ggalign: Bridging the Grammar of Graphics and Complex layout

Yun Peng

2024-11-24

Table of contents

Preface							
I	Ва	sics	7				
1	Intro	Introduction					
	1.1	Installation	8				
	1.2	General design	8				
	1.3	Align axes in ggplot	11				
	1.4	Getting Started	11				
2	stac		18				
	2.1	Input data	19				
	2.2	σ	20				
	2.3		26				
	2.4		32				
	2.5	active plot	34				
3	heatmap layout 3						
	3.1	input data	38				
	3.2	Main plot (heatmap body)	38				
	3.3		43				
	3.4		47				
	3.5	0 ,	51				
	3.6	1 1 1 = 1 1 1 1 1 2	52				
	3.7	1 = 0,7	53				
	3.8		55				
			55				
		3.8.2 Annotation Stack Size	57				
4	Layout customize						
	4.1	0 -0 -1 0	62				
	4.2	0 = "	63				
	4.3	8 = 4 4 4 4	64				
	4.4	O = ""	70				
	4.5	align_reorder()	78				

5	Plot	initialize	81
	5.1	ggalign()	81
	5.2	ggfree()	85
		5.2.1 Cross panel sumamry	88
	5.3	ggwrap() and inset()	91
	5.4	align_dendro()	92
	5.5	Plot titles	98
6	Anno	otate observations	101
	6.1	Links	101
	6.2	ggmark()	
7	anad	I-layout	114
•	7.1	Introduction	
	7.2	Annotations	
	7.3	quad_discrete()	
	7.4	quad_continuous()	
	7.5	quad_layout()	
	•••	· - v	
8		1 · · · = · · V · · · · · · ·	125
	8.1	Add quad_layout() to stack_layout()	
	8.2	Control sizes	131
9	circle	e layout	135
	9.1	radial	136
	9.2	spacing	139
10	Once	rators	141
10	-	Addition operator	
		logical AND operator	
		Subtraction operator	
	10.5	10.3.1 quad_layout()	
		10.3.2 stack_layout()	
	10 4	with_quad()	
	10.1	10.4.1 quad_layout()	
		10.4.2 stack_layout()	
11	Sche		1 - 1
11			154 154
			$154 \\ 159$
		-	
	11.5	scheme_align()	
		11.3.1 guides	
		11.3.3 free_labs	
		11.0.0 HED 1aus	1 I Z

12	Diffe	erence with ggplot2	174
	12.1	Position Scales	174
		12.1.1 breaks	174
		12.1.2 labels	181
	12.2	theme	187
	12.3	Facets	188
13	Plot	Composer	190
	13.1	Plot Assembly	190
	13.2	Empty area	192
	13.3	Controlling the grid	193
	13.4	Guide legends	195
	13.5	free_guide	196
П	Ad	vanced	199
14	ggnr	ncoplot	200
		Input data	
		oncoPrint Customization	
		Advanced Data Handling	
		Integration with maftools	
		Specialized Geoms	
		14.5.1 geom_subtile()	
		14.5.2 geom_draw2()	
	14.6	ggalign attributes	
		Integration with GISTIC results from maftools	
111	Cas	ses	218
15	Com	nplete examples	219
13			219
			_
	10.2	15.2.1 Based on dendrogram	
		15.2.2 Based on kmeans	220
			221
		15.2.4 Based on an ordering weights	222
	15.3		223
		Multiple heatmaps	$\frac{225}{225}$
	10.4	15.4.1 Horizontal layout	$\frac{225}{225}$
		·	226
	15.5	marginal plots	

IV	Co	mplexHeatmap	229				
16	6 A Single Heatmap						
	16.1	Colors	232				
	16.2	Titles	243				
	16.3	Clustering	244				
		16.3.1 Distance methods	244				
		16.3.2 Clustering methods	246				
		16.3.3 Render dendrograms					
	16.4	Set row and column orders	249				
	16.5	Seriation	250				
		Dimension labels					
	16.7	Heatmap split	256				
		16.7.1 Split by k-means clustering	256				
		16.7.2 Split by categorical variables					
		16.7.3 Split by dendrogram					
		16.7.4 Order of slices (panels)					
		16.7.5 Titles for splitting (facet strip text)					
		16.7.6 Graphic parameters for splitting					
17	More	e examples	266				
	17.1	Add more information for gene expression matrix	266				
		The measles vaccine heatmap					

Preface

Welcome to ggalign documents. Examples in the book are generated under version 0.0.5.9000.

In the world of data visualization, aligning multiple plots in a coherent and organized layout is often a challenging task, especially when dealing with complex datasets that require precise alignment across rows, columns, and even within plot elements. While existing tools provide some solutions, they often fall short in offering the flexibility, control, and simplicity that users need to create intricate and beautiful plots. This is where ggalign comes in.

The ggalign package, built on top of the powerful ggplot2 framework, is designed to solve this very problem. It offers a suite of functions specifically crafted for aligning and organizing plots with minimal effort. Whether you need to align observations based on statistical measures, group plots by categorical factors, or fine-tune the layout to match the precise needs of your data, ggalign gives you the tools you need to create polished, publication-ready visualizations.

This book serves as both an introduction to the ggalign package and a comprehensive guide to mastering its features. Whether you're a beginner or an experienced user of ggplot2, you'll find detailed explanations, step-by-step tutorials, and real-world examples to help you leverage the full potential of ggalign in your work.

Throughout this book, we will cover everything from basic concepts to advanced layout customizations, focusing on key functions like stack_layout(), align_* series (including align_group(), align_order(), and align_hclust()), and how to combine them with other ggplot2 layers to create aligned plots. Additionally, you'll learn how to adapt ggalign for different data types and scenarios, allowing you to develop flexible, dynamic visualizations tailored to your specific needs.

By the end of this book, you will be equipped to use ggalign effectively in your own projects, whether for scientific research, data analysis, or any other field where data visualization is key. Our goal is to provide you with the knowledge and confidence to tackle complex visualization challenges and transform your datasets into clear, impactful, and visually appealing plots.

Thank you for choosing ggalign. We hope this book will inspire you to explore the endless possibilities that come with aligned data visualization.

Part I

Basics

1 Introduction

ggalign extends ggplot2 by providing advanced tools for aligning and organizing multiple plots, particularly those that automatically reorder observations, such as dendrogram. It offers fine control over layout adjustment and plot annotations, enabling you to create complex visualizations while leveraging the familiar grammar of ggplot2.

1.1 Installation

You can install ggalign from CRAN using:

```
install.packages("ggalign")
```

Alternatively, install the development version from r-universe with:

```
install.packages("ggalign",
    repos = c("https://yunuuuu.r-universe.dev", "https://cloud.r-project.org")
)
```

or from GitHub with:

```
# install.packages("remotes")
remotes::install_github("Yunuuuu/ggalign")
```

1.2 General design

The core feature of ggalign lies in its integration of the grammar of graphics into advanced visualization through its object-oriented Layout system. The package provides three main Layout classes:

- the StackLayout class: Put plots horizontally or vertically.
- the QuadLayout class: Arranges plots in the four quadrants (top, left, bottom, right) around a main plot. This layout is ideal for designs that require supplementary plots or annotations surrounding a central figure.

• the CircleLayout class: Positions plots in a circular arrangement.

Each Layout class supports the alignment of both discrete (ordinal) and continuous variables. Depending on the desired alignment across multiple plots within a layout, the following variants are available:

For StackLayout:

- stack_discrete(): Align discrete variable along the stack.
- stack_continuous(): Align continuous variable along the stack.

stack_layout() integrates the functionalities of stack_discrete() and stack_continuous()
into a single interface. For simplicity, throughout this book, I will refer to both variants as
stack_layout().

For QuadLayout:

- quad_continuous/ggside: Align continuous variables in both horizontal and vertical directions.
- quad_discrete: Align discrete variables in both horizontal and vertical directions.
- quad_layout: Aligns discrete or continuous variables either horizontally or vertically.

The functions quad_continuous() and quad_discrete() are specialized versions of quad_layout() (Section 7.5): quad_continuous() sets both xlim and ylim arguments, while quad_discrete() does not set xlim or ylim. For simplicity, throughout this book, I will refer to all of the variants as quad_layout().

For CircleLayout:

- circle_discrete(): Align discrete variable in the circle.
- circle_continuous(): Align continuous variable in the circle.

circle_layout() integrates the functionalities of circle_discrete() and circle_continuous() into a single interface. For simplicity, throughout this book, I will refer to both variants as circle_layout().



Figure 1.1: General design of ggalign

1.3 Align axes in ggplot

Before introducing the package, let's first discuss how to align axes in ggplot2. For continuous axes, alignment is straightforward because we just need to ensure that all plots have the same limits. This can be achieved in the *_continuous() function by setting the limits (or xlim/ylim) argument.

Aligning discrete axes is more challenging because we must first ensure that the number of unique discrete values is the same across all plots, and that the ordering is consistent. In ggplot2, it's difficult to align discrete axes, since some plots may self-ordered. This issue is compounded when working with long-formatted data frames, which is required by ggplot2.

To address this, the ggalign package typically uses matrix input for layouts that require aligning discrete axes (*_discrete() functions). Each row of the matrix represents a unique discrete value (observation), and the number of rows corresponds to the total number of unique values (number of observations). To ensure consistent ordering across plots, the internal just need to reorder the rows. This approach is also well-suited for functions like quad_layout()/ggheatmap(), where axes need to be aligned in both directions simultaneously, as matrices can be transposed (i.e., switching rows to columns and columns to rows). When drawing the plot, the ggalign package will melt the matrix into a long-formatted data frame. You may worry about that the matrix format may not fully support all input requirements for plots, the ggalign package provides a function (scheme_data()) to transform the plot data as needed to fit your specific use case.

We often refer to "observations" when discussing discrete axes. Observations typically correspond to discrete variables, such as data points or samples. In the rest of the book, I will use "observations" and "discrete variables" interchangeably. You can assume that any reference to "observations" also applies to discrete variables.

1.4 Getting Started

```
library(ggalign)
#> Loading required package: ggplot2
```

The usage of ggalign is simple if you're familiar with ggplot2 syntax, the typical workflow includes:

- 1. Initialize the layout.
- 2. Customize the layout with:
 - align group(): Group observations into panel with a group variable.
 - align_kmeans(): Group observations into panel by kmeans.

- align_order(): Reorder layout observations based on statistical weights or by manually specifying the observation index.
- align_reorder: Reorder observations using an arbitrary statistical function
- align_hclust(): Reorder or group observations based on hierarchical clustering.

3. Adding plots with:

- align_dendro(): Add a dendrogram to the plot, and reorder or group observations based on hierarchical clustering.
- ggalign(): Initialize a ggplot object and align the axes.
- ggmark(): Add a plot to annotate selected observations.
- ggcross(): Initialize a ggplot object to connect two different layout crosswise
- ggfree(): Initialize a ggplot object without aligning the axes.
- 4. Layer additional ggplot2 elements such as geoms, stats, or scales.

```
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

Every *_layout() function accepts default data, which will be inherited by all plots within the layout.

Here's a simple example:

```
stack_discretev(small_mat) +
   align_dendro() +
   theme(axis.text.x = element_text())
   3
```

- (1) We initialize a vertical stack.
- (2) Reorder the observations based on hierarchical clustering and add a dendrogram tree.
- (3) Add x-axis text.



This function produces a simple dendrogram. The stack_discretev() function initializes a vertical stack and aligns discrete variables. It is simply an alias for stack_discrete("v"). By default, the axis text on the axis used for alignment is removed. This is because it's often unclear which plot should display the axis text, as typically, we want it to appear in only one plot. However, you can easily use the theme() function to control where the axis text appears.

Internally, align_dendro() will reorder the observations based on the dendrogram, and other plots in the layout will follow this ordering.

```
stack_discretev(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(.discrete_x, value), stat = "identity") +
    theme(axis.text.x = element_text())
5
```

- (1) We initialize a vertical stack.
- (2) Reorder the observations based on hierarchical clustering and add a dendrogram tree.
- (3) Create a new ggplot in the layout, and use data based on the sum of the layout data.
- (4) Add a bar layer.
- (5) Add x-axis text.



The data in the underlying ggplot object of ggalign() function contains following columns (more details will be introduced in the Section 5.1):

- .panel: the group panel for the aligned axis. It means x-axis for vertical stack layout, y-axis for horizontal stack layout.
- .x/.y and .discrete_x/.discrete_y: an integer index of x/y coordinates and a factor of the data labels (only applicable when names exists).
- .names and .index: A character names (only applicable when names exists) and an integer of index of the original data.
- value: the actual value (only applicable if data is a matrix or atomic vector).

It is recommended to use .x/.y, or .discrete_x/.discrete_y as the x/y mapping.

align_dendro() can also split the observations into groups.

```
stack_alignv(small_mat) +
   align_dendro(k = 3) +
   ggalign(data = rowSums) +
   geom_bar(aes(.discrete_x, value, fill = .panel), stat = "identity") +
   scale_fill_brewer(palette = "Dark2", name = "Group") +
   theme(axis.text.x = element_text())

6
```

1) We initialize a vertical stack.

- (2) Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree.
- (3) Create a new ggplot in the layout, and use data based on the sum of the layout data.
- 4 Add a bar layer.
- **5** Set fill scale palette.
- (6) Add x-axis text.



One common visualization associated with the dendrogram is the heatmap. You can use ggheatmap() to initialize a heatmap layout. When grouping the observations using $align_dendro(k = 3)$, a special column named branch is added, which you can use to color the dendrogram tree.

```
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = branch), k = 3) +
    scale_fill_brewer(palette = "Dark2")

#> > heatmap built with `geom_tile()`
```

- (1) We initialize a heatmap layout.
- (2) adjust the x-axis label theme element.
- (3) we initialize an annotation in the left side of the heatmap body, and set it as the active context, in this way, all following addition will be directed to the left annotation.

- (4) Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree, coloring the tree by branch.
- (5) Set fill scale palette.



ggheatmap() will automatically add axis text in the heatmap body, so you don't need to manually adjust axis text visibility using theme(axis.text.x = element_text())/theme(axis.text.y
= element_text()).

We can also arrange the dendrogram in a circular layout to visualize hierarchical relationships in a more compact and aesthetically pleasing way.

- 1 We initialize a circle layout and set the inner radius.
- (2) Create a new ggplot in the layout, and use data the same with the layout data.
- 3 Add a tile layer, the matrix input will be converted into a long formated data frame with column .column_index indicates the column index of the original matrix.
- (4) Set fill scale palette.

- (5) Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree, coloring the tree by branch.
- (6) Set color scale palette.



Having explored the core principles of ggalign, you should now be familiar with its basic workflow. In the next chapter, we'll introduce the StackLayout, a powerful tool for arranging multiple plots in a stacked fashion—either horizontally or vertically—while maintaining full control over their alignment. We'll explore how StackLayout and its various functions can give you even greater flexibility in creating sophisticated layouts.

2 stack layout

stack_layout() arranges plots either horizontally or vertically. Based on whether we want to align the discrete or continuous variables, there are two types of stack layouts:

- stack_discrete(): Align discrete variable along the stack.
- stack_continuous(): Align continuous variable along the stack.

stack_layout() integrates the functionalities of stack_discrete() and stack_continuous()
into a single interface. The first argument for these three functions is direction which
should be a single string indicating the direction of the stack layout, either "h"(horizontal)
or "v"(vertical).

Several aliases are available for convenience:

- stack_discretev: A special case of stack_discrete that sets direction = "v".
- stack discreteh: A special case of stack discrete that sets direction = "h".
- stack_continuousv(): A special case of stack_continuous that sets direction = "v".
- stack_continuoush(): A special case of stack_continuous that sets direction = "h".



```
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

2.1 Input data

As discussed in Section 1.3, when aligning discrete variables, we typically use a matrix. For continuous axes, we can still use the long-formatted data frame, which is the same as in ggplot2.

- For stack_continuous(), a data frame is required, and the input will be automatically converted using fortify_data_frame() if needed.
- For stack_discrete(), a matrix is required, and the input will be automatically converted using fortify_matrix() if needed.

By default, fortify_data_frame() will invoke the ggplot2::fortify() function for conversion. Note, for matrix, it will be converted to a long-formatted data frame which is different from the ggplot2::fortify().

stack_discrete()/stack_continuous() will set up the layout, but no plot will be drawn
until you add a plot element:

```
stack_discretev(small_mat) +
    layout_annotation(
        theme = theme(plot.background = element_rect(color = "red"))
)
# the same for `stack_continuous()`
```

- (1) initialize a vertical stack layout.
- (2) Add a plot background in for the entire layout.



In this example, we use layout_annotation() to insert a plot background in the entire layout, it can also be used to control the theme of title, subtitle, caption (layout_title()), guides, margins, panel.border, panel.spacing.

2.2 Layout Customize

When aligning discrete variables, the package offers a suite of align_* functions designed to give you precise control over the ordering and groups. Instead of detailing each align_* function individually, we will focus on the general usage and how to combine them with stack_discrete().

Here, we remain take align_dendro() as an example, it can reorder the observations, split them into groups, and can add a plot for visualization.

When used for stack_discreteh(), the observations are aligned along the y-axis:

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.



When used for stack_discretev(), the observations are aligned along the x-axis:

- 1 initialize a vertical stack layout.
- 2 reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.



When align_dendro() is added to the layout, it performs following actions:

- 1. reorder the observations.
- 2. set the active plot to the dendrogram.

The active plot refers to the plot that subsequent ggplot2 components will target. In this case, the active plot is the dendrogram, and any new layers added will be applied to it. For instance, we can add additional layers to visualize the dendrogram's structure or data. The default data underlying the ggplot object of align_dendro() consists of the dendrogram node data. It is also possible to use the dendrogram's edge data for customization, which I will introduce in Section 5.4.

```
stack_discreteh(small_mat) +
   align_dendro() +
   geom_point()

①
3
```

- 1 initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.
- (3) add a point layer to the dendrogram



The active argument controls whether a plot should be set as the active plot. It accepts an active() object with the use argument to specify if the plot should be active when added.

```
stack_discreteh(small_mat) +
    align_dendro(active = active(use = FALSE)) +
    geom_point()

#> Error in `chain_layout_add()`:

#> ! Cannot add `geom_point()` to the horizontal `stack_discrete()`

#> i No active plot component

#> i Did you forget to initialize a <ggplot> object with `ggalign()` or

*> `ggfree()`?
```

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, but don't set the active plot to this plot.
- (3) try to add a point layer to the dendrogram, should fail due to no active plot

Usually, you don't need to set this manually, as the active context is automatically applied only for functions that add plot areas. You can inspect whether a align_* function will add a plot by print it:

```
align_dendro()
#> <Class: AlignDendro AlignHclust Align AlignProto>
```

```
#> plot: yes
#> reorder: yes
#> split: no
```

You might find it confusing that we mentioned align_dendro() will split observations into groups, while the print output shows split = "no". This happens because we haven't specified the k/h argument in align_dendro().

```
align_dendro(k = 3L)
#> <Class: AlignDendro AlignHclust Align AlignProto>
#> plot: yes
#> reorder: yes
#> split: yes
```

You don't need to explicitly provide data to align_dendro(). By default, it inherits data from the layout. However, you can always provide another data source, but note that this package uses the concept of number of observations (NROW()). When aligning the observations, you must ensure the number of observations is consistent across all plots.

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on hierarchical clustering, add a dendrogram tree, and set the active plot to this one, using self-provided data. This should fail because the number of observations is inconsistent.
- (3) try to add y-axis text to the dendrogram.

```
set.seed(123)
stack_discreteh(small_mat) +
    align_dendro(data = matrix(rnorm(70), nrow = 7)) +
    theme(axis.text.y = element_text())
3
```

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot, using self-provided data
- (3) add y-axis text to the dendrogram.



Alternatively, you can provide a function (or purrr-lambda) that will be applied to the layout's matrix. For layouts that align observations, a matrix is always required, so the data input must be in matrix form.

```
set.seed(123)
stack_discreteh(small_mat) +
    align_dendro(data = ~ .x[sample(nrow(.x)), ]) +
    theme(axis.text.y = element_text())
    3
```

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot, using self-provided data function
- 3) add y-axis text to the dendrogram.



Without adding another plot, it's difficult to appreciate the benefits. Let's now explore how to incorporate a plot.

2.3 Plot initialize

There are two primary functions for adding plots:

- ggalign(): Initialize a ggplot object and align the axes.
- ggfree(): Initialize a ggplot object without aligning the axes.

Both functions initialize a ggplot object and, by default, set the active plot when added to the layout.

```
stack_discreteh(small_mat) +
   align_dendro() +
   ggalign(data = rowSums) +
   geom_bar(aes(value, .discrete_y), stat = "identity") +
   theme(axis.text.y = element_text())

①
```

- (1) initialize a horizontal stack layout.
- (2) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.

- (3) initialize a ggplot object, and set the active plot to this plot, using self-provided data function
- 4 add a bar to the plot
- (5) add y-axis text



You can build the plot separately and then add it to the layout:

```
my_bar <- ggalign(data = rowSums) +
    geom_bar(aes(value, .discrete_y), stat = "identity") +
    theme(axis.text.y = element_text())
stack_discreteh(small_mat) +
    align_dendro() +
    my_bar</pre>

①
```

- (1) Create the bar plot.
- (2) initialize a horizontal stack layout.
- (3) reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.
- (4) Add another bar plot to the layout.



The active argument can also control the place of the plot area to be added. It accepts an active() object with the order argument to specify the order of the plot area.

```
stack_discreteh(small_mat) +
    align_dendro() +
    ggalign(data = rowSums, active = active(order = 1)) +
    geom_bar(aes(value, .discrete_y), stat = "identity") +
    theme(axis.text.y = element_text())
```



You can also stack plots vertically using stack_discretev():

```
stack_discretev(small_mat) +
   align_dendro() +
   ggalign(data = rowSums) +
   geom_bar(aes(.discrete_x, value), stat = "identity") +
   theme(axis.text.y = element_text())
```



ggfree() focuses on layout integration without enforcing strict axis alignment.

```
stack_discretev() +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```



The & operator applies the added element to all plots in the layout, similar to its functionality in the patchwork package.

For stack_continuous(), only free plots (ggfree()) can be added. This layout arranges plots in one row or column without enforcing axis alignment:

```
stack_continuousv(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```



By default, ggfree() will also inherit data from the layout and call fortify_data_frame() to convert the data to a data frame. So, note that if the layout data is a matrix, it will be converted into a long-formatted data frame.

2.4 Plot Size

Both ggalign() and ggfree() functions have a size argument to control the relative width (for horizontal stack layout) or height (for vertical stack layout) of the plot's panel area.

```
stack_continuousv(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = 2) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = 1) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```



Alternatively, you can define an absolute size by using a unit() object:

```
stack_continuousv(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = unit(1, "cm")) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```



2.5 active plot

As mentioned earlier, the active plot refers to the plot that subsequent ggplot2 components will target. The package provide two functions to work with active plot for stack_layout.

- stack_switch(): switch the active context
- stack_active(): An alias for stack_switch(), which sets what = NULL

The stack_switch() function accepts the what argument, which can either be the index of the plot added (based on its adding order) or the plot name specified via the active() object using the name argument.

Note that the what argument must be explicitly named, as it is placed second in the function signature. This is because, in most cases, we don't need to switch the active plot, manually—adjusting the order of plot additions typically suffices.

```
stack_alignh(small_mat) +
   align_dendro() +
   ggalign(data = rowSums) +
   geom_bar(aes(value, .discrete_y), stat = "identity") +
   stack_switch(what = 1) +
   geom_point() +
```

```
theme(axis.text.y = element_text()) +
layout_title(title = "switch by integer")
```

switch by integer



```
stack_alignh(small_mat) +
    align_dendro(active = active(name = "tree")) +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .discrete_y), stat = "identity") +
    stack_switch(what = "tree") +
    geom_point() +
    theme(axis.text.y = element_text()) +
    layout_title(title = "switch by string")
```

switch by string



In the example, we use layout_title() to insert a title for the entire layout. Alternatively, you can add a title to a single plot with ggtitle().

By setting what = NULL (or alias stack_active()), we remove the active plot. This is particularly useful when the active plot is a nested Layout object, as any additions would otherwise be directed to that nested Layout. By removing the active plot, you can continue adding components to the StackLayout. We'll introduce the nested layout of StackLayout in Chapter 8.

In the next chapter, we will dive into the HeatmapLayout, which can take the StackLayout as input. Heatmap layouts offer additional features for aligning observations in both directions. Let's move ahead and explore how heatmaps can be seamlessly integrated into your layout workflows.

3 heatmap layout

The heatmap_layout() function provides a powerful way to create customizable heatmaps in R using ggplot2. This chapter will guide you through its usage.

heatmap_layout() is a specialized version of quad_discrete(), which itself is a specific variant of QuadLayout (quad_layout()) designed to align discrete variables both horizontally and vertically. We introduce heatmap_layout() directly, as it is more familiar to many users, especially those experienced with popular heatmap packages like pheatmap and ComplexHeatmap.



heatmap_layout() simplifies the creation of heatmap plots by integrating essential elements for a standard heatmap layout, ensuring that the appropriate data mapping and visualization layers are automatically applied. ggheatmap() is an alias for heatmap_layout().

```
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)</pre>
```

```
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

3.1 input data

As mentioned in Section 1.3, we typically require a matrix for the Layout which need align discrete variables. Internally, fortify_matrix() will be used to process the data. You can provide a numeric or character vector, a data frame, or any other data type that can be converted into a matrix using as.matrix().

```
ggheatmap(small_mat)
#> > heatmap built with `geom_tile()`
```



3.2 Main plot (heatmap body)

The ggheatmap()/quad_layout() functions arrange plots in the Quad-Side layout of the main plot. When the layout is initialized, a ggplot object is automatically created for the main plot.

For ggheatmap(), the matrix input will be converted into a long-format data frame when drawing. The data in the underlying ggplot object includes the following columns:

- .xpanel and .ypanel: the column and row panel
- .x and .y: the x and y coordinates
- .row_names and .column_names: A factor of the row and column names of the original matrix (only applicable when names exist).
- .row_index and .column_index: the row and column index of the original matrix.
- value: the actual matrix value.

The default mapping will use aes(.data\$.x, .data\$.y), but can be customized using mapping argument.

By default, the main plot is regarded as the active plot, meaning you can add ggplot2 elements directly to the main plot.

```
ggheatmap(small_mat) +
    geom_point() +
    scale_fill_viridis_c()
#> > heatmap built with `geom_tile()`
```



By default, ggheatmap() adds a heatmap layer. If the matrix has more than 20,000 cells (nrow * ncol > 20000), it uses geom_raster() for performance efficiency; for smaller matrices, geom_tile() is used. You can explicitly choose the layer by providing a single string ("raster" or "tile") in the filling argument.

ggheatmap(small_mat, filling = "raster")



ggheatmap(small_mat, filling = "tile")



Note, the filling layer will always use mapping of aes(.data\$.x, .data\$.y), if you want to customize filling, you can set filling = NULL, which will remove the filling layer and allow you to add custom filling geoms.

```
ggheatmap(small_mat, filling = NULL) +
  geom_tile(aes(fill = value), color = "black", width = 0.9, height = 0.9)
```



A heatmap pie charts can be easily drawn:

```
get.seed(123)
ggheatmap(matrix(runif(360L), nrow = 20L), filling = NULL) +
    geom_pie(aes(angle = value * 360, fill = value))
```



For more complex customizations of pie charts, you can try using ggforce::geom_arc_bar() instead.

3.3 rasterization

When working with large heatmaps, it's often beneficial to rasterize the heatmap body layer. You can achieve this by using the raster_magick() function. The res argument controls the resolution of the raster image. By default, the res argument matches the resolution of the current device, but specifying a different value can help reduce the resolution of the rasterized heatmap body.

```
ggheatmap(small_mat, aes(.x, .y), filling = NULL) +
  raster_magick(geom_tile(aes(fill = value)), res = 50)
```



By leveraging raster_magick(), you can also perform image post-processing using the magick package. This allows for custom image resizing with filters.



Note: When using magick::image_resize(), you should specify the geometry argument to resize the image. If only the filter is specified, it will only distort the image data (though subtle). For more information on image resizing, refer to ImageMagick's resize documentation.

You can also rasterize all plots in the layout directly with raster_magick(). This method is defined for both ggheatmap()/quad_layout() and stack_layout() objects.

Additionally, You can use external packages like ggrastr or ggfx to rasterize the heatmap body.

```
ggheatmap(small_mat, filling = FALSE) +
ggrastr::rasterise(geom_tile(aes(fill = value)), dev = "ragg")
```



Likewise, you can also rasterize all plots in the layout directly with ggrastr::rasterise() for both ggheatmap()/quad_layout() and stack_layout().

```
ggrastr::rasterise(ggheatmap(small_mat), dev = "ragg")
#> > heatmap built with `geom_tile()`
```



Furthermore, ggfx offers many image filters that can be applied to ggplot2 layers. See the package for the details.

3.4 annotations

In ggheatmap()/quad_layout(), annotations are handled by a stack_layout() object and can be positioned at the top, left, bottom, or right of the main plot (heatmap body). quad_layout() can align discrete or continuous variables either horizontally or vertically. Since the annotation handling is the same for both ggheatmap() and quad_layout(), we introduce the annotation functionality for both together.

The annotations will always follow the alignment of the ggheatmap()/quad_layout() in the current direction. For horizontal alignment of discrete variables, stack_discreteh() will be used for left and right annotations, while stack_continuoush() will be used for continuous variables. The same logic applies to vertical alignment—stack_discretev() or stack_continuousv() will be used for top and bottom annotations, depending on whether the variables being aligned are discrete or continuous.

By default, ggheatmap()/quad_layout() do not activate an annotation, You can use quad_anno() to activate an annotation, directing all subsequent additions to the specified annotation position. The quad_anno() function has the following aliases:

• anno_top: A special case of quad_anno() with position = "top".

- anno_left: A special case of quad_anno() with position = "left".
- anno_bottom: A special case of quad_anno() with position = "bottom".
- anno_right: A special case of quad_anno() with position = "right".

When quad_anno() is added to a ggheatmap()/quad_layout(), it will try to create a new stack_layout(). For top and bottom annotations, stack_discretev() or stack_continuousv() will be used; for left and right annotations, stack_discreteh() or stack_continuoush() will be applied.

Additionally, quad_anno() will set the active context to the annotation. This means that subsequent additions will be directed to the annotation rather than the main plot. We use the term active context in contrast to active plot (as discussed in Section 2.2), since the annotation is a Layout object but not a single plot.

```
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```



By default, the annotation stack_layout() will try to inherit data from ggheatmap()/quad_layout(). If we need align discrete variables vertically, this means the data from ggheatmap()/quad_layout() should be a matrix, the column annotations will also require a matrix and the matrix from ggheatmap()/quad_layout() will be transposed for use in the column annotations.

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    anno_top() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```



You can further customize the layout design or add new plots in the annotation stack, as described in Chapter 2.

```
ggheatmap(small_mat) +
    # in the heatmap body, we set the axis text theme
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    # we set the active context to the right annotation
    anno_right() +
    # in the right annotation, we add a dendrogram
    align_dendro(k = 3L) +
    # in the dendrogram, we add a point layer
    geom_point(aes(color = branch))
#> > heatmap built with `geom_tile()`
```



In this example:

- anno_right() initialize the right annotation stack, and change the active context to the right annotation of the heatmap.
- align_dendro(k = 3L) adds a dendrogram to the annotation and sets itself as the active plot in the annotation stack.
- geom_point(aes(color = branch)) is then added to this active plot within the annotation stack, here, it means the align_dendro() plot.

ggheatmap() aligns discrete variable both horizontally and vertically, so it's safe to always
use quad_anno() directly, as all annotations require a matrix, and the layout data is also a
matrix. However, for quad_layout(xlim = NULL) and quad_layout(ylim = NULL) (which
I'll discuss in more detail in a Chapter 7), which only align discrete variable in one direction,
the data in the layout may not fit the data for the annotation.

- `quad_layout(xlim = NULL)`: aligning discrete variable in horizontal direction, for column annotations, we ll need a data frame for `stack_continuous()`.
- `quad_layout(ylim = NULL)`: aligning discrete variable in vertical direction, for row annotations, we ll need a data frame for `stack_continuous()`.

In both cases, quad_anno() won't initialize the annotation by default, instead, you must provide the annotation manually.

3.5 Adding stack layout

To add a stack_layout() to the ggheatmap(), we must prevent the automatic creation of annotations by quad_anno() by setting initialize = FALSE.

```
my_stack <- stack_discreteh(small_mat) +
    align_dendro(aes(color = branch), k = 3L)

ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(initialize = FALSE) +
    my_stack
#> > heatmap built with `geom_tile()`
```



Note when aligning the discrete variable, you must ensure the number of observations is consistent in the direction. So for column annotations, you need transpose the data manually.

```
my_stack <- stack_discretev(t(small_mat)) +
    align_dendro(aes(color = branch), k = 3L)
ggheatmap(small_mat) +
    anno_top(initialize = FALSE) +
    my_stack
#> > heatmap built with `geom_tile()`
```



3.6 quad_active()

To remove the active context and redirect additions back to the heatmap body, you can use quad_active().

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    anno_top() +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    quad_active() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c()
#> > heatmap built with `geom_tile()`
```



3.7 quad_switch()/hmanno()

We also provide quad_switch()/hmanno() (heatmap annotation) which integrates quad_active() and quad_anno() into one function for ease of use. Feel free to use any of these functions to streamline your annotation process.

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    quad_switch("t") +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    quad_switch() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c() +
    layout_title("quad_switch()")
#> > heatmap built with `geom_tile()`
```

quad_switch()



```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    hmanno("t") +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    hmanno() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c()+
    layout_title("hmanno()")
#> > heatmap built with `geom_tile()`
```



3.8 Plot Size

3.8.1 Heatmap Body Size

You can specify the relative sizes of the heatmap body using the width and height arguments in the ggheatmap() function.

```
ggheatmap(small_mat, height = 2) +
   anno_top() +
   align_dendro()
#> > heatmap built with `geom_tile()`
```



Alternatively, the quad_active() function allows you to control the heatmap body sizes.

```
ggheatmap(small_mat) +
   quad_active(height = 2) +
   anno_top() +
   align_dendro()
#> > heatmap built with `geom_tile()`
```



3.8.2 Annotation Stack Size

The quad_anno() function allows you to control the total annotation stack size. The size argument controls the relative width (for left and right annotations) or height (for top and bottom annotations) of the whole annotation stack.

```
ggheatmap(small_mat) +
   anno_top(size = 1) +
   align_dendro()
#> > heatmap built with `geom_tile()`
```



You can also specify it as an absolute size using unit():

```
ggheatmap(small_mat) +
   anno_top(size = unit(30, "mm")) +
   align_dendro()
#> > heatmap built with `geom_tile()`
```



Note that the size of an individual plot (Section 2.4) does not affect the total annotation stack size. You must adjust the annotation size using the method described above.

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(size = unit(30, "mm")) +
   layout_title("plot size")
#> > heatmap built with `geom_tile()`
```

plot size



```
ggheatmap(small_mat) +
    anno_top(size = unit(30, "mm")) +
    align_dendro() +
    layout_title("annotation size")
#> > heatmap built with `geom_tile()`
```

annotation size



In this chapter, we explored the usage of heatmap layout. These features provide a strong foundation for visualizing matrix-based data in a structured way. However, as your visualization needs grow more complex, the ability to further customize and fine-tune the layout becomes essential.

In the next chapter, we will dive into the Layout Customize functionalities, where you can gain full control over your plot's layout.

4 Layout customize

For layouts that can align observations, the package provides a suite of align_* functions designed to give you precise control over the observations. These functions allow you to reorder observations or partition them into multiple groups.

Currently, there are four key align_* functions available for layout customization:

- align_group: Group and align plots based on categorical factors.
- align kmeans: Group observations by k-means clustering results.
- align_order: Reorder layout observations based on statistical weights or allows for manual reordering based on user-defined ordering index.
- align_reorder: Reorder observations using an arbitrary statistical function.
- align_hclust: Reorder or group observations based on hierarchical clustering.

Note that these functions do not add plot areas and set the active context, meaning you cannot incorporate ggplot2 elements directly into these objects. You can inspect the behavior of any align_* function by printing it.

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

4.1 align_group()

The align_group() function allows you to split the observations into groups.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE))
#> > heatmap built with `geom_tile()`
```



Note that all align_* functions which split observations into groups must not break the previous established groups. This means the new groups must nest in the old groups, usually they cannot be used if groups already exist.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE)) +
    align_group(sample(letters[1:5], ncol(small_mat), replace = TRUE))
#> Error in `setup_design()`:
#> ! `align_group()` disrupt the previously established panel groups of the
#> top annotation `stack_discrete()`
```

4.2 align_kmeans()

The align_kmeans() function split the observations into groups based on k-means clustering.

```
set.seed(1234)
ggheatmap(small_mat) +
  anno_top() +
```

```
align_kmeans(3L)
#> > heatmap built with `geom_tile()`
```



4.3 align_order()

The align_order() function reorder the observations based on the summary weights.

In this example, we order the rows based on their means. By default, the ordering is in ascending order according to the summary weights. You can reverse the order by setting reverse = TRUE.

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans) +
    layout_title(title = "reverse = FALSE")
#> > heatmap built with `geom_tile()`
```

reverse = FALSE



```
ggheatmap(small_mat) +
   anno_left() +
   align_order(rowMeans, reverse = TRUE) +
   layout_title(title = "reverse = TRUE")
#> > heatmap built with `geom_tile()`
```

reverse = TRUE



Additionally, you can provide the ordering integer index or character index directly:

```
set.seed(1234)
my_order <- sample(nrow(small_mat))
print(rownames(small_mat)[my_order])
#> [1] "row4" "row2" "row5" "row7" "row3" "row1" "row6"
```

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(my_order)
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat) +
   anno_left() +
   align_order(rownames(small_mat)[my_order])
#> > heatmap built with `geom_tile()`
```



Some align_* functions also accept a data argument. It's important to note that all align_* functions treat rows as the observations. This means NROW(data) must match the number of observations along the axis used for alignment. The data argument can also accept a function (supporting purrr-like lambda syntax), which will be applied to the layout matrix.

As mentioned in Section 3.4, for top and bottom annotations, the data matrix of quad_layout()/ggheatmap() is transposed to create the annotation stack_layout(). Therefore, you can use rowMeans() to calculate the mean value across all columns.

```
ggheatmap(small_mat) +
   anno_top() +
   align_order(rowMeans)
#> > heatmap built with `geom_tile()`
```



Some align_* functions that reorder observations include an argument called strict. This argument is especially useful when previous groups have already been established. If previous groups have been created and strict = FALSE, the function will reorder the observations within each group.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE))+
    align_order(rowMeans, strict = FALSE)
#> > heatmap built with `geom_tile()`
```



Note that we always prevent reordering the observations into two different orderings. If you want to apply two different orderings, you should use cross_layout() instead, which I'll introduce in a later chapter:

```
set.seed(1234)
another_order <- sample(ncol(small_mat))
ggheatmap(small_mat) +
    anno_top() +
    align_order(rowMeans) +
    align_order(another_order)
#> Error in `setup_design()`:
#> ! `align_order()` disrupt the previously established ordering index of
#> the top annotation `stack_discrete()`
```

4.4 align_hclust()

The align_hclust() function is designed to reorder observations and group them based on hierarchical clustering. Unlike align_dendro(), align_hclust() does not add a dendrogram tree to the plot. All the arguments intruduced here can also be used by align_dendro().

```
ggheatmap(small_mat) +
   anno_top() +
   align_hclust()
#> > heatmap built with `geom_tile()`
```



Hierarchical clustering is performed in two steps: calculate the distance matrix and apply clustering. You can use the distance and method argument to control the builind process.

There are two ways to specify distance metric for clustering:

- specify distance as a pre-defined option. The valid values are the supported methods in dist() function and coorelation coefficient "pearson", "spearman" and "kendall". The correlation distance is defined as 1 cor(x, y, method = distance).
- a self-defined function which calculates distance from a matrix.

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(distance = "pearson") +
    layout_title("pre-defined distance method (1 - pearson)")
#> > heatmap built with `geom_tile()`
```

pre-defined distance method (1 - pearson)



```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(distance = function(m) dist(m)) +
    layout_title("a function that calculates distance matrix")
#> > heatmap built with `geom_tile()`
```

a function that calculates distance matrix



Method to perform hierarchical clustering can be specified by method. Possible methods are those supported in hclust() function. And you can also provide a self-defined function, which accepts the distance object and return a hclust object.

```
ggheatmap(small_mat) +
   anno_top() +
   align_hclust(method = "ward.D2")
#> > heatmap built with `geom_tile()`
```



You can specify k or h to split observations into groups, which work similarly to cutree():

```
ggheatmap(small_mat) +
   anno_top() +
   align_hclust(k = 3L)
#> > heatmap built with `geom_tile()`
```



You can reorder the dendrogram based on the mean values of the observations by setting reorder_dendrogram = TRUE.

```
h1 <- ggheatmap(small_mat) +
        ggtitle("reorder_dendrogram = TRUE") +
        anno_top() +
        align_hclust(k = 3, reorder_dendrogram = TRUE)
h2 <- ggheatmap(small_mat) +
        ggtitle("reorder_dendrogram = FALSE") +
        anno_top() +
        align_hclust(k = 3)
align_plots(h1, h2)
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



In this example, we use align_plots() to arrange the layouts, which functions similarly to cowplot::align_plots() and patchwork::wrap_plots(). However, unlike those, align_plots() can be directly used with quad_layout() and stack_layout(), ensuring proper alignment by plot panel. Additionally, align_plots() can align pheatmap and ComplexHeatmap objects, though they won't align by panel area in the same way as ggplot2 plots.

align_hclust() can also perform clustering between groups, meaning it can be used even if there are existing groups present in the layout, in this way, you cannot specify k or h:

```
set.seed(3L)
column_groups <- sample(letters[1:3], ncol(small_mat), replace = TRUE)
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_hclust()
#> > heatmap built with `geom_tile()`
```



You can reorder the groups by setting reorder_group = TRUE, which reorders the hierarchical clusters based on the group tree:

```
ggheatmap(small_mat) +
   anno_top() +
   align_group(column_groups) +
   align_hclust(reorder_group = TRUE)
#> > heatmap built with `geom_tile()`
```



If you specify k or h, this will always turn off sub-clustering. The same principle applies to align_hclust(), where new groups must be nested within the previously established groups.

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_hclust(k = 2L)

#> Error in `setup_design()`:

#> ! `align_hclust()` disrupt the previously established panel groups of

#> the top annotation `stack_discrete()`
```

4.5 align_reorder()

The align_reorder() function enables the reordering of observations based on a specified statistical function. This function accepts a data argument and computes a statistic that determines the new ordering of the observations. The resulting order is extracted using the order2() function.

Foe example, you can also use hierarchical clustering to reorder the observations like this:

```
ggheatmap(small_mat) +
   anno_left() +
   align_reorder(hclust2)
#> > heatmap built with `geom_tile()`
```



Here, hclust2() is a special function used by align_hclust() to calculate the distance matrix and perform hierarchical clustering.

In addition to hierarchical clustering, the align_reorder() function can also handle objects from the seriation package. For example, you can use the seriate() function with a method like "BEA_TSP" to reorder the observations based on a specific seriation algorithm:

```
ggheatmap(small_mat) +
    anno_left() +
    align_reorder(seriation::seriate, method = "BEA_TSP", data = abs)
#> > heatmap built with `geom_tile()`
```



This approach gives you the flexibility to apply different statistical or algorithmic methods for reordering observations, which can be especially useful for exploring complex patterns in data.

With the align_* functions in place, we've covered how to reorder and group observations within a layout. Now, having set the stage for proper observation alignment, we can move on to the next essential step: plot initialize. In the following chapter, we'll dive deeper into how to set up your plots within the layout.

5 Plot initialize

There are following primary functions for adding plots:

- ggalign(): Create a ggplot object and aligns the axes.
- ggfree(): Create a ggplot object without alignment.
- align_dendro(): Create a ggplot object of dendrogram tree, and align the observation. In addition, it can reorder and group the observations.

All these functions will set the active plot when added to the layout.

Both ggmark() and align_dendro() can only for discrete variables.

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

5.1 ggalign()

ggalign is the package name but it's also a function in the package. ggalign() is similar to ggplot in that it initializes a ggplot data and mapping. ggalign() allowing you to provide data in various formats, including matrices, data frames, or simple vectors. By default, it will inherit from the layout. If a function, it will apply with the layout data.

The underlying plot data will be created using fortify_data_frame(), which, By default, calls ggplot2::fortify() to build the data frame. Additional methods have been added for atomic vector and matrix. For atomic vector, it will convert it to a data frame with following columns:

- .names: the names for the vector (only applicable if names exist).
- value: the actual value of the vector.

When data is a matrix, it will automatically be transformed into a long-form data frame, where each row represents a unique combination of matrix indices and their corresponding values. The resulting data frame will contain the following columns:

- .row_names and .row_index: the row names (only applicable when names exist) and an integer representing the row index of the original matrix.
- .column_names and .column_index: the column names (only applicable when names exist) and column index of the original matrix.
- value: the actual value.

When alignning continuous variables, ggalign() will apply the limits set by the layout when drawing.

```
ggheatmap(small_mat) +
    anno_top() +
    ggalign(data = rowSums) +
    geom_point(aes(y = value))
#> > heatmap built with `geom_tile()`
```



When aligning discrete variables, ggalign() always applies a default mapping for the axis of the data index in the layout. Specifically, This mapping is aes(y = .data\$.y) for horizontal stack (including left and right annotation) and aes(x = .data\$.x) for vertical stack (including top and bottom annotation).

The following columns will be added to the data frame to align discrete variables:

• .panel: The panel for the aligned axis. Refers to the x-axis for vertical stack_layout() (including top and bottom annotations), and the y-axis for horizontal stack_layout() (including left and right annotations).

- .names ([vec_names()][vctrs::vec_names]) and .index ([vec_size()][vctrs::vec_size()]/[NROW()]): Character names (if available) and the integer index of the original data.
- .x/.y and $.discrete_x/.discrete_y$: Integer indices for x/y coordinates, and a factor of the data labels (only applicable when names exist).

It is recommended to use .x/.y, or .discrete_x/.discrete_y as the x/y mapping.

When aligning discrete variables, almost all functions that add plots will generate data containing two key columns:

- .panel: used to create ggplot2 facets.
- .index: serves to match the data. It is useful for aligning observations or linking different data sources. You can use .index to merge data and create a new data frame, using method provided in Section 11.2.

Additionally, if names exist, the .names column will also be created.

```
stack_continuous("v", mtcars, limits = continuous_limits(c(2, 4))) +
    ggalign(mapping = aes(wt, mpg)) +
    geom_point() +
    ggalign(mapping = aes(wt, mpg)) +
    geom_point() +
    theme(axis.text.x = element_text())
```



When aligning discrete variables, ensure that the number of rows in the data matches the number of observations along the axis used for alignment:

```
my_df <- mtcars[seq_len(ncol(small_mat)), ]
ggheatmap(small_mat) +
    anno_top() +
    ggalign(data = my_df) +
    geom_point(aes(y = cyl))
#> > heatmap built with `geom_tile()`
```



If data = NULL, the data in the underlying ggplot object only contains following columns: .panel, .index, .names, .x/.y, .discrete_x/.discrete_y. You can use it to integrate additional elements, such as block annotation or customized panel title, into your layout.

```
ggheatmap(small_mat) +
   anno_top(size = unit(1, "cm")) +
   align_kmeans(centers = 3L) +
   ggalign(data = NULL) +
   geom_tile(aes(y = 1L, fill = .panel, color = .panel)) +
   theme_no_axes("y")
#> > heatmap built with `geom_tile()`
```



The function theme_no_axes() can be used to quickly remove axis text and titles from the plot.

5.2 ggfree()

The ggfree() function allows you to incorporate a ggplot object into your layout. Unlike ggalign(), which aligns every axis value precisely, ggfree() focuses on layout integration without enforcing strict axis alignment.

Internally, the function also uses fortify_data_frame() to transform the input into a data frame.

```
ggheatmap(small_mat) +
    anno_top() +
    ggfree(mtcars, aes(wt, mpg)) +
    geom_point()
#> > heatmap built with `geom_tile()`
```



Since ggfree() may use axes that are not aligned with the primary plot's axes, the axis titles will not be removed automatically. If you want to remove the axis titles, you must do so manually.

Alternatively, you can directly input the ggplot object.

```
ggheatmap(small_mat) +
   anno_top() +
   ggfree(ggplot(mtcars, aes(wt, mpg))) +
   geom_point()
#> > heatmap built with `geom_tile()`
```



You can also add the ggplot object directly without using ggfree(). However, doing so will limit control over the plot (like plot area size, and active components):

```
ggheatmap(small_mat) +
   anno_top() +
   ggplot(mtcars, aes(wt, mpg)) +
   geom_point()
#> > heatmap built with `geom_tile()`
```



5.2.1 Cross panel sumamry

When used in quad_layout()/ggheatmap(), if the data is inherited from the quad_layout() and the other direction aligns discrete variables, following columns will be added:

- .extra_panel: Provides the panel information for the column (left or right annotation) or row (top or bottom annotation).
- .extra_index: The index information for the column (left or right annotation) or row (top or bottom annotation).

This is useful if you want to create summary plot using another axis panel groups.

```
set.seed(1234)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +

# in the right annotation
    anno_right() +
    align_kmeans(2) +

# in the top annotation
    anno_top(size = 0.5) +
    ggfree() +
```

```
geom_boxplot(aes(.extra_panel, value, fill = .extra_panel)) +
    scale_fill_brewer(palette = "Dark2", name = "row groups") +
    theme_no_axes("x")
#> > heatmap built with `geom_tile()`
```



This approach replicates the functionality of ComplexHeatmap::anno_summary(), but is versatile enough to be used with any heatmap, not just single-column or single-row heatmaps.

5.3 ggwrap() and inset()

The ggwrap() function allows you to wrap objects that can be converted into a grob, turning them into a ggplot for plotting. Further you can still add ggplot elements like title, subtitle, tag, caption, and geoms using the same approach as with normal ggplots (using ggtitle(), labs(), geom_*()) as well as styling using theme(). This enables you to pass these wrapped objects into ggfree().

```
library(grid)
ggheatmap(small_mat) +
    anno_top() +
    # `ggwrap()` will create a `ggplot` object, we use `ggfree` to add it into the layout
    ggfree(data = ggwrap(rectGrob(gp = gpar(fill = "goldenrod")), align = "full"))
#> > heatmap built with `geom_tile()`
```



You can also integrate base plots, pheatmap, ComplexHeatmap, e.g.

Additionally, you can add any graphics as a inset to a ggplot using the inset() function. Like ggwrap(), inset() can accept any object that can be converted to a grob.

```
ggheatmap(small_mat) +
   anno_top() +
   ggfree(data = ggwrap(rectGrob(gp = gpar(fill = "goldenrod")), align = "full")) +
   # we can then add any inset grobs (the same as ggwrap, it can take any objects
   # which can be converted to a `grob`)
   inset(rectGrob(gp = gpar(fill = "steelblue")), align = "panel") +
   inset(textGrob("Here are some text", gp = gpar(color = "black")),
        align = "panel"
   )
   *> heatmap built with `geom_tile()`
```



5.4 align_dendro()

align_dendro() is an extension of align_hclust() that adds a dendrogram to the layout. All functions of align_hclust() introduced in Section 4.4 can be used with align_dendro(). Here, we focus on the plot-related function.

align_dendro() will initialize a ggplot object, the data underlying the ggplot object contains the dendrogram node data with dendrogram edge data attached in a special attribute.

dendrogram node and edge contains following columns:

- .panel: Tree branch groups, used to create ggplot2 facet.
- .names and .index: a character names (only applicable when names exists) and an integer index of the original data.
- label: node label text
- x and y: x-axis and y-axis coordinates for current node or the start node of the current edge.
- xend and yend: the x-axis and y-axis coordinates of the terminal node for current edge.
- branch: which branch current node or edge is. You can use this column to color different groups.
- leaf: A logical value indicates whether current node is a leaf.

By default, plot_dendrogram is set to TRUE, meaning a geom_segment() layer will be added using the dendrogram edge data when drawing. Note that this layer is always added to the first.

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro()
#> > heatmap built with `geom_tile()`
```



You can choose between two dendrogram types: "rectangle" (default) and "triangle". However, if there are any groups in the stack, "rectangle" will be used.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(type = "triangle")
#> > heatmap built with `geom_tile()`
```



You can also manually add the dendrogram tree using the edge data by setting plot_dendrogram = FALSE. In this case, you can access the dendrogram edge data with ggalign_attr(). The edge data is stored in the edge field of ggalign_attr():

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(plot_dendrogram = FALSE) +
   geom_segment(
       aes(x = .data$x, y = .data$y, xend = .data$xend, yend = .data$yend),
       data = function(x) ggalign_attr(x, "edge")
   )
#> > heatmap built with `geom_tile()`
```



When there are multiple groups, a branch column will be available. This can be used to color the nodes or edges based on the group source.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    geom_point(aes(color = branch))
#> > heatmap built with `geom_tile()`
```



align_dendro() will draw dendrogram tree in each group when previous group exists.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_kmeans(2) +
    align_dendro(aes(color = branch))
#> > heatmap built with `geom_tile()`
```



You can merge the sub-tree by settting merge_dendrogram = TRUE.

```
ggheatmap(small_mat) +
   anno_top() +
   align_kmeans(2) +
   align_dendro(aes(color = branch), merge_dendrogram = TRUE)
#> > heatmap built with `geom_tile()`
```



5.5 Plot titles

ggplot2 only allow add titles in the top or add caption in the bottom. we extends this capability, allowing you to place titles around any border of the plot using the patch_titles() function.

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(aes(color = branch), k = 3) +
   patch_titles(
      top = "top patch title",
      left = "left patch title",
      bottom = "bottom patch title",
      right = "right patch title"
   )
#> > heatmap built with `geom_tile()`
```



The appearance and alignment of these patch titles can be customized using ggplot2::theme():

- plot.patch_title/plot.patch_title.*: Controls the text appearance of patch titles. By default, plot.patch_title inherit from plot.title, and settings for each border will inherit from plot.patch_title, with the exception of the angle property, which is not inherited.
- plot.patch_title.position/plot.patch_title.position.*: Determines the alignment of the patch titles. By default, plot.patch_title.position inherit from plot.title.position, and settings for each border will inherit from plot.patch_title. The value "panel" aligns the patch titles with the plot panels. Setting this to "plot" aligns the patch title with the entire plot (excluding margins and plot tags).

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(aes(color = branch), k = 3) +
   patch_titles(
      top = "top patch title",
      left = "left patch title",
      bottom = "bottom patch title",
      right = "right patch title"
) +
   theme(
      plot.patch_title.position = "plot",
```

```
plot.patch_title = element_text(hjust = 0)
)
#> > heatmap built with `geom_tile()`
```



Now that you know the general approach to adding a plot, it would be useful to explore how to add annotation plots for specific observations. Let's move on to the next chapter.

6 Annotate observations

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

To add an annotation plot for specific observations, we must first know how to select the observations to be linked.

6.1 Links

The pair_links() function is used to define pairs of connected observations. In the formula, the left side (before the ~) represents hand1—the observations on the left (for a horizontal stack layout) or top (for a vertical stack layout). The right side (after the ~) represents hand2—the observations on the right (for horizontal stack) or bottom (for vertical stack).

These defined pairs will either be linked together, or each group in the pair will be linked separately within the same plot area.

Both sides of the formula can be specified using integer or character indices of the original data (before reordering).

The print method showed various informations:

- The object class and the number of paired groups.
- The specific groups in each pair.
- The total number of groups (rather than the number of paired groups). For example, in this case, there are 4 groups in total since each pair consists of 2 groups.

If only the left-hand side of the formula exists, you can input it directly.

```
pair_links(1:2, c("a", "b"))
#> <ggalign_pair_links>
#> A total of 2 pairs of link groups
#>
#> hand1 ~ hand2
#> 1: 1:2 ~
#> 2: c("a", "b") ~
#>
#> A total of 2 link groups
```

To specify links in the right only, you must use formula:

```
pair_links(~ 1:2, ~ c("a", "b"))
#> <ggalign_pair_links>
#> A total of 2 pairs of link groups
#>
#> hand1 ~ hand2
#> 1: ~ 1:2
#> 2: ~ c("a", "b")
#>
#> A total of 2 link groups
```

For integer indices, wrap them with I() to follow ordering from the layout.

```
pair_links(I(1:2))
#> <ggalign_pair_links>
#> A total of 1 pair of link groups
#>
#> hand1 ~ hand2
#> 1: I(1:2) ~
#>
#> A total of 1 link group
```

You can wrap the whole formula.

```
pair_links(I(1:2 ~ 3:4))
#> <ggalign_pair_links>
#> A total of 1 pair of link groups
#>
#> hand1 ~ hand2
#> 1: I(1:2) ~ I(3:4)
#>
#> A total of 2 link groups
```

range_link() function can be used to define a range of observations, which accepts two argument that specify the lower and upper bounds of the range. These bounds should be defined as a single integer or character.

```
pair_links(range_link(1, "a"))
#> <ggalign_pair_links>
#> A total of 1 pair of link groups
#>
#> hand1 ~ hand2
#> 1: range_link(1, "a") ~
#>
#> A total of 1 link group
```

In this case, the left-hand is defined as the range between the 1st observation and the observation named "a".

You can also use waiver() to inherit values from the opposite group.

```
pair_links(range_link(1, "a") ~ waiver())

#> <ggalign_pair_links>
#> A total of 1 pair of link groups
#>
#> hand1 ~ hand2
#> 1: range_link(1, "a") ~ waiver()
#>
#> A total of 2 link groups
```

(1) inherit values from the left hand

```
pair_links(waiver() ~ range_link(1, "a"))

#> <ggalign_pair_links>

#> A total of 1 pair of link groups

#> hand1 ~ hand2

#> 1: waiver() ~ range_link(1, "a")

#> #> A total of 2 link groups
```

(1) inherit values from the right hand

You can combine any of these into a list.

In this example, the left side combines the range between the 1st observation and "a" with the observations 4 and 5, and the observations "b" and "c".

6.2 ggmark()

ggmark() can be used to add annotation plot for the selected observations. ggmark accepts mark argument, which should be a mark_draw() object to define how to draw the links. Currently, two internal functions mark_line(), mark_tetragon() can be used to quickly draw line and quadrilateral links used to connect the selected observations and the plot panel.

By default, when no manual observations were selected, ggmark() will select all observations and split them based on the groups defined in the layout.

The data underlying the ggplot object generated by ggmark() is similar to that of ggalign() (Section 5.1), but it differs in that it does not include the .x, .y, and .discrete_x/.discrete_y columns. Instead, a special column named .hand is added, which is a factor with levels c("left", "right") for horizontal stack layouts or c("top", "bottom") for vertical stack layouts. This column indicates the position of the linked observations.

Note: Only data for selected observations are retained.

You can adjust the link size by using the plot.margin argument.

```
set.seed(123)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(hjust = 0, angle = -60)) +
    anno_right() +
    align_kmeans(3L) +
    ggmark(mark_line()) +
    geom_boxplot(aes(.names, value)) +
    theme(plot.margin = margin(l = 0.1, t = 0.1, unit = "npc"))
#> > heatmap built with `geom_tile()`
```



```
set.seed(123)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(hjust = 0, angle = -60)) +
    anno_right() +
    align_kmeans(3L) +
    ggmark(mark_tetragon()) +
    geom_boxplot(aes(.names, value)) +
    theme(plot.margin = margin(1 = 0.1, t = 0.1, unit = "npc"))
#> > heatmap built with `geom_tile()`
```



If you manually provide the linked observations, you can use the <code>group1</code> and <code>group2</code> arguments to control whether the layout panel groups and their ordering should be used to create the annotations.

```
set.seed(123)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(hjust = 0, angle = -60)) +
    anno_right() +
    align_kmeans(3L) +
    ggmark(mark_tetragon(1:3), group1 = TRUE) +
    geom_boxplot(aes(.names, value)) +
    theme(plot.margin = margin(1 = 0.1, t = 0.1, unit = "npc"))
#> > heatmap built with `geom_tile()`
```



By default, ggmark() uses facet_wrap to define the facet, and you can use it to control the facet apearance (just ignore the facets argument). We prefer facet_wrap() here because it offers flexibility in positioning the strip on any side of the panel, and typically, we only want to a single dimension to create the annotate the selected observations. However, you can still use facet_grid() to create a two-dimensional plot. Note that for horizontal stack layouts, the row facets, or for vertical stack layouts, the column facets will always be overwritten.

```
set.seed(123)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(hjust = 0, angle = -60)) +
    anno_right() +
    align_kmeans(3L) +
    ggmark(mark_line()) +
    geom_boxplot(aes(.names, value, fill = .names)) +
    facet_wrap(vars(), scales = "free", strip.position = "right") +
    theme(plot.margin = margin(l = 0.1, t = 0.1, unit = "npc"))
#> > heatmap built with `geom_tile()`
```



You can further customize the appearance of link lines and quadrilaterals using the element argument in mark_line()/mark_tetragon() function:

- Link lines can be customized using the element_line().
- Link ranges can be customized using the element_polygon().

By default, vectorized fields in element_line() and element_polygon() will be recycled to match the total number of groups.





You can wrap the element with I() to recycle it to match the drawing groups. The drawing groups typically correspond to the number of observations for element_line(), as each observation will be linked with the plot panel.



For element_polygon(), the drawing groups usually align with the defined groups. However, if the defined group of observations is separated and cannot be linked with a single quadrilateral, the number of drawing groups will be larger than the number of defined groups.



For stack_layout(), we usually don't need to specify the observations for hand2, since it should match hand1. This is because all plots in stack_discrete() should maintain the same ordering index. However, specifying hand2 becomes useful in stack_cross(), where different orderings are involved.

```
stack_discreteh(small_mat) +
    align_dendro(aes(color = branch), k = 3L) +
    scale_x_reverse(expand = expansion()) +
    theme(plot.margin = margin()) +
    ggmark(mark_line(4:6 ~ waiver(), 1:2 ~ waiver())) +
    geom_boxplot(aes(.names, value, fill = .names)) +
    theme(plot.margin = margin(l = 0.1, t = 0.1, r = 0.1, b = 0.1, unit = "npc")) +
    align_dendro(aes(color = branch), k = 3L) +
    scale_x_continuous(expand = expansion()) +
    theme(plot.margin = margin())
```



Now, let's move on to the next chapter, where we will introduce quad_layout() in full. While we've already introduced ggheatmap()—a specialized version of quad_layout()—most of the operations discussed in Chapter 3 can also be applied to quad_layout(). In the next section, we'll delve into quad_layout() and explore its full functionality.

7 quad-layout

quad_layout() arranges plots around the quad-sides of a main plot, aligning both horizontal and vertical axes, and can handle either discrete or continuous variables.



```
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

7.1 Introduction

Depending on whether you want to align discrete or continuous variables in the horizontal and vertical direction, there are four main types of quad_layout():

Alignment of Observations	horizontal	vertical	Data Format
<pre>quad_continuous()/ggside() quad_layout(xlim =) quad_layout(ylim =) quad_discrete()/ggheatmap()</pre>	continuous	continuous	data frame
	discrete	continuous	matrix
	continuous	discrete	matrix
	discrete	discrete	matrix

7.2 Annotations

Annotation is typically handled using a stack_layout(). Depending on whether you want to align observations in the specified direction, different stack_layout() are compatible (Section 3.5). Below is a table outlining the compatibility of various layout types for annotations:

Annotations	left and right	top and bottom
<pre>quad_continuous()/ggside() quad_layout(xlim =) quad_layout(ylim =) quad_discrete()/ggheatmap()</pre>	<pre>stack_continuous() stack_discrete() stack_continuous() stack_discrete()</pre>	<pre>stack_continuous() stack_continuous() stack_discrete() stack_discrete()</pre>

7.3 quad_discrete()

quad_discrete() aligns discrete variables in both horizontal and vertical directions. It serves
as the base version of ggheatmap()/heatmap_layout() and does not automatically add default
layers or mappings.

The underlying ggplot data of the main plot is the same with ggheatmap()/heatmap_layout() (Section 3.2), it is recommended to use .y, or .discrete_y as the y mapping and use .x or .discrete_x as the x mapping in the main plot.

```
quad_discrete(small_mat, aes(.x, .y)) +
    geom_tile(aes(fill = value)) +
    scale_fill_viridis_c()
```



7.4 quad_continuous()

quad_continuous() align continuous variables and is functionally equivalent to the ggside package. For convenience, ggside() is provided as an alias for quad_continuous(). This layout is particularly useful for adding metadata or summary graphics along a continuous axis.

```
ggside(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    # initialize top annotation
    anno_top(size = 0.3) +
    # add a plot in the top annotation
    ggalign() +
    geom_density(aes(displ, y = after_stat(density), colour = class), position = "stack") +
    # initialize right annotation
    anno_right(size = 0.3) +
    # add a plot in the right annotation
    ggalign() +
    geom_density(aes(x = after_stat(density), hwy, colour = class),
        position = "stack"
    ) &
    theme bw()
```



ggside() allows facetting for the main plot, which should also be applied to the annotations for proper alignment.

```
i2 <- iris
i2$Species2 <- rep(c("A", "B"), 75)
ggside(i2, aes(Sepal.Width, Sepal.Length, color = Species)) +
    geom_point(size = 2) +
   facet_grid(Species ~ Species2) +
   anno_top(size = 0.3) +
    ggalign() +
    geom_density(aes(Sepal.Width, y = after_stat(density), colour = Species),
        position = "stack"
    ) +
   facet_grid(cols = vars(Species2)) +
    anno_right(size = 0.3) +
   ggalign() +
    geom_density(aes(x = after_stat(density), Sepal.Length, colour = Species),
        position = "stack"
    facet_grid(rows = vars(Species)) &
    theme_bw()
```



If an annotation contains multiple plots, it can be tedious to add the same element to each one individually. One way to simplify this is by creating an external stack_layout() and adding the desired elements using the & operator. Then, you can add this stack_layout() to the quad_layout(). In Chapter 10, I will introduce another more powerful operator that seamlessly combines with the + operator, allowing you to add elements to multiple plots at once.

7.5 quad_layout()

This function arranges plots around the quad-sides of a main plot, aligning both horizontal and vertical axes, and can handle either discrete or continuous variables.

- If xlim is provided, a continuous variable will be required and aligned in the vertical direction. Otherwise, a discrete variable will be required and aligned.
- If ylim is provided, a continuous variable will be required and aligned in the horizontal direction. Otherwise, a discrete variable will be required and aligned.

```
quad_layout(small_mat, xlim = NULL) +
    geom_boxplot(aes(value, .discrete_y, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_layout(xlim = ...)")
```

quad_layout(xlim = ...)



```
quad_layout(small_mat, ylim = NULL) +
    geom_boxplot(aes(.discrete_x, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_layout(ylim = ...)")
```

quad_layout(ylim = ...)



As discussed in Section 3.4, quad_anno() will always attempt to initialize a stack_layout() with the same alignment as the current direction. For top and bottom annotations in quad_layout(xlim = ...), and left and right annotations in quad_layout(ylim = NULL), quad_anno() will not initialize the annotation due to inconsistent data types.

```
quadh <- quad_layout(small_mat, xlim = NULL) +
    anno_top()
#> Warning: `data` in `quad_layout()` is a double matrix, but the top annotation stack need
#> a <data.frame>, won't initialize the top annotation stack
quadv <- quad_layout(small_mat, ylim = NULL) +
    anno_left()
#> Warning: `data` in `quad_layout()` is a double matrix, but the left annotation stack
#> need a <data.frame>, won't initialize the left annotation stack
```

Manual adding of a stack_layout() is required in such cases, you can set initialize = FALSE to prevent the warning message.

```
quadh <- quad_layout(small_mat, xlim = NULL) +
    anno_top(initialize = FALSE)
quadv <- quad_layout(small_mat, ylim = NULL) +
    anno_left(initialize = FALSE)</pre>
```

```
quadh +
    stack_continuous("v", mpg) +
    # add a plot in the top annotation
    ggalign(mapping = aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class)) +
    quad_active() +
    geom_boxplot(aes(value, .discrete_y, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2")+
    layout_title("quad_alignh()")
```



```
quadv +
    stack_continuous("h", mpg) +
    # add a plot in the left annotation
    ggalign(mapping = aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class)) +
    quad_active() +
    geom_boxplot(aes(.discrete_x, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_alignv()")
```



Alternatively, you can set initialize = TRUE, which will initialize the annotation stack layout with no data. In this case, you must provide data in each plot within the annotation.

```
quad_layout(small_mat, xlim = NULL) +
    geom_boxplot(aes(value, .discrete_y, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2") +
    anno_top(initialize = TRUE) +
    ggalign(data = mpg, aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class))+
    layout_title("quad_alignh()")
```



```
quad_layout(small_mat, ylim = NULL) +
    geom_boxplot(aes(.discrete_x, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    anno_left(initialize = TRUE) +
    ggalign(data = mpg, aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class))+
    layout_title("quad_alignv()")
```



In the next chapter, we will explore even more advanced techniques for combining multiple quad_layout()s. These methods will provide you with the tools to manage more complex plot arrangements and make your visualizations even more flexible and powerful.

8 A list of quad_layout()

Similarly, stack_layout() can be added to a quad_layout(), and a quad_layout() can also be added to a stack_layout().

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

note: when stack_layout() contains a nested quad_layout(), it cannot be used within the annotation of another quad_layout().

8.1 Add quad_layout() to stack_layout()

Here is a summarized table showing which quad_layout() can be used with each stack_layout():

```
stack_discreteh(ack_discretes(ack_continuous))
quad_continuous()/ggside()
quad_layout(xlim =
...)
quad_layout(ylim =
...)
quad_layout(ylim =
...)
```

As long as the alignment is consistent across both stack_layout() and quad_layout(), you can directly add quad_layout().

```
stack_discreteh(small_mat) +
    ggheatmap()
#> > heatmap built with `geom_tile()`
```



When ggheatmap()/quad_layout() is added to a stack_layout(), it will also set the active context to itself, which means subsequent addition will be directed to ggheatmap()/quad_layout(). One exception is the ggheatmap()/quad_layout() itself, which cannot be added to another quad_layout(). In this case, they will be added directly to the stack_layout().



The data of ggheatmap()/quad_layout() can inherit from the stack_layout(), but the data format must match. Otherwise, you will need to manually provide the data.

```
quad_continuousv_plot <- quad_layout(small_mat, xlim = NULL) +
    geom_boxplot(aes(value, .discrete_y, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2")
# `stack_continuous()` need a data frame
stack_continuousv() +
    quad_continuousv_plot +
    quad_continuousv_plot</pre>
```



When ggheatmap()/quad_layout() is added to a vertical stack_layout(), the inherited matrix is transposed before use. This is because the columns of ggheatmap()/quad_layout() must match the number of observations in stack_layout().

```
stack_discretev(small_mat) +
    ggheatmap() +
    ggheatmap()
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



stack_discrete() ensures that all plots aligned along the stack have the same ordering index
or groups.

We can customize the layout in the stack_discrete() directly, or in quad_layout(). As introduced in Section 2.5, you can easily switch from the ggheatmap()/quad_layout() to the stack_layout() using stack_active().

```
stack_discretev(small_mat) +
    ggheatmap() +
    ggheatmap() +
    anno_bottom(size = 0.2) +
    align_dendro(aes(color = branch), k = 3)+
    layout_title("dendrogram in ggheatmap()")
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

dendrogram in ggheatmap()



```
stack_discretev(small_mat) +
    ggheatmap() +
    ggheatmap() +
    stack_active() +
    align_dendro(aes(color = branch), k = 3, size = 0.2) +
    scale_y_reverse() +
    layout_title("dendrogram in stack_layout()")
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

dendrogram in stack_layout()



When applied to a stack_layout(), the orientation of the dendrogram may need to be manually adjusted.

8.2 Control sizes

A numeric or a unit object of length 3 should be provided in stack_discrete()/stack_continuous() when placing a quad_layout(). For vertical stack_layout(), this means quad_layout() with left or right annotations; for horizontal stack_layout(), this means quad_layout() with top or bottom annotations. The first size controls the relative width/height of the left or top annotation, the second controls the relative width/height of the main plot, and the third controls the relative width/height of the right or bottom annotation.

By default the three rows/columns will have equal sizes.

```
stack_discretev(small_mat) +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity") +
```

```
ggheatmap() +
   theme(axis.text.x = element_text(angle = -60, hjust = 0))
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



```
heat1 <- ggheatmap(t(small_mat)) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity")

stack_discretev(small_mat, sizes = c(1, 2, 1)) +
    heat1 +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0))

#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



In this way, the width/height of main plot specified in quad_active() or quad_layout()/ggheatmap() won't work.

```
stack_discretev(small_mat) +
    ggheatmap(width = unit(2, "null")) + # not work
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity") +
    ggheatmap(width = unit(2, "null")) + # not work
    theme(axis.text.x = element_text(angle = -60, hjust = 0))
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



Now that you've learned how to combine quad_layout() and stack_layout() in various configurations, you're ready to explore a new, exciting way to organize and visualize your data: the circle layout. This layout offers a unique, radial perspective that can be particularly useful for visualizing hierarchical data or creating visually engaging plots.

9 circle layout

circle_layout() arranges plots in a circular, each plot will occupy one circle track. Based on whether we want to align the discrete or continuous variables, there are two types of circle layouts:

- circle_discrete(): Align discrete variable along the circle.
- circle_continuous(): Align continuous variable along the circle.

circle_layout()



```
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

9.1 radial

the core argument of circle_layout() is radial, which should be a coord_radial() object that defines the global parameters for coord_radial() across all plots in the layout. The parameters start, end, direction, and expand will be inherited and applied uniformly to all plots within the layout. The parameters theta and r.axis.inside will always be ignored and will be set to "x" and TRUE, respectively, for all plots.

```
circle_discrete(small_mat, radial = coord_radial(end = pi / 2, expand = FALSE)) +
    ggalign() +
    geom_tile(aes(y = .column_index, fill = value)) +
    scale_fill_viridis_c()
```



In essence, you can think of circle_layout() as a radial version of stack_vertical(). When rendered, circle_layout() uses coord_radial() to arrange each plot along a circular track. The radial argument in circle_layout() controls the overall inner radius. Plots are added from the outermost to the innermost position.

```
circle_discrete(small_mat, radial = coord_radial(inner.radius = 0.1)) +
    align_dendro() +
    ggalign() +
    geom_tile(aes(y = .column_index, fill = value)) +
    scale_fill_viridis_c()
```



The size of each plot within the circle can be adjusted using the size argument (Section 2.4). However, it's important to note that circle_layout() only supports relative size adjustments, meaning all size values are interpreted relative to one another, even for unit objects. By default, all plots have a relative size of 1.

```
circle_discrete(small_mat, radial = coord_radial(inner.radius = 0.1)) +
    align_dendro(size = 0.5) +
    ggalign() +
    geom_tile(aes(y = .column_index, fill = value)) +
    scale_fill_viridis_c()
```



Although circle_layout() does not currently support splitting observations into different facets, you can still indicate different groups visually. For instance, you can add a tile to represent each group. If you're using a dendrogram, you can color the branches to indicate different groups.

```
circle_discrete(small_mat, radial = coord_radial(inner.radius = 0.1)) +

# add a dendrogram
align_dendro(aes(color = branch), k = 3L, size = 0.5) +
scale_color_brewer(palette = "Dark2") +

# add a heatmap
ggalign() +
geom_tile(aes(y = .column_index, fill = value)) +
scale_fill_viridis_c() +

# add a single tile for each group
ggalign(NULL, size = 0.1) +
geom_tile(aes(y = 1L, fill = .panel)) +
theme_no_axes("y") +
scale_fill_brewer(palette = "Dark2", guide = "none")
```



9.2 spacing

The spacing between plot tracks in circle_layout() is controlled panel.spacing.r parameter. The panel.spacing.r parameter sets the spacing between the individual plot panels (tracks) within the circle layout. Increasing this value will add more space between the tracks. Specifically, the panel.spacing.r of the outer plot determines the spacing between it and the inner plot. By modifying this value, you can control the overall visual separation between tracks. Note, you should also remove the scale expansion if you remove all spaces.

```
circle_discrete(small_mat, radial = coord_radial(inner.radius = 0.1)) +

# add a dendrogram
align_dendro(aes(color = branch), k = 3L, size = 0.5) +
no_expansion("y") +
scale_color_brewer(palette = "Dark2") +

# add a heatmap
ggalign() +
geom_tile(aes(y = .column_index, fill = value)) +
scale_fill_viridis_c() +
no_expansion("y") +
```

```
theme(panel.spacing.r = unit(0, "mm")) +

# add a single tile for each group
ggalign(NULL, size = 0.1) +
geom_tile(aes(y = 1L, fill = .panel)) +
scale_fill_brewer(palette = "Dark2", guide = "none") +
no_expansion("y") +
theme(panel.spacing.r = unit(0, "mm"))
```



The no_expansion() function is used to remove expansion around specific borders. It accepts a string with one or more of the following values: "t" (top), "l" (left), "b" (bottom), "r" (right), "x" (both left and right), and "y" (both bottom and top).

Now we have explored all the Layout defined in the package, next, we will build on these concepts and explore even more advanced strategies for integrating elements across multiple plots or annotations in a layout.

10 Operators

In ggalign, operators are used to manage and manipulate the plot elements in various layouts. These operators help you build complex visualizations by adding, or modifying elements across multiple plots in a layout. You may be familiar with the + and & operators, this section will expand on their usage, as well as introduce the subtraction operator (-) and how to apply it in different contexts.

- 1. Addition Operator (+): Adds elements to the active plot in the active layout.
- 2. Logical AND Operator (&): Applies elements to all plots in the layout.
- 3. Subtraction Operator (-): Allows you to add elements or modify them across multiple plots in the layout

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

10.1 Addition operator

+: adds elements to the active plot in the active layout.

The + operator is straightforward and should be used as needed.

In stack_layout(), a nested layout will only occur if you pass a quad_layout() into stack_layout() (Chapter 8). If the active context in the stack_layout() is quad_layout(), this quad_layout() is treated as the active layout, and the + operator will add elements to it, following the same principles as in quad_layout(). Otherwise, the stack_layout() itself is treated as the active layout, and the + operator will add elements directly to this plot.

circle_layout() works similarly to stack_layout(), but it does not support nested layouts.
Therefore, the active layout in circle_layout() will always be the circle_layout() itself.

```
stack_alignh(small_mat) +
    align_dendro() +
    geom_point() +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```



In quad_layout(), four nested layouts are available for annotations: top, left, bottom, and right. If no active annotation is set, quad_layout() treat itself as the active layout. Since there is only one main plot in the quad_layout(), the main plot is always treated as the active plot in this context. Otherwise, the active annotation is treated as the active layout, and the + operator will add elements to it, following the same principles as in stack_layout().

```
ggheatmap(small_mat) +
    scale_fill_viridis_c() +
    anno_left(size = 0.2) +
    align_dendro() +
    anno_right(size = 0.2) +
    align_dendro()
#> > heatmap built with `geom_tile()`
```



10.2 logical AND operator

& applies elements to all plots in the layout including plots in the nested layout.

The & operator works similarly to patchwork, applying an element across all plots in a layout. Since & has lower precedence than +, it's generally best to use it at the end of an expression or you should wrap it in parentheses when needed.

```
# Initialize the heatmap
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    # Add a dendrogram in the left annotation and split the dendrogram into 3 groups
    align_dendro(aes(color = branch), k = 3L) +
    anno_right(size = 0.2) +
    # Add a dendrogram in the right annotation and split the dendrogram into 3 groups
    align_dendro(aes(color = branch), k = 3L) &
    # Set color scale for all plots
    scale_color_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
```



10.3 Subtraction operator

The – operator is more powerful than the & operator, enabling flexible addition of elements to multiple plots. While its use might initially seem unintuitive, the reason behind this is that – shares the same precedence group as +, which allows it to seamlessly combine with +.

10.3.1 quad_layout()

For ggheatmap()/quad_layout(), if the active context is the ggheatmap()/quad_layout() itself (no active annotation), the - operator behaves similarly to &. It applies the specified elements to all plots within the layout.

```
# Initialize the heatmap
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    anno_right(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # Remove any active annotation
    quad_active() -
```

```
# Set color scale for all plots, since the active layout is the `ggheatmap()`/`quad_layo
scale_color_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
```



If the active layout is an annotation stack, the – operator will only add the elements to all plots in the active annotation stack:

```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    align_dendro(aes(color = branch), k = 3L) -
    # Modify the the color scales of all plots in the left annotation
    scale_color_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
```



10.3.2 stack_layout()

For stack_layout(), if the active layout is the stack_layout() itself, - applies the elements to all plots in the layout except the nested ggheatmap()/quad_layout().

