Plot Multiple Plots on Same Graph in R (3 Examples)

Authored by

Mohammed looti

October 30, 2025

Mastering Multi-Panel Data Visualization in R

The ability to efficiently compare and contrast different data series is fundamental to effective data visualization. In the statistical programming environment of R, particularly when utilizing the powerful capabilities of the Base R Plotting System, researchers and analysts frequently need to display multiple plots or lines within a single graphical output. This technique allows for immediate visual assessment of relationships, trends, and differences between datasets, making the resulting graphics clearer and more informative than scattered individual plots. Achieving this flexibility in visualization relies on understanding R's core graphical parameters and functions.

Whether your goal is to superimpose several lines on a shared coordinate system or to arrange distinct plots neatly into a grid layout, R provides straightforward methods to manage the graphical device output. These methods are indispensable for creating publication-quality figures that clearly communicate complex data structures. We will explore the three primary approaches available in Base R for managing and displaying simultaneous plots, ensuring you can tailor your output precisely to your analytical needs.

You can use the following methods to plot multiple visualizations on the same graphical device in R:

Method 1: Plot Multiple Lines on Same Graph: This approach involves drawing multiple data series (e.g., lines or points) onto a single set of axes, ideal for direct comparison when variables share the same scale.

Method 2: Create Multiple Plots Side-by-Side: This technique uses the graphical parameters to segment the plotting area into a horizontal grid, allowing distinct plots to be viewed simultaneously.

Method 3: Create Multiple Plots Stacked Vertically: Similar to the side-by-side arrangement, this method configures the plotting area into a vertical grid, often preferred when tracking changes over time across different categories.

Method 1: Overlaying Multiple Data Series on a Single Graph

The simplest and most direct way to compare trends is by drawing multiple lines onto the same set of axes. This method is particularly suitable when the variables being plotted share common units and scales. The key distinction in this method lies in the functions used: the first dataset must be plotted using the standard plot() function, which initializes the graphical device, sets the axis limits, and draws the primary plot elements. Subsequent data series are then added using additive functions like `lines()`, `points()`, or `text()`, which do not reset the existing plot environment.

When overlaying plots, careful consideration must be given to visual aesthetics. Differentiating the

lines using distinct colors (`col`), line types (`lty`), or plotting characters (`pch`) is crucial for readability. Furthermore, including a legend is often necessary to correctly identify which line corresponds to which dataset. This approach maintains a tight visual comparison because all data points are referenced against the same x and y coordinates.

The following R snippet illustrates the fundamental difference between the initializing function (`plot()`) and the additive function (`lines()`) when drawing two separate time series onto a single graph.

```
#plot first line
plot(x, y1, type='l')

#add second line to plot
lines(x, y2)
```

Method 2: Creating Side-by-Side Plots (Horizontal Arrangement)

When the datasets you wish to display are fundamentally distinct, or if comparing them on shared axes would lead to overplotting or conflicting scales, arranging them in a multi-panel layout is the preferred solution. In R, managing the layout of multiple figures (known as 'faceting') is primarily achieved using the par() function, specifically by setting the `mfrow` or `mfcol` graphical parameters. The `mfrow` parameter dictates the number of rows and columns for the subsequent plots, arranging them in a row-major order (filling across rows first).

To create a side-by-side arrangement, we define the graphical device to have one row and two columns using the command `par(mfrow = c(1, 2))`. Once this parameter is set, every subsequent call to a plotting function (like `plot()`, `hist()`, or `barplot()`) will draw the visualization sequentially into the next available subplot area. After both plots have been generated, the layout configuration remains active until the graphical device is closed or the `par()` settings are reset.

This method is particularly powerful for showing comparisons of distribution, scale, or relationship structure that might be obscured if the data were forced onto a single plot. It is essential, however, to ensure that if the plots are meant for direct comparison, the y-axis limits (`ylim`) are synchronized manually across all subplots to avoid misleading visual interpretations.

```
#define plotting area as one row and two columns
par(mfrow = c(1, 2))

#create first plot
plot(x, y1, type='l')
```

#create second plot
plot(x, y2, type='l')

Method 3: Stacking Plots Vertically

While horizontal arrangements are common, vertical stacking of plots is often preferred when the variable on the shared x-axis (e.g., time) is the primary focus, and the viewer needs to scan down to observe the corresponding behavior of different variables. To achieve this vertical layout, we again utilize the par() function, but this time, we define the layout to have two rows and one column: `par(mfrow = c(2, 1))`. This configuration instructs R to divide the plotting area vertically, filling the top plot first, followed by the bottom plot.

When stacking plots, a common challenge is managing the margins, especially if the plots share an x-axis or if the default titles and labels conflict. The `par()` function provides several parameters to control spacing, most notably the `mar` argument, which specifies the size of the margins (bottom, left, top, right) in lines of text. Adjusting `mar` is critical to reduce redundant white space between the stacked graphs and ensure the overall figure is compact and visually appealing. For example, reducing the top margin of the bottom plot and the bottom margin of the top plot can create a tighter linkage between them.

By carefully controlling the layout and margins, analysts can create compelling vertical visualizations that highlight sequential or concurrent processes. As with Method 2, if the scales should be consistent for comparative purposes, the `ylim` parameter must be manually set for both plots to span the overall minimum and maximum data values across both series.

#define plotting area as two rows and one column
par(mfrow = c(2, 1))

#create first plot
plot(x, y1, type='l')

#create second plot
plot(x, y2, type='l')

Practical Implementation: Plotting Multiple Lines on a Shared Axis (Example 1)

In this first practical example, we demonstrate how to effectively combine two distinct data series, `y1` and `y2`, which track different trajectories over a common index `x`, onto a single graph. This technique requires initializing the plot using the primary `plot()` function while simultaneously

setting the required visual parameters, such as axis labels and the color of the first line. By using `type='l'`, we instruct R to draw a line plot.

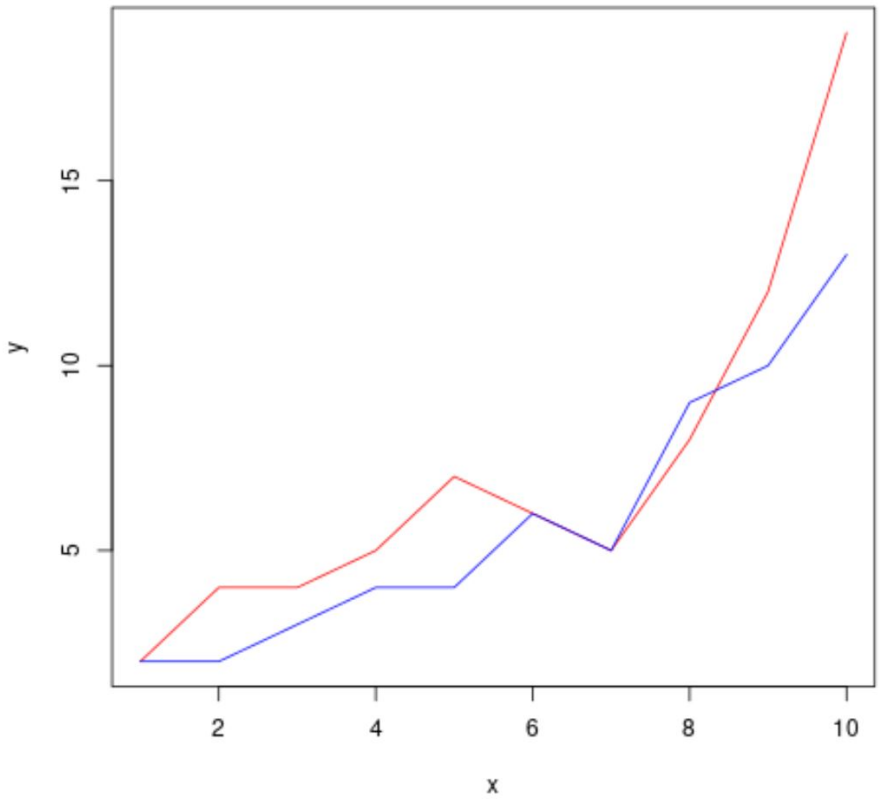
Once the primary plot framework is established, the `lines()` function is invoked to introduce the secondary dataset, `y2`. Crucially, the `lines()` function does not redraw the axes or the plot boundary; it merely overlays the new data points. We assign a different color to the second line (e.g., blue) to ensure clear differentiation from the first (red). This overlaying method is optimal for identifying concurrent trends or potential crossovers between the two data series, provided their magnitudes are reasonably similar.

Note that for professional figures, a legend should typically be added using the `legend()` function to map the colors to the data series names, although this specific example focuses on the core plotting mechanism.

#define data to plot
x <- 1:10
y1 <- c(2, 4, 4, 5, 7, 6, 5, 8, 12, 19)
y2 <- c(2, 2, 3, 4, 4, 6, 5, 9, 10, 13)

#plot first line
plot(x, y1, type='l', col='red', xlab='x', ylab='y')

#add second line to plot
lines(x, y2, col='blue')



Practical Implementation: Comparing Datasets in a Horizontal Grid (Example 2)

When the goal shifts from direct overlay to distinct, yet simultaneous, comparison, arranging plots side-by-side using a multi-figure layout is necessary. This example utilizes the par() function with the `mfrow` parameter set to `c(1, 2)`, establishing a single row containing two columns. This preparation ensures that the next two plotting commands will draw independent figures within this horizontal grid.

The code proceeds by generating the first plot (`y1`) and then the second plot (`y2`). A critical consideration for comparative visualization is maintaining consistency in the visual dimensions. If the two plots have different natural y-axis ranges, comparing their vertical magnitude becomes distorted. To address this, we define the `ylim` argument in the second plot to explicitly match the range of the first plot (or, ideally, the combined range of both datasets). This synchronization ensures that a vertical distance of one unit appears the same height in both the left and right panels, facilitating accurate visual judgment.

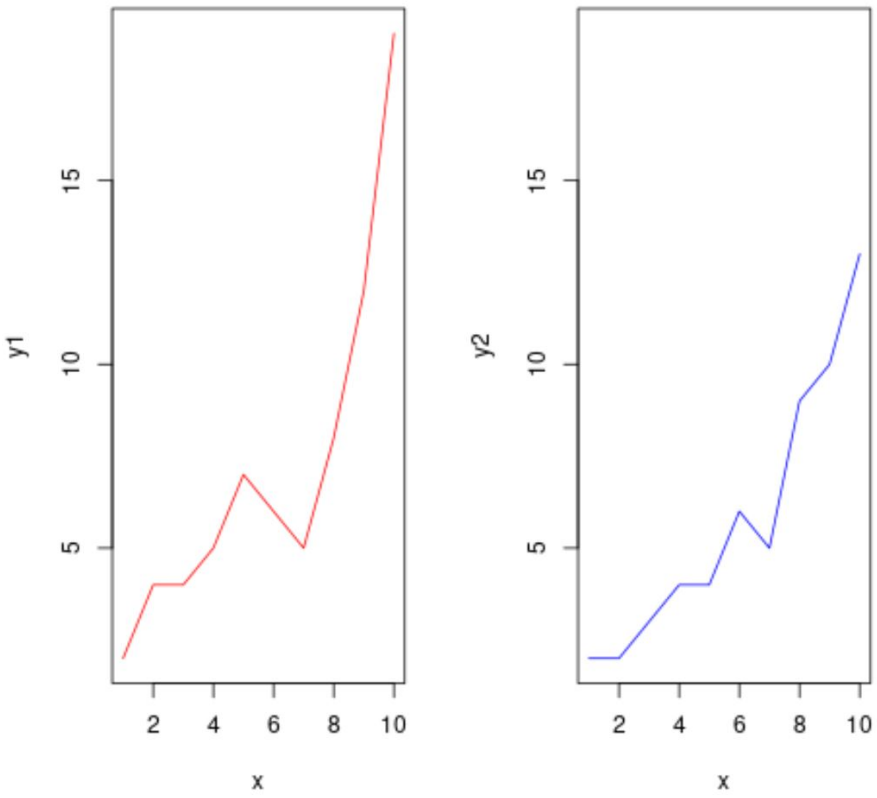
By applying the `par(mfrow = c(1, 2))` command, we efficiently segment the visualization space, allowing for clear, separated presentation of the two data series while retaining their proximity for easy comparison.

```
#define data to plot
x <- 1:10
y1 <- c(2, 4, 4, 5, 7, 6, 5, 8, 12, 19)
y2 <- c(2, 2, 3, 4, 4, 6, 5, 9, 10, 13)

#define plotting area as one row and two columns
par(mfrow = c(1, 2))

#create first line plot
plot(x, y1, type='l', col='red')

#create second line plot
plot(x, y2, type='l', col='blue', ylim=c(min(y1), max(y1)))
```



Note that we used the `ylim` argument in the second plot to ensure that the two plots had the same y-axis limits, which is vital for maintaining an accurate visual scale between the horizontally arranged figures.

Practical Implementation: Vertical Stacking with Margin Control (Example 3)

This final example shifts the focus to vertical arrangement, setting the par() function parameter `mfrow` to `c(2, 1)`, indicating two rows and one column. This layout is excellent for presenting time series data where the x-axis is shared implicitly but the datasets themselves may have different characteristics or units, making vertical separation desirable.

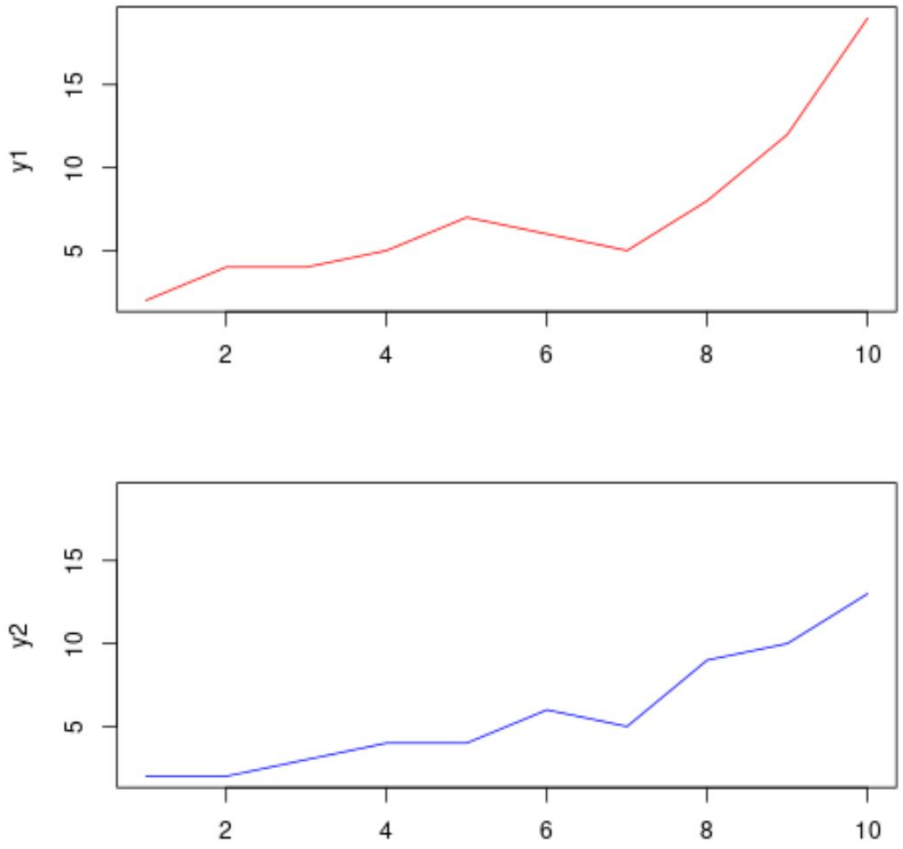
When plots are stacked vertically, the default margin settings often result in excessive white space, particularly between the plots themselves, or cause the titles and axis labels to overlap. To refine the aesthetics and maximize the data-to-ink ratio, we introduce the `mar` argument. The `mar` vector specifies the size of the margins in the order: bottom, left, top, and right. The default setting is `c(5.1, 4.1, 4.1, 2.1)`.

In the provided code, we use `par(mfrow = c(2, 1), mar = c(2, 4, 4, 2))`. By significantly reducing the bottom margin (from 5.1 to 2) and slightly adjusting the others, we achieve a much tighter, more cohesive visualization where the two plots appear closely linked by their shared horizontal dimension. As in Example 2, the `ylim` parameter is employed in the second plot to ensure visual scale consistency, even with the vertical separation.

```
#define data to plot
x <- 1:10
y1 <- c(2, 4, 4, 5, 7, 6, 5, 8, 12, 19)
y2 <- c(2, 2, 3, 4, 4, 6, 5, 9, 10, 13)

#define plotting area as two rows and one column
par(mfrow = c(2, 1), mar = c(2, 4, 4, 2))
#create first line plot
plot(x, y1, type='l', col='red')

#create second line plot
plot(x, y2, type='l', col='blue', ylim=c(min(y1), max(y1)))
```



Note that we used the **mar** argument to specify the (bottom, left, top, right) margins for the plotting area, optimizing the display of the vertically stacked figures.

Note: The default margin setting in R is `mar = c(5.1, 4.1, 4.1, 2.1)`. Adjusting these values is a crucial step in fine-tuning multi-panel visualizations.

Additional Resources for R Visualization

The techniques covered--overlaying, horizontal faceting, and vertical stacking--form the foundation of advanced visualization using Base R graphics. Mastering the par() function and its parameters like `mfrow` is essential for controlling the output device. For those looking to further enhance their R visualization skills, exploring other graphical packages, such as `ggplot2`, which handles faceting in a conceptually different but highly structured manner, is recommended.

The following tutorials explain how to perform other common tasks in R: