Understanding the `par()` Function: A Comprehensive Guide to R Graphics Parameters

Authored by

Mohammed looti

November 3, 2025

The **par() function** in **R** is perhaps the most fundamental tool for controlling the aesthetic and structural elements of base graphics. It serves as the primary mechanism for setting or querying global graphical parameters, granting users comprehensive control over the appearance and layout of their visualizations. Critically, this function enables the efficient simultaneous display of multiple plots within a single graphical window, a necessity for comparative data analysis and reporting.

Achieving a multi-figure layout relies heavily on specific arguments such as **mfrow** or `mfcol`, which define the precise dimensions of the plotting area (rows and columns). Understanding how these parameters interact with subsequent plotting commands is key to generating structured graphical reports. The following example illustrates the basic syntax required to establish a grid of figures before creating the individual plots that will populate that grid.

#Define the plot area as four rows and two columns (4x2 grid)
par(mfrow = c(4, 2))

#Create sequential plots which will fill the defined grid row by row
plot(1:5)
plot(1:20)
...

The subsequent sections will provide detailed, practical examples demonstrating how to leverage the power of **par()**, moving beyond simple layout control to customize essential elements such as plot margins, axis labeling, and text scaling, ultimately enabling the creation of high-quality, customized figures suitable for publication.

Understanding the par() Function in R Graphics

The **par()** function operates by managing the overall state of the active **R** graphics device. Unlike arguments passed directly into functions like `plot()`, which affect only that specific visualization, **par()** settings are global and sticky, meaning they persist and influence every subsequent plot until they are explicitly reset. This global scope is what makes **par()** such a powerful tool for establishing consistent visual rules across an entire analysis workflow. While **par()** accepts dozens of arguments for fine-tuning visual output--controlling everything from line types to background colors--the parameters governing figure layout are often the most frequently used.

For managing multi-panel configurations, the parameter **mfrow** is indispensable. This argument instructs the graphics device to arrange multiple figures in a specific grid format. When utilizing **mfrow**, the function requires a two-element numeric vector, conventionally specified as `c(R, C)`, where R is the desired number of rows and C is the number of columns. The key operational

difference of **mfrow** is that plots are added sequentially to the figure starting from the top-left position and proceeding horizontally, filling up the first row completely before moving to the next, similar to reading a book.

Implementing this function effectively is crucial for advanced statistical reporting and comparative data analysis. Whether a researcher needs to efficiently examine diagnostic plots for model validation, compare results generated by different algorithms, or view the evolution of a time series across multiple sub-periods, arranging figures side-by-side or stacked vertically ensures immediate and effective visual contrast. Mastering the layout controls provided by **par()** is a foundational skill for anyone producing reproducible and insightful graphics in R.

Example 1: Creating Multi-Panel Layouts with mfrow

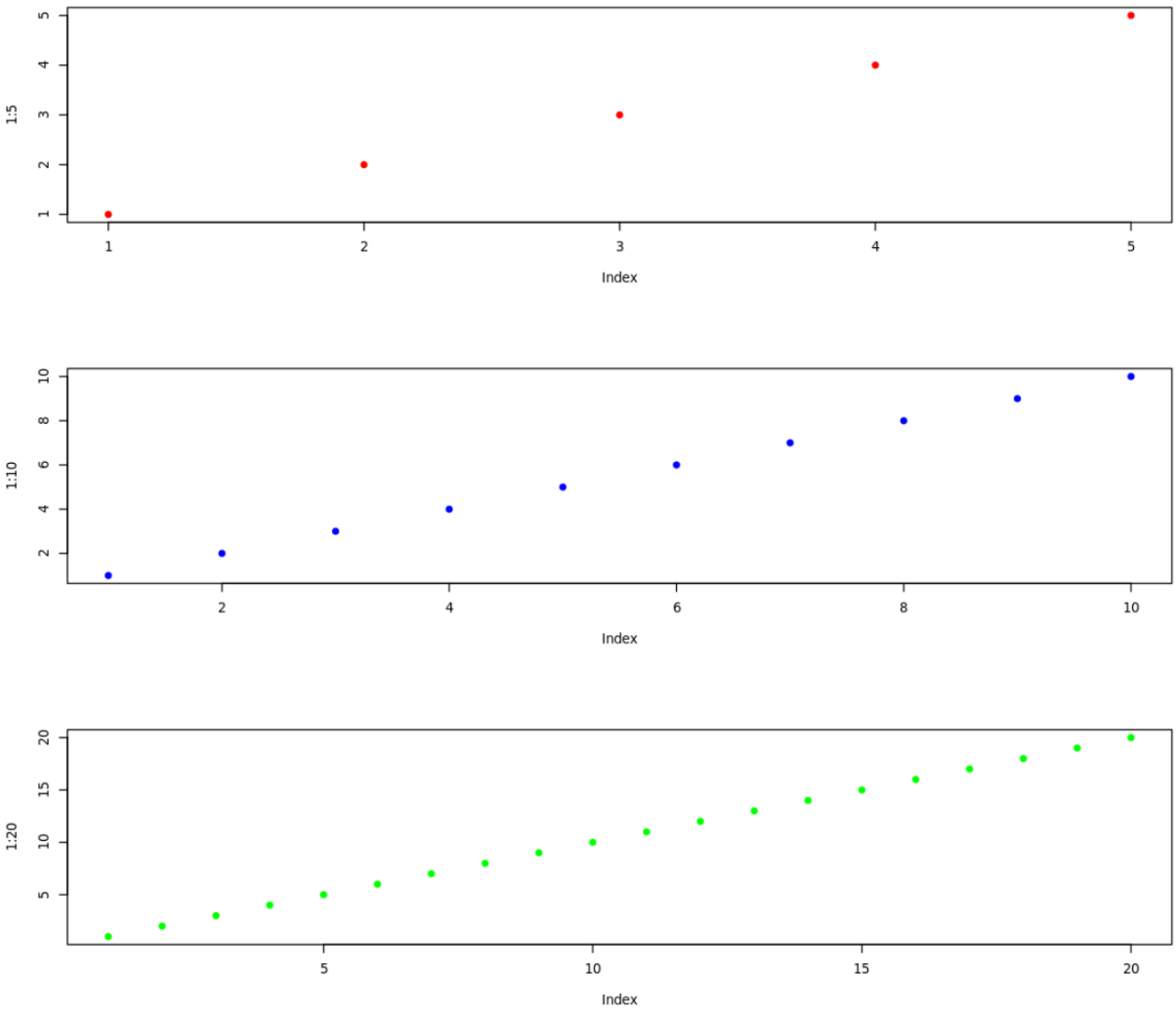
This first practical example demonstrates the straightforward application of the **par()** function to create a vertically stacked array of plots. For tasks requiring direct comparison of data series (e.g., comparing three different distributions or model residuals), a vertical orientation often proves most effective. We define a layout comprising three rows and a single column, ensuring that our visualizations are efficiently stacked within the figure region.

To establish this specific orientation, we execute the command **par(mfrow = c(3, 1))**. Once this command is run, the graphics device is prepared to receive exactly three plots, each occupying one slot in the vertical arrangement. Subsequent calls to plotting functions, such as **plot()**, will automatically place the generated graph into the next available position defined by the **mfrow** structure. Within the individual `plot()` commands, we utilize local arguments like `pch` (point character) and `col` (color) solely to customize the appearance of that specific data series, demonstrating the separation between global (par) and local (plot) parameters.

#Define the plot area for 3 rows and 1 column
par(mfrow = c(3, 1))

#Create three distinct plots
plot(1:5, pch=19, col='red')
plot(1:10, pch=19, col='blue')
plot(1:20, pch=19, col='green')

The resulting graphic, depicted below, clearly illustrates the immediate effect of the `mfrow = c(3, 1)` setting. All three plots are aligned vertically and share the common space within the graphical device. This structure provides an immediate visual context for comparing the underlying data trends (5, 10, and 20 data points), achieving maximum visual efficiency.



Example 2: Controlling Plot Margins with the mar Parameter

While **mfrow** governs the arrangement of plots, precise control over the spacing surrounding each figure is managed using the **mar argument** within the **par()** function. Manipulating margins is a frequent requirement when preparing graphics for formal submission, especially when plots have lengthy axis labels, descriptive titles, or require precise alignment in constrained publication layouts. Adjusting the margins ensures that all text elements are legible and do not overlap the data region or adjacent plots.

