Check for updates

Tutorial

ASSOCIATION FOR PSYCHOLOGICAL SCIENCE

Advances in Methods and Practices in Psychological Science
July-September 2024, Vol. 7, No. 3, pp. 1–26
© The Author(s) 2024
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/25152459241267927
www.psychologicalscience.org/AMPPS

Visualization of Composite Plots in R Using a Programmatic Approach and *smplot2*

Seung Hyun Min
Affiliated Eye Hospital, School of Optometry and Ophthalmology, Wenzhou Medical University, Wenzhou, China

Abstract

In psychology and human neuroscience, the practice of creating multiple subplots and combining them into a composite plot has become common because the nature of research has become more multifaceted and sophisticated. In the last decade, the number of methods and tools for data visualization has surged. For example, R, a programming language, has become widely used in part because of *ggplot2*, a free, open-source, and intuitive plotting library. However, despite its strength and ubiquity, it has some built-in restrictions that are most noticeable when one creates a composite plot, which currently involves a complex and repetitive process with steps that go against the principles of open science out of necessity. To address this issue, I introduce *smplot2*, an open-source R package that integrates *ggplot2*'s declarative syntax and a programmatic approach to plotting. The package aims to enable users to create customizable composite plots by linearizing the process of complex visualization. The documentation and code examples of the *smplot2* package are available online (https://smin95.github.io/dataviz).

Keywords

composite plots, *ggplot2*, linear workflow, programmatic approach, subplotting

Received 4/18/24; Revision accepted 7/6/24

With modern software tools, there has been a surge in the number of methods and tools through which researchers and clinicians can perform data visualization, an important skill in scientific research. For instance, R, a programming language (R Core Team, 2021), has become exponentially prevalent for statistical data visualization in the last 15 years in part because of *ggplot2*, a plotting library that was introduced in 2009 by Hadley Wickham (2009). Its citation count has towered over that of Python's *matplotlib* (see Fig. 1), an extensive, flexible, but challenging low-level plotting library that was first introduced by John Hunter in 2007 (Hunter, 2007). The reason for the recent rise of *ggplot2* is that the library is free, open-source, and intuitive for users. Layers of graphics can be added sequentially on a plotting space to produce complex plots. The details of the philosophy behind *ggplot2*, which is better known as the "grammar of graphics," are well explained in a tutorial in this journal (Nordmann et al., 2022). In brief, as long as users know how to add a layer of points, a layer of lines, and other specific layers sequentially using *ggplot2*'s declarative syntax, they will be able to plot their data in both simple and complex fashions with a high level of customization without applying the programmatic approach, such as creating loops and functions (Hehman & Xie, 2021). Furthermore, because of the active community of users, there exist diverse third-party R packages (Mowinckel & Vidal-Piñeiro, 2020; Patil, 2021; Tang et al., 2016), which complement *ggplot2*, that provide shortcut functions for plotting, allowing users to plot data in just a few lines of codes in wide-ranging ways. These factors have made R, rather than Python, a preferable tool for data visualization for researchers and clinicians across disciplines and levels of experience.

Corresponding Author:
Seung Hyun Min, Affiliated Eye Hospital, School of Optometry and Ophthalmology, Wenzhou Medical University, Wenzhou, China
Email: seung.min@eye.ac.cn

Creative Commons NonCommercial CC BY-NC: This article is distributed under the terms of the Creative Commons Attribution-NonCommercial 4.0 License (https://creativecommons.org/licenses/by-nc/4.0/), which permits noncommercial use, reproduction, and distribution of the work without further permission provided the original work is attributed as specified on the SAGE and Open Access pages (https://us.sagepub.com/en-us/nam/open-access-at-sage).

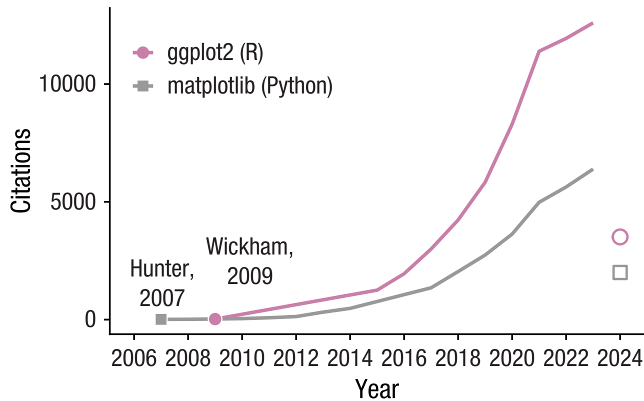


Fig. 1. Year-to-year citation count of two major plotting libraries in R and python: *ggplot2* and *matplotlib*. The year 2024 shows a partial count of the citations. Each point denotes the time point when the authors have published an article regarding their software. Citation counts were collected from Google Scholar on April 4, 2024.

Built-In Restrictions of *ggplot2*

In psychology and human neuroscience, the practice of creating multiple subplots and combining them into one composite plot is common (Kubilius, 2014). This method of data visualization is known as "subplotting." In the last few decades, it has become more widespread as research has become increasingly sophisticated, as demonstrated by the recent trend of including more variables and conditions in experiments, conducting collaborations with other laboratories if possible, and implementing multiple methodologies for data collection and analysis (Lin & Lu, 2023). These, in turn, create data sets with complicated structures, thereby requiring complex forms of data visualizations. However, as a high-level plotting library—which does not require users to plot each detail of the plot separately—*ggplot2* has some built-in restrictions that are most noticeable when one creates a composite plot.

Currently, creating a composite plot in *ggplot2* is complex for several reasons. First, although *ggplot2* allows for flexible customization of individual plots with concise codes, it is not compatible with the most well-known programmatic approach—iteration using a *for loop*—unless unorthodox methods are used. Consequently, users unfamiliar with proper methods may struggle with applying iterations in *ggplot2*.

Second, *ggplot2* provides limited options for subplotting. A typical *ggplot2* operation returns a single plot object that can be easily manipulated or stored. Although `facet_wrap()` and `facet_grid()` support data allocation into multiple subplots (facets) within a single plot object, *ggplot2* limits aesthetic customization of these subplots within a faceted plot. For example, assigning subsets of data to different subplots using multiple or hierarchical variables or applying dynamic color schemes for each variable level is challenging. To circumvent this, users might need to restructure their data frames to visualize data as intended, but this affects only components that map data variables to aesthetics, not elements such as axis limits, background themes, or coordinate systems.

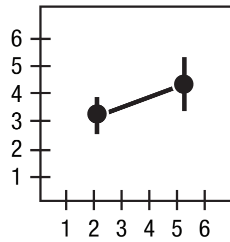
For plotting unique visual elements that have no relation to the given data across panels, a third method has been used. It involves combining separate *ggplot2* plot objects into one composite figure using libraries such as *cowplot* (Wilke, 2019) and *patchwork* (Pedersen, 2019). This method enables users to draw a composite plot flexibly but requires them to code each subplot separately (see Pseudocode 1), resulting in repetitive scripts, albeit with minor differences (compare Plot 1 and Plot 8 in Pseudocode 1). In addition, this approach restricts the aesthetic control of the composite figure, such as its layout, annotations (including legends), and marginal space (see Fig. 2), further encouraging users to code each subplot individually.

Put together, although *ggplot2* has enjoyed its widespread user base, for visualizing a composite plot, users have had to write repetitive scripts, seek third-party packages, or resort to a vector graphics editor, straying from the recommended practices for scientific reproducibility:

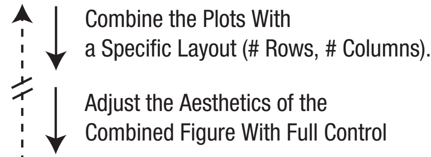
```
# Pseudocode 1: Composite plot in Figure 2
  using ggplot2
# Generate each plot separately
plot1 <- ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping =
  aes(<MAPPINGS>)) +
  . . . +
  <THEME_FUNCTION>(<LESS SPACING>) + #
    unique for this panel
  <THEME_FUNCTION>(<REMOVE X-TICKS>) + #
    unique for this panel
  <ANNOTATE_FUNCTION>(<TEXT, SHAPE
    ANNOTATIONS>)
# Repeat for plot2, plot3, . . .. , plot7

plot8 <- ggplot(data = <DATA>) +

  <GEOM_FUNCTION>(mapping =
  aes(<MAPPINGS>)) +
  . . .
  <THEME_FUNCTION>(<LESS SPACING>) +
  <THEME_FUNCTION>(<REMOVE X-TICKS>) +
    # unique for this panel
  <THEME_FUNCTION>(<REMOVE Y-TICKS>) +
    # unique for this panel
  <ANNOTATE_FUNCTION>(<TEXT, SHAPE
    ANNOTATIONS>)
```

Python's Matplotlib

Make Eight Subplots Together Using Loops



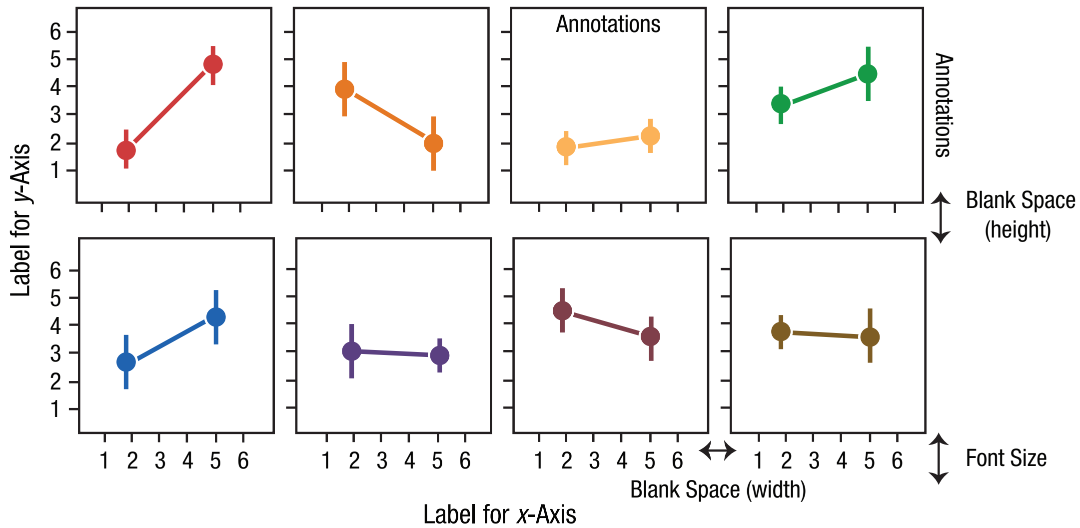
A Programmatic Approach Is Encouraged



Unnecessary to Go Back to Each Plot

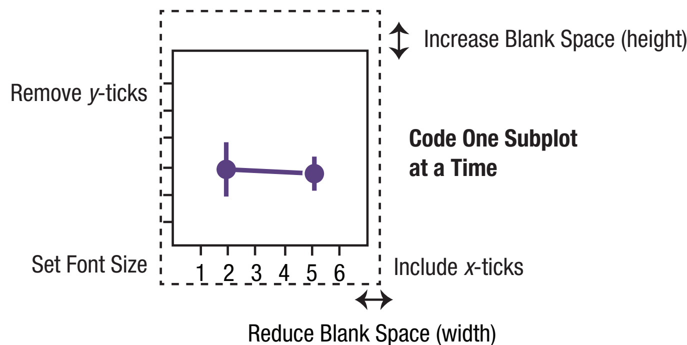
Final Figure

Title of a Composite Plot



Fine-Tune Each Plot Again After Checking the Final Figure

Combine With Other Separate Subplots



R's ggplot2

Reduce Blank Space (width)

Fig. 2. A comparison of the standard routines for subplotting in between *matplotlib* from Python and *ggplot2* from R. In Python, it is standard to generate multiple panels using iterative or functional programming approach. After the plots have been combined, the specific aesthetics of the composite plot, such as the number of rows and columns, the common legend, and *x*-axis and *y*-axis labels, can be adjusted without modifying the individual plots. Furthermore, it provides a full flexibility for text, shape, and other types of annotations to be added on the combined image. The process of visualizing a composite plot is linear in Python's *matplotlib* given its clear starting and ending points. However, in R's *ggplot2*, the process often requires users to go back and forth between the stages of creating individual plots and then combining them. Users are encouraged to plot one graph at a time and then combine all plots together as late as possible. The goal of *smplot2* is to simplify the process of complex data visualizations by resolving these issues.

```
library(<THIRD_PARTY_PACKAGES>)

multi_plot <- <COMBINE_FUNCTION>(plot1,
  plot2, plot3, plot4, plot5,
 plot6, plot7, plot8)

# Check if the multi_plot looks OK. If
  not, revise the codes that generate each
  plot.
```

On the other hand, the workflow for subplotting and creating a composite plot is simpler and more concise in Python's *matplotlib* because it requires programmatic approaches such as building loops (see Pseudocode 2) and custom functions. Applying a programmatic approach ensures a full flexibility for graphical aesthetics because users can then allocate subsets of data to unique panels using any number and combinations of variables and dynamically control the aesthetics, such as color, without writing repetitive scripts:

```
# Pseudocode 2: Composite plot in Figure
  2 using Python's matplotlib
fig, ax = plt.subplots(nrows = 2, ncols =
  4, sharex = True, sharey = True)

for <PLOT INDEX> in range(<NUMBER OF
  PLOTS>): # 8 iterations
 ax[<PLOT INDEX>].<PLOTTING_
  FUNCTIONS>(<DATA>, <COLOR>)

fig.subplots_adjust(hspace = 0.2, wspace =
  0.1) # more spacing between rows than
  columns
fig.text(x = 0.5, y = 0.95, 'Title of a
  Composite Figure')
```

The first line of Pseudocode 2 determines the structure of the composite figure. The data themselves are plotted within a *for loop* at each panel, iterating for the length of the total number of subplots (eight total; see Fig. 2). Because of the programmatic approach, the color and other aesthetics in the plot for each panel can be different, yielding more flexibility. In addition, although the codes that generate the panels are identical, the panels actually look different from one another; some have y-axis ticks or x-axis ticks (or both; see Fig. 2) because the layout of the combined figure has already been established in the beginning. Furthermore, the aesthetics of the composite figure can be controlled, such as the amount of blank space between panels (`hspace` and `wspace` in Pseudocode 2). Finally, after the panels have been combined, a common legend and annotations (texts, shapes, points, patches, lines, etc.) can be added anywhere in the composite figure. In short, *matplotlib* offers flexibility both at the level of each panel and the composite figure, making it possible for the workflow of generating a composite plot to be linear, with its clear start and resolution. This versatility of control and a structured workflow for performing complex visualizations are missing in *ggplot2*.

