How to Easily Save Multiple R Plots to a Single PDF

Authored by

stats writer

December 4, 2025

The ability to generate high-quality visualizations is one of the foundational strengths of the R programming language. However, efficiently managing and exporting these visualizations into a cohesive, distributable format like a PDF can sometimes be a challenge, especially when dealing with multiple plots. This tutorial provides an in-depth guide on utilizing R's built-in functionality--specifically the powerful combination of the `pdf()` and `dev.off()` functions--to bundle several distinct graphical outputs into a single, professional Portable Document Format file.

The standard workflow in R involves directing graphical output to a specific visualization environment known as a graphics device. When you execute a plotting command, R automatically draws the plot on the currently active device, which is usually an interactive screen window. To capture this output persistently, we must explicitly open a file-based device, such as the PDF device. This process ensures that all subsequent plotting commands are rerouted from the screen to the designated output file. This method is highly flexible, allowing users to control not only the content but also the physical characteristics of the final document, including size, resolution, and orientation, making it indispensable for producing publication-ready graphics.

We will explore how to manage plot layouts using the specialized `par()` function to arrange multiple graphs on a single page, a technique particularly useful for comparative analyses or dashboards. Furthermore, we will demonstrate the default behavior of saving plots sequentially, where each new plot command initiates a new page within the PDF document. Understanding this fundamental mechanism is key to mastering visualization output control in R, enabling complex data narratives to be constructed and disseminated with ease and precision.

Understanding R Graphics Devices

In R, a graphics device is an abstract concept representing the location where graphical output is rendered. This device can be screen-based (like RStudio's plot pane or an X11 window) or file-based (like a PDF, PNG, or JPEG file). Before any plotting occurs, a device must be active. By default, R opens a screen device automatically when the first plot command is executed interactively. However, when the goal is persistence--saving the image to a file--we must manually initiate a file device using functions tailored for the desired output format, such as `pdf()`, `png()`, or `jpeg()`. This manual initiation is crucial because it tells R precisely where to direct the pixel or vector information generated by plotting commands like `plot()`, `hist()`, or `ggplot()`.

The primary advantage of vector-based devices like the PDF device is scalability; the graphics are defined mathematically rather than by a fixed grid of pixels. This means that PDF plots can be scaled indefinitely without loss of quality, which is essential for academic publishing and professional reports. When you call `pdf()`, you are essentially opening a channel--a context--where R temporarily stores all plotting instructions. Crucially, R maintains a list of active devices, and plots are only sent to the one most recently opened. This structure allows sophisticated control

over the plotting environment, ensuring that the separation between interactive exploration and final output generation is maintained cleanly.

It is vital to remember that merely calling `pdf()` does not immediately create the file content. Instead, it prepares the system to capture subsequent plot calls. If you forget to close this channel, the file will often remain empty or incomplete, and the plots will not be saved correctly. This requirement for a defined beginning (opening the device) and a defined end (closing the device) enforces a clear, robust workflow for plot exportation, minimizing errors and ensuring reproducibility, which is a core tenet of effective data science using R.

The Role of the pdf() Function

The `pdf()` function is the starting point for saving plots to a PDF file in R. It belongs to the `grDevices` package, which is part of R's base distribution. When executed, `pdf()` opens the PDF graphics device and typically requires at least one argument: `file`, which specifies the full path and filename for the output document. This explicit file path ensures R knows exactly where to deposit the final graphics output once the device is closed. If no arguments are provided, R will usually create a file named "Rplots.pdf" in the current working directory, though specifying the path is highly recommended for script clarity and control.

Beyond the file path, the `pdf()` function offers extensive customization through various arguments. Key arguments include `width` and `height`, which define the dimensions of the pages in inches (the default unit), allowing precise control over the visual presentation. Furthermore, arguments like `paper` (e.g., "a4", "letter") or `onefile` (which defaults to `TRUE`, allowing multiple plots to be stored in the same file) manage document characteristics. For users needing specialized font handling, `pdf()` also supports arguments for embedding fonts and managing color spaces, ensuring that the graphics maintain fidelity across different operating systems and viewing environments.

When you execute `pdf(file="path/to/my_plots.pdf")`, R performs several actions: it registers the new PDF device as the current active device, suspends output to any previous screen device, and prepares the file structure. All subsequent commands that generate graphics--whether they are base R plots, lattice plots, or ggplot2 visualizations--are interpreted by R and translated into PostScript commands that define the PDF content. This buffering process continues until the device is explicitly closed, which is why the pairing of `pdf()` and `dev.off()` is non-negotiable for a successful file generation.

Controlling Output with dev.off()

The counterpart to `pdf()` is the `dev.off()` function. This function is arguably the most crucial step in the entire plotting workflow, especially when saving to a file. Its primary role is to close the currently active graphics device. When executed, `dev.off()` signals to R that the sequence of

plotting commands intended for that specific device is complete. For file-based devices like PDF, this action triggers the finalization process: R writes all buffered graphics data, closes the file handle, and returns control of the graphical output stream back to the previous device (usually the screen or R console).

Failure to execute `dev.off()` leads to several common issues. First, the output file might remain open and locked by R, preventing other programs (or even the user) from viewing or modifying it. Second, and more importantly, the graphical data stored in R's internal buffers might not be properly flushed to the disk, resulting in an incomplete, corrupted, or even empty PDF file. Therefore, always treat the opening and closing device commands as a mandatory parenthesis around your plot generation code, ensuring the integrity and accessibility of the resulting file.

The elegance of `dev.off()` lies in its universality; it works to close any active device, regardless of whether it was opened using `pdf()`, `png()`, or other device functions. When working with multiple devices simultaneously (an advanced technique), it is good practice to check which device is currently active using `dev.cur()` or explicitly specify which device to close using its device number, although calling `dev.off()` without arguments typically closes the most recently opened device, which is the standard procedure for saving plot sequences.

Basic Syntax for Saving Multiple Plots

The overall structure for exporting multiple plots to a single PDF document in R is straightforward, following the open-plot-close paradigm. This structure involves defining the target path, invoking `pdf()` to start the capture, running all necessary plot commands (often within a loop for efficiency), and concluding with `dev.off()`. The critical element here is that every plot command executed between the opening and closing functions is recorded sequentially in the output file.

If no special layout parameters are set, each separate plot command will automatically create a new page within the PDF document. However, if the user wishes to consolidate several smaller plots onto one page for comparative display, they must utilize layout management functions, primarily `par()`, before the plots are generated. This allows for extremely flexible document generation, suitable for everything from long appendices containing hundreds of individual figures to concise summary reports.

The following basic syntax outlines the standard process. Note how the loop structure allows us to generate four separate plots using the `plot(x, y)` command, all directed to the PDF file specified by the `destination` variable. This sequence ensures that all four randomized visualizations are bundled together before the graphics device is safely terminated.

You can use the following basic syntax to save multiple plots to a PDF in R:

```
#specify path to save PDF to
destination = 'C:UsersBobDocumentsmy_plots.pdf'

#open PDF
pdf(file=destination)

#specify to save plots in 2x2 grid
par(mfrow = c(2,2))

#save plots to PDF
for (i in 1:4) {
x=rnorm(i)
y=rnorm(i)
plot(x, y)
}

#turn off PDF plotting
dev.off()
```

The following examples show how to use this syntax in practice, demonstrating both single-page and multi-page plot organization.

Understanding the par() Function for Layouts

When generating multiple plots, it is often desirable to display them side-by-side or stacked on a single page for direct comparison. This layout management is achieved in base R graphics using the `par()` function, which controls global graphical parameters. Specifically, the argument `mfrow` (or `mfcol`) dictates how subsequent plots should be arranged on the current graphics device page. The `mfrow` parameter takes a vector of two numbers, `c(nrows, ncols)`, which defines the number of rows and columns in the plot matrix. Plots are then filled row-wise (left to right, then top to bottom).

For instance, setting `par(mfrow = c(2, 2))` configures the current device--in this case, the PDF page opened by `pdf()`--to expect four plots arranged in two rows and two columns. Once this parameter is set, the next four plot commands executed will sequentially fill these four slots on the page. If a fifth plot command is issued while `mfrow` remains active, R will automatically create a new page in the PDF file and begin filling the grid on that new page. This provides a seamless way to paginate documents while maintaining a consistent layout within each page.

It is important to note the scope of `par()`. Changes made using `par()` persist until they are explicitly reset or until the device is closed. Best practice often involves saving the existing

graphical parameters before modifying them and restoring them afterwards, especially in functions or scripts that need to maintain a predictable environment. However, when working within the confined scope of a `pdf()` and `dev.off()` block dedicated solely to output generation, simply closing the device with `dev.off()` is sufficient, as this action automatically cleans up all device-specific settings, including `mfrow`.

Example 1: Saving Multiple Plots to a Single Page

This example demonstrates the critical use of `par()` to consolidate several visualizations onto one sheet of the final PDF document. This approach is ideal for presenting summary statistics or a series of related comparisons that need to be viewed simultaneously. By setting `par(mfrow = c(2,2))` immediately after opening the PDF device, we instruct R to divide the page into a 2x2 matrix before executing the plotting loop. The subsequent four calls to `plot(x, y)` are then directed to the four available panels on that single page.

The code segment below clearly defines the necessary steps: device setup, parameter modification, plot generation, and device teardown. The use of a `for` loop is a standard and efficient way in R to automate the creation of a sequence of similar graphics, varying only slightly based on the loop counter `i`. This method ensures all plots are generated rapidly and captured correctly within the specified grid layout.

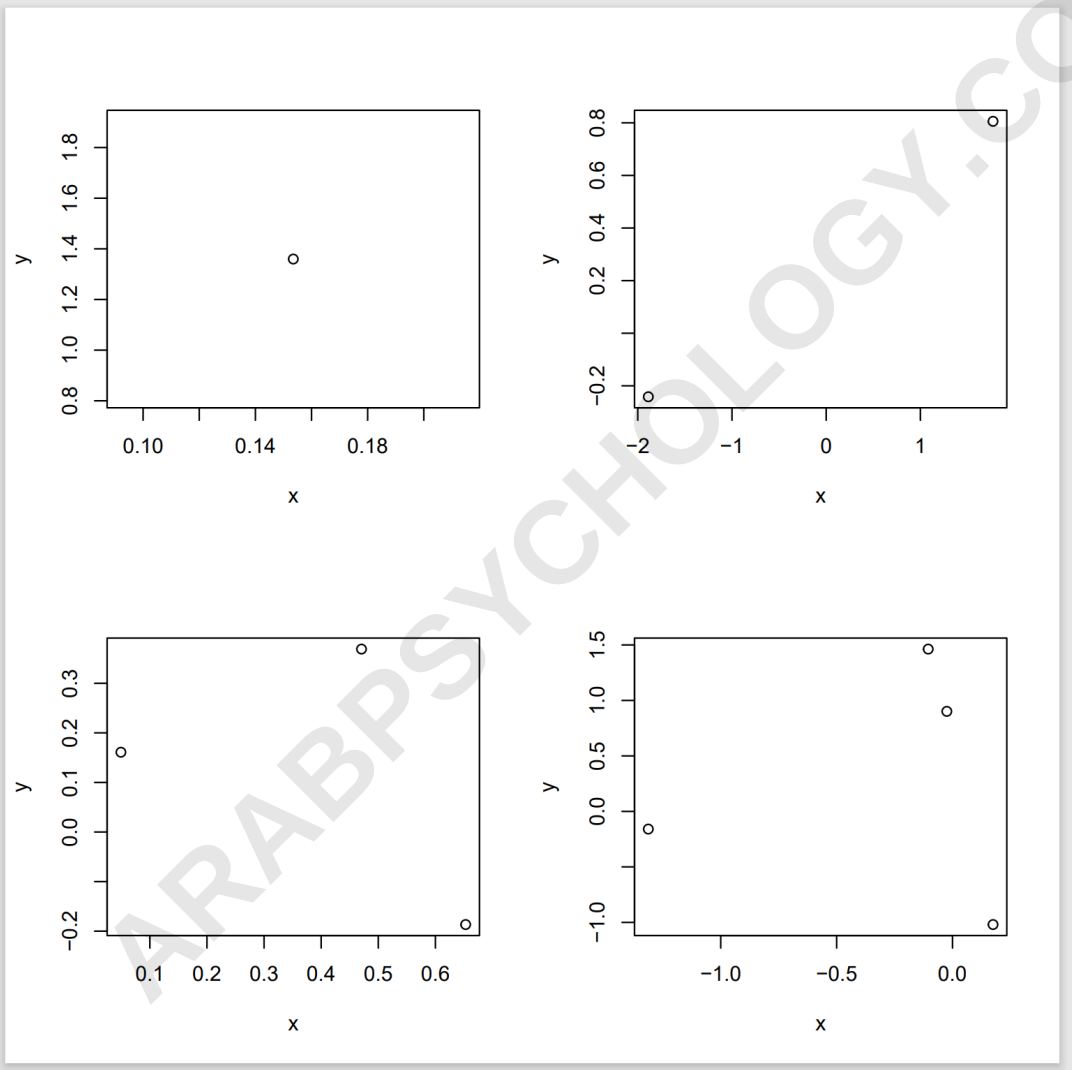
Upon reviewing the output PDF file, we confirm that all four randomized scatter plots, generated sequentially in the loop, are neatly arranged within the confines of the single page, utilizing the specified 2x2 grid structure. This confirms the successful application of the `par(mfrow)` setting in conjunction with the PDF device.

The following code shows how to save several plots to the same page in a PDF:

```
#specify path to save PDF to
destination = 'C:UsersBobDocumentsmy_plots.pdf'

#open PDF
pdf(file=destination)

#specify to save plots in 2x2 grid
par(mfrow = c(2,2))

#save plots to PDF
for (i in 1:4) {
x=rnorm(i)
y=rnorm(i)
```

```
plot(x, y)
}

#turn off PDF plotting
dev.off()
```

Once I navigate to the PDF in the specified location on my computer, I find the following one-page PDF with four plots on it:



Example 2: Saving Multiple Plots to Different Pages in PDF

In contrast to the previous example, sometimes the visualizations are complex or large enough that they require their own dedicated page. If the goal is to save multiple plots sequentially, with each plot occupying one full page within the output PDF, the layout management setting controlled by

`par()` must be omitted. By default, when the `pdf()` device is active, every single call to a plotting function signals R to start a new page before drawing the graph, assuming no layout matrix has been set.

To achieve this multi-page output, we simplify the workflow. We open the PDF device using `pdf(file=destination)`, then proceed directly to the loop containing the plot commands. Crucially, the line `par(mfrow = c(2,2))` is deliberately removed. Without this configuration, R treats the available space on the device as a single panel for the current plot. Once the plot is finished, the next plot command automatically advances the document to the subsequent page, creating a clean separation for each figure.

This method is generally preferred for generating appendices, individual figure files for manuscripts, or when the plots contain dense information that would be illegible if scaled down to fit a multi-panel layout. After the four plots are generated by the loop, the mandatory execution of `dev.off()` ensures that the file is properly closed and saved, resulting in a document with four distinct pages, each holding one plot.

To save multiple plots to different pages in a PDF, I can simply remove the **par()** function:

```
#specify path to save PDF to
destination = 'C:UsersBobDocumentsmy_plots.pdf'

#open PDF
pdf(file=destination)

#save plots to PDF
for (i in 1:4) {
x=rnorm(i)
y=rnorm(i)
plot(x, y)
}

#turn off PDF plotting
dev.off()
```

Once I navigate to the PDF in the specified location on my computer, I find the a four-page PDF with one plot on each page.

Customizing PDF Output Dimensions and Metadata

Achieving a professional result often requires more than just correctly arranging the plots; it

requires fine-tuning the output dimensions and metadata. The `pdf()` function provides arguments to precisely control the physical size of the output pages. By utilizing the `width` and `height` arguments, the user can specify the dimensions in inches. For example, `pdf(file="my_plots.pdf", width=8.5, height=11)` would create a document conforming to standard US letter size. This customization is essential when plots are destined for journals or reports that mandate specific figure sizes.

Further control is provided by the `paper` argument, which offers shortcuts for common paper sizes like "a4" or "letter", overriding manual `width` and `height` settings if specified. If you are integrating the PDF into LaTeX or other typesetting systems, ensuring the dimensions match the document layout is crucial to prevent scaling issues. Additionally, the `title` and `creator` arguments allow the embedding of metadata within the PDF file itself. While these arguments do not affect the visual representation of the plots, they contribute to the document's professional integrity and searchability.

Finally, for those requiring specific color profiles or font embedding, `pdf()` includes arguments like `colormodel` and `fonts`. While the defaults are generally adequate, customizing these features ensures that the graphical fidelity--especially regarding color accuracy and textual representation--is preserved across different viewing platforms. Mastering these customization options transforms the simple act of saving plots into a robust part of a professional data reporting workflow in R.

Summary of Workflow Best Practices

To ensure consistent and reliable PDF generation in R, adopting a disciplined workflow is critical. The first best practice is always to define the output path clearly and assign it to a variable (e.g., `destination`), which enhances script readability and makes it easier to manage file locations. Second, the pairing of `pdf()` and `dev.off()` must be treated as an atomic operation, enclosing the entire plotting sequence. Never forget to close the graphics device; failure to do so is the most common error leading to unusable output files.

Third, when employing the `par()` function for multi-panel layouts (using `mfrow` or `mfcol`), ensure that the number of plots generated matches or exceeds the space defined by the layout matrix. If you define a 2x2 grid but only generate three plots, the fourth panel will remain blank. If you generate five plots, the fifth plot will correctly wrap to a new page, maintaining the 2x2 layout on the second page.

Finally, remember that the PDF device generates vector graphics, which means they are resolution-independent. However, file size can still grow rapidly if you embed complex, high-resolution raster images (such as heatmaps derived from massive matrices) within the vector format. While R handles this efficiently, monitoring file size is advisable for documents intended for web distribution or email. By adhering to these structured practices, R users can streamline their

data visualization output and produce professional, high-quality documents consistently.