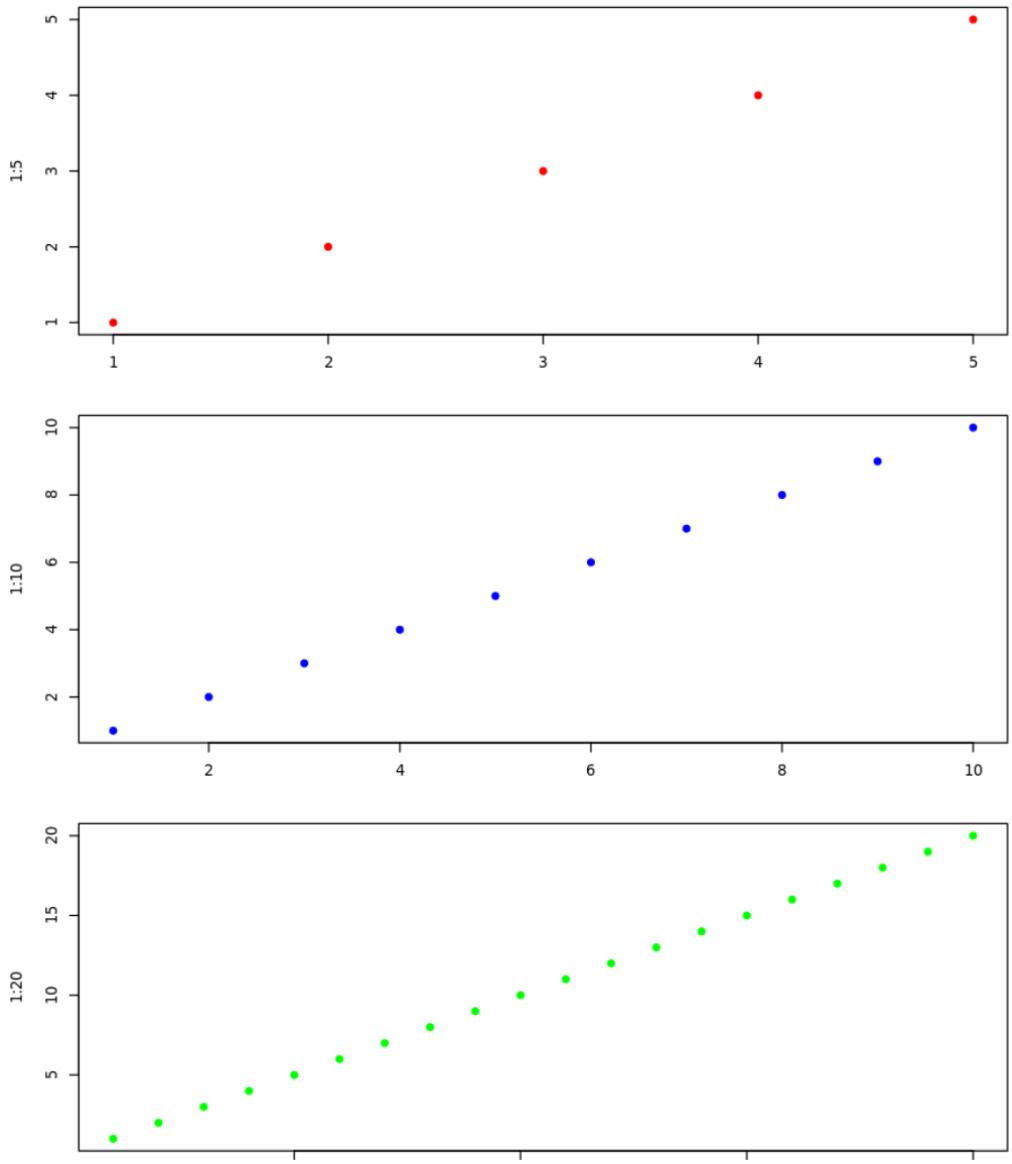
The **mar** argument accepts a vector of exactly four values, which define the margin size in lines for each side of the plot area, following the sequence: bottom, left, top, and right. By default, R initializes the margins to `mar = c(5.1, 4.1, 4.1, 2.1)`, providing standard spacing appropriate for most default labels. When customization is required, these values must be carefully chosen to accommodate the required space.

In this specific demonstration, we modify the margins drastically to highlight their impact on the plot dimensions. We significantly reduce the bottom margin (setting it to 0.5 lines) to minimize the space beneath the X-axis, while simultaneously creating an extremely wide margin on the right side (setting it to 20 lines). This extreme setting clearly shows how margin consumption dictates the effective size of the actual plotting space available for the data visualization.

#Define plot area with 3 rows, 1 column, and custom margins (tiny bottom, huge right)
par(mfrow = c(3, 1), mar = c(0.5, 4, 4, 20))

#Create plots using the new margin settings
plot(1:5, pch=19, col='red')
plot(1:10, pch=19, col='blue')
plot(1:20, pch=19, col='green')

Upon viewing the resulting figure below, it is noticeable that the plots appear significantly narrower compared to Example 1. This contraction is a direct consequence of the large **mar** value assigned to the right side (20 lines), which consumes a substantial portion of the total horizontal space available within the figure region. This example underscores the principle that margins define the boundaries of the plot area, and large margin values necessarily reduce the space allocated to the data itself.



Example 3: Scaling Text and Labels using cex Parameters

Ensuring optimal readability is paramount for any visualization intended for presentations, reports, or printed materials. The `cex` family of parameters within **par()** provides the necessary control to scale the size of various text elements globally. These scaling factors operate as multipliers relative to the default size of 1. A value greater than 1 enlarges the text, while a value less than 1 shrinks it.

To achieve precise text scaling, we differentiate between two key parameters: **cex.lab** and **cex.axis**. The argument **cex.lab** adjusts the size of the axis labels (the descriptive titles, such as "X-axis" or "Y-variable"), while **cex.axis** modifies the size of the tick labels (the numerical values or categories displayed along the axes). Both of these parameters default to a scaling factor of 1,

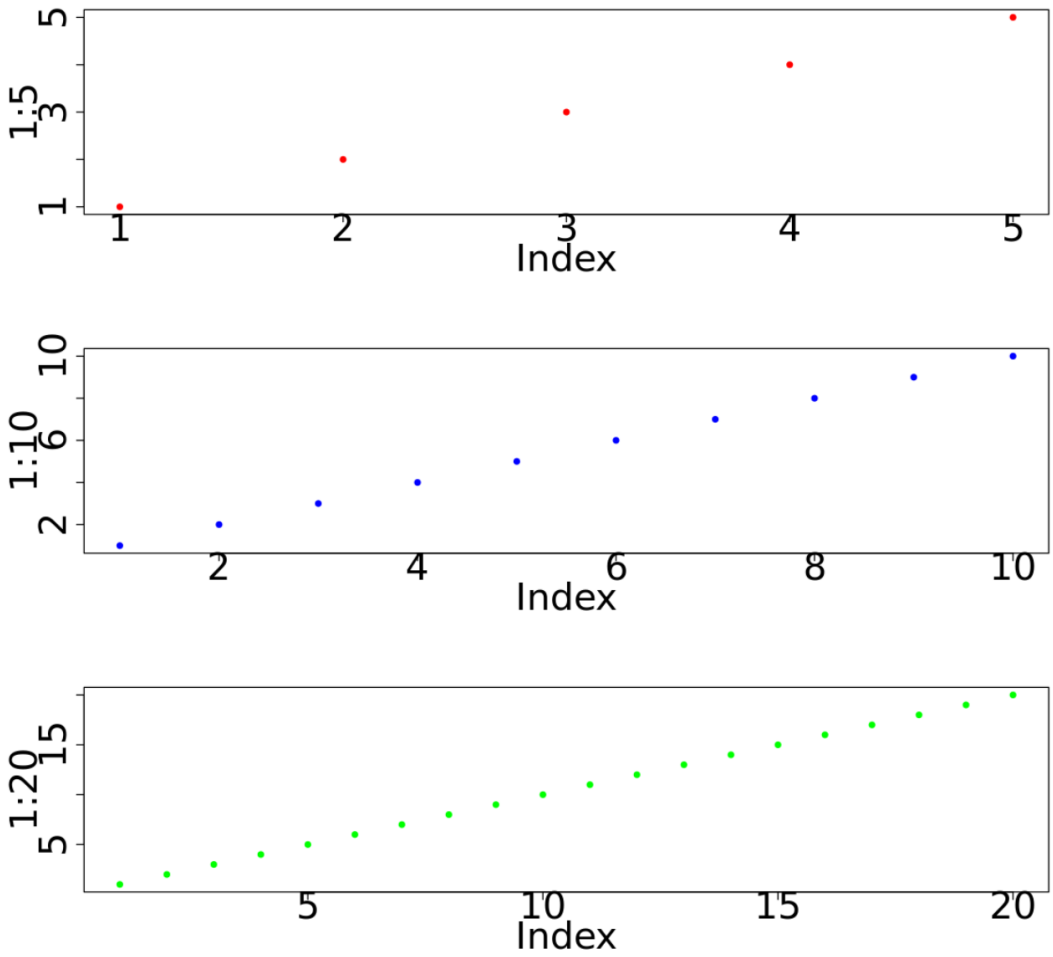
providing the standard text size.

In this scenario, we demonstrate a significant enlargement by setting both **cex** parameters to 3. This substantial increase in text size necessitates a critical adjustment to the plot margins. Specifically, we must increase the left margin substantially (using **mar** = `c(5, 10, 4, 1)`). This expanded left margin is required solely to accommodate the newly enlarged Y-axis tick labels, preventing them from overlapping the plot area itself. This illustrates the crucial interdependence between text scaling and margin adjustments.

#Define plot area with custom margins and large axis/tick labels
par(mfrow = c(3, 1), mar = c(5, 10, 4, 1), cex.axis = 3, cex.lab = 3)

#Create plots utilizing the scaled text sizes
plot(1:5, pch=19, col='red')
plot(1:10, pch=19, col='blue')
plot(1:20, pch=19, col='green')

The resulting visualization confirms the successful scaling of all axis and tick labels, making the figure highly readable even from a distance. The coordinated use of **mar** and the `cex` parameters ensures that the expanded text is properly contained within the figure boundaries, maintaining a clean and professional appearance.



Essential Cleanup: Resetting Graphical Parameters with dev.off()

A critical consideration when utilizing the **par()** function is the persistence of its settings. Because **par()** modifies global graphical settings, these customized parameters--such as the `mfrow` layout or customized `mar` values--remain actively enforced for all subsequent plots created in the current **R** session. If these settings are not explicitly reverted to the default state, any future visualization will inherit the unintended, customized layout, potentially leading to errors or misaligned figures.

To ensure that the graphical environment is clean, predictable, and ready for future commands, it is absolutely essential to reset the graphical device immediately after applying custom **par()** settings for a multi-panel figure. The most reliable and universally recommended method for achieving this necessary cleanup is by executing the **dev.off() function**.

The **dev.off()** command performs a dual function: it closes the active graphical device (which might be the plot window, a PDF file, or a PNG output file) and, in doing so, effectively removes all temporary global parameter changes that were applied via **par()**, returning the settings to their original, default state. This simple action prevents unexpected behavior in future visualizations and

is crucial for maintaining reproducible code environments.

#Reset par() options by closing and reopening the graphical device
dev.off()

Developing the habit of pairing any global **par()** modification with an immediate call to **dev.off()** upon completion of the customized figure is a fundamental practice for any analyst relying on base R graphics. This discipline ensures that your environment remains clean, preventing frustrating debugging efforts caused by inherited plot parameters.

Conclusion: Mastering Base R Graphics Customization

The **par()** function offers unparalleled, comprehensive control over the base **R** plotting system. By mastering the core set of parameters discussed--specifically **mfrow** for defining figure layout, **mar** for precise control over spacing and margins, and the **cex** family for scaling text elements--users can transform standard, default plots into highly customized, publication-quality graphics tailored precisely to their documentation needs.

A robust understanding of how to manage the global graphical device context is fundamental for anyone relying on base R plotting capabilities for serious data analysis. The consistent and deliberate use of **par()** to achieve desired visual effects, combined with the crucial step of cleaning up the environment via **dev.off()**, ensures an efficient, predictable, and fully reproducible visualization workflow, which is vital in any scientific or statistical setting.

For those seeking to explore the vast array of other arguments available within this function and to delve into more advanced graphical customization techniques, consulting the official R documentation is highly recommended:

Official R Documentation for par()
Base R Graphics Manual