A Need for a Solution: *smplot2*

Although the grammar of graphics interface in *ggplot2* simplifies the code for a standalone figure, it can complicate the workflow when multiple *ggplot2* outputs are combined into one composite figure, which has restricted flexibility for aesthetics. Users have been encouraged to separately code each subplot and combine the subplots as late as possible. This is concerning because *ggplot2* has been widely used (see Fig. 1) and research routines in psychology and human neuroscience have become more sophisticated.

To address this issue, I introduce *smplot2*, an open-source R package that integrates the practice of data visualization in *ggplot2* and the programmatic approach to plotting. This package gives users equal levels of control over both individual subplots and a composite plot. It has more than 40 functions at the time of writing (see 300+ examples in https://smin95.github.io/dataviz), but for brevity, in this tutorial, I primarily discuss how it can linearize the workflow of visualizing elegant composite plots using a programmatic approach and maximize the flexibility for aesthetics in *ggplot2*. All examples here are created with aesthetic defaults of the *smplot2* package, which are clean and appropriate for research articles across various fields and data structures. The functions of *smplot2* have been optimized for subplotting to maximize the visibility of data in a composite plot by controlling the extent of blank spacing, scaling, and the relative text size. I hope that this tutorial can empower readers to perform complex and expressive data visualizations of a composite plot using a structured workflow.

Aim and structure of the tutorial

The aim of this tutorial is not to reiterate the contents of the package's documentation from the web in its entirety or introduce *ggplot2* (Hehman & Xie, 2021; Wickham, 2016; Wickham et al., 2023). Instead, I aim to present a new workflow for the visualization of a composite plot in *ggplot2* with a programmatic approach and the *smplot2* package. In the first section, I briefly introduce some of the visualization functions of *smplot2*, such as its background themes, that improve aesthetics for subplotting. Then, in the next three sections, I demonstrate how researchers can produce subplots in

Table 1. Summary of the Tutorial

Steps to create composite figures	Functions to use	Comments
1. Construct `lapply()` iteration codes	`lapply()`	During each iteration, the data should be filtered for each level of the variable. If the user wishes to subplot with more than one variable, then use nested `lapply()` functions. The order of the figures that will be generated depends on the structure of the nested `lapply()` function.
2. Draw figures iteratively using `lapply()`	`geom_*()` functions from *ggplot2*, plotting and thematic functions from *smplot2* (e.g., `sm_hgrid()`) or other functions from third-party packages	Set the limits of *y*- and *x*-axes to be identical for all panels.
3. Make a title, *x*-axis label, *y*-axis label for the composite figure	`sm_common_title(),` `sm_common_xlabel(),` `sm_common_ylabel()` – optional functions	The arguments x and y can be adjusted to set their coordinates (0 to 1).
4. Put the subplots, title, and axes labels together	`sm_put_together()`	The input for plots should be a list. The number of columns `(ncol)` and rows `(nrow)` must be specified. Blank space between panels can be controlled using `wmargin` and `hmargin` inputs (negative values mean less spacing).
5. Make a common legend manually or quickly from a sample plot and then add the legend to the composite figure	`sm_common_legend()` builds a highly customizable legend. `sm_add_legend()` then adds it to the composite figure. If no legend is provided as input, `sm_add_legend()` can derive a legend from a given sample plot.	If customization is important, use `sm_common_legend()`.
6. If needed, add annotations on the final plot	`sm_add_text()` for text annotations, `sm_add_point()` for point annotations, `annotate()` for the rest, and functions from other packages	For coordinates, `x` and `y` values should be from 0 to 1.
7. Save the figure.	`ggsave()` from the *ggplot2* package	`height` and `width` set the size of the image.

ggplot2 iteratively and then combine them into a composite plot using a linear process (similar as shown in Fig. 2 for Python's *matplotlib*) with three examples. The examples become increasingly more sophisticated to demonstrate there is no limit to how users can create and customize composite figures. The tutorial is summarized in Table 1.

Target audience

In this tutorial, I assume that readers have some basic knowledge of R and *ggplot2* and some experience with working with data frames using functions such as `filter()`, `group_by()`, `%>%`, and `summarise()`. They do not need to be familiar with concepts of programming or be fluent in any other programming languages, such as Python. Although the examples in this tutorial use randomly generated data based on human-vision studies, readers across disciplines will be able to adapt the codes/examples in this tutorial easily for their own purpose.

Readers who have not used *ggplot2* and R should read Chapters 2 and 3 of the package's documentation webpage (https://smin95.github.io/dataviz) before starting this tutorial. The chapters provide a step-by-step guide on how to install RStudio and use *ggplot2*. Individuals who have not worked with data frames in R are recommended to read the tutorial by Nordmann et al. (2022) or the early sections of Chapter 7 of the documentation webpage (Sections 7.1 and 7.2). Completing these two prerequisites for the tutorial would take about 2 to 3 hours.

Installation requirements for this tutorial

These two packages—*tidyverse* (Wickham et al., 2019) and *smplot2*—should be downloaded for the completion of the tutorial from the Comprehensive R Archive

Network (CRAN). The *tidyverse* package is a suite of multiple packages, such as *ggplot2* (for plotting and saving visualizations), *dplyr* (for working with data frames), and *readr* (for reading external data files):

```
install.packages(c('tidyverse','smplot2'
  )) # smplot2 version: 0.2.4
```

Open-science practices

With more than 300 examples, *smplot2* has been documented online in detail (https://smin95.github.io/dataviz); there are 12 are chapters devoted to the package at the time of writing. The documentation webpage was created using the *bookdown* package for reproducibility (source codes in https://www.github.com/smin95/dataviz). The codes in the tutorial and their outputs are posted online (https://www.smin95.com/smplot2doc).

Introduction to *smplot2*: Background Themes

First and foremost, one should load the two packages to memory:

```
library(tidyverse)
library(smplot2)
```

smplot2 offers various plotting and thematic functions. In this section, only the thematic functions are discussed. For more information about the plotting functions (raincloud plot, slope chart, forest plot, Bland-Altman plot, etc.), see examples in Chapters 3 through 6 from the documentation webpage.

In this example, a randomly generated data set is used as shown below:

```
set.seed(2022) # Set seed for generating
  random data
df <- data.frame(
  Subject = rep(paste0('S', 1:16),
    times = 3),
  Value = c(
  rnorm(n = 16, mean = 0, sd = 1.5), # Day 1
  rnorm(n = 16, mean = 5, sd = 1.7), # Day 2
  rnorm(n = 16, mean = 10, sd = 2.0) # Day 3
 ),
  Time = rep(paste("Day", 1:3), each = 16)
 )

head(df)
## Subject Value Time
## 1   S1  1.3502130 Day 1
## 2   S2 -1.7600187 Day 1
## 3   S3 -1.3462280 Day 1
## 4   S4 -2.1667521 Day 1
## 5   S5 -0.4965204 Day 1
## 6   S6 -4.3509435 Day 1
```

The data frame is stored in the object `df`. Each row of the `df` object represents a single observation from each `Subject` and `Time`. The column `Subject` stores identifiers for all subjects in the form of character strings; the column `Value` stores the dependent variable, which is the value of interest in this example; the column `Time` contains all identifiers for the independent variable, which has three levels, in the form of character strings: `Day 1`, `Day 2`, and `Day 3`.

In this section, raincloud plots are drawn using the function `sm_raincloud()` to present the different themes (see Fig. 3). Each subject's data (as points), the sample's distribution (in violin plots), median, and first and third quartiles (in boxplots) are typically displayed in a raincloud plot. A black dot below the boxplot for Day 1 denotes that an outlier is present. Details of this function are described in Chapter 6 of the documentation webpage. Here, the data from the `Value` column are plotted as a function of `Time`. One can map the aesthetics (i.e., `fill`) within the `ggplot()` function so that each unique color represents each level of `Time` (see the codes for Fig. 3).

Figures 3a through 3c show a different background theme. The theme with major horizontal grids is used in Figure 3a by default because `sm_raincloud()` implements the theme automatically. However, this can be overwritten if users add another theme function modularly to a *ggplot2* object (ex. `sm_classic()` is added to generate Fig. 3). These thematic functions provide minimalistic aesthetics and have `borders` and `legends` arguments. The former, if set to `borders = TRUE`, will print the border of the panel. The latter, if set to `legends = TRUE`, will print the legend of the standalone plot. There are several background themes in the package:

- `sm_hgrid()` is a theme with horizontal major grids (Fig. 2a).
- `sm_vgrid()` is a theme with vertical major grids.
- `sm_hvgrid_minor()` is a theme with horizontal and vertical grids (major and minor).
- `sm_classic()` is a theme with a standard *y*-axis on the left side and *x*-axis at the bottom (Fig. 2b).
- `sm_minimal()` is a theme with no grids (Fig. 2c).

```
# Figure 3A - Major horizontal grids
ggplot(data = df, mapping = aes(x = Time,
  y = Value, fill = Time)) +
  sm_raincloud() + # Default
  scale_fill_manual(values = sm_color
  ('blue','darkred','viridian'))
```

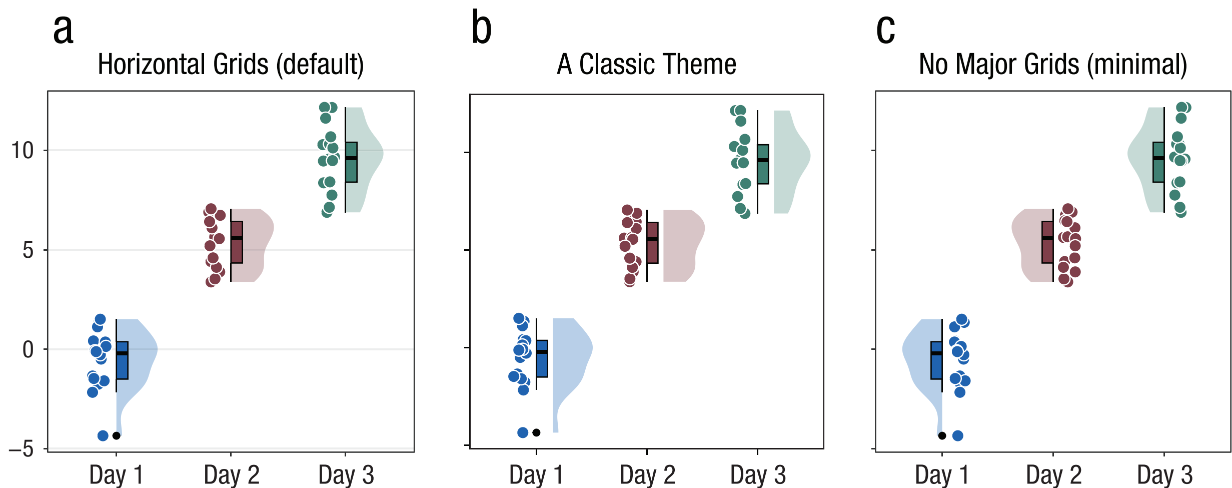


Fig. 3. (a) A default raincloud plot with a background theme that has major horizontal grids. (b) A raincloud plot with a classic theme. (c) A raincloud plot with a minimal theme (no grid).

```
# Figure 3B - Classic theme
ggplot(data = df, mapping = aes(x = Time,
  y = Value, fill = Time)) +
  sm_raincloud(sep_level = 3) + #
  Separates the graphical components
  sm_classic() +
  scale_fill_manual(values = sm_color
  ('blue','darkred','viridian'))
# Figure 3C - White background with no
  grids
ggplot(data = df, mapping = aes(x = Time,
  y = Value, fill = Time)) +
  sm_raincloud(which_side = 'l') +
    # Changes the raincloud plot's facing
    direction
  sm_minimal() +
  scale_fill_manual(values = sm_color
    ('blue','darkred','viridian'))
```

The themes have been developed to optimize the discernability of each plotting feature (e.g., relative text size, blank spacing) even when multiple subplots are combined into one composite figure. Here, for instance, the three examples of the raincloud plot have been combined into one figure using the function `sm_put_together()` (codes not shown), which I discuss extensively in the later sections. To foreshadow, `sm_put_together()`, which combines subplots into a composite figure (as described later in the text), essentially interacts with these themes and other functions to optimize the aesthetics so that each plotting feature is discernible in a multipanel, composite figure. For this reason, these functions are discussed before creating a composite plot. The hex codes of the three colors in Figure 3 are from the `sm_color()` function, which returns primarily colors with high visibility, an important factor when one creates a composite plot.

In the next three examples in which composite plots are created, I strictly use the thematic functions `sm_hgrid()`, `sm_minimal()`, and `geom_*()` to plot data in the form of lines and points (i.e., simplest type of data visualization) so that users across all levels of experience and background can understand the codes without knowing the plotting functions of *smplot2*.

Example 1: Subplotting Data Using One Variable

Simulated data set

Amblyopia is a visual deficit with origins in the primary visual cortex (Min et al., 2022). The simulated data here represent visual health in individuals with amblyopia and normal vision at various experimental conditions and types of visual stimuli. They are used throughout the rest of the tutorial.

```
df2 <- read_csv('https://www.smin95.com/
amblyopia_random2.csv')

df2_amb <- df2 %>% filter(Group ==
  'Amblyopia') %>%
  mutate(logSF = log2(SF)) %>%
  mutate(Condition = factor(Condition,
    levels = c('One','Two','Three')))

head(df2_amb)
## # A tibble: 6 × 6
## Subject absBP SF Group   Condition logSF
```

```
## <chr>  <dbl>  <dbl>  <chr>  <fct>  <dbl>
## 1 A1   0.168   0.5 Amblyopia Three  -1
## 2 A1   1.37    1   Amblyopia Three   0
## 3 A1   1.29    2   Amblyopia Three   1
## 4 A1   2.67    4   Amblyopia Three   2
## 5 A1   0.0111  8   Amblyopia Three   3
## 6 A2   0.0136  0.5 Amblyopia Three  -1
```

This data set should be loaded to memory using the code above. Throughout the tutorial, I use `%>%` operator, which is known as the "pipe." It allows the data frame from the previous operation of data transformation to be carried over, or piped, to the next operation. This reduces the burden of users from supplying the input data frame for each operation. To begin with, extract the data from the `df2` object only for individuals in the `Group == 'Amblyopia'` by using `filter()`. Next, the continuous variable `SF`, which is an acronym for spatial frequency, is converted into log2 scale using `mutate()`, which creates another column (`logSF`) based on the existing column (`SF`) in the data frame `df2`. Through this logarithmic operation in `mutate()`, a new column `logSF` is created, with equal spacing along its scale. Then, the data type of the `Condition` column is changed using `mutate()`; it is initially a string, but `mutate()` converts it into a factor and then reorders the level of the variable to its numerical order (`'One'`-`'Two'`-`'Three'`). After the data transformations, a newly formed data frame is stored in the object `df2_amb`, whose first six rows are displayed in the tutorial. Readers can double-check by comparing their own printed values of the `df2_amb` object with those in the tutorial.

The column `absBP`, which is short for absolute balance point, contains data of the dependent variable (y-axis). It is a measure of visual health. The higher the value is, the worse the vision is.

In this example, I allocate data to each panel by using the variable `Subject` (i.e., subplotting with one variable). In other words, each panel will display the data of each subject (`absBP` as a function of `logSF`).

lapply()

`lapply()` is one of the `apply` functions from base R. It applies a function to a list or a vector and returns a list with the same length as the input. A list is a data structure of an object that can contain different types of elements, such as strings, numbers, and lists. Essentially, `lapply()` is similar to how a *for loop* works, but it returns a list as output. Because *ggplot2* objects can be stored in a list but not in other types of vectors, I use `lapply()` to perform iterations. Pseudocode 3 shows the basic syntax of `lapply()`:

```
# Pseudocode 3
<OUTPUT> <- lapply(<INPUT>, <FUNCTION>,
  <ADDITIONAL ARGUMENTS>)
```

The input can be either a list or a vector. If the input has a length of 5 (i.e., five elements), then the function will be run five times, and an output list that has a length of 5 will be returned. In this case, the function will be plotting the data, with specific mapping and aesthetics, and generate *ggplot2* objects. I plot each of the nine individuals' data, so I run the function nine times (i.e., nine iterations). The returning output should therefore have a length of 9, each of which is a plot. Additional arguments can be passed to the function, but in this tutorial, there will not be any additional arguments, so these can be ignored.

First, I create an input object that specifies the nine subjects in the Amblyopia `Group`. The data frame `df2` contains a column of identifiers for subjects. These subjects have identifiers `A1` to `A9`. These can be recreated as vector `subj_list` by concatenating the string `'A'` with a sequence numbers `1:9` (1 to 9 in integers) using the function `paste0()`. The elements in the vector `subj_list` will contain subject identifiers that are also found in the `Subject` column of the data frame `df2`.

In the `lapply()` structure, through which I plot the data of each subject on a separate panel, there should be two parts. The approach is used throughout the rest of the tutorial, and it can be widely applicable across designs and disciplines.

The first part filters data using the index of the iteration. Here, `iSubj` is the index of the iteration, and it starts from 1 and ends at 9 as specified by `1:length(subj_list)`. During each iteration, the index is used to retrieve the element of the object `subj_list`, for example, `A9` from `subj_list[iSubj]` when `iSubj = 9`. The extracted subject identifier is then used to filter for each subject's data before plotting begins (e.g., `filter(Subject == subj_list[iSubj])`). The filtered data are stored in the object `subj_data`, which will be used by the subsequent plotting functions; so plotting will use only the filtered data from each subject.

The second part of the lapply() function plots the filtered data. The variables are mapped to aesthetics, and the appearance of the plot is customized using functions from *ggplot2*. Here, the specifications are set so that the data frame to be used is `subj_data`, `x` is `logSF`, and `y` is `absBP`, which is the outcome of interest in this simulated data set. Moreover, the aesthetic `group` is mapped to the variable `Condition` of the data frame `subj_data` so that the points are connected with lines for each condition. In addition, within the `ggplot()` function, the `shape`, the filling color of the points

(`fill`), and `color` of the lines are all set to be unique for each condition. In other words, all three conditions will be plotted in one panel of each subject at once.

In this example, each condition is coded to a unique shape with the function `scale_shape_manual()`. The first condition is coded to the shape value of 21 (circle with borders), the second to the value of 22 (square with borders), and the third to the value of 23 (triangle with borders). Because these shapes have borders, the argument `fill` determines their filling color, and `color` determines their border color, which is set to `transparent`. Likewise, with `scale_fill_manual()`, each condition is color coded to a specific filling color. The colors are specified in the object `cList`, which is defined outside the `lapply()` function using the `sm_palette()` function that returns three colors here. This function returns default colors of the package and is equivalent to `sm_color()` except that it takes the number of colors as input instead of character strings specifying the colors. Users are encouraged to find their own color schemes from other packages, such as *RColorBrewer* and *viridis*, available in the R ecosystem:

```
subj_list <- paste0("A", 1:9) # 9 subjects
cList <- sm_palette(3) # Three colors from
  smplot2 (defaults)

indv_plots <- lapply(1:length(subj_list),
  function(iSubj) {
  # First part: Filter for each subject's
    data during each iteration
  subj_data <- df2_amb %>%
  filter(Subject == subj_list[iSubj])

  # Second part: Plot each subject's data
  ggplot(data = subj_data, aes(
  x = logSF, y = absBP, group = Condition,
  shape = Condition, fill = Condition,
    color = Condition
 )) +
  geom_line(linewidth = 1) +
  geom_point(size = 5, color =
    "transparent") +
  scale_color_manual(values = cList) +
  scale_fill_manual(values = cList) +
  scale_shape_manual(values = c(21, 22,
    23)) +
  sm_hgrid() +
  scale_y_continuous(limits = c(0, 3)) +
  scale_x_continuous(
  limits = c(-1.3, 3.3),
  labels = c(0.5, 1, 2, 4, 8)
 )
})
```

As `lapply()` performs function each time, a plot will be generated. Each plot will get stored in the object `indv_plots`, which is a list. Because `length(subj_list)` is 9 and the input for the `lapply()` function is a digit from 1 to 9 (`1:length(subj_list)`), there will be nine iterations and hence, nine plots that will be generated.

When one codes for multiple subplots in a `lapply()` function (second part of the structure), it is important to make the limits of *x*- and *y*-axes identical. In the `lapply()` function, both have been specified using `scale_y_continuous()` (*y*-axis: 0 to 3) and `scale_x_continuous()` (*x*-axis: –1.3 to 1.3). If there is no specification of the limits, each plot will have its own limit based on each subject's data. In addition, notice that although users plot data as a function of the variable `logSF`, which has values of –1, 0, 1, 2, and 3, the tick labels of the *x*-axis are displayed as 0.5, 1, 2, 4, and 8. This is because the `labels` argument has been supplied with these specifications in the function `scale_x_continuous()`. Essentially, these inputs mask over the true values of the *x* ticks on the plot. This is a common method of plotting data in human-vision studies because the visual system has been known to process information nonlinearly (Baker et al., 2012), and it is specific to the examples in the tutorial.

To display a plot from the object `indv_plots`, users can type the name of the list `indv_plots` in the console or subset for one specific subject's plot using double brackets (e.g., `indv_plots[[3]]`). This individual plot still has ticks for *x*- and *y*-axes and their labels. However, these will be removed automatically later or resized during the generation of a composite plot:

```
# Figure not shown
indv_plots[[3]] # Print the third plot
  from the list
```

Next, users define the title and common *x*- and *y*-axes labels of the composite figure that they will create. As their names suggest, `sm_common_title()` sets the title of the combined figure, `sm_common_xlabel()` sets the common *x*-axis label of the combined plot, and `sm_common_ylabel()` sets the common *y*-axis label of the combined figure. In these three functions, `x` and `y` control the location of the texts. Their defaults are set to `x = 0.5, y = 0.5,` which refers to the center origin of their respective areas (`x` and `y` do not refer to the coordinate relative to the combined figure):

```
# Figure 4 - Set the title and axis labels
  of the composite figure
title <- sm_common_title("Individual data
  (subplotting with one variable)", x =
  0.53, y = 0.52)
```

```
xlabel <- sm_common_xlabel("Spatial
  frequency (c/deg)", x = 0.52)
ylabel <- sm_common_ylabel("Visual
  deficit")
```

Notice that this process is highly similar to what is often used in Python's *matplotlib*, where `fig` refers to the object that stores the combined figure (see Pseudocode 4):

```
# Pseudocode 4: Titles and axis labels in
  Python's matplotlib (Figure 4)
fig.suptitle('Individual data', x, y)
fig.text(x, y, 'Spatial frequency (c/
  deg)') # x-axis label
fig.text(x, y, 'Visual deficit', rotation
  = 90) # y-axis label
```

In Python's *matplotlib*, the text labels and the title get added to `fig` (the composite plot) using the object-oriented approach. Here, use `sm_put_together()`, which is essentially a layout function that creates a composite figure from individual plots. The output from `sm_put_together()` must be stored in an output object (e.g., `plots_tgd`). Three inputs must be provided to run `sm_put_together()`: (a) the list object, which stores all plots (e.g., `all_plots` argument = `indv_plots`), (b) the number of columns (e.g., `ncol` argument = `3`), and (c) the number of rows (e.g., `nrow` argument = `3`). There are optional arguments as well, such as `title`, `xlabel`, and `ylabel`. For instance, if `title` is not supplied as input, then no space will be allocated for a common title in the composite figure. Here, I supply `title` in `sm_put_together()`. In addition, the extent of blank spacing (i.e., margin) in both width (`wmargin`) and height (`hmargin`) can be adjusted (see Fig. 2). In this example, they are set as negative values to minimize the spacing between subpanels:

```
# Figure 4 - Combine subplots into a speci-
  fied layout
plots_tgd <- sm_put_together(
  all_plots = indv_plots, title = title,
    xlabel = xlabel,
  ylabel = ylabel, ncol = 3, nrow = 3,
  wmargin = -4.5, hmargin = -4.5
)
```

Users can save the figure using `ggsave()` (see Fig. 4). Supply the name of the image file in strings (`'together1.pdf'`) as the function's first input and the object of the composite figure (`plots_tgd`) as its second input. This forces the function to save the object `plots_tgd` as `together1.pdf` in the directory. In addition, I set the dimension of the image so that it has a `height` and a `width` of 9 inches:

```
# Figure 4 - Save the composite output as a
  vector file
ggsave("together1.pdf", plots_tgd,
  width = 9, # inches
  height = 9
)
```

Immediately in Figure 4, one can notice that the function `sm_put_together()` has removed extraneous tick labels and titles from both axes in the inner panels. This was possible because I had provided the layout of the composite figure that was to be constructed in `sm_put_together()`, which is similar to how *matplotlib* controls the layout of the figure (see Pseudocode 2). Although the default of the function removes the extraneous ticks in inner panels (`remove_ticks = 'some'` in `sm_put_together()`), this option can be overwritten so that all ticks are kept (`remove_ticks = 'none'`) or removed (`remove_ticks = 'all'`). The order of the subpanels follows that of plots that are generated by the `lapply()` code chunk. In this case, I set the layout to be (`ncol = 3, nrow = 3`), so axis ticks in the second, third, fifth, and sixth panels in the composite figure are removed.

One can also label each panel by annotating each subject's identifier (e.g., A1 for Subject 1 in `Amblyopia`; see Fig. 5). There are two ways of achieving this. The first way is revisit and modify the code chunk that generates the nine points iteratively using `lapply()`, but this goes against the aim of linearizing the process of subplotting. Therefore, use the function `sm_panel_label()` to label each panel:

```
# Figure 5 - Add subject identification
  label (ex. A1) in each panel
indv_plots_label1 <- sm_panel_label(
  all_plots = indv_plots, x = 0.15, y =
    0.85,
  panel_pretag = "A", panel_tag = "1",
  text_color = "black"
)
```

The function `sm_panel_label()` has a few arguments, some of which are similar to those of the former function. For the first argument `all_plots`, users must provide the list vector (`indv_plots`) that stores all plots. Next, `x` and `y` determine the location of the panel label; 0.5 is the origin of the panel (i.e., the center of each subplot). `panel_tag` sets the string for enumeration. In this example, I set `panel_tag = "1"` so that the first panel will have "1" labeled, but the next one will have "2." There are other options to enumerate each

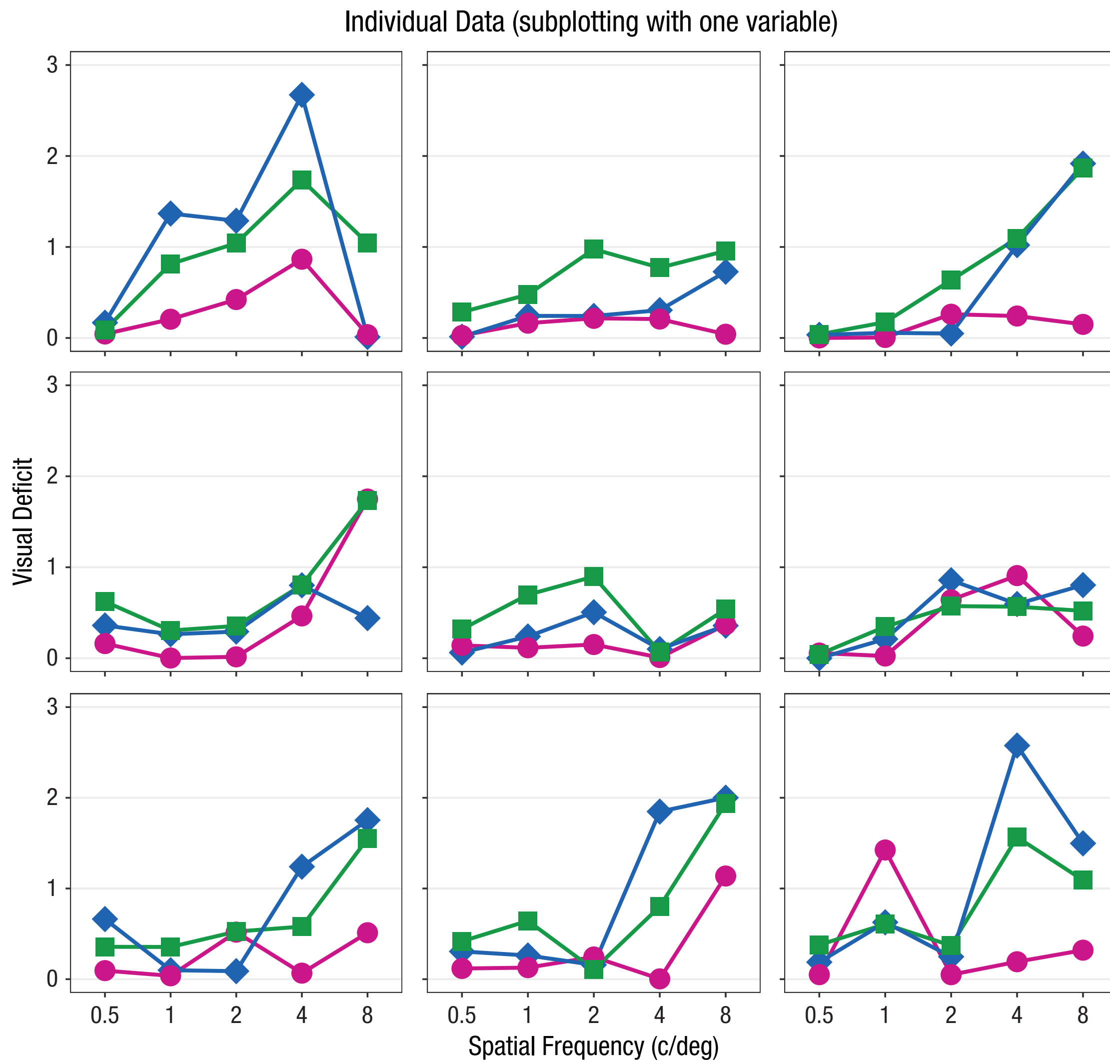


Fig. 4. A composite plot with three columns and three rows. Each panel plots each subject's data across all three conditions. A legend is absent in this figure.

panel, such as (a) `panel_tag = "A"` for uppercase letters, (b) `panel_tag = "a"` for small case letters, (c) `panel_tag = "I"` for upper roman numerals, and (d) `panel_tag = "i"` for lower roman numerals. These options of the `panel_tag` argument were included as inspired by the `plot_annotation()` function of the *patchwork* package (Pedersen, 2019). In addition, there are tag labels that can be set to be consistent across panels: `panel_pretag` and `panel_posttag`. As their names imply, `panel_pretag` comes before `panel_tag`, and `panel_posttag` comes after `panel_tag`. These two arguments can be any string at any length. To label each panel using the subject's identifier that is consistent with those in the data frame `df2` (e.g., `A1` and `A3`), `panel_pretag` should be "A." Then, we store the output from `sm_panel_label()` in the object `indv_plots_label1`. The differences between `plot_annotation()` and `sm_panel_label()` are that in `sm_panel_label()`, (a) the locations can be specified in `x` and `y` coordinates within but not outside each panel, (b) annotations can be added multiple times (as demonstrated in this example) in sequence, and (c) the plot input must be a single list object rather than separate *ggplot2* objects.

One can also label each panel so that the first panel has "a)" and the second panel has "b)." To do so, use the function `sm_panel_label()` again, in which the user provides the `indv_plots_label1` object as input and sets `panel_tag = 'a'` and `panel_posttag = ')'`. This creates labels with a small alphabet that is followed by a parenthesis in each panel. The final output is then saved in the `indv_plots_label2` object, which is the end result of running `sm_panel_label()` twice:

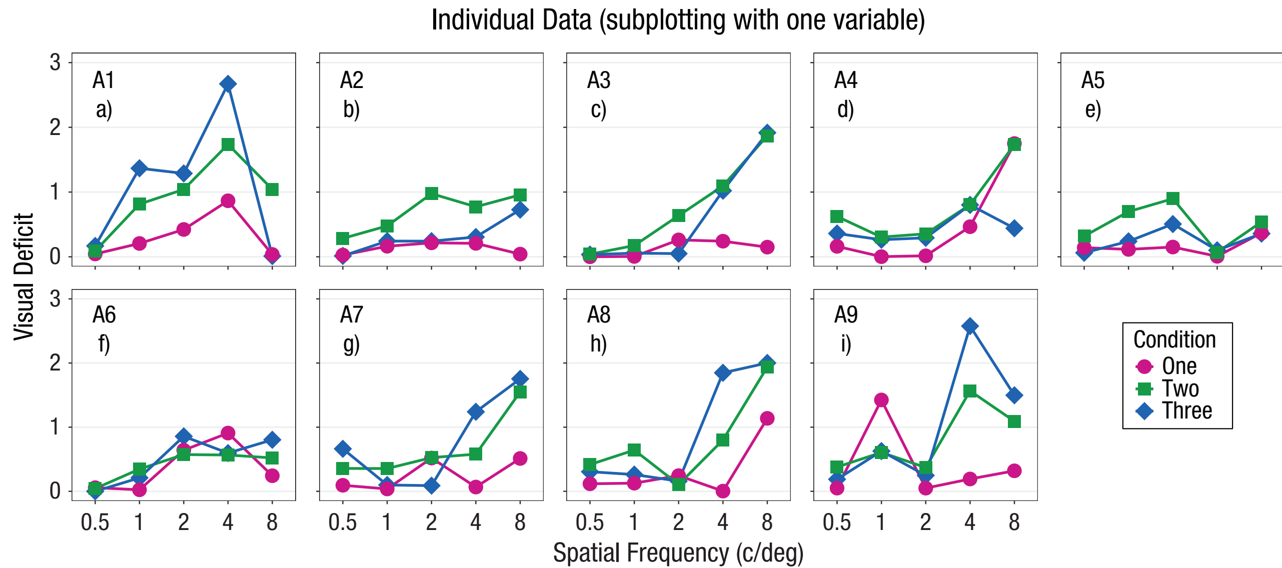


Fig. 5. A composite plot with two rows and five columns. Nine panels display each subject's data in the amblyopia group, and the last panel shows the legend, representing each condition with a unique color.

```
# Figure 5 - Add panel label in small alpha-
  bets followed by a bracket
indv_plots_label2 <- sm_panel_label(
  all_plots = indv_plots_label1, x = 0.15,
    y = 0.7,
  panel_tag = "a", panel_posttag = ")",
  text_color = "black", fontface = "bold"
)
```

Next, one can sort the nine panels into a layout with five columns and two rows (`ncol = 5, nrow = 2`) using the function `sm_put_together()`. One also add the common title, common *x*-axis label, and common *y*-axis label by directly supplying character strings rather than using `sm_common_*()` functions. This option is less flexible, but it is more convenient; the text size can still be adjusted using the `labelRatio` argument, where 1 refers to the default size but not its location. The `labelRatio` argument does not affect the size of text labels created from `sm_common_*()` functions. `sm_put_together()` also supports combining subplots with secondary *x*- and *y*-axes (not shown in the tutorial); `xlabel2` and `ylabel2` should be provided to set the titles for these axes:

```
# Figure 5 - Combine the subplots into one
  figure
plots_tgd2 <- sm_put_together(
  all_plots = indv_plots_label2,
  title = "Individual data (subplotting
    with one variable)",
  xlabel = "Spatial frequency (c/deg)",
  ylabel = "Visual deficit", ncol = 5,
    nrow = 2,
  wmargin = -2, hmargin = -2, labelRatio =
    0.9
)
```

Now that a composite figure has been created with individual subplots and labels, one can add a common legend in the combined figure `plots_tgd2`. There are two ways to do so using *smplot2*. There is a quick way and a slow but highly customizable way. They both involve the function `sm_add_legend()`. To preview, readers can compare the legend in Figure 5 from the quick method with the legend in Figure 6 from the slow method.

The first method of adding a legend basically forces `sm_add_legend()` to derive a legend from a reference plot so that users do not have to manually make it. To make the legend using the quick method, users should provide some inputs for some arguments. The output from `sm_put_together()` (`plots_tgd2`) must be supplied as input for the argument `combined_plot`. The coordinate of the legend can be specified using `x` (horizontal coordinate of the legend) and `y` (vertical coordinate of the legend) arguments. In addition, a reference plot from which the legend can be derived must be supplied for the argument `sampleplot` (i.e., one plot from `indv_plots`). In this example, the coordinate is set to be within the area of the empty 10th panel (`x=0.92, y=0.35`); the sample plot is derived from the first subject's plot (`indv_plots[[1]]`). The `direction` argument (i.e., orientation) of the legend is specified to

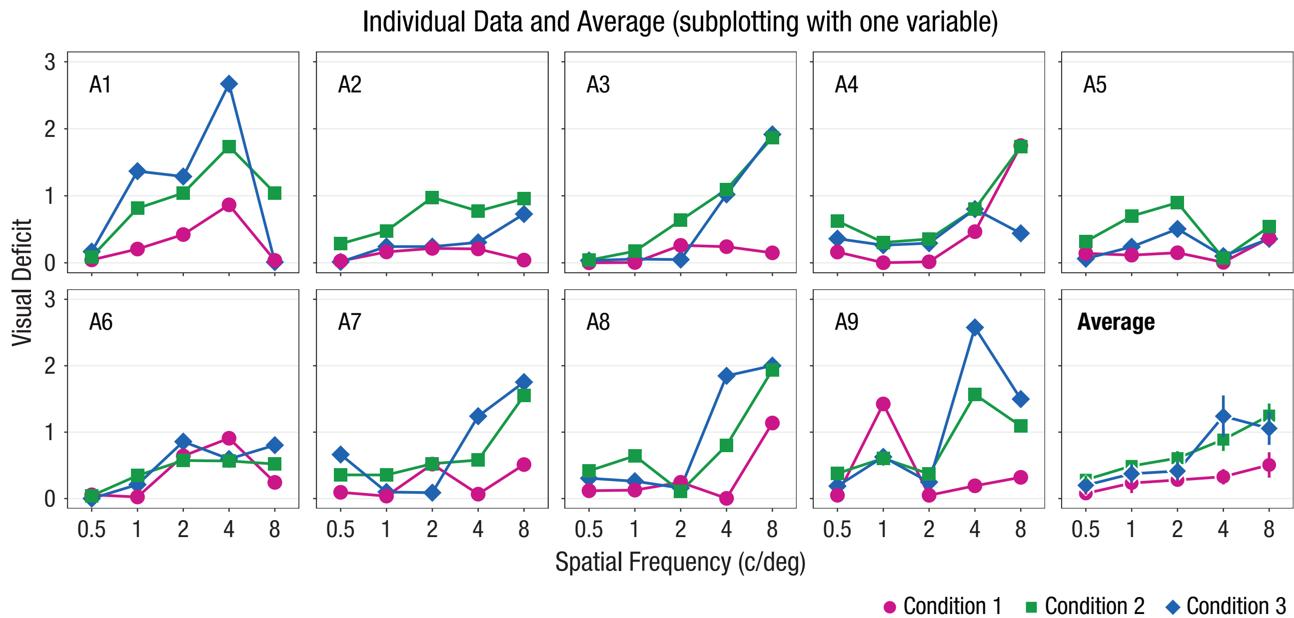


Fig. 6. A composite plot with two rows and five columns with a common legend that is located at the bottom-right area of the figure. The first nine panels display each subject's data from the amblyopia group, and the last panel shows the average data with error bars, which represent standard errors.

be `vertical`, not `horizontal`. `legend_spacing` is an argument that can set the extent of blank space within the legend to prevent overcrowding. If `border = FALSE`, then the border of the legend will be removed. The font size of the legend can be adjusted using the argument `font_size`. The code below stores the output from `sm_add_legend()` in the object `plots_tgd2_legend` and then saves the figure as a vector file using the `ggsave()` function with specified `width` and `height`:

```
# Figure 5 - Legend in the area of the 10th
  panel
plots_tgd2_legend <- sm_add_legend(
  combined_plot = plots_tgd2, x = 0.92, y
    = 0.35,
  sampleplot = indv_plots[[1]], direction
    = "vertical",
  legend_spacing = 1, border = TRUE, font_
    size = 13
)

# Figure 5 - Save the composite figure as
  a vector file
ggsave("together2.pdf", plots_tgd2_legend,
  width = 15, # inches
  height = 6.6
)
```

Two observations can be made from the legend in Figure 5. First, the legend's title matches to one of the column's name (`Condition`) in the data frame `df2`. Second, labels within legends are identical to the string characters that are provided in the `Condition` column of `df2`. These similarities indicate that the legend's title and labels have been automatically generated according to the given data frame. If the legend is created in this quick approach (by forcing `sm_add_legend()` to derive one from a sample plot), the title and the labels cannot be customized, although the title can be removed:

```
# Compute the average and standard error for
  each SF and Condition level
df2_amb_avg <- df2_amb %>%
  group_by(logSF, Condition) %>%
  summarise(
  avgBP = mean(absBP),
  stdErr = sm_stdErr(absBP), .groups =
    "drop"
)

head(df2_amb_avg)
## # A tibble: 6 × 4
##    logSF Condition  avgBP stdErr
##    <dbl> <fct>      <dbl> <dbl>
## 1     -1 One       0.0769 0.0182
## 2     -1 Two       0.283  0.0649
## 3     -1 Three     0.199  0.0720
## 4      0 One       0.234  0.151
## 5      0 Two       0.491  0.0705
## 6      0 Three     0.374  0.135
```

Because there is one empty panel that is available for plotting (10th panel of Fig. 5), users can add an additional panel that shows the average data of the nine subjects with error bars (e.g., standard error). This panel showing the average data should have the same *x*- and *y*-limits as those of the individual subjects' panels. Next, one can compute the average and standard errors of the data from nine individuals for each independent variable (`logSF`) and experimental condition (`Condition`) and store the resulting data frame into the object `df2_amb_avg`. The initial step can be achieved using functions from the *dplyr* package, such as `group_by()` and `summarise()`. `group_by()` does not change the data frame at the surface level. Instead, it changes its underlying structure so that the following functions that will be called later for computations within `summarise()` will be done separately for each grouped variable's level. The two computations—mean and standard error—are conducted using the functions `mean()` and `sm_stdErr()`, respectively. The latter is a shortcut function from the *smplot2* package. The grouping will remain even after the computation has been performed, so it is crucial to undo the grouping by setting `.groups = 'drop'` in `summarise()`. For more information about these functions, see Chapter 7 of the documentation webpage (https://smin95.github.io/dataviz):

```
# Figure 6 - 10th panel showing the average
  data
avg_plot <- ggplot(data = df2_amb_avg,
  aes(
  x = logSF, y = avgBP, group = Condition,
  shape = Condition, fill = Condition,
    color = Condition
)) +

  geom_line(linewidth = 1) +
  geom_point(size = 5, color = "white",
    stroke = 1) +
  geom_linerange(aes(ymin = avgBP -
    stdErr, ymax = avgBP + stdErr),
    linewidth = 1) +
  scale_color_manual(values = sm_
    palette(3)) +
  scale_fill_manual(values = sm_
    palette(3)) +
  scale_shape_manual(values = c(21, 22,
    23)) +
  sm_hgrid() +
  scale_y_continuous(limits = c(0, 3)) +
  scale_x_continuous(
  limits = c(-1.3, 3.3),
  labels = c(0.5, 1, 2, 4, 8)
 ) +
  annotate("text", label = "Average", x =
    -0.3, y = 2.65, size = 5.5,
  fontface='bold')
```

With the newly created data frame `df2_amb_avg`, one can plot the average data using the same mapping specifications as those in the individual plots in the `lapply()` function. Average data are plotted as points with white borders using the `geom_point()` function. The lines are drawn to join the points with `geom_line()`, and the error bars without caps are displayed using `geom_linerange()`, which is a useful function for indicating intervals of some range. The aesthetic mapping is defined in `geom_linerange()` so that vertical lines with certain ranges can be plotted at each level of `logSF` (*x*-axis); explicitly specify the minimum (`ymin = avgBP - stdErr`) and maximum (`ymax = avgBP + stdErr`) of the vertical range to be equal to the range of the standard error of the average data. Do not use `lapply()` function here because one plot needs to be made.

```
# Figure 6 - Combine all the subplots into
  a composite plot
all_plots <- list(indv_plots_label1,
  avg_plot)

plots_tgd3 <- sm_put_together(
  all_plots = all_plots,
  title = "Individual data and average
    (subplotting with one variable)",
  xlabel = "Spatial frequency (c/deg)",
  ylabel = "Visual deficit", ncol = 5,
    nrow = 2,
  wmargin = -4.5, hmargin = -4.5,
    labelRatio = 0.9
)
```

The limits of both *x*- and *y*-axes and the thematic background (i.e., `sm_hgrid()`) are set to be identical to those of the individual plots. In addition, annotate the average plot with the bolded text `'Average'` using `annotate()`, where users can specify its coordinate to be at the top left of the panel (`x = -0.3, y = 2.65`) in the units of the plotted data (`x = logSF`, `y = avgBP`). The plot output is then saved in the object `avg_plot`.

Then, store all 10 plots (9 individuals' plots in `indv_plots_label1` + 1 average plot in `avg_plot`) that have been generated into one list using the function `list()` and then assign the output to the object `all_plots`. The `all_plots` object will be the input for `sm_put_together()`, which will create a composite plot using the plots, title, and axis labels with a layout (`ncol = 5` and `nrow = 2`).

Because there are 10 panels to plot in a layout with five columns and two rows, there should already be a limited amount of available space for the legend (for the final output, see Fig. 6). To effectively use the remaining plotting space, users will have to build and customize a legend using the function `sm_common_legend()` rather than relying on the automatically generated legend from `sm_add_legend()`. After creating a legend manually, users can then add it to the composite plot using `sm_add_legend()` at a specific location within the combined figure. This option requires more work, but it is more flexible.

To do so, users need to essentially create a new plot using the standard procedure of *ggplot2* (see codes below). This includes setting the mapping `x` and `y` variables to certain aesthetics. Points are also drawn using `geom_point()` so that they are included in the legend. The legend labels have also been changed, as specified in the two `scale_*()` functions. Finally, users finish creating the legend by using `sm_common_legend()`, which essentially hides all features of a normal graph, such as points and axis lines, that will be plotted otherwise. As a result, the output `legend2` prints the legend components only when it gets called. I set the legend to have a `horizontal` orientation with no borders (`border = FALSE`). The text size of the legend can also be adjusted using the argument `textRatio`, which has been set to 1.1 in this example; this means that the text size of the legend is 1.1 times larger than the default from a given theme. Finally, `legend_spacing` controls the amount of blank space in the legend:

```
# Figure 6 - Create a legend manually
legend2 <- ggplot(data = df2_amb, aes(
  x = logSF, y = absBP, group = Condition,
  shape = Condition, fill = Condition
)) +

  geom_point(size = 4.5, color = "white") +
  scale_fill_manual(
  values = sm_palette(3),
  labels = c(
  "Condition 1 ", "Condition 2 ",
  "Condition 3 "
 )
 ) +
  scale_shape_manual(
  values = c(21, 22, 23),
  labels = c(
  "Condition 1 ", "Condition 2 ",
  "Condition 3 "
 )
 ) +
  sm_common_legend(
  title = FALSE, direction = "horizontal",
    border = FALSE,
  textRatio = 1.1, legend_spacing = .9
 )
```

The customized legend can be added to the composite plot with the function `sm_add_legend()` at a specific coordinate (`x = 0.84, y = 0.05`; bottom-right region of Fig. 6). Because I have manually created the legend with `sm_common_legend()`, there is no need to supply inputs for other arguments in `sm_add_legend()`, such as `direction`, `border`, and `sampleplot`, all of which will be ignored. The final output—a composite figure that shows both individual plots and a panel that shows the average data (Fig. 6)—is saved using `ggsave()` from the *ggplot2* package:

```
# Figure 6 - Save the figure with a legend
  as a vector file
plots_tgd3_legend <- sm_add_legend(
  combined_plot = plots_tgd3, legend =
    legend2, x = 0.84,
  y = 0.05
)

ggsave("together3.pdf", plots_tgd3_legend,
  width = 15, # inches
  height = 6.6
)
```

Readers might realize that they could also generate Figures 4 through 6 with `facet_wrap()`. Indeed, when subplotting data using one variable, using `facet_wrap()` might be simpler. However, the advantage of using *smplot2*'s pipeline with `lapply()` is that it remains very similar even if more variables or `lapply()` functions are added (next two examples).

Example 2: Subplotting Data Using Two Variables

Thus far, I have explored only a relatively simple way of assigning data to each panel. In this example, I allocate data to each panel using two variables (`Condition` and `Subject Group`).

In this example, the same data set (`df2`) will be used, albeit with some data transformations. Average data at each level of condition and subject group will be plotted. There are three experimental conditions and two groups, totaling to six combinations of levels from the two variables. Therefore, the data will be allocated to six separate panels:

```
df2_avg <- df2 %>%
  mutate(logSF = log2(SF)) %>%
  mutate(Condition = factor(Condition,
    levels = c("One", "Two", "Three")))
    %>%
  group_by(logSF, Condition, Group) %>%
  summarise(
  avgBP = mean(absBP),
  stdErr = sm_stdErr(absBP), .groups =
    "drop"
 )

head(df2_avg)
## # A tibble: 6 × 5
##   logSF Condition Group     avgBP  stdErr
##   <dbl> <fct>     <chr>     <dbl>  <dbl>
## 1    -1 One       Amblyopia 0.0769 0.0182
## 2    -1 One       Normal    0.149  0.0491
## 3    -1 Two       Amblyopia 0.283  0.0649
## 4    -1 Two       Normal    0.287  0.0707
## 5    -1 Three     Amblyopia 0.199  0.0720
## 6    -1 Three     Normal    0.244  0.0868
```

To begin with, the original data frame `df2` is transformed similarly as in the previous example by creating another column for spatial frequency in log-scale to achieve equal spacing (`logSF` column) and reordering the level of the `Condition` column to its proper, numerical order by converting it into factor from strings (`'One'`-`'Two'`-`'Three'`). Next, the codes compute the average and standard error for each combination of the two variables. This is possible because the underlying structure of the data frame is transformed using `group_by()` so that subsequent computations for average and standard error on these data in `summarise()` are performed according to the specified groupings. As in Example 1, the mean is computed using `mean()`, and the standard error is computed using `sm_stdErr()`.

In this example, there will be two levels of `lapply()` structure in the code fragment because I perform subplotting with two variables (`Group` and `Condition`). Hence, the code structure will have one inner function and one outer function. This is better known as a nested structure, which involves using functions in a hierarchical fashion. The outer function will iterate around the variable `Group`, and the inner function will iterate around `Condition`. This structure of the nested functions will affect the order in which the plots will be generated and stored in the object `avg_plots`. Specifically, plots from the first level of `Group` and all three levels of `Condition` will be generated first, followed by those from the second level of `Group`.

With the structure of the nested `lapply()` functions in mind, users can first create vectors that contain string elements that match the identifiers of `Group` and `Condition` columns from the `df2_avg` data frame. These are then stored in `group_list` and `cond_list` objects, respectively. Each iteration of the nested functions will filter the average data based on the selected element of `group_list` and `cond_list` from their indices, for example, `Group == group_list[[iGroup]]`, where `iGroup` = 1, and therefore, `Amblyopia`:

```
# Figure 7 - Visualize each subplot
group_list <- c("Amblyopia", "Normal")
cond_list <- c("One", "Two", "Three")
shape_list <- c(21, 22, 23) # Shape for
  each condition
cList <- list(
  c("#ddc7d8", "#d3a7c0", "#b7729a"), #
    Color for each subject group
  c("#bababa", "#999999", "#636262")
)

avg_plots <- lapply(1:length(group_list),
  function(iGroup) {
  lapply(1:length(cond_list),
    function(iCond) {
  # First part: Filter average data for
    each group & condition during each
    iteration
 currData <- df2_avg %>%
  filter(Condition == cond_list[iCond])
    %>%
  filter(Group == group_list[iGroup])

  # Second part: Plot the filtered average
    data
 pp <- ggplot(data = currData, aes(x =
   logSF, y = avgBP)) +
  geom_area(fill = cList[[iGroup]]
    [[iCond]], alpha = 0.3) +
  geom_line(linewidth = 1, color =
    cList[[iGroup]][[iCond]]) +
  geom_point(
  size = 5, shape = shape_list[[iCond]],
    color = "white",
  fill = cList[[iGroup]][[iCond]], stroke
    = 1
 ) +
  geom_linerange(aes(ymin = avgBP -
    stdErr, ymax = avgBP + stdErr),
  linewidth = 1, color = cList[[iGroup]]
    [[iCond]]
 ) +
  scale_y_continuous(limits = c(0, 1.6)) +
```

```
  scale_x_continuous(
  limits = c(-1.3, 3.3),
  labels = c(0.5, 1, 2, 4, 8)
 ) # pp is the intermediate plot output

  # Third part (optional): Apply different
    themes based on subject grouping
  if (group_list[iGroup] == "Amblyopia") {
 pp + sm_minimal() # No grids for
   Amblyopia
 } else {
 pp + sm_hgrid() # Horizontal grids for
   Control
 }
 })
})
```

In addition, shapes are set to be unique for each of the three experimental conditions; their values are stored in the `shape_list` vector, and each value will get selected during the iteration for each condition to specify the shape when plotting (e.g., `shape = shape_list[[iCond]]`). The color palettes for the two subject groups are set to be different, and the intensity of the color is set to increase as a function of `Condition`. The color values (in hex codes) are stored in the list vector `cList`, which contains six different colors that have been separated into two vectors (one for each `Group`). Therefore, if the iteration has an index for the first level of `Group` and the second level of `Condition`, the corresponding color will be `cList[[1]][[2]]`, where `cList[[1]]` contains three colors that are in the pink palette in the increasing intensity. Here, the first level of `Group` is `Amblyopia` because the first element of `group_list` is `Amblyopia`. Finally, using `if` conditional statements, users can allow subplots only of `Normal` group's data to have horizontal grids but not those of `Amblyopia`. This is possible because the intermediate plotting output is stored as in the second part of the `lapply` function. Thematic functions are then added modularly to `pp` in the third part of the `lapply` function, creating a final output. The third part is optional to perform subplotting, and it can be useful to set specific customizations. Integrating the programmatic approach for plotting allows users to dynamically control aesthetics, such as color, shape, and theme (Fig. 7), which is very difficult to do in *ggplot2* unless users code plots separately.

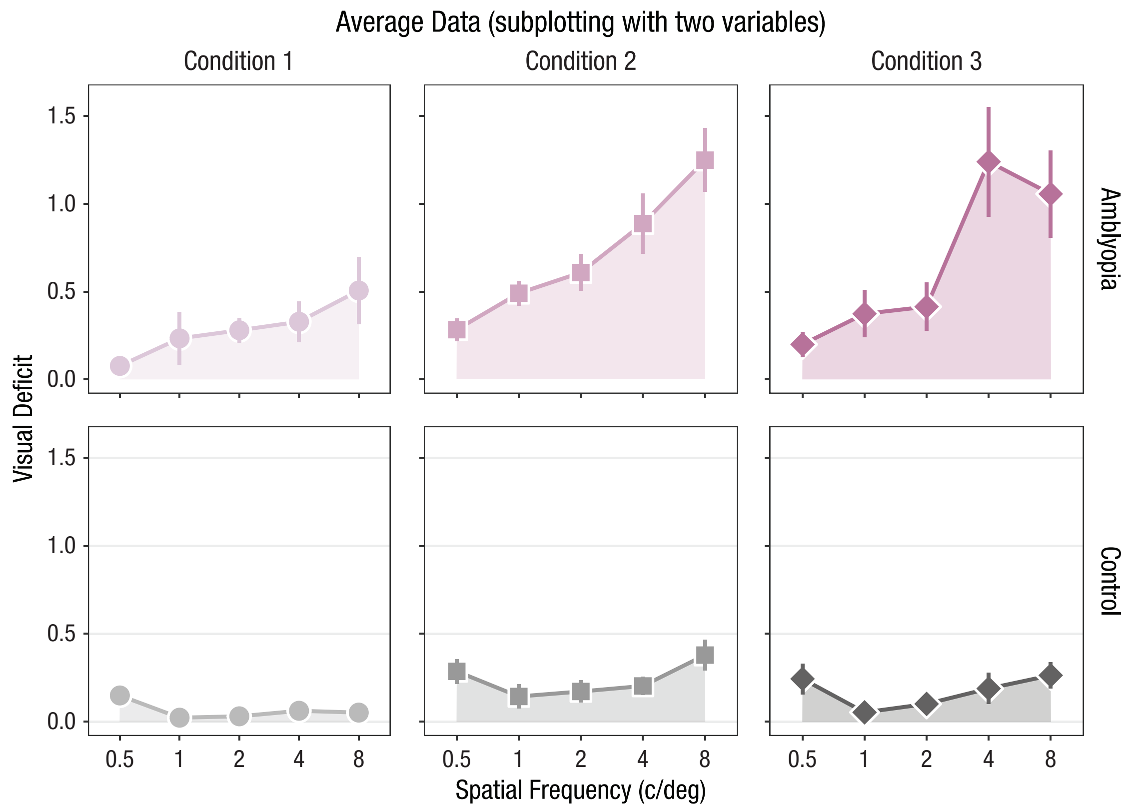


Fig. 7. A composite plot with two rows and three columns, showing the average data from each condition and group with error bars (standard error). The first row shows data of the amblyopia group, whereas the second row shows the data of the normal group, as specified in the `lapply()` function. The main and secondary titles have been added as annotations.

As in Example 1, the `lapply()` function must contain two parts. The first part filters for the data of interest, which are average data at each group and condition. The filtered data are stored in the object `currData`. Then, the second part of the function plots the data from `currData`. Specifically, the average data are plotted as points using `geom_point()`, whereas the associated standard error values are drawn in vertical lines using `geom_linerange()` (as explained in Example 1). Here, I use an additional function from the *ggplot2* package: `geom_area()`, which plots area (i.e., filled line plots). The function essentially fills the area below the lines of the plot with colors. Coloring the area is useful to illustrate the magnitude of the data. In this example, I make it transparent to some extent by setting `alpha = 0.3`; if `alpha = 1`, the colored area will be opaque. Furthermore, the *x*- and *y*-axes limits are set to be identical for all panels using `scale_x_continuous()` and `scale_y_continuous()` functions because the panels will get combined into one composite figure with shared tick labels. Finally, the theme is set to `sm_hgrid()` to optimize the aesthetics of each panel for subplotting:

```
avg_plots # A list of list: each element has
  three plots
```

Notice that in this example, the object `avg_plots` is a *list of list*, where each element is a list containing three plots. It has a length of 3 even if it stores six plots in total. However, the function `sm_put_together` still recognizes it as a list with six elements (i.e., plots) because the function automatically flattens each element if the element is a list itself. There is no need to manually reorganize the structure of the object `avg_plots` for the function `sm_put_together()` to operate. It is important to be aware that the order of the plots that will be used in the composite figure from `sm_put_together` is `avg_plots[[1]][[1]]`, `avg_plots[[1]][[2]]`, `avg_plots[[1]][[3]]`, `avg_plots[[2]][[1]]`, `avg_plots[[2]][[2]]`, and `avg_plots[[2]][[3]]`. If one were to subplot these panels in a 2 × 3 figure, then three plots from `avg_plots[[1]]` will be on the first row, whereas three plots from `avg_plots[[2]]` will be on the second row.

Next, I set *y*-axis label of the combined figure as in Example 1 by directly providing character strings in `sm_put_together()`. However, for the `xlabel`, I use the output created from `sm_common_xlabel()`, demonstrating that both options of labeling the axes can work in concert. Here, the argument `wRatio` controls the width of the leftmost column to those of other columns. The value exceeds the value of 1 because the panels in the leftmost column have *y*-axis ticks, capturing additional plotting space. If an input for this argument is missing, the function by default adjusts a width ratio using the information about the composite plot, such as the number of lines and characters in the tick labels. The argument `ylabel2` has an input of an empty string because if the input is supplied in any form (even when it is empty), some space will be spared on the right side of the composite plot (area for labels of secondary *y*-axis), where users will add labels of the two subject groups. As previously noted, `labelRatio` affects only axis labels that are created directly from `sm_put_together()`, so it will adjust the size of `ylabel` but not `xlabel`:

```
# Figure 7 - Combine the subplots and specify
  the layout
xlabel <- sm_common_xlabel("Spatial
  frequency (c/deg)", x = 0.52)
avg_plots_tgd <- sm_put_together(
  all_plots = avg_plots,
  title = "", # Spare space for title
  xlabel = xlabel,
  ylabel = "Visual deficit",
  ylabel2 = "", # Spare space for group
    label
  ncol = 3, nrow = 2, wRatio = 1.1,
    wmargin = -2, hmargin = -2,
  labelRatio = 0.95 # Text size of the
    ylabel
)
```

In this example, notice that I also did not supply a character string for the main title (`title` argument) of the combined plot in `sm_common_title()`. By putting an empty string, I merely allocated some space for the title at the top of the figure, where I will add text annotations using `sm_add_text()`. I can set the coordinate of the title to be at the center of the *x*-axis and top along the *y*-axis (`x = .55, y = .98`, where 0.5 represents the origin of the combined figure) and its `fontface` to be `bold`. The text annotation itself can be defined using the `label` argument within `sm_add_text()`.

As for the group labels, one can also use `sm_add_text()` to denote the two subject groups by setting the orientations of the texts at 270° relative to the horizontal axis using the `angle` argument. Position them on the right side of the composite figure by setting `x = 0.93`.

Next, because I have assigned data to multiple panels using two variables (groups and conditions), it leaves one more variable (`Condition`) to label in the composite plot. Here, I can add a subtitle at the top of each column where I label each condition (as shown in Fig. 7) using `sm_add_text()`. When using `sm_add_*()` functions, the coordinate is uniform regardless of the size of the composite plot output that is generated from `sm_put_together()` or `sm_add_legend()` (0 to 1;

`x = 0.5, y = 0.5` is the center); essentially, the annotations can be added to the composite figure similarly as to how `geom` objects can be added together to form a *ggplot2* object with a common coordinate:

```
# Figure 7 - Add text annotations
avg_plots_tgd1 <-
avg_plots_tgd + # Composite plot
  sm_add_text(
  label = "Average data (subplotting with
    two variables)", # Main title
  x = 0.53, y = 0.98, fontface = "bold",
    size = 17
 ) +
  sm_add_text(label = "Condition 1", x =
    0.25, y = 0.92, size = 14) + # Sub-
    title for Column 1
  sm_add_text(label = "Condition 2", x =
    0.51, y = 0.92, size = 14) + # Sub-
    title for Column 2
  sm_add_text(label = "Condition 3", x =
    .78, y = .92, size = 14) + # Sub-title
    for Column 3
  sm_add_text(label = "Control", x = 0.93,
    y = 0.335, angle = 270, size = 15) + #
    Group label
  sm_add_text(label = "Amblyopia", x =
    0.93, y = 0.705, angle = 270, size =
    15) # Group label
```

The final figure is stored in the object `avg_plots_tgd1`, which then gets saved as an image using the `ggsave()` function from the *ggplot2* package:

```
# Figure 7 - Save the final output as a vec-
  tor file
ggsave("avg_together.pdf", avg_plots_tgd1,
  width = 9.6, # inches
  height = 6.4
)
```

Example 3: Complex Subplotting Using Separate `lapply()` Functions

In this example, using the data frames `df2_amb` and `df2_avg` from the previous examples, I create a composite figure that plots the data of individuals in the `Amblyopia` group in a slightly more complex way that is not currently possible with *ggplot2* or its third-party packages that enhance its faceting functions. This time, I allocate the data from each condition of each individual to a unique panel and plot the average data for each condition on a unique panel:

```
# Figure 8 - Generate three subplots for
  each subject
subj_list <- paste0("A", 1:9) # 9 subjects
cond_list <- c("One", "Two", "Three")
shape_list <- c(21, 22, 23)
cond_cList <- c("#ddc7d8", "#d3a7c0",
  "#b7729a")

indv_plots <- lapply(1:length(subj_list),
  function(iSubj) {
lapply(1:length(cond_list),
  function(iCond) {
# First part: Filter data for each subject
  & condition during each iteration
subj_data <- df2_amb %>%
filter(Subject == subj_list[iSubj]) %>%
filter(Condition == cond_list[iCond])

# Second part: Plot the filtered data
  ggplot(data = subj_data, aes(x = logSF,
    y = absBP)) +
  geom_area(fill = cond_cList[[iCond]],
    alpha = 0.3) +
  geom_line(linewidth = 1, color = cond_
    cList[[iCond]]) +
  geom_point(
  size = 5, shape = shape_list[[iCond]],
  color = "transparent", fill =
    cond_cList[[iCond]]
 ) +
  sm_hgrid() +
  scale_y_continuous(limits = c(0, 3)) +
  scale_x_continuous(
  limits = c(-1.3, 3.3),
  labels = c(0.5, 1, 2, 4, 8)
 ) +
  annotate("text",
  label = paste0("A", iSubj), x = -0.9,
    y = 2.65, size = 5.5,
  hjust = 0
)

 })
})
```

As the title for this example implies, I create two separate `lapply()` functions to build a composite figure. With the data frame `df2_amb`, the first function will iterate through each subject's data at each condition, creating 27 plots (9 subjects × 3 conditions). With the data frame `df2_avg`, another `lapply()` function will iterate through the average data at each condition, generating three plots (three conditions). These outputs will then be combined

and stored in a single object, which will then be used as input by the layout function `sm_put_together()` to create a composite plot of 30 subplots, all of which will have the same *x*- and *y*-axes limits.

To generate a plot for each subject at each condition using `df2_amb`, a nested `lapply()` structure should be used, with one inner function and one outer function (as described in Example 2). Data can be filtered similarly as in Example 2 during each iteration. The structure of the nested functions will determine the order in which the figure outputs will be created and stored in the output (i.e., `indv_plots`). In this example, the first three outputs will be plots using data from the first element of `subj_list` and all three elements from `cond_list` because the latter has been used to iterate through the inner `lapply()` function. Aesthetics can also be dynamically controlled using the programmatic approach. For example, different shapes and colors can be set to represent each condition, as specified by the order of objects `shape_list` and `cond_cList`. The figures that are generated from this `lapply()` function are stored in the `indv_plots` object.

In Example 1, I annotated each panel with the subject's identifier using the function `sm_panel_label()` because I had forgotten to include codes that add panel label inside the nested `lapply()` code fragment (Figs. 5 and 6). Here, I use an alternative method; the annotation label on each panel is defined within the `lapply()` code fragment that generates the individual panels in sequence. Specifically, this can be performed using the function `annotate()`, which is from the *ggplot2* package. Specify its annotation type as `text` for the first input and the `label` as each subject's identifier by concatenating the string `A` with the index of the subject during each iteration (`iSubj`) with the function `paste0()`. The coordinates of `x` and `y` are set in the units of the data that are plotted so that the label annotations are on the top left of each panel. The argument `hjust` aligns the text to the left because it is set as 0. If `hjust` is set to 1, the text label will be aligned to the right. After writing the codes, readers can check if the `indv_plots` object correctly stores each subject's plot at each condition with the panel label of each subject's identifier (e.g., A1 in the first plot of `indv_plots`):

```
# Figure 8 - Generate a subplot for each
  condition's average across subjects
avg_plots_amblyopia <-
  lapply(1:length(cond_list),
  function(iCond) {
# First part: Filter average data for each
  condition
  currData <- df2_avg %>%
  filter(Group == "Amblyopia") %>%
  filter(Condition == cond_list[iCond])

# Second part: plot the filtered data
  ggplot(data = currData, aes(x = logSF, y
    = avgBP)) +
  geom_area(fill = cond_cList[[iCond]],
    alpha = 0.25) +
  geom_line(linewidth = 1, color = cond_
    cList[[iCond]]) +
  geom_point(
  size = 5, shape = shape_list[[iCond]],
    color = "white",
  fill = cond_cList[[iCond]], stroke = 1
 ) +
  geom_linerange(aes(ymin = avgBP -
    stdErr, ymax = avgBP + stdErr),
  linewidth = 1, color =
    cond_cList[[iCond]]
 ) +
  sm_hgrid() +
  scale_y_continuous(limits = c(0, 3)) +
  scale_x_continuous(
  limits = c(-1.3, 3.3),
  labels = c(0.5, 1, 2, 4, 8)
 ) +
  annotate("text",
  label = "Average", x = -0.9, y = 2.65,
    size = 5.5,
  hjust = 0, fontface = "bold"
 )
})
```

Next, construct codes that generate plots using the average data (from `df2_avg`) at each condition. This requires a single `lapply()` structure, looping through each condition. Data can be filtered similarly as in the previous examples. For the average panels, I establish the aesthetics so that the points have white border lines, thereby accentuating the error bars. In addition, I add annotations with the bolded text `'Average'.` There will be three iterations total from this `lapply()` function. The order of the figure outputs will follow the order of the elements in `cond_list`. The three output figures are stored in the object `avg_plots_amblyopia`.

In the `lapply()` function, as that in Example 2, the average data are plotted as points using `geom_point()`, the lines that connect the points are drawn using `geom_line()`, the areas below the lines are filled with colors and some transparency using `geom_area()`, and the range of standard error across subjects is displayed in vertical lines using `geom_linerange()`. The ticks and limits of both *x*- and *y*-axes are set to be consistent across panels:

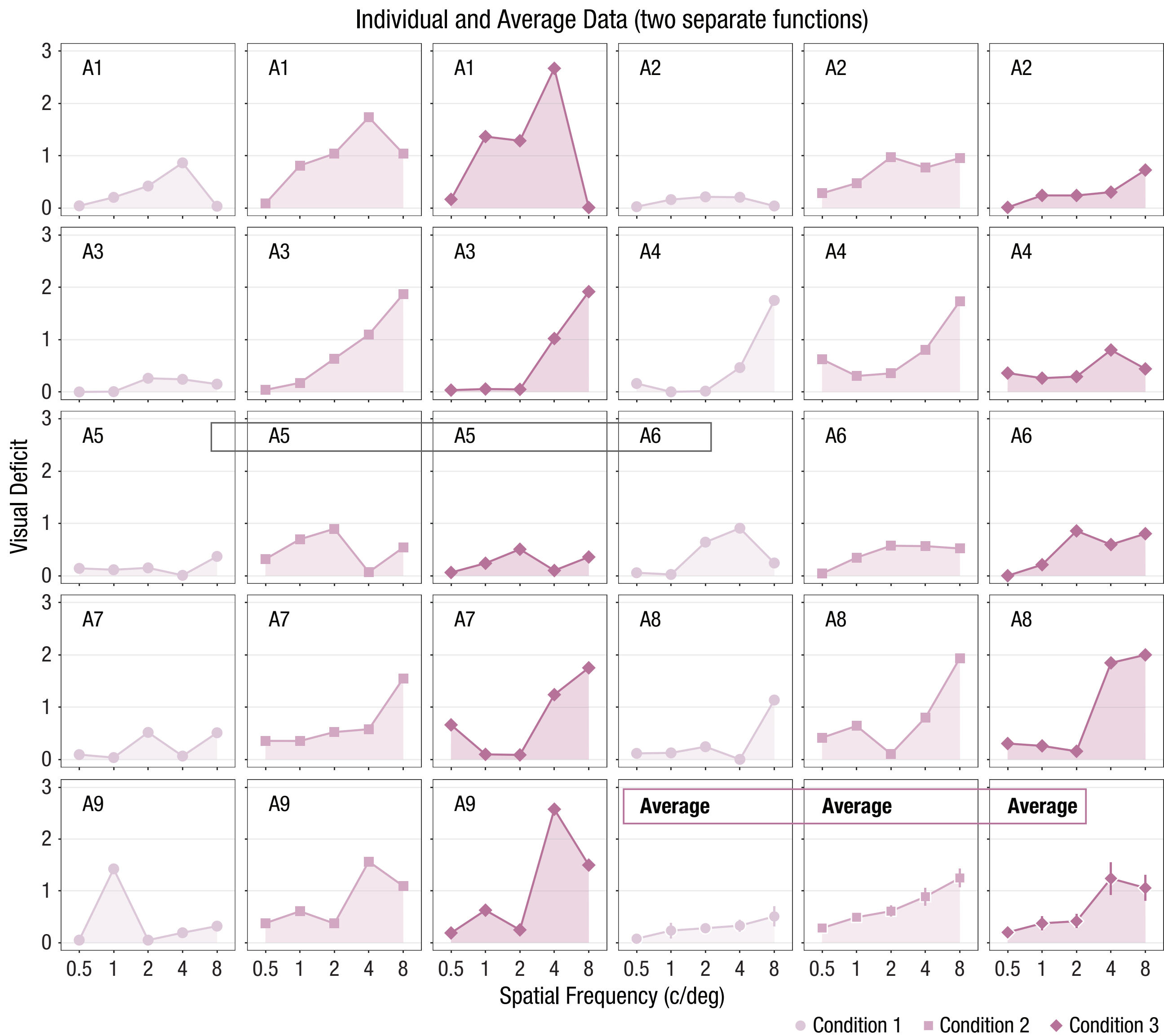


Fig. 8. A composite plot with five rows and six columns. Twenty-seven panels display each subject's data from each condition, and the last three panels show the average data with error bars, which denote standard errors. A borderless legend is placed at the bottom-right corner of the composite figure.

```
# Figure 8 - Put together all subplots
  from individual and average data
all_plots1 <- list(indv_plots, avg_plots_
  amblyopia) # Combine all plot outputs in
  a list

composite_plot <- sm_put_together(
  all_plots = all_plots1,
  title = "Individual and average data
    (two separate functions)",
  xlabel = "Spatial frequency (c/deg)",
  ylabel = "Visual deficit",
  ncol = 6, nrow = 5, wmargin = -5,
    hmargin = -5,
  labelRatio = 0.95 # Text size of the
    axes' label
)
```

I then combine two objects (`indv_plots` and `avg_plots_amblyopia`) from the two `lapply()` structures into a single object (`all_plots1`) using the function `list()`. The object `all_plots1` will then be used as input for `sm_put_together()`. Notice that because the `indv_plots` list has been generated from a nested `lapply()` structure, each of the nine elements in the list contains three plots (hence, 27 plots total). Conversely, `avg_plots_amblyopia` is from a single `lapply()` function, so there are three elements in the list, and each

element stores one plot (hence, three plots total). In other words, these two lists have different underlying structures. However, this is not an issue because `sm_put_together()` will automatically flatten different structures of list (e.g., list of list) into a uniform list structure, thereby making it easier for users to use the function when they have used multiple, separate `lapply()` structures to generate numerous subplots.

Thirty panels that have been generated with the two separate `lapply()` iterations are combined using `sm_put_together()`, where `ncol = 6, nrow = 5` are set as the layout of the composite figure. In addition, the main title, *x*-axis label, and *y*-axis label are all defined and then integrated into its composite form. Next, create a custom legend using `sm_common_legend()`, as has been done in the previous examples. This legend is set to have a `horizontal` orientation with no border (`border = FALSE`). The legend is coded so that the `fill` and `shape` aesthetics are mapped to each level of `Condition`. Then, one can add it to the object `composite_plot` using the function `sm_add_legend()` at the bottom-right corner of the composite plot (`x=0.85, y=0.065` of `composite_plot`):

```
# Figure 8 - Make a legend
  legend3 <- ggplot(data = df2_amb, aes(
  x = logSF, y = absBP, group = Condition,
  fill = Condition, shape = Condition
)) +
  geom_point(size = 5, color = "white") +
  scale_fill_manual(
  values = cond_cList,
  labels = c("Condition 1 ", "Condition 2
    ", "Condition 3 ")
) +
  scale_shape_manual(
  values = c(21, 22, 23),
  labels = c("Condition 1 ", "Condition 2
    ", "Condition 3 ")
) +
  sm_common_legend(
  direction = "horizontal", border =
    FALSE,
  textRatio = 1.2, legend_spacing = 0.9
)

composite_plot2 <- sm_add_legend(combined_
  plot = composite_plot, x = 0.85, y =
  0.065,
  legend = legend3)
```

Finally, users can add other types of annotations (besides `sm_add_text()` and `sm_add_point()`) using *ggplot2* functions directly. Here, I add two rectangles to the composite plot using the function `annotate()`. The coordinate system works similarly to how `sm_add_*()` functions work; when x and y are 0.5, annotations are drawn at the origin of the composite plot. The `annotate()` function requires inputs for some arguments. The first input is the type of `geom`, which has to be written as `'rect'` to draw a rectangle on the plot; the coordinates of the rectangle are specified with `xmin`, `xmax`, `ymin`, and `ymax` arguments, all of which should be from 0 to 1. Their border `color` and `fill`ing color can also be specified. In this example, both of these rectangles have no filling color (`fill = NA`) but have different border colors at different locations (set with `x` and `y` inputs). They span areas of multiple subplots, demonstrating that users have full control over aesthetics. Outputs from `sm_put_together()` and `sm_add_legend()` are treated as a single layer of *ggplot2* with a normalized coordinate from 0 to 1, so users can also use functions from third-party packages to perform particular types of annotations:

```
# Figure 8 - Add annotations of shapes
composite_plot2b <- composite_plot2 + #
  Composite plot with legend
  annotate("rect", # Rectangle 1
  xmin = 0.23, xmax = 0.63, ymin = 0.54,
    ymax = 0.57, fill = NA,
  color = "#636262", linewidth = 0.8
 ) +
  annotate("rect", # Rectangle 2
  xmin = 0.56, xmax = 0.93, ymin = 0.22,
    ymax = 0.25,
  fill = NA, color = "#b7729a", linewidth
    = 0.8
 )
```

Finally, the final figure is stored in the object `composite_plot2b`, which is then saved as an image using `ggsave()` with defined `width` and `height` of the composite figure (Fig. 8):

```
# Figure 8 - Save the composite graph as a
  vector file
ggsave("composite_plot.pdf",
  composite_plot2b,
  width = 18, # inches
  height = 16.5
)
```

Through these examples, I have shown that the workflow for complex data visualization in *ggplot2* can be structurally linear, with its clear beginning and resolution. In addition, the examples have illustrated that the limitations of how users can allocate different subsets of data to distinct subplots and dynamically control the aesthetics are not determined by what *ggplot2* and its third-party

Table 2. Contributions of *smplot2*

Feature	*ggplot2* without *smplot2*	*smplot2*	*matplotlib*
1. Learning curve	Moderate	Flat	Steep
2. Shortcut functions for drawing various types of plots	Yes	Yes	Yes
3. Composite plots (from multiple *ggplot2* objects)	Yes	Yes	Yes
4. Programmatic approach for plotting	Not ideal	Ideal	Ideal
5. Control for the aesthetics of the combined figure	Some	Yes	Yes
6. Legend addition to the combined figure	Some	Yes	Yes
7. Annotations in the combined figure	Some	Yes	Yes
8. Linear process of subplotting	No	Yes	Yes

Note: Although there are many other functionalities in *ggplot2* and third-party packages, these are not mentioned here because they are not relevant to *smplot2*. This table applies to instances when multiple *ggplot2* objects (outputs) are combined into one composite plot rather than when a faceted plot is generated as a single *ggplot2* object.

packages are capable of but by their own ability to apply the programmatic approach using the `lapply()` function. I hope that this will empower readers to programmatically perform subplotting in creative and limitless ways. For the summary of the tutorial, see Table 1.

Discussion

In this tutorial, I have demonstrated how *smplot2* can improve the user experience for data visualization using *ggplot2* in coding both standalone and composite plots. Specifically, the package can be useful for both beginners who wish to visualize their data with elegant aesthetics and advanced users who wish to structure their workflow for drawing composite figures with programmatic approaches and extend their level of customization. In the long term, the package can provide users a flexible and programmatic approach of plotting data that could yield more diverse, expressive, and powerful visualizations across different fields, including psychology and human neuroscience.

Key advantages of smplot2

The *smplot2* package can provide benefits to both entry-level and advanced R users.

To begin with, a major advantage of *smplot2* for incoming users, as noted by a recent review from a group of clinicians (Gandhi et al., 2024), is that it flattens the learning curve of *ggplot2* (Item 1 in Table 2). The visualization functions are flexible, and their aesthetics have been optimized for the general format of scientific journals (Min & Zhou, 2021). More than 300 reproducible examples are provided in the documentation page (https://smin95.github.io/dataviz), so users can freely use and modify these codes for their own purposes. In addition, the codes of the package have been reviewed for quality and stability across different computing systems by CRAN. Some of the functions that users from eclectic fields and levels of experience have used are raincloud plots (Chen et al., 2023; Gómez-Robles et al., 2024), regression analyses (Hamad et al., 2024; Ilyés et al., 2024), and forest plots (Grobler & Kramer, 2023) in both standalone and composite forms.

For users with working knowledge of R and *ggplot2*, *smplot2* has potential to affect how they perform complex and sophisticated data visualizations. Specifically, it provides key functions for them to integrate the practices of data visualization using *ggplot2* and the programmatic approach because *smplot2* overcomes the limited flexibility of aesthetics at the level of composite figures in *ggplot2*. That is, it provides a complete, flexible, and linear workflow for combining multiple *ggplot2* outputs into a composite plot. It also integrates the programmatic approach, which can generate multiple *ggplot2* outputs, into the visualization pipeline by handling different (nested) structures of list objects from `lapply()` functions or other methods (e.g., `map()`) that are compatible with *ggplot2*. Furthermore, it enables users to adjust marginal space and annotate both within and across subplots in any form after a composite plot has been constructed, encouraging users to apply the programmatic approach rather than to create each plot separately. Therefore, it will motivate users to search for solutions within their scripts rather than find third-party packages that resolve issues in plotting. In sum, the package has potential to empower users by allowing them to create more customizable, dynamic, and expressive figures; promoting the reproducibility of complex visualization routines; and linearizing the workflow of visualizing a composite plot.

Numerous packages, such as *ggfortify* (Tang et al., 2016), *ggstatsplot* (Patil, 2021), and *GGally* (Schloerke et al., 2021), have been developed to allow users to easily plot data using different types of graphs in a few lines of codes (shortcut functions; Item 2 in Table 2), thereby extending the functionalities of *ggplot2* and flattening the steep learning curve for beginners. There are also packages, such as *grid*, *patchwork*, and *gridExtra*, that

provide functions for users to create composite figures in *ggplot2* in various layouts from combining discrete *ggplot2* objects (Item 3 in Table 2). This approach has been widely used so that users can achieve a maximum flexibility of aesthetics of the combined figure (see Fig. 1). Nevertheless, they do not offer significant versatility for users after subplots (multiple *ggplot2* objects) have been combined into a composite plot, thereby encouraging users to create plots separately and shirking away from applying programmatic practices (Item 4 in Table 2). For instance, after multiple *ggplot2* objects are combined into one form, controlling the positions of legends and annotations and extent of margin between subplots in the combined figure becomes more difficult (Items 5–7 in Table 2), a task that can be easily performed in Python's *matplotlib*. This has made users implement practices that go against the principles of open science, such as using a vector graphics software (e.g., Adobe Illustrator) to annotate the final figure generated from R. These restrictions can now be lifted with *smplot2*, which linearizes the workflow for complex data visualizations (Item 8 in Table 2) and elevates the level of customization for aesthetics in situations in which users want to stitch multiple *ggplot2* objects together to construct a composite plot.

R or Python?

The dispute about which of *ggplot2* and *matplotlib* is better for data visualization has been ongoing for some time (Ozgur et al., 2017). A well-known plotting package that complements *matplotlib*, *seaborn* (Waskom, 2021) has captivated a wide user base in Python with its beautiful aesthetics and shortcut functions for plotting. The two libraries (*seaborn* and *matplotlib*) embrace the programmatic approach, requiring users to apply iterations and conditional statements to plot data. Although this steepens the learning curve for users, it increases flexibility for aesthetics, allowing users to dynamically control each component of the figure. A comparable library with *matplotlib* in R is *ggplot2*, which is convenient for plotting different types of graphs without the requirement for users to understand concepts of programming, such as loops and functional methods, primarily because of its layered approach. This simplicity and the fact that *ggplot2* can generally reproduce figures from *matplotlib* with fewer lines of code have expanded its user base rapidly (see Fig. 1). However, this layered approach comes at a cost because it hinders users from controlling the aesthetics using the programmatic approach. Although *ggplot2* can be superior in many aspects of visualizations to *matplotlib*, notably for concisely plotting different types of graphs with its declarative syntax, its design can complicate the workflow for users when it comes to subplotting and creating composite figures, leaving Python's *matplotlib* slightly more suitable for performing complex visualizations (for their comparisons, see Table 2).

Throughout the tutorial, I have compared Python's *matplotlib* and R's *ggplot2* closely to demonstrate that the gap between *ggplot2* and *matplotlib* has been minimized with *smplot2* in the realms of subplotting and flexibility. With the arrival of *smplot2*, it is now possible to linearize the process of subplotting with its clear starting and ending points because the package integrates the interface of *ggplot2* and the programmatic approach.

Why use the programmatic approach?

So far, the programmatic approach has not been ideal in *ggplot2* because it creates various *ggplot2* objects that need to be joined together using other packages. Unfortunately, the level of aesthetic control decreases steeply from when a plot is built as a single *ggplot2* output to when multiple outputs are combined into a composite figure, encouraging users to generate each subplot separately. However, in this tutorial, I have demonstrated the efficiency of the programmatic approach with three examples using *smplot2*.

I support this plotting method for several reasons. First, complex data visualizations, such as composite plots, can be performed concisely. Second, it increases code readability and reproducibility because the pipeline remains very similar regardless of the number of variables or `lapply()` functions. Third, it lifts aesthetic limitations of composite plots by integrating the native R programming practices with *ggplot2*'s declarative syntax.

For example, users can create a complex composite plot, such as a lower triangular matrix form, without relying on external packages. They can use `lapply()` function to create empty panels in specific panels and then combine them using `sm_put_together()`. The panels will be arranged in a triangular layout. To create an empty plot, users can type `ggplot(NULL) + sm_common_legend()` (see examples in Chapter 7 of the documentation webpage), which has no aesthetic mappings (no legend). After creating a composite plot with `sm_put_together()`, users can also add annotations in any forms at any coordinates within the composite figure using the layered approach of *ggplot2*. In summary, *smplot2*'s programmatic approach with `lapply()` can empower users to perform sophisticated visualizations with *ggplot2*.

Closing remarks

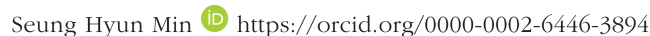
In this tutorial, I have introduced *smplot2*, an R package that provides a structured workflow for plotting by integrating a programmatic approach and visualization and

layout functions for advanced data visualizations. The defaults of the plots generated by the package are simple and minimalistic and have also been optimized for subplotting so that individual components of the figure are still clearly visible in a composite plot. In addition, the functions introduce a linear process of creating a composite figure by giving users full control of aesthetics at multiple stages of plotting in *ggplot2*. I hope that the package can encourage more users to use R as part of their visualization routines.

Transparency

Action Editor: Pamela Davis-Kean
Editor: David A. Sbarra
Author Contribution
Seung Hyun Min: Conceptualization; Software; Visualization; Investigation; Writing - Original Draft; Writing - Review and Editing; Funding Acquisition.
Declaration of Conflicting Interests
The author(s) declared that there were no conflicts of interest with respect to the authorship or the publication of this article.
Funding
This work was supported by a National Natural Science Foundation of China grant (No. 32350410414) and a National Foreign Expert Project (No. QN2022016002L) fund.
Open Practices
This article has received the badge for Open Materials. More information about the Open Practices badges can be found at http://www.psychologicalscience.org/publications/badges.

ORCID iD


Seung Hyun Min https://orcid.org/0000-0002-6446-3894

Acknowledgments

I thank *smplot2* users who have given feedback and raised issues about the package since its inception. I am also grateful to Mengting Chen, Chenyan Zhou, and Shiqi Zhou, who tested the package numerous times during its development.

References

Baker, D. H., Wallis, S. A., Georgeson, M. A., & Meese, T. S. (2012). Nonlinearities in the binocular combination of luminance and contrast. *Vision Research*, *56*, 1–9.

Chen, Y., Chen, Y., Tao, C., Zhou, S., Chen, H., Huang, P.-C., Hess, R. F., & Zhou, J. (2023). Temporal synchrony discrimination is abnormal in dichoptic but not monocular visual processing in treated anisometropic amblyopes. *Ophthalmic and Physiological Optics*, *43*(2), 263–272. https://doi.org/10.1111/opo.13090

Gandhi, M. A., Tripathy, S. P., Pawale, S. S., & Bhawalkar, J. S. (2024). A narrative review with a step-by-step guide to R software for clinicians: Navigating medical data analysis in cancer research. *Cancer Research, Statistics, and Treatment*, *7*(1), 91–99. https://doi.org/10.4103/crst.crst_313_23

Gómez-Robles, A., Nicolaou, C., Smaers, J. B., & Sherwood, C. C. (2024). The evolution of human altriciality and brain development in comparative context. *Nature Ecology & Evolution*, *8*(1), 133–146.

Grobler, B., & Kramer, M. (2023). The acute effects of schoolbag loading on posture and gait mechanics in 10-to 13-year-old children: A cohort from the North West province. *Children*, *10*(9), Article 1497. https://doi.org/10.3390/children10091497

Hamad, M. J., Alcaraz, P. E., & de Villarreal, E. S. (2024). Effects of combined uphill–downhill sprinting versus resisted sprinting methods on sprint performance: A systematic review and meta-analysis. *Sports Medicine*, *54*(1), 185–202.

Hehman, E., & Xie, S. Y. (2021). Doing better data visualization. *Advances in Methods and Practices in Psychological Science*, *4*(4). https://doi.org/10.1177/25152459211045334

Hunter, J. D. (2007). matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95.

Ilyés, A., Paulik, B., & Keresztes, A. (2024). Discrimination of semantically similar verbal memory traces is affected in healthy aging. *Scientific Reports*, *14*(1), Article 17971. https://doi.org/10.1038/s41598-024-68380-0

Kubilius, J. (2014). A framework for streamlining research workflow in neuroscience and psychology. *Frontiers in Neuroinformatics*, *7*, Article 52. https://doi.org/10.3389/fninf.2013.00052

Lin, Z., & Lu, S. (2023). Exponential authorship inflation in neuroscience and psychology from the 1950s to the 2020s. *American Psychologist*. Advance online publication. https://doi.org/10.1037/amp0001216

Min, S. H., Chen, Y., Jiang, N., He, Z., Zhou, J., & Hess, R. F. (2022). Issues revisited: Shifts in binocular balance depend on the deprivation duration in normal and amblyopic adults. *Ophthalmology and Therapy*, *11*(6), 2027–2044.

Min, S. H., & Zhou, J. (2021). smplot: An R package for easy and elegant data visualization. *Frontiers in Genetics*, *12*, Article 802894. https://doi.org/10.3389/fgene.2021.802894

Mowinckel, A. M., & Vidal-Piñeiro, D. (2020). Visualization of brain statistics with R packages ggseg and ggseg3d. *Advances in Methods and Practices in Psychological Science*, *3*(4), 466–483.

Nordmann, E., McAleer, P., Toivo, W., Paterson, H., & DeBruine, L. M. (2022). Data visualization using R for researchers who do not use R. *Advances in Methods and Practices in Psychological Science*, *5*(2). https://doi.org/10.1177/25152459221074654

Ozgur, C., Colliau, T., Rogers, G., & Hughes, Z. (2017). MatLab vs. Python vs. R. *Journal of Data Science*, *15*(3), 355–371.

Patil, I. (2021). Visualizations with statistical details: The 'ggstatsplot' approach. *Journal of Open Source Software*, *6*(61), Article 3167. https://doi.org/10.21105/joss.03167

Pedersen, T. L. (2019). *Package 'patchwork'* [R package]. https://cran.r-project.org/web/packages/patchwork

R Core Team. (2021). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Schloerke, B., Cook, D., Larmarange, J., Briatte, F., Marbach, M., Thoen, E., Elberg, A., & Crowley, J. (2021). *GGally:*

Extension to 'ggplot2' [R package Version 2.2.1]. https://github.com/ggobi/ggally

Tang, Y., Horikoshi, M., & Li, W. (2016). ggfortify: Unified interface to visualize statistical results of popular R packages. *The R Journal*, *8*(2), 474–485.

Waskom, M. L. (2021). seaborn: Statistical data visualization. *Journal of Open Source Software*, *6*(60), Article 3021. https://doi.org/10.21105/joss.03021

Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. Springer.

Wickham, H. (2016). *ggplot2: Elegant graphics for data analysis*. Springer-Verlag. https://ggplot2.tidyverse.org

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., . . . Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), Article 1686. https://doi.org/10.21105/joss.01686

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science*. O'Reilly Media, Inc.

Wilke, C. O. (2019). *Cowplot: Streamlined plot theme and plot annotations for 'ggplot2'*. https://CRAN.R-project.org/package=cowplot