# Learning to Plot Tables in R with gridExtra

Authored by

**Mohammed looti**

March 2, 2026

In the realm of R programming for data analysis, effective communication often requires more than just graphical representations. While visualizations like scatterplots or bar charts excel at conveying trends, presenting the underlying raw data simultaneously can significantly enhance clarity and trustworthiness. Analysts frequently encounter scenarios where they need to plot a detailed **table** directly alongside or underneath a resulting **chart**, ensuring that the viewer has immediate access to the numeric values that generated the visual output. This integrated approach is crucial for reporting and peer review, bridging the gap between abstract visualization and concrete data points.

Fortunately, achieving this seamless integration in R is straightforward, thanks primarily to powerful extension packages. The core functionality we will explore relies heavily on the **gridExtra** package, which provides essential tools for arranging multiple graphical objects--including custom tables--on a single plot canvas. The following comprehensive guide details the necessary steps and functions required to successfully plot a **data frame** as a table alongside any standard R chart, providing practical examples using standard R syntax.

This tutorial will walk through a complete example, demonstrating how to initialize the required packages, structure the data, define both the visualization and the tabular object, and finally, employ specialized functions to coordinate their placement within the final output device. Understanding these methods is vital for creating polished, publication-ready graphics that convey both visual insights and numerical precision.

## The Necessity of Combining Tables and Plots in Data Visualization

Data visualization is the art of representing information visually, making complex data sets accessible and understandable. However, relying solely on graphical elements can sometimes lead to misinterpretation, especially when dealing with small sample sizes or highly nuanced datasets. By incorporating the raw data **table** directly into the visualization output, we eliminate ambiguity and provide immediate context for every point or bar shown on the chart. This dual presentation ensures that stakeholders, reviewers, or readers can quickly verify the data points without having to reference external documents or separate data dumps.

In professional statistical reporting or academic publications, clarity is paramount. Imagine presenting a time-series plot where the exact values for certain critical dates are necessary for decision-making; a side-by-side table makes this information instantly available. Furthermore, when working within the powerful visualization environment of ggplot2, the ability to merge its output with other non-plot elements, such as text annotations or structured tables, is a highly desirable feature. The **gridExtra** package acts as the necessary bridge, translating the tabular data structure into a graphical object compatible with R's grid system.

This technique is particularly valuable when the underlying **data frame** is concise and contains the

primary variables of interest. If the table were excessively long, this approach might clutter the visualization; however, for summary statistics, small experimental results, or the raw input for a small chart, combining them into a single figure is the most efficient method for data presentation, significantly improving the overall informational density of the graphic.

## Essential R Packages: gridExtra and ggplot2

To successfully combine a chart and a table in R, we rely on two foundational packages. The first is **ggplot2**, which is the industry standard for creating elegant and highly customizable statistical graphics. We use **ggplot2** to define the initial visualization, such as a **scatterplot**, leveraging its grammar of graphics approach. Defining the plot object is the first crucial step in this process, as it generates the visualization component that we intend to arrange with the tabular component.

The second, and arguably the most vital package for this specific task, is **gridExtra**. This package extends R's core graphics capabilities by providing tools to arrange multiple graphical objects created using the "grid" system. Unlike traditional base R graphics, the grid system allows for precise control over layout and placement, enabling us to treat a plot, a legend, or even a structured table as independent graphical elements that can be combined and positioned flexibly. The functions provided by **gridExtra**, specifically **tableGrob** and **grid.arrange**, are central to our workflow.

The role of **tableGrob()** is transformative: it takes a standard R object, such as a **data frame**, and converts it into a "grob" (graphical object), which is essentially a visual representation of the table suitable for printing or arranging. Subsequently, **grid.arrange()** acts as the layout manager, taking the plot object generated by **ggplot2** and the grob object generated by **tableGrob()**, and arranging them according to specified layout parameters, thereby rendering the final combined image seamlessly onto the plotting device.

## Practical Example: Setting Up the Data and Initial Scatterplot

Before combining elements, we must first establish the data structure we intend to visualize and tabulate. For this demonstration, we will define a simple **data frame** containing seven observations across two numeric variables, 'x' and 'y'. This concise structure is ideal for showing the underlying data alongside a chart, as it is small enough to be easily readable within the confines of a single graphic. The code snippet below illustrates the creation and subsequent inspection of this primary data object in R, confirming its dimensions and initial values.

The use of a **data frame** is standard practice in R for organized data manipulation, serving as the universal input for most statistical modeling and visualization functions, including those within the **ggplot2** ecosystem. Once this structure is defined, we can proceed to the visualization stage, where we aim to create a simple **scatterplot** to illustrate the relationship between the 'x' and 'y'

variables, providing the primary visual insight for our combined figure. The integrity of the data structure is paramount, as errors here will propagate to both the plot and the generated table.

The following R code block initializes the example dataset, ensuring that the necessary input is ready for conversion into both a plot object and a tabular grob object. This initial setup is the foundation upon which the subsequent arrangement using the **gridExtra** package will be built. Note the use of clear variable names, which facilitates readability during the definition of the plot and table objects later in the script.

**#create data frame**
**df <- data.frame(x=c(1, 2, 3, 4, 5, 6, 7),**
**y=c(3, 4, 4, 8, 6, 10, 14))**

#view data frame
df

x y
1 1 3
2 2 4
3 3 4
4 4 8
5 5 6
6 6 10
7 7 14

## Implementing the Default Layout: Vertical Arrangement

With the **data frame** established, the next logical step involves defining the two graphical components: the **scatterplot** and the tabular representation. We start by loading the necessary libraries, **gridExtra** and **ggplot2**, which must be executed before calling any of their respective functions. The scatterplot is defined using standard **ggplot2** syntax, creating an object (`my_plot`) that represents the visualization without immediately rendering it. This object-oriented approach is critical, as it allows us to manipulate the plot before final rendering.

Subsequently, we use the specialized function **tableGrob()** from the **gridExtra** package. This function takes our input **data frame** (`df`) and transforms it into a grid graphical object (`my_table`). This object is now recognized by R's grid system as a plot element, making it compatible with arrangement functions. It is important to note that **tableGrob()** handles the formatting, alignment, and styling of the table automatically, although advanced users can customize these aesthetics through additional arguments.

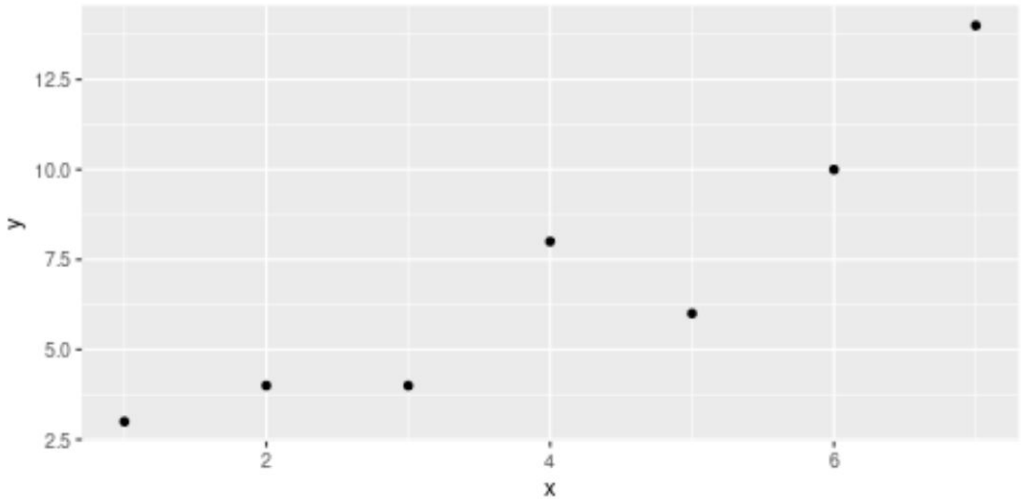
Finally, the coordination is handled by **grid.arrange()**. When supplied with the plot object and the table grob object (`my_plot` and `my_table`), the function's default behavior is to stack them vertically, arranging them in the same column. This is the simplest and often the most readable layout, especially when the table is intended to serve as an appendix directly referencing the data points above it. The execution of **grid.arrange(my_plot, my_table)** renders the combined output to the graphics device, resulting in a single figure that incorporates both the visual trend and the numerical foundation.

**library(gridExtra)**
**library(ggplot2)**

#define scatterplot
my_plot <- ggplot(df, aes(x=x, y=y)) +
geom_point()

#define table
my_table <- tableGrob(df)

#create scatterplot and add table underneath it
grid.arrange(my_plot, my_table)




| | x | y |
|---|---|---|
| 1 | 1 | 3 |
| 2 | 2 | 4 |
| 3 | 3 | 4 |
| 4 | 4 | 8 |
| 5 | 5 | 6 |
| 6 | 6 | 10 |
| 7 | 7 | 14 |

## Function Deep Dive: Understanding tableGrob() and grid.arrange()

A deeper understanding of the functions involved is essential for customization. The **tableGrob()** function is not merely a display tool; it is a highly configurable engine for rendering data structures graphically. It inherits capabilities from the underlying `grid` package, allowing for modification of cell colors, font styles, alignment, and theme adherence. When creating `my_table`, the function analyzes the dimensions of the input **data frame** and automatically calculates the optimal cell sizes and padding required to fit the content cleanly within the graphic device, ensuring the resulting table is visually appealing and legible, regardless of the output resolution.

The **grid.arrange()** function, conversely, is the primary layout manager, designed to integrate multiple grob objects. Its simplicity in the basic use case (stacking vertically) hides its powerful potential for complex multi-panel arrangements. Fundamentally, **grid.arrange()** uses a simple grid layout system, often defaulting to a single column unless otherwise specified. This function is instrumental because it coordinates the viewports--the specific regions on the graphic device--that each component occupies, ensuring that the plot and the table do not overlap and are scaled

appropriately relative to one another.

In essence, this workflow separates the creation of visual components from their assembly. We use **ggplot2** to perfect the chart, **tableGrob()** to perfect the tabular representation, and then rely on **grid.arrange()** to act as the final editor, dictating the precise positioning of these finalized elements. This modular approach allows for robust error handling and easier modification of individual components without needing to redesign the entire figure.

The sequence of operations required to produce the vertically arranged figure involves three distinct, critical steps:

We utilized the standard **ggplot()** function suite to define and generate the primary **scatterplot** object (`my_plot`).
We employed **tableGrob()** to convert the raw numerical **data frame** into a compatible graphical table object (grob).
We finalized the display using **grid.arrange()**, which orchestrates the placement of both the plot and the table sequentially on the graphic output device, defaulting to a single column arrangement.

## Advanced Layout Control: Arranging Plots Side-by-Side

While the default vertical stacking provided by **grid.arrange()** is useful, data visualization often benefits from a horizontal arrangement, displaying the chart and the table side-by-side. This layout is particularly advantageous when both components are relatively narrow, maximizing the use of screen or page width. To achieve this horizontal orientation, we must explicitly control the column structure using the `ncol` argument within the arrangement mechanism.

When using **grid.arrange()** for complex layouts, especially when integrating objects that might require internal layout management (like a table grob), it is often cleaner to nest the arrangement definition within **arrangeGrob()**. The **arrangeGrob()** function creates a composite grob object that defines the desired layout (in this case, two columns: `ncol=2`) before passing it to **grid.arrange()** for final rendering. This separation maintains clarity, defining the layout structure first and then executing the output display.

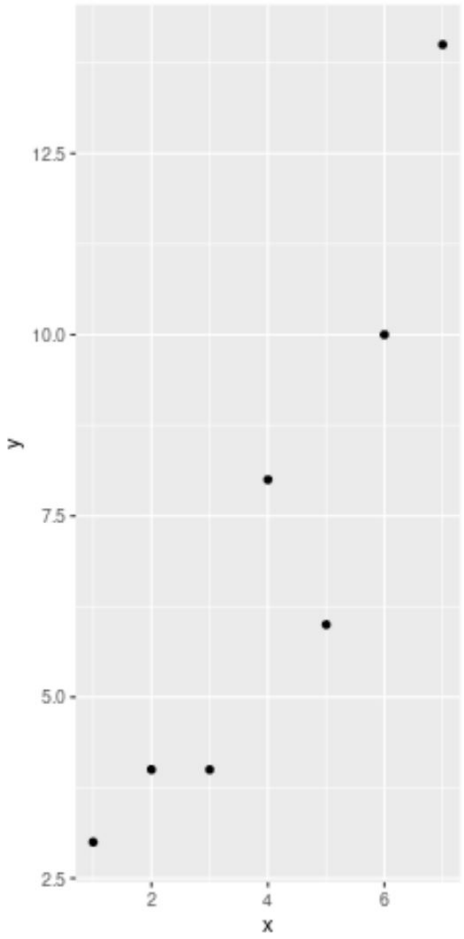
By specifying `ncol=2`, we instruct the layout engine to place the first object (`my_plot`) in the first column and the second object (`my_table`) in the second column. This powerful feature allows analysts complete flexibility in presentation style, transitioning the table from a footnote to a direct companion of the visualization. This adjustment in layout is crucial for figures destined for journals or presentations where space efficiency and visual balance are important design considerations.

**library(gridExtra)**
**library(ggplot2)**

```
#define scatterplot
my_plot <- ggplot(df, aes(x=x, y=y)) +
geom_point()

#define table
my_table <- tableGrob(df)

#create scatterplot and add table next to it
grid.arrange(arrangeGrob(my_plot, my_table, ncol=2))
```




| | x | y |
|---|---|---|
| 1 | 1 | 3 |
| 2 | 2 | 4 |
| 3 | 3 | 4 |
| 4 | 4 | 8 |
| 5 | 5 | 6 |
| 6 | 6 | 10 |
| 7 | 7 | 14 |

As demonstrated in the resulting figure, the numerical **table** is now correctly positioned to the side of the **scatterplot**, providing a cleaner, more horizontal composition compared to the previous arrangement where the table was placed beneath the main graphic. This spatial adjustment significantly impacts how the reader processes the combined information.

## Further Exploration and Resources

Mastering the combination of graphical and tabular elements using **gridExtra** opens up numerous possibilities for advanced reporting in R. While this tutorial focused on aligning a simple **scatterplot** with its input **data frame**, the principles apply universally. Users can combine various plot types (e.g., bar charts, box plots, heatmaps) with customized summary statistics tables or results from statistical models, all within a single, cohesive figure.

For those looking to expand their skills, exploring the arguments within **tableGrob()** allows for deep aesthetic customization, enabling the creation of tables that match specific branding or publication requirements, including alternating row colors or precise font control. Similarly, studying the advanced layout capabilities of **grid.arrange()**--such as using the `widths` or `heights` arguments--provides the ability to manually control the precise spatial distribution and relative size allocated to each component within the final graphic.

The integration of data visualization and raw data presentation is an indispensable skill for any R user focused on clear, defensible data communication. Utilizing the structured approach detailed here ensures that all generated figures are not only visually appealing but also numerically transparent. We highly recommend exploring the official documentation for both the **gridExtra** and **ggplot2** packages to uncover the full extent of their capabilities.

## Additional Resources

The following tutorials explain how to perform other common data visualization tasks in R, building upon the foundational knowledge provided here: