Learning to Plot and Compare Functions Using R: A Comprehensive Tutorial

Authored by

Mohammed looti

November 13, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning to Plot and Compare Functions Using R: A Comprehensive Tutorial*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=24156

The Necessity of Comparative Visualization in R

In analytical disciplines, ranging from advanced mathematics to complex statistical modeling, the ability to compare multiple functions visually is indispensable. Plotting two or more mathematical expressions on the same coordinate plane in R provides immediate insight into their relative behaviors, including rates of growth, inflection points, and asymptotic tendencies across a defined domain. This capability is absolutely crucial for researchers and analysts who need to swiftly evaluate the performance of different model specifications or understand how varying parameters affect underlying relationships.

The most efficient and streamlined method for achieving this comparative data visualization in R, without requiring the manual generation of large datasets, is through the use of the **Base Graphics System**. This native system offers robust tools for generating high-quality plots directly from function expressions. Specifically, the **curve()** function serves as the primary mechanism, evaluating an algebraic expression across a sequence of x-values and rendering the resulting curve. Mastering this function is the key to overlaying multiple functions effectively.

The core challenge in comparative plotting is ensuring that subsequent functions are added to the existing plot rather than replacing it entirely. This is managed by the critical **add** argument. By setting **add=TRUE** for every function plotted after the initial one, we instruct R to superimpose the new curve onto the current graph window. Neglecting this argument results in the complete loss of the previous visualization, defeating the objective of direct comparison.

Utilizing the curve() Function and R's Base Graphics

The powerful curve() function is an integral component of R's built-in Base Graphics System, meaning it is instantly accessible without the requirement of external package installation. When you execute the first call to **curve()**, R performs two essential tasks: it evaluates the expression and initializes the plot environment, setting up the coordinate axes based on the specified range (domain and codomain). This initial call effectively draws the first function and prepares the canvas for subsequent overlays.

Any additional function intended for the same graph must be included via a new **curve()** command, but with the explicit inclusion of the **add=TRUE** argument. If **add=TRUE** is omitted, R defaults to creating a completely new plot window, erasing the work done by the first call. Therefore, proper sequencing and the consistent application of this argument are foundational to successful comparative plotting. Furthermore, it is best practice to define clear axis labels using the **xlab** and **ylab** arguments in the initial function call to ensure the resulting graph is immediately understandable to the viewer.

Consider the basic scenario of comparing two common polynomial functions: a cubic function

($y=x^3$) and a quartic function ($y=x^4$). The following structure demonstrates the necessary syntax, where the first line initializes the plot and the second line adds the subsequent function using the required argument:

curve(x^3, from=1, to=50, xlab='x', ylab='y')
curve(x^4, from=1, to=50, xlab='x', ylab='y', add=TRUE)

This code block successfully superimposes both curves, allowing for a side-by-side assessment of their growth rates across the specified domain. The differentiation between the initialization step and the layering step, marked by **add=TRUE**, is the most crucial takeaway from this foundational example.

Synchronizing Boundaries: The Importance of `from` and `to` Arguments

For any visual comparison to be valid and meaningful, all functions must be evaluated and displayed over the identical range of input values. This synchronization is managed explicitly by the **from** and **to** arguments within the curve() function. These arguments specify the lower and upper bounds, respectively, of the x-axis domain across which the function expression is calculated.

Maintaining consistency in these bounds is paramount. If, for instance, the first function is plotted from 1 to 50, and the second function is inadvertently plotted from 1 to 100, R will scale the axes according to the wider range (1 to 100). This inconsistent scaling will distort the visual representation of the first function, leading to potentially inaccurate interpretations regarding its behavior or magnitude relative to the second function. Therefore, analysts must ensure that the **from** and **to** values are carefully chosen to be appropriate for all functions under comparison and that they are exactly synchronized across every **curve()** call used for the current plot.

In our example comparing $y=x^3$ and $y=x^4$, both functions were deliberately plotted using the range **from=1** and **to=50**. This guarantees that any visible difference in the curves is attributable solely to the mathematical nature of the functions themselves, not to artifacts introduced by inconsistent plotting ranges or poor axis definition. Consistent boundary definition is a hallmark of professional data visualization.

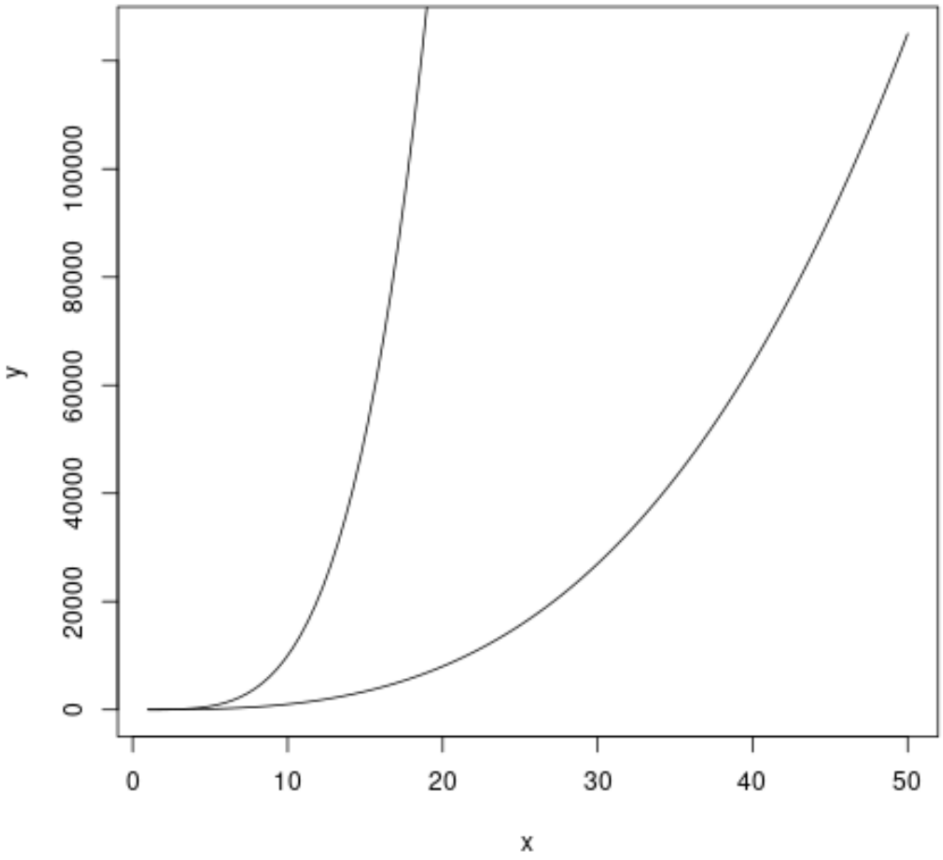
Practical Demonstration: Initial Unstyled Plotting

To fully grasp the mechanism described, let us execute the required R code to plot the simple cubic and quartic functions, $y=x^3$ and $y=x^4$. This concrete example illustrates precisely how R handles the initialization and subsequent layering of functions onto a single coordinate plane, setting the stage for further enhancements.

curve(x^3, from=1, to=50, xlab='x', ylab='y')

curve(x^4, from=1, to=50, xlab='x', ylab='y', add=TRUE)

Executing this syntax produces the initial, unstyled chart, confirming that both functions are correctly displayed on the same coordinate plane:



While the graph confirms that the superposition was technically successful, a significant problem remains: without any visual differentiation, it is impossible for the viewer to determine which line corresponds to the cubic function and which corresponds to the quartic function. Both lines appear as solid black curves. This lack of clarity significantly hinders the interpretability of the comparison and necessitates the application of graphical parameters to enhance the plot's readability.

Enhancing Readability: Controlling Line Color and Style

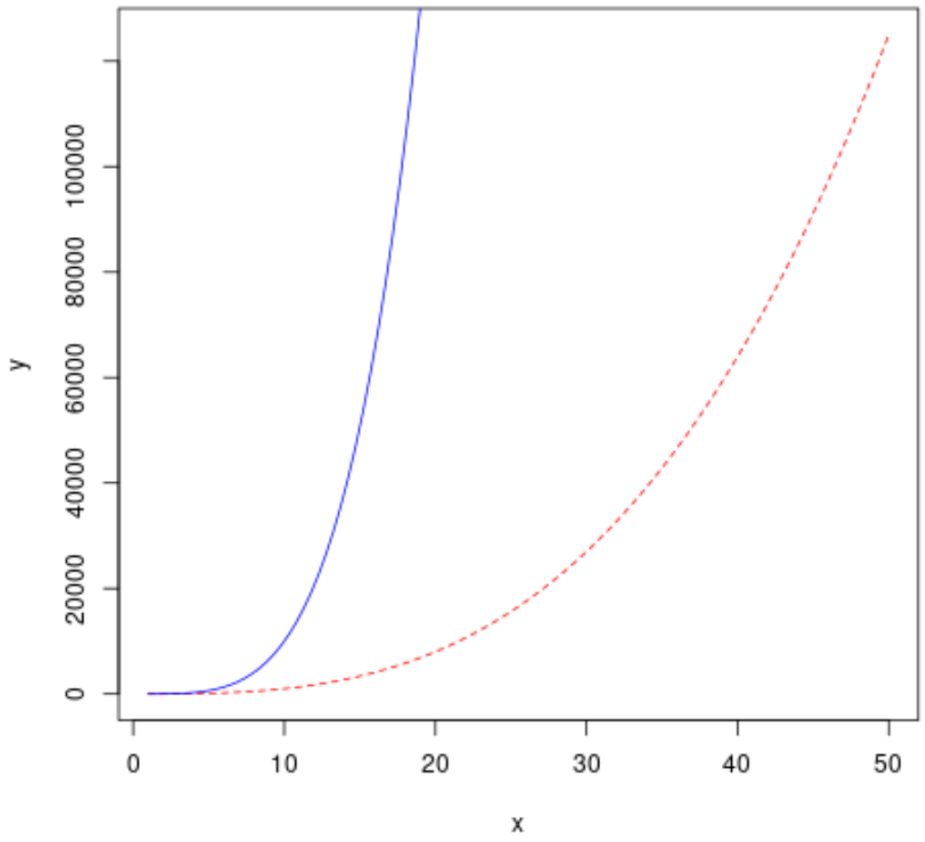
When producing high-quality data visualization, the clarity of distinguishing between multiple variables is non-negotiable. Using only position to differentiate curves is insufficient, especially when they overlap or diverge similarly. To resolve the ambiguity presented in the previous step, it is highly recommended to leverage the line customization arguments available within the curve() function. Specifically, the **col** argument allows the user to specify a distinct color for each line, and

the **lty** argument permits the selection of a specific line type pattern (e.g., solid, dashed, dotted).

By applying contrasting visual elements, we can create an unambiguous graph that drastically improves the viewer's ability to track and compare the behavior of each curve across the domain. For example, we can assign a solid blue line to the quartic function and a dashed red line to the cubic function. This intentional use of stylistic differentiation transforms a basic, confusing plot into a professional, easily deciphered analytical tool. The following syntax demonstrates how to incorporate these crucial stylistic parameters into our existing code structure:

```
curve(x^3, from=1, to=50, xlab='x', ylab='y', col='red', lty='dashed')
curve(x^4, from=1, to=50, xlab='x', ylab='y', col='blue', add=TRUE)
```

Applying these enhancements yields a significantly improved chart, where the visual separation provided by the unique color and line style for each function makes the graph substantially easier to interpret immediately:



Mastering Interpretation: Incorporating the legend() Function

While assigning unique colors and line types makes differentiation possible, a comparative plot

remains incomplete and unprofessional without a clear key explaining what those visual elements represent. The legend() function, also part of the R Base Graphics System, is the standard and necessary mechanism for adding this explanatory key to any plot. The legend must meticulously map the applied visual properties (color, line type, or fill) back to their corresponding functional expressions.

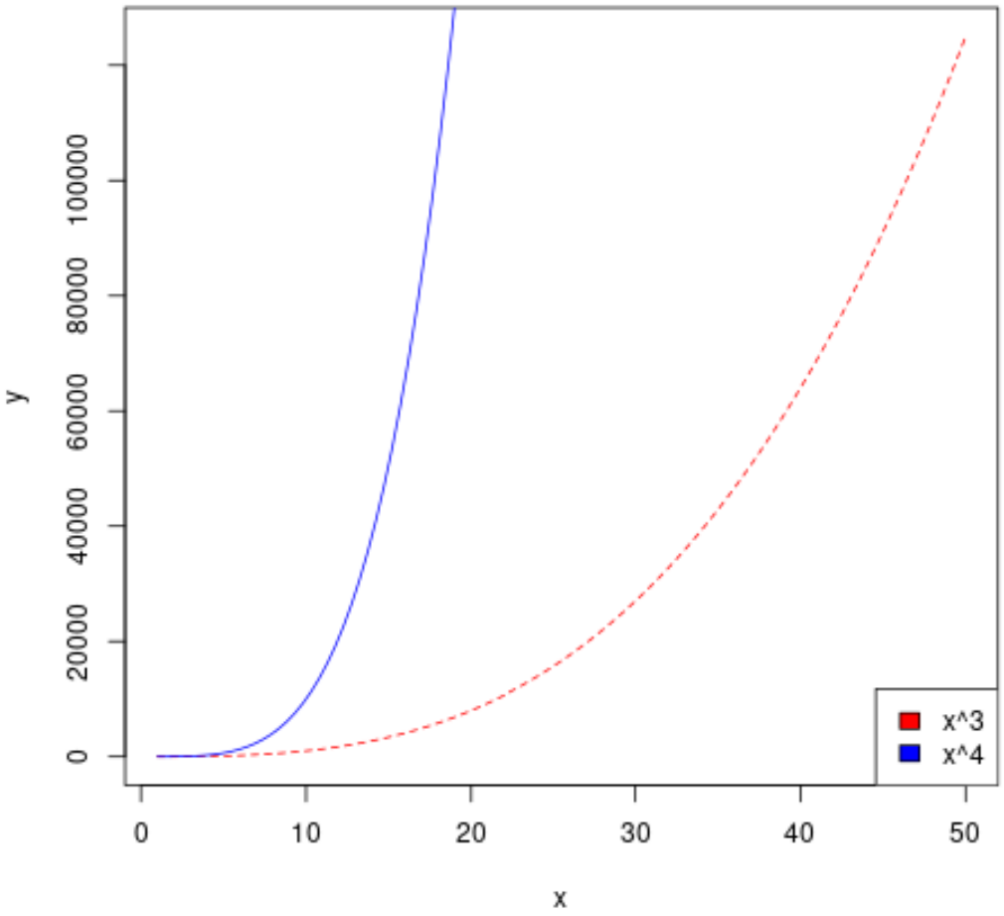
When constructing the legend, several key parameters must be specified: the location on the plot (e.g., 'bottomright', 'topleft', or specific coordinates), the visual cues used (e.g., **fill** for colors, **lty** for line types), and the actual text labels for the legend entries. Crucially, the order of the colors/fills and labels provided to the **legend()** function must perfectly match the order in which the functions were defined and styled in the preceding **curve()** calls. This strict adherence to order ensures that the red dashed line is correctly identified as x^3 and the blue solid line as x^4.

The following integrated code block provides the complete sequence, executing the plot generation with styling and then adding the necessary explanatory legend:

```
#plot two functions on the same graph
curve(x^3, from=1, to=50, xlab='x', ylab='y', col='red', lty='dashed')
curve(x^4, from=1, to=50, xlab='x', ylab='y', col='blue', add=TRUE)

#add legend to display which lines represent which function
legend('bottomright', fill=c('red', 'blue'), legend=c('x^3', 'x^4'))
```

This comprehensive approach yields the final, professionally structured chart, confirming that the plot communicates maximum clarity and interpretability:



Mastering the combined use of the **curve()** function, applying strategic visual arguments (**col**, **lty**, **add=TRUE**), and implementing the **legend()** function is fundamental for effective and professional comparative data visualization in R.

Additional Resources for R Data Visualization

For those seeking to expand their knowledge of plotting and data handling in R, the following resources and tutorials explain how to perform other common visualization tasks and deepen understanding of the R programming language:

<!--

Featured Posts

-->