, annotations, scales)	`custmz`
Custom Theme	Fine-grained styling of non-data elements via arguments passed to `theme()`	...

Table 1. Parameter Architecture of the R Graph Library.

SCATTER PLOT

The `r_f_scatter()` function provides a unified framework for generating scatterplots that adhere to predefined standards. Its purpose is to enhance analytical efficiency and reproducibility by integrating a comprehensive suite of customization options – such as grouping variable, legend control, statistical model overlays, and theme settings – within a single function. This reduces redundancy and ensures consistency across figures.

1. FUNCTION OVERVIEW

```
r_f_scatter <- function(Data, X_var, Y_var,
                        Grouping_var = NULL, Grouping_order = NULL,
                        X_axis_label = waiver(), Y_axis_label = waiver(),
                        show.legend = TRUE,
                        legend_title = "", legend_label = waiver(),
                        legend_border = TRUE,
                        legend_location = "bottom",
                        legend_ncol,
                        legend_margin = c(1,5,1,5),
                        graph_outline = TRUE,
                        point_size = 2.5, point_opacity = 1,
                        point_shape = NULL, point_color = NULL,
                        facet_var, facet_label = waiver(),
                        ticks_label_size = 10, axis_label_size = 11,
                        legend_label_size = 10, legend_fontface = "plain",
                        add_model = NULL, model_formula = "y ~ x",
                        model_args = list(), loess_span = NULL,
                        output_data = FALSE, custmz = NULL, ...)
```

2. METHODOLOGY

The function begins by normalizing grouping behavior in a deterministic manner. When a grouping variable and an explicit display order are provided, the grouping variable is coerced into a factor using a specified level sequence to guarantee consistent legend ordering across figures.

```
Data <- Data %>%
  mutate({{Grouping_var}} := factor({{Grouping_var}}, levels = Grouping_order))
```

If no grouping variable is provided, a dummy grouping factor is internally generated and the legend display is suppressed.

```
Data <- Data %>% mutate(.dummy_group = factor(1))
Grouping_var <- quo(.dummy_group)
show.legend <- FALSE
```

This allows the function to seamlessly support both single-group and multi-group scenarios. In single group settings, unnecessary legend elements are removed; in multi-group settings, the grouping variable drives visual differentiation.

Additionally, when no grouping variable is present and no explicit point color is defined, the function defaults to black points.

```
if ((missing(Grouping_var) || length(legend_label) == 1) && (is.null(point_color))) {
  point_color <- "black"
}
```

This design ensures stable aesthetic mapping across runs and prevents unintended legend generation.

After preprocessing, the base `ggplot` object is constructed using a unified aesthetic mapping in which both shape and color are mapped to the grouping variable.

```
ggplot(
  aes(x = {{X_var}}, y = {{Y_var}},
      shape = {{Grouping_var}},
      color = {{Grouping_var}})) +
  geom_point(# redacted code)
```

Encoding grouping through both color and shape increases the generality and flexibility, as groups can be distinguished by either attribute or a combination of the two.

Manual aesthetic scales are applied only when explicitly requested, maintaining flexibility without overriding automated behavior.

```
if (!is.null(point_shape))
  scale_shape_manual(# redacted code)

if (!is.null(point_color))
  scale_color_manual(# redacted code)
```

Beyond static scatter visualization, the function optionally supports statistical overlays via the `add_model` argument, which directly interfaces with `geom_smooth()`. This means users can apply the same smoothing methods supported by *ggplot2*, such as linear regression or loess smoothing.

```
if (!is.null(add_model)) {
  p <- p + geom_smooth(# redacted code)
}
```

Extensibility is achieved through a layered customization mechanism. The `custmz` argument appends complete `ggplot` components after core plot construction.

```
for (custm in custmz) {
  p <- p + custm
}
```

This enables structural additions, such as reference lines, coordinate transformations, or scale modifications, without altering internal logic.

The function then integrates a centralized styling layer built on `theme_bw()`, such as standardized typographic settings, optionally drawing panel borders, etc. For instance,

```
theme_bw() +
theme(
  panel.grid.major = element_blank(),
  panel.grid.minor = element_blank(),
  panel.border = if (graph_outline) element_rect(linewidth = 0.3) else element_blank(),
  axis.text = element_text(size = ticks_label_size),
  axis.title = element_text(size = axis_label_size),
  legend.text = element_text(size = legend_label_size, face = legend_fontface),
  plot.background = element_rect(colour = "black"),
  ...
)
```

This ensures consistent formatting across figures without repeating theme settings for every plot. The ellipsis (…) allows users to add extra styling options to the internal `theme()` call, enabling refinement of non-data elements while preserving a consistent overall structure. A list of supported arguments is available in the *ggplot2* `theme()` documentation.

Stratified displays are supported through `facet_grid()` with free y-scaling.

```
facet_grid(
  stats::as.formula(paste(paste(facet_var, collapse = "+"), "~ .")),
  scales = "free_y", space = "free_y", switch = "y"
)
```

The use of `free_y` and `space = "free_y"` allows panel heights to adjust dynamically according to data range. Clipping is disable at the final stage, e.g., `p$coordinates$clip <- "off"` to prevent truncation of annotations or inside-positioned legends, ensuring complete graphical rendering. Additionally, when `output_data = TRUE`, the function supports downstream QC by exporting the plotting dataset as an `.rds` file.

Importantly, many of these structural components are implemented as library-wide standards rather than function-specific logic. Core behaviors such as grouping normalization, optional factor ordering via `Grouping_order`, default color assignment when no grouping variable is present, automatic legend suppression for single-group displays, faceting support, and predefined theme settings, are designed to be consistently inherited across plotting functions where applicable.

3. EXAMPLES

Figure 2 plots baseline AST versus maximum AST (RA2ULN) for safety-population subjects, where each point represents an individual patient and treatment groups are distinguished by shape. Both axes are displayed on a logarithmic scale with reference lines marking clinically relevant ULN multiples (1x, 3x, 5x, 8x, 20x).

```
# Customize additional elements for the plot
# Modify the customizations to fit your needs
extra_custom <- c(
  # Apply a logarithmic scale to the y-axis with specified breaks and labels
  scale_y_log10(breaks = c(0.01, 0.1, 1, 10, 100),
                labels = c(0.01, 0.1, 1, 10, 100)),
  # Apply a logarithmic scale to the x-axis with specified breaks and labels
  scale_x_log10(breaks = c(0.01, 0.1, 1, 10, 100),
                labels = c(0.01, 0.1, 1, 10, 100)),
  # `xlim = c(0.01, 100)`: focus the x-axis on the range of interest
  # Similarly for `ylim = c(0.01, 100)`
  # `clip = "off"`: allow elements to extend outside the plotting area
  coord_cartesian(xlim = c(0.01, 100), ylim = c(0.01, 100), clip = "off"),
  # Add horizontal lines for specific ULN levels
  geom_hline(yintercept = c(1, 3, 5, 8, 20),
             linetype = "dotted", linewidth = 0.3),
  annotate("text", x = 0.0065, y = c(1.2, 3.5, 5.8, 9.2, 24),
           label = paste("", c("ULN", "3xULN", "5xULN", "8xULN", "20xULN")),
           size = 2.8, hjust = 0),
  # Add vertical lines for specific ULN levels
  geom_vline(xintercept = c(1, 3, 5, 8, 20),
             linetype = "dotted", linewidth = 0.3),
  annotate("text", y = 0.0075, x = c(1, 3, 5, 8, 20),
           label = paste("", c("ULN", "3xULN", "5xULN", "8xULN", "20xULN")),
           size = 2.8, hjust = 0)
)

p1 <- r_f_scatter(
  Data = f_lb_ast_base, X_var = Baseline, Y_var = Maximum,
  Grouping_var = TRT01A,
  X_axis_label = "Baseline AST/ULN",
  Y_axis_label = "Maximum AST/ULN",
  point_shape = c(4, 17),
  point_color = rep("black", 2),
  output_data = TRUE, custmz = extra_custom,
  panel.grid.major.x = element_line(colour = "grey", linewidth = 0.1),
  panel.grid.major.y = element_line(colour = "grey", linewidth = 0.1)
)
```

Any extra theme arguments passed through …

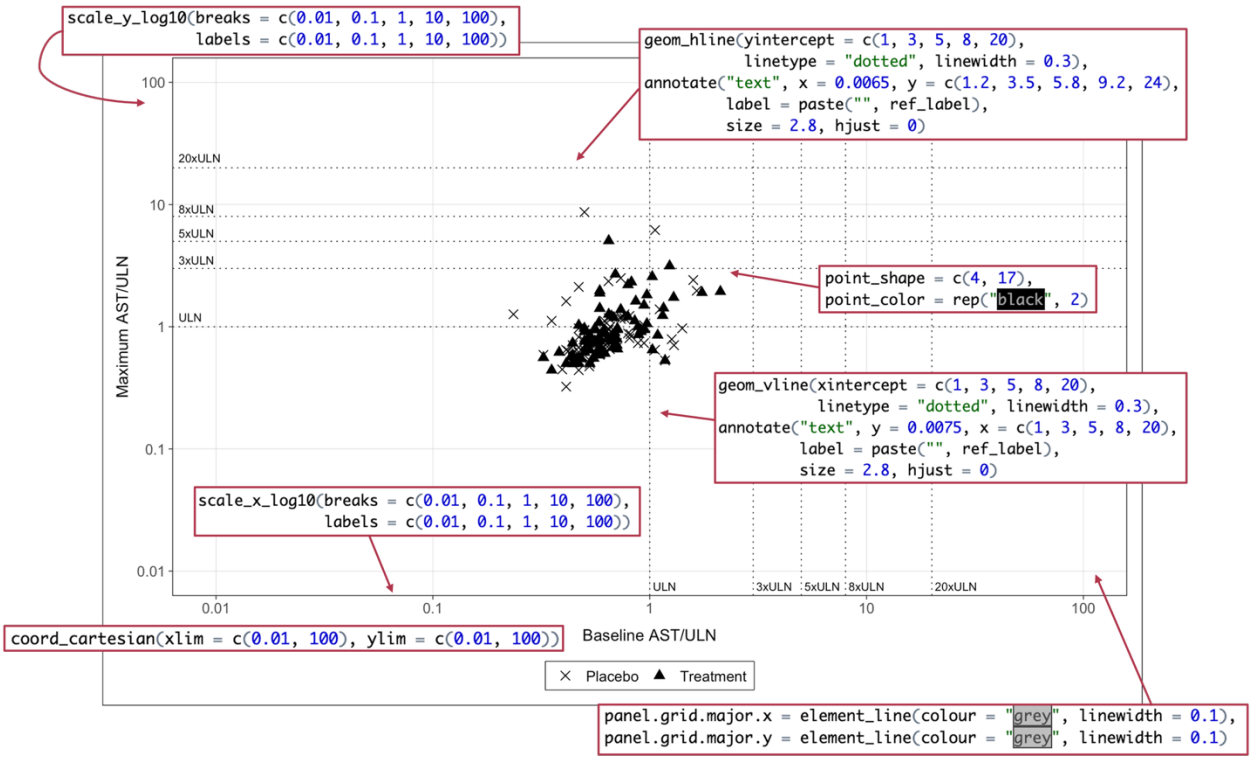

Figure 2. Scatterplot of maximum aspartate transaminase (AST) versus baseline during the treatment-emergent period, generated using R.

In this example, the custmz argument is used to append full `ggplot` components, such as `scale_x_log10()`, `scale_y_log10()`, `coord_cartesian()`, `geom_hline()`, and `geom_vline()`, which control axis transformation, limits, breaks, reference lines, and annotation layers. These are structural modifications that affect the geometry and coordinate system of the plot.

By contrast, the additional arguments passed through … (e.g., `panel.grid.major.x` and `panel.grid.major.y`) are passed directly into the internal `theme()` call, allowing targeted stylistic refinement without rewriting the plot or stacking external theme calls.

Instead of introducing numerous axis-specific arguments such as `x_lower`, `x_upper`, `x_trans`, `x_breaks`, or `x_labels` into the function signature, axis behavior is controlled through native *ggplot2* scale and coordinate functions inside `custmz`. As a result, transformations, limits, breaks, labels, and spacing can be managed in a single cohesive call while preserving a compact and stable function interface.

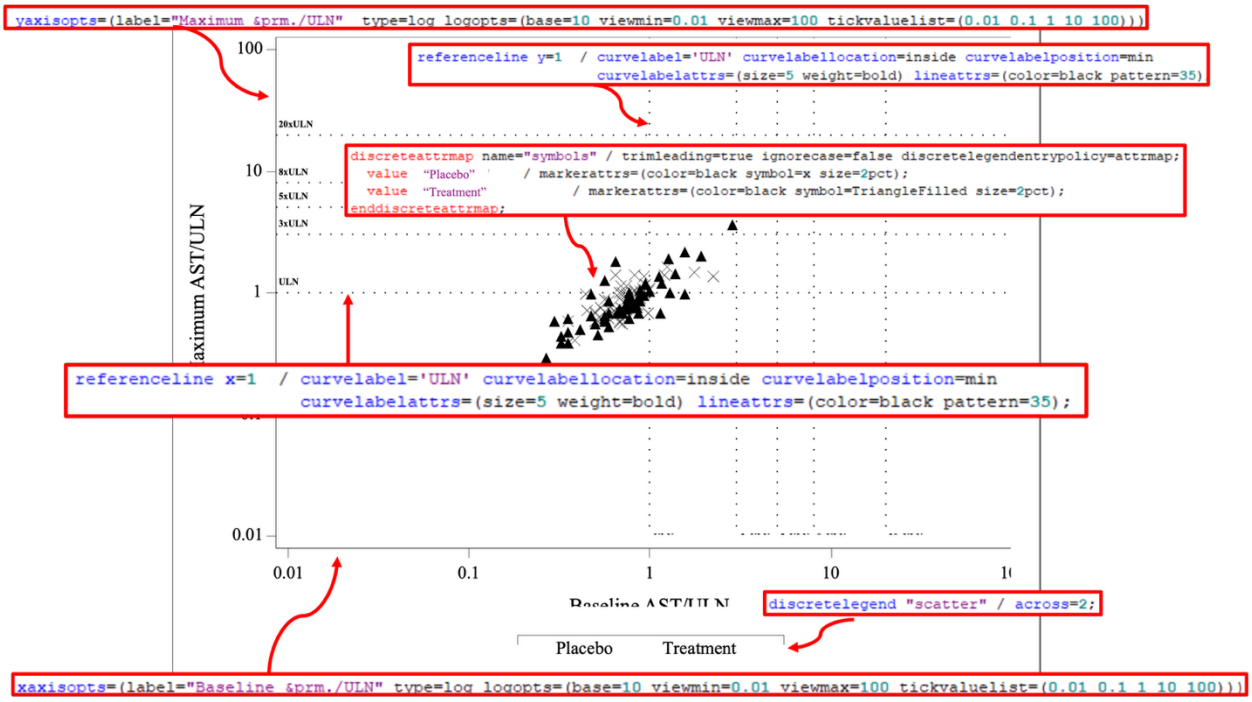

Figure 3. SAS output corresponding to Figure 2.

Compared with SAS, the `geom_hline()` in R allows multiple horizontal reference lines to be added in a single call by providing a vector to `yintercept`, such as `geom_hline(yintercept = c(1, 3, 5, 8, 20))`. This vectorized approach makes it easy to draw several clinically relevant thresholds at once while applying consistent styling (e.g., linetype, color, linewidth) in one statement. Labels for these lines can also be added programmatically using `annotate()` with matching vectors, keeping the structure concise and scalable. In contrast, SAS typically requires separate `referenceline` statements or more verbose configuration within procedural blocks, making the code longer and less modular when multiple reference lines are needed.

BAR PLOT

The `r_f_bar()` function is designed to construct structured bar charts with flexible support for grouping, summarization, and uncertainty display. It supports both pre-aggregated data and raw observations, allowing summary statistics and error bars to be computed internally when needed. The function further incorporates optional grouped bar positioning, horizontal orientation, legend confirmation, and faceting controls.

1. FUNCTION OVERVIEW

```
r_f_bar <- function(#--- Shared arguments (same as r_f_scatter)
                    # Data, X_var, Y_var, etc.
                    bar_width = 0.6,
                    bar_color = NULL, bar_opacity = 1,
                    point_size = 1.5,
                    show.errorBar = FALSE, isSummarized = TRUE,
                    meanmed_var, error_var,
                    summary_fun = mean,
                    errorBar_type = "SD",
                    errorBar_top_only = FALSE,
                    errorBar_width = 0.2,
                    errorBar_linewidth = 0.4,
                    bar_pos = position_dodge(),
                    isHorizontal = FALSE,
                    output_data = FALSE, custmz = NULL, ...)
```

2. METHODOLOGY

A central mechanism in `r_f_bar` is the `isSummarized` argument, which determines how input data are processed. When `isSummarized = TRUE`, the function assumes that summary statistics (e.g., mean and corresponding variability) are already provided in the dataset. In this case, bar heights are drawn directly using `geom_bar(stat = "identity")`. In contrast, when `isSummarized = FALSE`, the function operates on raw observations and internally computes summary statistics using a specified `summary_fun`, (e.g., mean, median, sum, length) via `stat_summary()`. However, it is important to use caution and verify that the algorithm being used in R aligns with the intended statistical definition and analysis requirements. The same mechanism can also be applied to other functions in the library, such as line plots.

```
if (isSummarized) {
  ggplot(aes(x = {{X_var}}, y = {{Y_var}}, fill = {{Grouping_var}})) +
    geom_bar(# redacted code)
} else {
  ggplot(aes(x = {{X_var}}, y = {{Y_var}}, fill = {{Grouping_var}})) +
    stat_summary(# redacted code)
}
```

The second major component is uncertainty visualization, controlled by `show.errorBar`. For summarized inputs (`isSummarized = TRUE`), error bars are computed directly from user-provided center (`meanmed_var`) and error variables (`error_var`). The function supports both the conventional two-sided error bar (mean ± error) and a one-sided "top-only" variant through `errorBar_top_only`.

```
if (show.errorBar) {
  if (!errorBar_top_only) {
    # Two-sided: mean ± error
    geom_errorbar(# redacted code)
  } else {
    # Top-only: mean to mean + error
    geom_errorbar(# redacted code)
  }
  # Overlay point to mark the mean/median explicitly
  geom_point(# redacted code)
}
```

To display confidence intervals (CI), users must pre-calculate the CI limits and set `isSummarized = TRUE`, and provide corresponding `meanmed_var` and `error_var`.

For raw data (`isSummarized = FALSE`), error bars are calculated internally using `stat_summary()`. In the two-sided case, SD or SE intervals are generated via `mean_sdl` or `mean_se` depending on `errorBar_type`. In the top-only case, custom `fun.data` functions return (ymin, ymax) where ymin is the center and ymax is the center ± SD/SE. A summary point is then overlaid using the same `summary_fun` to ensure the center marker aligns with the bar height definition. CI are not supported in this raw-data mode.

```
if (!errorBar_top_only) {
  # Two-sided: mean ± error
  if (toupper(errorBar_type) == "SD") {
    p <- p + stat_summary(
      fun.data = mean_sdl, # redacted code
    )
  } else if (toupper(errorBar_type) == "SE") {
    p <- p + stat_summary(
      fun.data = mean_se, # redacted code
    )
  }
} else {
  # Top-only: mean to mean + error
  if (toupper(errorBar_type) == "SD") {
    p <- p + stat_summary(
      fun.data = function(x) {# redacted code},
      # redacted code
    )
  } else if (toupper(errorBar_type) == "SE") {
    p <- p + stat_summary(
      fun.data = function(x) {# redacted code},
      # redacted code
    )
  }
}

# Center marker matches summary_fun
p <- p + stat_summary(
  fun = summary_fun, geom = "point", # redacted code
)
```

In addition to faceting, the function supports alternative layouts through `isHorizontal`, which flips the coordinate system with `coord_flip()` for horizontal bars.

When `isSummarized = FALSE`, the function uses `ggplot_build(p)$data` to extract the computed plotting data instead of saving the original raw dataset. This ensures that the exported output reflects the exact internally calculated values, such as mean, SD, or SE, that are ultimately displayed in the figure.

```
if (output_data) {
  if (isSummarized) {
    saveRDS(Data, file_path)
  } else {
    # Retrieve computed plot data (post-statistical transformation)
    built <- ggplot_build(p)

    # Attach layer and geometry metadata
    geom_class <- # redacted code
    built$data <- # redacted code
    saveRDS(built$data, file_path)
  }
}
```

In this step, `geom_class` is generated to identify the geometry associated with each extracted plotting layer. It retrieves the geometry type from each layer in `p$layers`, producing a vector such as `"GeomLine"`, `"GeomPoint"`, or `"GeomErrorBar"`. These identifiers are then added to the extracted data so reviewers can clearly see which computed values belong to each plotted component.

3. EXAMPLES

Example 1

Figure 4 displays the distribution of subjects across categories of absolute change from baseline in BMI at week 24, stratified by treatment group. Each bar represents the total number of subjects in the corresponding category. Two implementation approaches are shown: `bar_p1a` uses pre-summarized counts (`df1_summary`), where the number of subjects (n) is computed for each category, while `bar_p1b`

uses raw subject-level data (`df1_raw`) and computes counts internally via `summary_fun = length`. Different sets of colors are applied to differentiate the two implementations.

```
extra_custom <- c(
  scale_y_continuous(breaks = breaks_width(2), expand = c(0, 0)),
  coord_cartesian(ylim = c(0, 15))
)

# Use pre-summarized data
bar_p1a <- r_f_bar(
  Data = df1_summary, X_var = chgcat, Y_var = n,
  Grouping_var = TRT01A,
  X_axis_label = expression("Absolute Change from Baseline in BMI (kg/m"^2*") at Week
24"),
  Y_axis_label = "Number of Subjects",
  bar_color = c("white", "grey"),
  custmz = extra_custom,
  axis.text.x = element_text(angle = -60, hjust = 0),
  legend.key.spacing.x = unit(0.8, "cm")
)

# Use raw data
bar_p1b <- r_f_bar(
  Data = df1_raw, isSummarized = FALSE,
  X_var = chgcat, Y_var = CHG,
  Grouping_var = TRT01A,
  X_axis_label = expression("Absolute Change from Baseline in BMI (kg/m"^2 * ") at Week
24"),
  Y_axis_label = "Number of Subjects",
  bar_color = c("#66C2A5", "#FC8D62"),
  summary_fun = length,
  bar_pos = position_dodge(preserve = "single"),
  custmz = extra_custom,
  axis.text.x = element_text(angle = -60, hjust = 0),
  legend.key.spacing.x = unit(0.8, "cm")
)
```

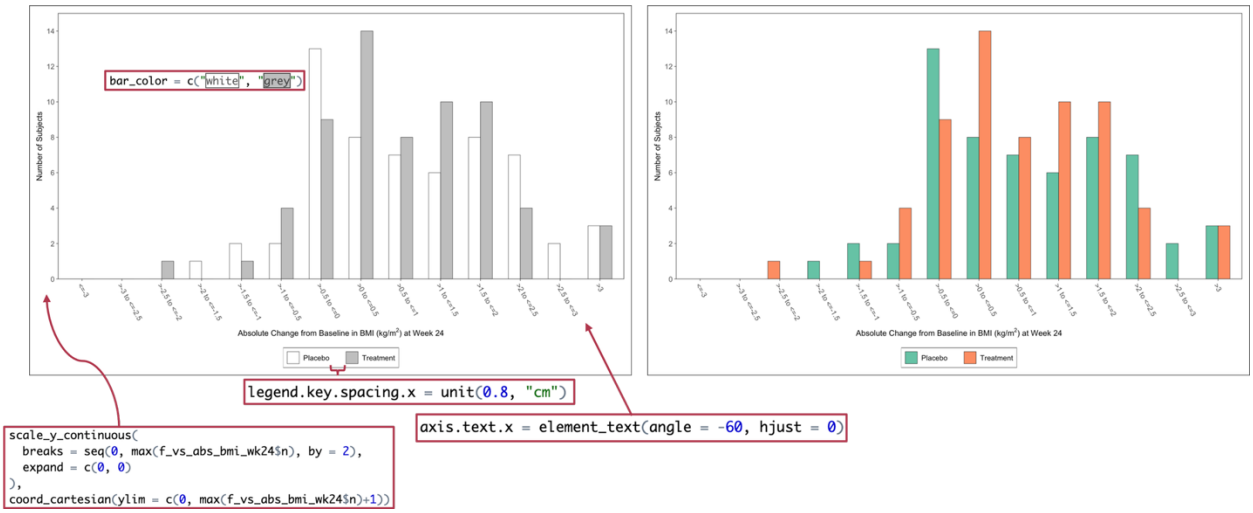

Figure 4. Bar plot of absolute change from baseline in BMI at week 24, generated using R. Left uses pre-summarized data, right uses raw data with internally computed counts.

In the SAS code shown in Figure 5, bar colors are specified through separate `style GraphData1/GraphData2` statements, and legend positioning is handled through an additional `discretelegend` statement. In our function template, both elements are defined directly within the same function call: `bar_color = c(…)` explicitly sets the fill colors and in order, and legend properties,

such as title, position, spacing, or suppression, are managed through arguments like `legend_title`, `legend_location`, and `show.legend`. This integration keeps aesthetic mapping and legend behavior localized within the plotting call, making the code more concise, easier to adjust, and readily reusable without redefining external style elements.

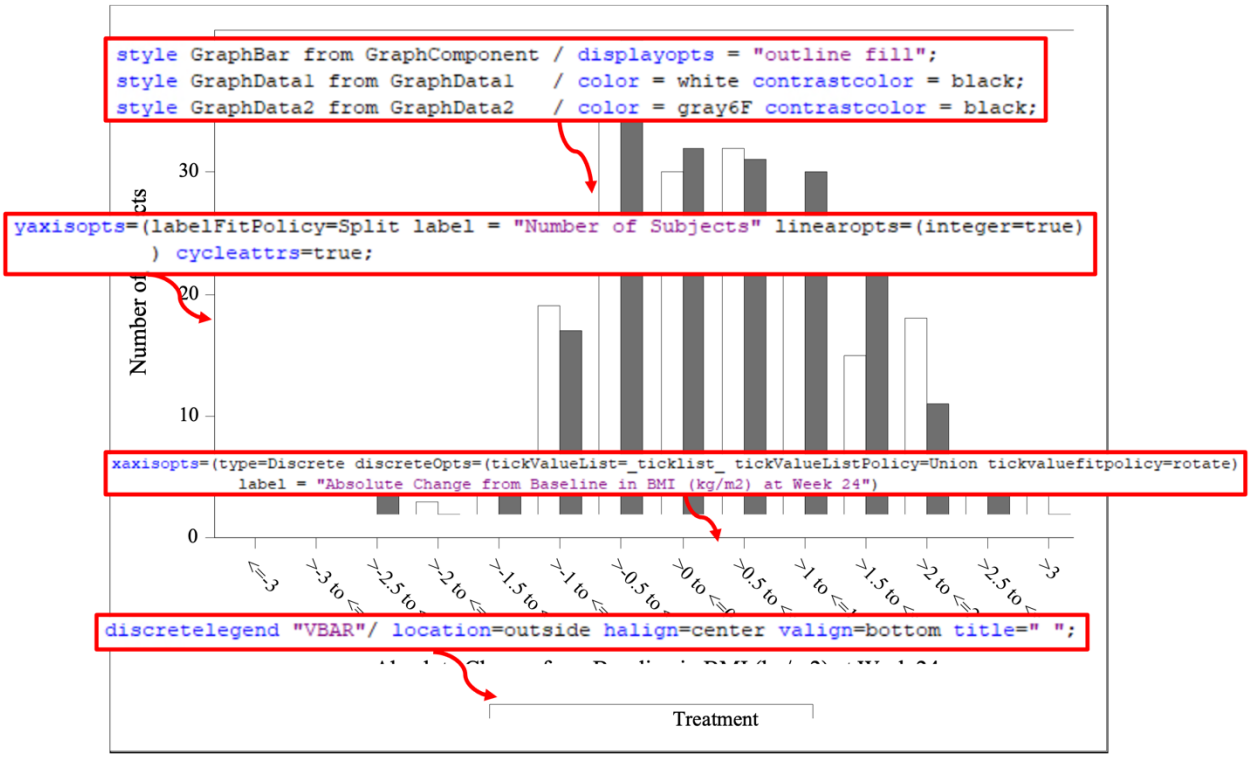

Figure 5. SAS output corresponding to Figure 4.

Example 2 – Error Bars

Figure 6 presents mean BMI at baseline and week 24 by treatment group, with error bars indicating variability around the mean. `bar_p2a` is based on pre-summarized data (`df2_summary`), where means and standard deviations are computed for each combination of AVISIT and TRT01A, whereas `bar_p2b` is generated from raw data (`df2_raw`) with means and standard deviations computed internally (`isSummarized = FALSE`).

```r
extra_custom <- c(
  scale_y_continuous(breaks = breaks_width(5), expand = c(0, 0)),
  coord_cartesian(ylim = c(0, 29))
)

# Use pre-summarized data
bar_p2a <- r_f_bar(
  Data = df2_summary, X_var = AVISIT, Y_var = mean_AVAL,
  Grouping_var = TRT01A,
  X_axis_label = "Analysis Visit",
  Y_axis_label = expression("Mean BMI (kg/m"^2*")"),
  bar_width = 0.2, bar_color = c("white", "grey"),
  show.errorBar = TRUE, meanmed_var = mean_AVAL, error_var = sd_AVAL,
  errorBar_width = 0.05,
  custmz = extra_custom,
  legend.key.spacing.x = unit(0.8, "cm")
)

# Use raw data
bar_p2b <- r_f_bar(
  Data = df2_raw, isSummarized = FALSE,
  X_var = AVISIT, Y_var = AVAL, Grouping_var = TRT01A,
  X_axis_label = "Analysis Visit",
  Y_axis_label = expression("Mean BMI (kg/m"^2*")"),
  bar_width = 0.2, bar_color = c("#66C2A5", "#FC8D62"),
  show.errorBar = TRUE, errorBar_width = 0.05,
  custmz = extra_custom,
  legend.key.spacing.x = unit(0.8, "cm")
)
```

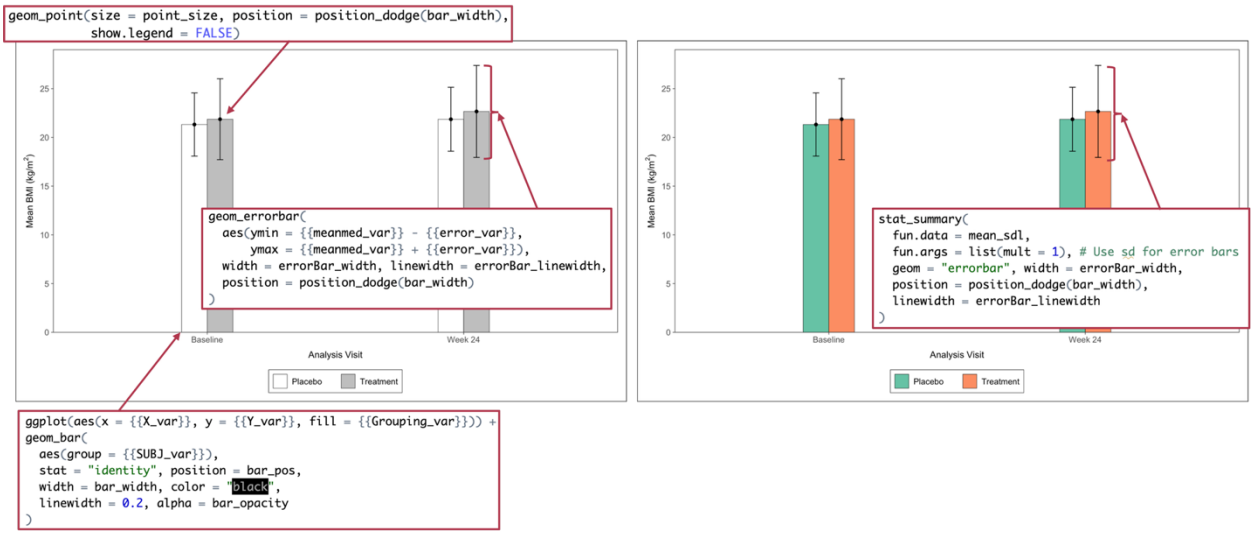

Figure 6. Bar plot of mean BMI at baseline and week 24 by treatment group, generated using R. Left uses pre-summarized data, right uses raw data with internally computed means and standard deviations.

Example 3 – Stratification

Figure 7 visualizes change-from-baseline outcomes through week 24 by baseline sweat chloride (SW) category (SWBLCAT), stratified by treatment group. `bar_p3a` uses pre-summarized results (`df3_summary`), where mean changes for SW and $ppFEV_1$ are already computed for each SWBLCAT. `bar_p3b` recreates the same figure from raw dataset (`df3_raw`) with `isSummarized = FALSE`, allowing mean change is calculated dynamically for each SWBLCAT and TRT01A using default `summary_fun = mean` applied to the raw CHG values.

```
# Use pre-summarized data
extra_custom <- c(
  geom_hline(data = subset(df3_summary, PARAMCD == "mean_SWCHG"),
             aes(yintercept = 0), linetype = "dashed", color = "black")
)

bar_p3a <- r_f_bar(
  Data = df3_summary, X_var = SWBLCAT, Y_var = mean_CHG,
  Grouping_var = TRT01A,
  X_axis_label = "Baseline Sweat Chloride Category",
  Y_axis_label = "",
  bar_width = 0.2, bar_color = c("white", "grey"),
  facet_var = "PARAMCD",
  facet_label = c(
    "mean_SPCHG" = "bold(atop(Change~from~Baseline~'in'~PPFEV[1], through~Week~'24'))",
    "mean_SWCHG" = "bold(atop(Change~from~Baseline~'in'~Sweat~Chloride,
through~Week~'24'))"
  ),
  custmz = extra_custom,
  legend.key.spacing.x = unit(0.8, "cm"),
  # Place facet strips outside
  strip.placement = "outside",
  # Customize strip background
  strip.background = element_rect(fill = "white", colour = "black")
) +
  # Set individual scales in facets (using ggh4x package)
  facetted_pos_scales(
    y = list(scale_y_continuous(limits = c(0, 16), breaks = breaks_width(5),
                                expand = c(0, 0)),
             scale_y_continuous(breaks = breaks_width(5)))
  )

# Use raw data
bar_p3b <- r_f_bar(
  Data = df3_raw, X_var = SWBLCAT, Y_var = CHG,
  Grouping_var = TRT01A,
  X_axis_label = "Baseline Sweat Chloride Category",
  Y_axis_label = "",
  bar_width = 0.2, bar_color = c("#66C2A5", "#FC8D62"),
  facet_var = "PARAMCD",
  facet_label = c(
    "SPCHG" = "bold(atop(Change~from~Baseline~'in', PPFEV[1]~through~Week~'24'))",
    "SWCHG" = "bold(atop(Change~from~Baseline~'in'~Sweat~Chloride, through~Week~'24'))"
  ),
  custmz = extra_custom,
  isSummarized = FALSE,
  legend.key.spacing.x = unit(0.8, "cm"),
  strip.placement = "outside",
  strip.background = element_rect(fill = "ivory", colour = "black"),
  panel.spacing = unit(2, "lines")
)

# Get current axis limits
plot_build <- ggplot_build(bar_p3b)
ylim <- plot_build$layout$panel_scales_y[[1]]$range$range

bar_p3b <- bar_p3b +
  facetted_pos_scales(
    y = list(scale_y_continuous(limits = c(0, ylim[2] + 2), breaks = breaks_width(5),
                                expand = c(0, 0.04)),
             scale_y_continuous(breaks = breaks_width(5)))
  )
```

`facetted_pos_scales()` enables panel-specific axis customization within a faceted plot. Rather than imposing identical y-axis limits, breaks, or expansions across all panels, it allows each facet to use its own `scale_y_continuous()` settings, ensuring that each outcome is displayed on an appropriate and

readable scale.

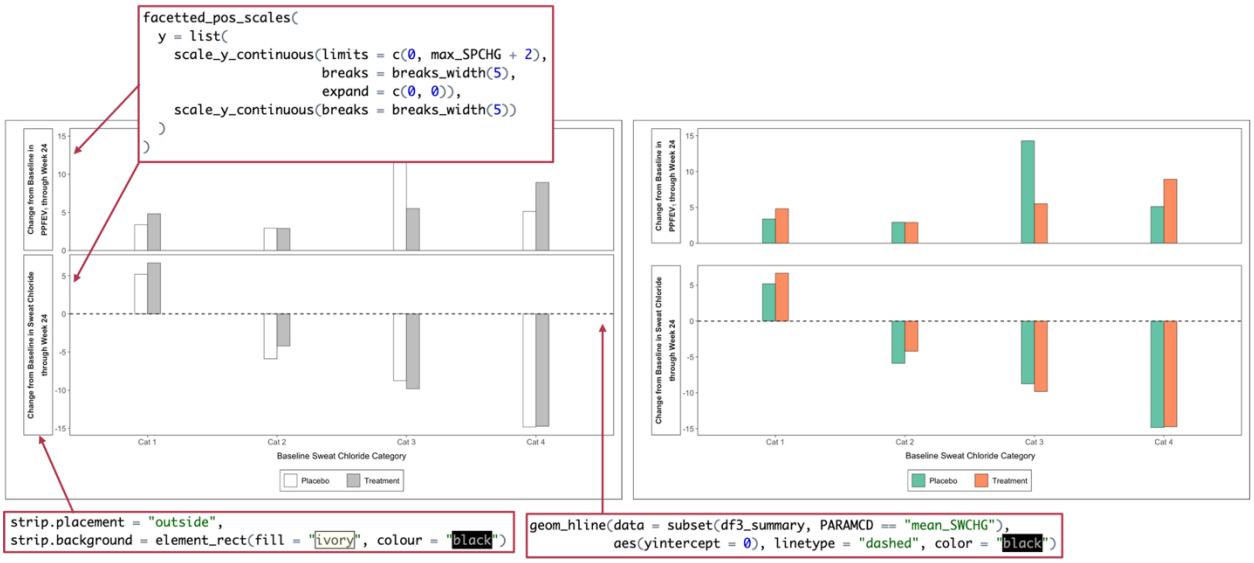

Figure 7. Bar plot of mean absolute change from baseline in ppFEV$_1$ and sweat chloride through week 24, generated using R. Left uses pre-summarized data, right uses raw data with internally computed means.

FOREST PLOT

The `r_f_forest()` function is built upon *forestploter* (v1.1.3) to produce customizable and flexible forest plots. The forest plot is generated directly from the structure of the input dataset, allowing CI to be displayed across multiple columns or grouped categories with ease. The function also supports CI column placement, alternating row banding, reference lines, optional arrow labels, legend control, and bold formatting for selected rows.

1. FUNCTION OVERVIEW

```
r_f_forest <- function(Data, est, lower, upper,
                       CI_spacing = 180, CI_location = 2,
                       X_axis_label,
                       Xref_line_val = 0, Xref_line_type = 1,
                       Xref_line_color = "black", Xref_line_width = 1,
                       point_size = 0.5, point_color = "black", point_shape = 16,
                       line_type = 1, line_width = 1, line_color = "black",
                       cell_spacing = c(0.6, 0.3),
                       show.legend = TRUE,
                       legend_title = "", legend_label = "",
                       legend_location = "bottom",
                       arrow_label = NULL,
                       X_limit, X_tick,
                       band_group, band_color,
                       bold_rows = NA, output_data = FALSE)
```

2. METHODOLOGY

Unlike the other plotting functions, `r_f_forest()` does not use explicit x and y aesthetic mappings. Instead, the `Data` argument represents the table structure to be displayed in the forest plot. The graphical output is generated based on the layout of this input dataset, rather than mapping variables to cartesian coordinates. The table should include the relevant text fields, such as subgroup labels and formatted statistics (e.g, LS mean differences with 95% CI). The column names are drawn as headers. The numeric vectors `est`, `lower`, and `upper` define the point estimates and corresponding CI.

To control where the CI graphic appears, the function inserts one or more temporary placeholder columns at a user-defined `CI_location`, where the widths of the CI region are controlled by `CI_spacing`. By default, setting `CI_location = 2` places the CI values in the second column of the table.

```
# Insert CI placeholder columns at specified locations
insert_after <- CI_location - 1

for (i in seq_along(insert_after)) {
  Data <- Data %>%
    mutate(!!paste0("tmp", CI_location[i]) := paste(rep(" ", CI_spacing), collapse = ""),
           .after = insert_after[i])
}
```

A custom theme object is created using `forest_theme()` to standardize formatting of CI lines, point markers, reference lines, table cell padding, and etc. For example,

```
tm <- forest_theme(
  core = list(padding = unit(cell_spacing, "in"),
              bg_params = list(fill = band_color)),
  ci_col = line_color,
  ci_fill = point_color,
  vertline_col = Xref_line_color,
  # redacted code
)
```

The main forest plot is constructed by passing the estimate, lower, and upper confidence bounds to the plotting engine.

```
args_list <- list(
  Data,
  est = est,
  lower = lower,
  upper = upper,
  ci_column = CI_location,
  vert_line = Xref_line_val,
  theme = tm
)
```

Optional row banding is supported through `band_group`. If grouping information is provided, alternating background colors are generated automatically. Selected rows can also be highlighted using bold formatting through post-processing.

```
p <- do.call(forest, args_list) %>%
  edit_plot(row = bold_rows, gp = gpar(fontface = "bold")) %>%
  # redacted code

p$coordinates$clip <- "off"
```

3. EXAMPLES

Example 1

Figure 8 shows subgroup least squares (LS) mean differences and their 95% CI. The subgroup labels and formatted CI text are drawn directly from the dataset, while the numeric estimates and bounds are used to render the graphical CI lines. Row banding is applied based on group information.

```
forest_p1 <- r_f_forest(
  Data = df[c("Subgroup", "LS mean difference (95% CI)")],
  est = df$lsmean, lower = df$Lower, upper = df$Upper,
  band_group = df$group,
  X_axis_label = "Difference in Absolute Change from Baseline in CFQ-R respiratory
domain score",
  legend_label = "Treatment vs Placebo",
  line_color = "#762a83",
  X_limit = c(-12, 17),
  X_tick = seq(-10, 15, by = 5),
  arrow_label = c("Placebo Better", "Treatment Better")
)
```

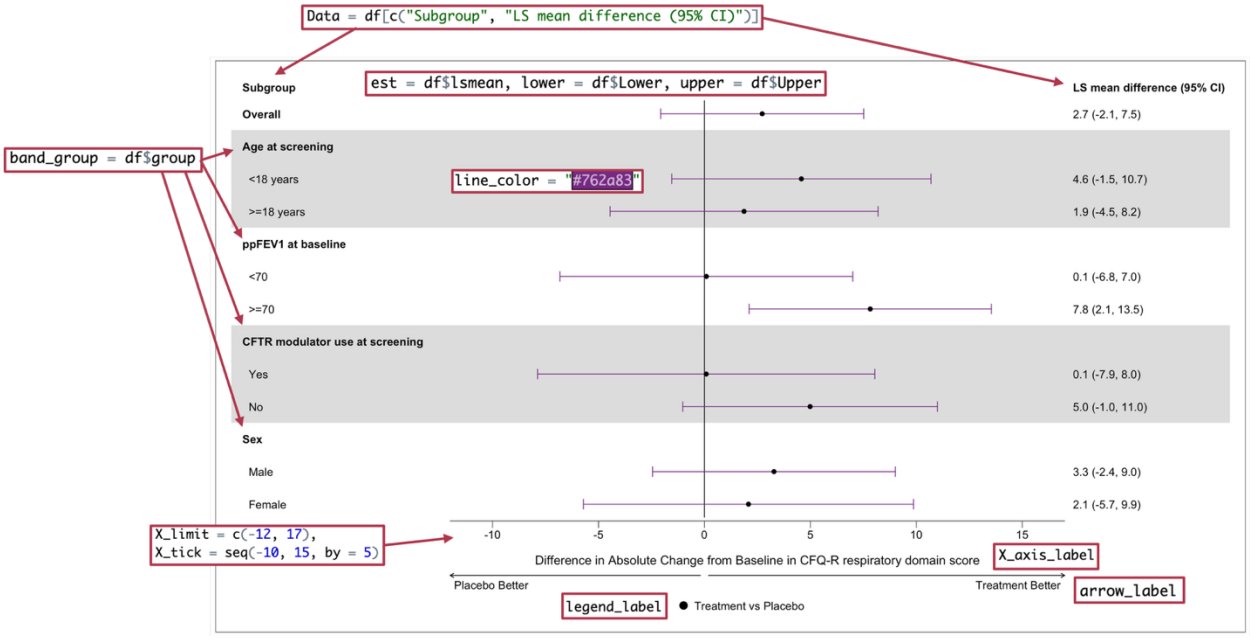

Figure 8. Forest plot of LS mean difference between treatments with 95% CI for absolute change from baseline in CFQ-R respiratory domain score through week 24 by subgroup, generated using R.

Example 2 – Multiple CI Columns

In contrast, Figure 9 extends the structure by displaying two outcomes side by side within the same table layout, which may originate from different datasets. Instead of a single set of estimates, the function accepts lists of estimates and confidence bounds to generate two CI panels simultaneously. The argument `CI_location = c(2, 4)` specifies the placement of the two graphical CI columns, enabling aligned dual-outcome comparison within a unified subgroup table.

```
forest_p2 <- r_f_forest(
  Data = cbind(sp[c("Subgroup", "LSM diff. (95% CI)")], sw["LSM diff. (95% CI)"]),
  est = list(sp$med, sw$med),
  lower = list(sp$low, sw$low),
  upper = list(sp$high, sw$high),
  CI_spacing = 80,
  band_group = sp$group,
  X_axis_label = c("Difference in Absolute Change from Baseline in ppFEV1",
                   "Difference in Absolute Change from Baseline in SwCl"),
  legend_label = "Treatmnet vs Placebo",
  CI_location = c(2, 4),
  line_color = "#762a83",
  X_limit = list(c(-10.5, 10.5), c(-19.5, 19.5)),
  X_tick = list(seq(-10, 10, by = 5), seq(-19, 19, by = 5)),
  cell_spacing = c(0.25, 0.18)
)
```

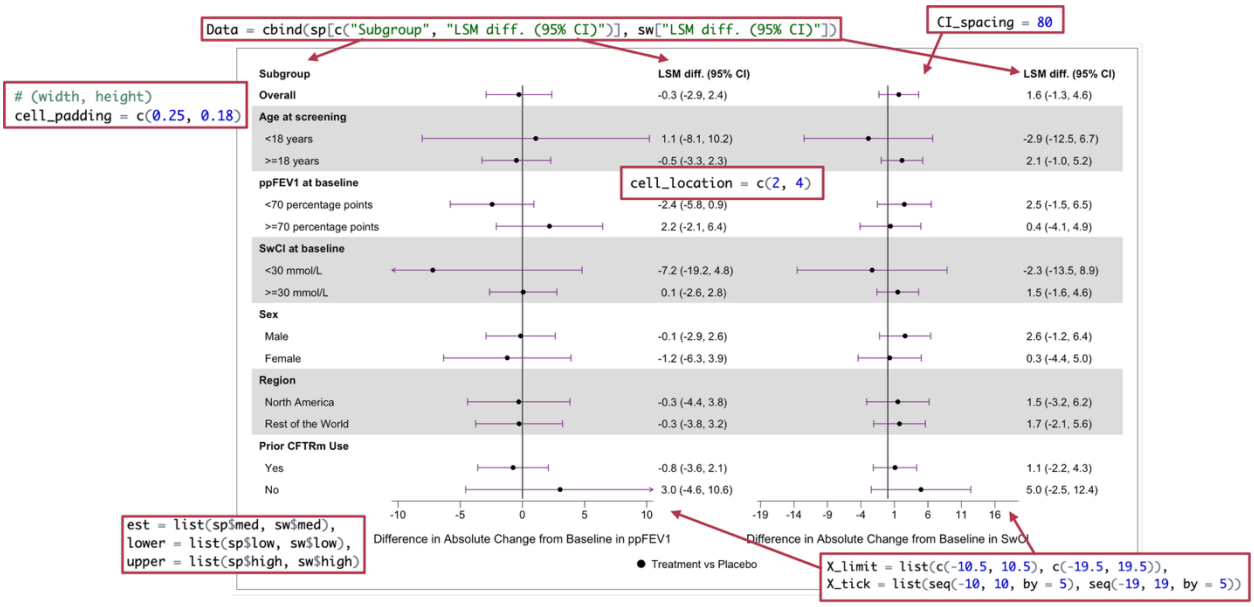

Figure 9. Forest plot of LS mean difference between treatments with 95% CI for absolute change from baseline in ppFEV$_1$ (percentage points) and sweat chloride (mmol/L) through week 24 by subgroup, generated using R.

CONCLUSION

This work presents a standardized R graph library for generating CSR figures through reusable plotting templates. By replacing repetitive figure-specific coding with a structured and consistent framework, the library improves efficiency, readability, and reproducibility across commonly used graph types. Its parameter architecture supports both standardization and controlled flexibility, allowing users to customize figures without disrupting the overall design. The examples show that CSR figures can be produced with shorter, clearer code while maintaining alignment with reporting standards. Hence, this template-based approach streamlines figure production, reduces formatting inconsistency, and provides a practical foundation for scalable, production-ready clinical graphics.

The library is designed to work seamlessly within reproducible and automated analysis pipelines, as each plotting function is parameter-driven and generates deterministic outputs from clearly defined inputs. This makes the templates well suited for scripted workflows such as batch figure generation, reproducible report builds, and version-controlled environments, where figures can be regenerated consistently across studies or analysis updates with minimal manual intervention. By standardizing plotting logic in reusable

templates, the library improves operational efficiency and ensures that figures remain consistent throughout the reporting lifecycle.

In addition, the use of standardized templates provides several advantages when integrated into large language model (LLM)-assisted workflows. Because the functions follow a consistent interface, LLMs can reliably call predefined templates instead of generating plotting code from scratch, reducing the risk of syntactic errors or inconsistent formatting. Templates also make outputs more predictable and auditable, since the logic and styling rules are fixed and version-controlled. This improves reproducibility, facilitates automated QC and validated steps, and enables scalable generation of figures across multiple datasets or studies. Overall, standardized templates help ensure that LLM-driven figure generation remains controlled, consistent, and aligned with reporting standards.

REFERENCES

Dayimu, A. 2025. “Introduction to forestploter.” Accessed September 18, 2025. https://cran.r-project.org/web/packages/forestploter/vignettes/forestploter-intro.html.

Weiss, J. “Creating swimmer plots with ease.” Accessed September 22, 2025. https://cran.r-project.org/web/packages/swimplot/vignettes/Introduction.to.swimplot.html.

Wickham, H. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

ACKNOWLEDGMENTS

We would like to thank Todd Case for his guidance, support, and encouragement throughout the project. We also sincerely thank Matt Finnemeyer for his invaluable feedback and editorial suggestions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Chunting Zheng
Syneos Health
Chunting_Zheng@vrtx.com

Xindai Hu
Vertex Pharmaceutical, Inc.
XindaiDara_Hu@vrtx.com

Margaret Huang
Vertex Pharmaceutical, Inc.
Margaret_Huang@vrtx.com