```
stack_alignv(small_mat) +
    align_dendro() +
    ggtitle("I'm from the parent stack") +
    ggheatmap() +
    # remove any active context
    stack_active() +
    align_dendro() +
    ggtitle("I'm from the parent stack") -
    # Modify the the color scales of all plots in the stack layout except the heatmap layout
    scale_color_brewer(palette = "Dark2") -
    # set the background of all plots in the stack layout except the heatmap layout
    theme(plot.background = element_rect(fill = "red"))
#> > heatmap built with `geom_tile()`
```



When the active layout is the nested ggheatmap()/quad_layout(), the - operator applies the elements to this nested layout, following the same principles as in the Section 10.3.1.

Want apply elements for plots in both stack_layout() and the nested ggheatmap()/quad_layout() at the same time? refer to the following section.

10.4 with_quad()

The with_quad() function adjusts the context in which elements are applied in ggheatmap()/quad_layout(). It allows you to control how objects such as themes, scales, and other plot elements are applied to specific annotation stacks or the main plot, without changing the currently active context.

This function accepts three arguments:

- 1. **x**: An object which can be added to the plot.
- 2. **position**: A string containing one or more of "t", "l", "b", and "r" specifies the context for applying x.
- 3. main: A single boolean value indicates whether x should also apply to the main plot within ggheatmap()/quad_layout(). Only used when position is not NULL.

10.4.1 quad_layout()

Default Behavior by wrapping object with with_quad():

- When ggheatmap()/quad_layout() has no active annotation stack, objects added via + or operate normally without with_quad().
- When the active annotation stack is set, with_quad() ensures the applied object also modifies:
 - The main plot (by default).
 - Opposite annotation stacks when using -.

By wrapping object with with_quad(), the + operator will apply the object not only to the active plot in the annotation stack, but also to the main plot unless specified by main argument otherwise.

```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro() +
    # apply the object not only to the active plot in the annotation stack, but
    # also to the main plot
    with_quad(theme(plot.background = element_rect(fill = "red")))
#> > heatmap built with `geom_tile()`
```



By wrapping object with with_quad(), the - operator will apply the object not only to that annotation stack but also to the opposite one (i.e., bottom if top is active, and vice versa). In these cases, the object will also be applied to the main plot by default unless specified by main argument otherwise.

```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # Change the active layout to the left annotation
    anno_top(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    anno_bottom(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) -
    # Modify the theme of all plots in the bottom and the opposite annotation
    # in this way, the `main` argument by default would be `TRUE`
    with_quad(theme(plot.background = element_rect(fill = "red")))
#> > heatmap built with `geom_tile()`
```



The position argument can be a string containing one or more of "t", "l", "b", and "r", indicating which annotation stack should be used as the context. When the position argument is manually set, the default value of the main argument will be FALSE.

```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # initialize the top annotation
    anno_top(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # initialize the bottom annotation
    anno_bottom(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) -
    # Modify the background of all plots in the left and top annotation
    with_quad(theme(plot.background = element_rect(fill = "red")), "tl")
#> > heatmap built with `geom_tile()`
```



Setting position to NULL change the context to the ggheatmap()/quad_layout() itself.

```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro() +
    # we apply the theme to the main plot only
    with_quad(theme(plot.background = element_rect(fill = "red")), NULL)
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat) +
    # initialize the left annotation
    anno_left(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # initialize the top annotation
    anno_top(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) +
    # initialize the bottom annotation
    anno_bottom(size = 0.2) +
    align_dendro(aes(color = branch), k = 3L) -
    # Modify the background of all plots
    with_quad(theme(plot.background = element_rect(fill = "red")), NULL)
#> > heatmap built with `geom_tile()`
```



10.4.2 stack_layout()

When the active layout is the stack_layout() itself, by default, by wrapping object with with_quad(), - operator will apply changes to all plots along the stack_layout(), which means if the stack layout is in horizontal, - operator will also add the element to the left and right annotation, if the stack layout is in vertical, - operator will also add element to the top and bottom annotation. In these cases, the object will also be applied to the main plot by default unless specified by main argument otherwise.

```
stack_alignv(small_mat) +
    align_dendro() +
    ggtitle("I'm from the parent stack") +
    ggheatmap() +
    anno_top() +
    align_dendro() +
    ggtitle("I'm from the nested heatmap") +
    # remove any active context
    stack_active() +
    align_dendro() +
    ggtitle("I'm from the parent stack") -
    # Modify the the color scales of all plots in the stack layout except the heatmap layout scale_color_brewer(palette = "Dark2") -
```

```
# set the background of all plots in the stack layout (including plots in the heatmap lay
with_quad(theme(plot.background = element_rect(fill = "red")))
#> > heatmap built with `geom_tile()`
```



+ operator won't do anything special when the active layout is the stack_layout() itself.

When the active layout is the nested ggheatmap()/quad_layout(), the +/- operator applies the elements to this nested layout, following the same principles as for ggheatmap()/quad_layout().

11 Schemes

Schemes control the actions of plots within the layout and can be applied either globally to the layout or individually to specific plots.

- To apply a scheme to a single plot, use the + operator.
- To set a scheme at the layout level, use the operator. Scheme set at the layout level will be inherited by all plots when rendering the layout.

scheme inherit properties from parent layout hierarchically.

The package currently provides three schemes, each prefixed with scheme_:

- scheme_theme: Sets the default theme for the plot.
- scheme_data: Transforms the plot data. Many functions in this package require a specific data format to align observations, scheme_data() helps reformat data frames as needed.
- scheme_align: Defines alignment specifications for plots within the layout.

colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>

```
library(ggalign)
#> Loading required package: ggplot2

set.seed(123)
small_mat <- matrix(rnorm(81), nrow = 9)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))</pre>
```

11.1 scheme_theme()

The scheme_theme() function extends theme() to set a default theme for plots, allowing you to input theme() elements directly or add the theme elements.

To set a scheme for a single plot, simply use the + operator:

```
ggheatmap(small_mat) +
          scheme_theme(plot.background = element_rect(fill = "red"))
#> > heatmap built with `geom_tile()`
```

- 1 initialize a heatmap layout
- 2 set the default theme, change the plot background for the main plot



You can use a theme() object directly in scheme_theme().

```
ggheatmap(small_mat, filling = FALSE) +
  geom_tile(aes(fill = value), width = 0.9, height = 0.9) +
  scheme_theme(theme_bw(), plot.background = element_rect(fill = "red"))
```



Note that scheme_theme() serves as the default theme and will always be overridden by any theme() settings applied directly to the plot. The default theme (scheme_theme()) is applied first, followed by any specific theme() settings, even if theme() is added before scheme_theme().

```
ggheatmap(small_mat) +
    # change the plot theme of the heatmap body
    theme(plot.background = element_rect(fill = "blue")) +
    # change the default theme of the heatmap body
    scheme_theme(plot.background = element_rect(fill = "red"))
#> > heatmap built with `geom_tile()`
```



By using the - operator with schemes, we apply the scheme directly to the active layout.

```
ggheatmap(small_mat) +
    # Change the active layout to the top annotation
    anno_top() +
    # add a dendrogram to the top annotation
    align_dendro() +
    # add a bar plot to the top annotation
    ggalign(aes(.discrete_x, value, fill = factor(.names)), data = rowSums) +
    geom_bar(stat = "identity") -
    # Change the default theme of the top annotation
    # All plots in the top annotation will inherit this default theme
    scheme_theme(plot.background = element_rect(fill = "red"))
#> > heatmap built with `geom_tile()`
```



Unlike individual ggplot2 elements, which will be added directly to each plot by – operator, **layout-level schemes** set by – operator are **inherited** by all plots in the layout when **rendered**. Any plot-specific schemes will override these layout-level schemes, regardless of the order in which they are added.

```
ggheatmap(small_mat) +
    # Change the active layout to the top annotation
    anno_top() +
    # add a dendrogram to the top annotation
    align_dendro() +
    # change the scheme_theme for the dendrogram plot
    scheme_theme(plot.background = element_rect(fill = "blue")) +
    # add a bar plot to the top annotation
    ggalign(aes(.discrete_x, value, fill = factor(.names)), data = rowSums) +
    geom_bar(stat = "identity") -
    # Change the default theme of the top annotation
    # All plots in the top annotation will inherit this default theme
    # But the plot-specific options will override these
    scheme_theme(plot.background = element_rect(fill = "red"))
#> > heatmap built with `geom_tile()`
```



11.2 scheme_data()

ggalign() requires the specific data format for its operations. If you need to transform or filter data for individual geoms, you can use the data argument within each geom. However, if you have multiple geoms and want a consistent transformation applied across all, you can utilize the scheme_data() function. This allows you to transform the default data for the entire plot.

The scheme_data() accepts a function that takes a data frame as input and returns a modified data frame. By default, scheme_data() will attempt to inherit from the parent layout if the data is inherited from it. However, there is one exception: align_dendro() will not inherit scheme_data() transformations by default.

```
set.seed(1234L)
ggheatmap(small_mat) +
   anno_top() +
   align_kmeans(3L) +
   # we add a bar plot
   ggalign() +
   # we subest the plot data
   scheme_data(~ subset(.x, .panel == 1L)) +
```

```
geom_bar(aes(y = value, fill = .row_names), stat = "identity")
#> > heatmap built with `geom_tile()`
```



11.3 scheme_align()

The scheme_align() function controls the align specifications for plots.

11.3.1 guides

By default, ggheatmap() will collect all guide legends on the side from which they originate.

```
heatmap_collect_all_guides <- ggheatmap(small_mat, width = 2, height = 2L) +
    # we set the legend to the left side
    scale_fill_gradient2(
        low = "blue", high = "red",
        name = "I'm from heatmap body",
        guide = guide_legend(position = "left")
    ) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    # we add a top annotation</pre>
```

```
anno_top() +
    # in the top annotation, we add a dendrogram
    align_dendro(aes(color = branch), k = 3L) +
    # we set the legends of the dendrogram to the left side
    scale_color_brewer(
        name = "I'm from top annotation", palette = "Dark2",
        guide = guide_legend(position = "left")
    ) +
    # we add a left annotation
    anno_left() +
    align_dendro(aes(color = branch), k = 3L) +
    # we set the legends of the dendrogram to the top side
    scale_color_brewer(
        name = "I'm from left annotation", palette = "Dark2",
        guide = guide_legend(position = "top", direction = "vertical")
    ) &
    # we remove all margins for all plots
   theme(plot.margin = margin())
heatmap_collect_all_guides
#> > heatmap built with `geom_tile()`
```



The guides argument schemes which side of guide legends should be gathered. In the following example, we'll collect the guide legends only on the top (t) sides:

```
heatmap_collect_all_guides -
    # we set global `guides` argument for `the heatmap layout`
    # we only collect guides in the top side
    with_quad(scheme_align(guides = "t"), NULL)
#> > heatmap built with `geom_tile()`
```



You can also apply the scheme_align() function directly to specific plots:

```
heatmap_collect_all_guides -
    # we set global `guides` argument for the heatmap layout
    # we only collect guides in the top side
    with_quad(scheme_align(guides = "t"), NULL) +
    # `+` apply it to the active plot
    # for the heatmap body, we collect guide in the left side
    with_quad(scheme_align(guides = "l"), NULL)
```



Note: The legend on the left side of the heatmap body is collected and positioned on the left side at the layout level.

If you're annoyed by the large space between the left annotation and the heatmap body, don't worry! This issue will be addressed in Section 11.3.2.

Now, Let's dive deeper into the guide collection process.

In the last example, we set the guides argument for the heatmap body. But what happens when we set the guides for the dendrogram in the top annotation?

```
heatmap_collect_all_guides -
    # we set global `guides` argument for `the heatmap layout`
    # we only collect guides in the top side in the heatmap layout
    with_quad(scheme_align(guides = "t"), NULL) +
    # `+` apply it to the active plot
    # for the dendrogram in the top annotation, we collect guide in the left side
    with_quad(scheme_align(guides = "l"), "t")
#> > heatmap built with `geom_tile()`
```


Nothing seems to have changed, right? This is because guide legends within annotation stacks are first collected by the annotation <code>stack_layout()</code> and only then passed to the top-level layout for further integration.

By default, the annotation stack inherits the guides arguments from the heatmap layout, followed by the inherited of individual plot in the annotation. So guides argument set at top-level (heatmap layout) will affect all guide collection behaviour.

In this example:

• The legend on the left side of the dendrogram in the top annotation is collected first at the annotation level.

- Since the heatmap layout is not set to collect legends from the left side, it remains at the left side within the annotation stack.
- For this specific case, the top annotation contains only one plot, so its annotation-level placement is identical to plot-level placement.

To override this, you can use the free_guides argument of the quad_anno()/anno_*() function. This differs from the guides argument in scheme_align(), which schemes the behavior for the plots in the layout. The free_guides argument specifies which guide legends from at the annotation stack layout level should be collected by the heatmap layout.

```
heatmap_collect_all_guides -
    # we set global `guides` argument for `the heatmap layout`
    # we only collect guides in the top side in the heatmap layout
    with_quad(scheme_align(guides = "t"), NULL) +
    # we also collect guides in the left side for the top annotation stack
    # in the heatmap layout
    anno_top(free_guides = "l") +
    # `+` apply it to the active plot
    # for the dendrogram in the top annotation, we collect guide in the left side
    scheme_align(guides = "l")
#> > heatmap built with `geom_tile()`
```



Note: The heatmap layout will only collect guide legends from the annotation stack if the stack layout collects its own guides first.

11.3.2 free_spaces

By default, ggheatmap() will align all elements of the plot, which can sometimes lead to unwanted spacing. Consider the following example:

```
ggheatmap(small_mat) +
    # add top annotation
    anno_top(size = unit(30, "mm")) +
    # add a dendrogram to the top annotation
    align_dendro() +
    # here, we use long labels for visual example
    scale_y_continuous(
        expand = expansion(),
        labels = ~ paste("very very long labels", .x)
    ) +
    # add left annotation
    anno_left(unit(20, "mm")) +
    # add a dendrogram to the left annotation
    align_dendro()
#> > heatmap built with `geom_tile()`
```



In this case, the left annotation stack is positioned far from the heatmap body due to the wide axis labels in the top annotation stack. This occurs because the top annotation stack is aligned with the heatmap body. To fix this, you can remove the left borders around the panel of the top annotation stack by setting free_spaces = "1".

```
ggheatmap(small_mat) +
    # add top annotation
    anno_top(size = unit(30, "mm")) -
    # we remove the spaces of the left borders in the top annotation
    scheme_align(free_spaces = "l") +
    # add a dendrogram to the top annotation
    align_dendro() +
    # here, we use long labels for visual example
    scale_y_continuous(
        expand = expansion(),
        labels = ~ paste("very very long labels", .x)
    ) +
    # add left annotation
    anno_left(unit(20, "mm")) +
    # add a dendrogram to the left annotation
    align_dendro()
#> > heatmap built with `geom_tile()`
```



One useful way to utilize free_spaces is to position the guide legends next to the annotations. (Note the guide legend from the bottom annotation):

```
heatmap_collect_all_guides +
    # reset the active context to the heatmap layout
    quad_active() -
    # we set global `guides` argument for the heatmap layout
    # we only collect guides in the top side
    scheme_align(guides = "t") +
    # `+` apply it to the current active plot
    # for the heatmap body, we collect guide in the left side
    scheme_align(guides = "l") -
    with_quad(scheme_align(free_spaces = "l"), "t")
#> > heatmap built with `geom_tile()`
```



Usually you want to apply free_spaces with the whole layout, instead of individual plots.

In ggheatmap()/quad_layout(), the behavior of the free_spaces and free_labs arguments differs from guides arguments in scheme_align() when inheriting from the parent layout:

- For top and bottom annotations, it inherits from the left ("1") and right ("r") axes.
- For left and right annotations, it inherits from the top ("t") and bottom ("b") axes.

11.3.3 free_labs

By default, we won't align the axis titles.

```
ggheatmap(small_mat) +
    ylab("Heatmap title") +
    anno_top(size = unit(30, "mm")) +
    align_dendro() +
    ylab("Annotation title")
#> > heatmap built with `geom_tile()`
```



To align axis titles, you can set free_labs = NULL. Alternatively, A single string containing one or more of axis positions ("t", "l", "b", "r") to indicate which axis titles should be free from alignment.

```
ggheatmap(small_mat) -
    scheme_align(free_labs = NULL) +
    ylab("Heatmap title") +
    anno_top(size = unit(30, "mm")) +
    align_dendro() +
    ylab("Annotation title")
#> > heatmap built with `geom_tile()`
```



12 Difference with ggplot2

ggalign focuses on aligning axes across multiple plots. While it builds on the ggplot2 framework, there are key differences in how scales, facets, and coordinates are handled. This vignette highlights these syntax differences.

12.1 Position Scales

To better fit the alignment-based layout, ggalign introduces adjustments to breaks, labels, and expand arguments.

For discrete values, ggalign introduces special syntax and handling for these axes, differing from the default behavior in ggplot2. These adjustments ensure that observations are properly aligned and operations are user-friendly.

The following syntax applies to the axes used to align observations (discrete values).

12.1.1 breaks

breaks and labels are typically handled similarly to discrete scales, as we focus on aligning observations (which should be regarded as discrete variables); no matter what you use is continuous scale or discrete scale.

breaks should be one of:

- `NULL` for no breaks
- `waiver()` for the default breaks (the full data index or `NULL` if no data names and `labels` is `waiver()`)
- A character vector of breaks (rownames / colunames of the matrix).
- A numeric vector of data index (must be an integer).
- A function that takes the data names or the data index as input and returns breaks as output. Also accepts rlang lambda function notation.

Default breaks without names:

```
no_names <- small_mat
colnames(no_names) <- NULL
ggheatmap(no_names) + scale_x_continuous()
#> > heatmap built with `geom_tile()`
```



No breaks:

```
ggheatmap(small_mat) + scale_x_continuous(breaks = NULL)
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat, filling = FALSE) +
   geom_tile(aes(.discrete_x, .discrete_y, fill = value)) +
   scale_x_discrete(breaks = NULL)
```



Character-based breaks use data names (or indices if names are absent)

```
ggheatmap(small_mat) +
    scale_x_continuous(breaks = c("column3", "column5")) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat, filling = FALSE) +
   geom_tile(aes(.discrete_x, .discrete_y, fill = value)) +
   scale_x_discrete(breaks = c("column3", "column5")) +
   anno_top() +
   align_dendro(k = 3L)
```



Integer-based breaks are interpreted as data indices:

```
ggheatmap(small_mat) +
    scale_x_continuous(breaks = c(3, 5)) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat, filling = FALSE) +
   geom_tile(aes(.discrete_x, .discrete_y, fill = value)) +
   scale_x_discrete(breaks = c(3, 5)) +
   anno_top() +
   align_dendro(k = 3L)
```



Floating numbers are invalid for breaks:

```
ggheatmap(small_mat) + scale_x_continuous(breaks = c(3.5, 5))
#> > heatmap built with `geom_tile()`
#> Error in `scale_x_continuous()`:
#> ! Can't convert from `breaks` <double> to <integer> due to loss of precision.
#> * Locations: 1
```

To interpret integers as plot-specific coordinate indices, wrap them with I():

```
ggheatmap(small_mat) +
    scale_x_continuous(breaks = I(3:4)) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



12.1.2 labels

labels should be one of:

- `NULL` for no labels
- `waiver()` for the default labels (data names)
- A character vector giving labels (must be same length as breaks)
- An expression vector (must be the same length as breaks). See $\ensuremath{\,\hat{}}$ for details.
- A function that takes the data names (or data index if data has no names) as input and returns labels as output. This can be also a rlang lambda function.

The default labels are the data names (or indices if names are absent):

```
ggheatmap(small_mat) + scale_x_continuous() +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



No labels:

```
ggheatmap(small_mat) +
    scale_x_continuous(labels = NULL) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



Character labels will be reordered based on the data's ordering:

```
ggheatmap(small_mat) +
    scale_x_continuous(labels = letters[seq_len(ncol(small_mat))]) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



To retain the original order of character labels, wrap them with I():

```
ggheatmap(small_mat) +
    scale_x_continuous(labels = I(letters[seq_len(ncol(small_mat))])) +
    anno_top() +
    align_dendro(k = 3L)
#> > heatmap built with `geom_tile()`
```



By default, labels correspond to breaks:

```
ggheatmap(small_mat) +
    scale_x_continuous(breaks = c(5, 3), labels = c("a", "b"))
#> > heatmap built with `geom_tile()`
```



To override the default matching, wrap the labels vector with I():

```
ggheatmap(small_mat) +
    scale_x_continuous(breaks = c(5, 3), labels = I(c("a", "b")))
#> > heatmap built with `geom_tile()`
```



12.2 theme

Although ggplot2 does not officially support vectorized input for theme elements, we can still utilize it. ggalign extends this feature, allowing theme elements to be vectorized and applied across panels.

```
ggheatmap(small_mat) +
    theme(
        axis.text.x = element_text(
            colour = c(rep("red", 4), rep("blue", 5))
    ),
    axis.ticks.x = element_line(
            colour = c(rep("red", 4), rep("blue", 5))
    ),
    axis.ticks.length.x = unit(rep(c(1, 4), times = c(4, 5)), "mm")
    ) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3L) +
    scale_y_continuous(expand = expansion()) &
    theme(plot.margin = margin())
#> Warning: Vectorized input to `element_text()` is not officially supported.
```

#> i Results may be unexpected or may change in future versions of ggplot2.
#> > heatmap built with `geom_tile()`



12.3 Facets

When working with facets, manual configuration of the panel using the facet_*() functions is not possible since the internal structure will use facet_grid() to set the row/column groups defined by align_*() functions. However, you can still use facet_grid() or facet_null() (if no panel) to control other arguments except aligned panels (rows in horizontal stack layout or cols in vertical stack layout, or both rows and cols in heatmap body).

A common use case is to modify the panel strip text. The default theme (theme_ggalign()) will always remove the panel strip text, you can override this behaviour with theme(strip.text = element_text()) to add the panel title in the plot area.

```
ggheatmap(small_mat) +
   facet_grid(labeller = labeller(.column_panel = function(x) letters[as.integer(x)])) +
   theme(strip.text = element_text()) +
   anno_top() +
   align_kmeans(centers = 3L)
#> > heatmap built with `geom_tile()`
```



13 Plot Composer

Special thanks to the patchwork project—many core codes of the plot composer process were adapted from patchwork. We have added new features to better implement ggalign's layout functions (stack_layout() and quad_layout()), including:

- free_align()
- free_border()
- free_guide()
- free_lab()
- free_space()
- free_vp()

These features have not been pushed to patchwork because they required significant modification of core code. We attempted to merge them, but the author of patchwork decided to implement some of these features independently. The latest version of patchwork now includes free_align(), free_lab(), and free_space() functionality under a single function: patchwork::free(). For more details, see: https://www.tidyverse.org/blog/2024/09/patchwork-1-3-0/.

The plot composer function in ggalign is align_plots(), which behaves similarly to cowplot::align_plots() and patchwork::wrap_plots(). However, you can directly use align_plots() with quad_layout()/ggheatmap() and stack_layout(), ensuring that they align correctly by plot panel. Additionally, align_plots() can align pheatmap and ComplexHeatmap objects, although they won't align by panel area with ggplot2.

```
library(ggalign)
#> Loading required package: ggplot2
```

13.1 Plot Assembly

We'll start with a few well-known example plots:

```
p1 <- ggplot(mtcars) +
        geom_point(aes(mpg, disp))
p2 <- ggplot(mtcars) +
        geom_boxplot(aes(gear, disp, group = gear))
p3 <- ggplot(mtcars) +
        geom_bar(aes(gear)) +
        facet_wrap(~cyl)
p4 <- ggplot(mtcars) +
        geom_bar(aes(carb))
p5 <- ggplot(mtcars) +
        geom_violin(aes(cyl, mpg, group = cyl))</pre>
```

Either add the plots as single arguments

```
align_plots(p1, p2, p3, p4, p5)
```



Or use bang-bang-bang to add a list of plots

```
align_plots(!!!list(p1, p2, p3), p4, p5)
```



13.2 Empty area

You can use NULL to indicate the empty area.

align_plots(p1, NULL, p2, NULL, p3, NULL)







13.3 Controlling the grid

Like patchwork, if no specific layout is provided, align_plots() will attempt to create a grid that is as square as possible, with each column and row taking up equal space:

 $align_plots(p1, p2, p3, p4, ncol = 3)$



To adjust the widths of columns, use:

align_plots(p1, p2, p3, p4, widths = c(2, 1))



13.4 Guide legends

By default, align_plots() won't collect any guide legends. You can use the guides argument to control which side of the guide legends should be collected. They will be collected to their original side. Here, we use patch_titles() to indicate the guide legend position (instead of using ggtitle()). patch_titles() can add titles on four sides, and the title will be placed between the plot panel and the guide legend.

```
p_right <- ggplot(mtcars) +</pre>
    geom_point(aes(hp, wt, colour = mpg)) +
    patch_titles("right") +
    labs(color = "right")
p_top <- p_right +</pre>
    patch_titles("top") +
    scale_color_continuous(
        name = "top",
        guide = guide_colorbar(position = "top")
p_left <- p_right +</pre>
    patch_titles("left") +
    scale_color_continuous(
        name = "left",
        guide = guide_colorbar(position = "left")
p_bottom <- p_right +</pre>
    patch_titles("bottom") +
    scale_color_continuous(
        name = "bottom",
        guide = guide_colorbar(position = "bottom")
align_plots(p_right, p_bottom, p_top, p_left, guides = "tlbr")
```





If align_plots() is nested in another align_plots(), the nested align_plots() will inherit the guides argument from the upper-level align_plots(). And the top-level align_plots() won't collect guide legends from plots within the nested align_plots() unless the nested align_plots() collects them first.

13.5 free_guide

The free_guide() function allows you to override the guides argument for a single plot.

```
align_plots(
    free_guide(p_right, NULL),
    free_guide(p_bottom, NULL),
    free_guide(p_top, NULL),
    free_guide(p_left, NULL),
    guides = "tlbr"
)
```



You can also specify which guide positions to be collected for individual plots.

```
align_plots(
    free_guide(p_right, "r"),
    free_guide(p_bottom, "b"),
    free_guide(p_top, "t"),
    free_guide(p_left, "l")
)
```





Part II Advanced

14 ggoncoplot

The ggoncoplot() function generates oncoPrint visualizations that display genetic alterations in a matrix format. This function is especially useful for visualizing complex genomic data, such as mutations, copy number variations, and other genomic alterations in cancer research.

```
library(ggalign)
#> Loading required package: ggplot2
```

14.1 Input data

The input should be a character matrix which encodes the alterations, you can use string of ";", ":", ",", and "|" to separate multiple alterations. Internally, ggoncoplot() will use fortify_matrix() to get such matrix.

```
mat <- read.table(</pre>
    textConnection(
         "s1,s2,s3
          g1, snv; indel, snv, indel
          g2,,snv;indel,snv
          g3,snv,,indel;snv"
    ),
    row.names = 1, header = TRUE, sep = ",", stringsAsFactors = FALSE
)
mat
                                              s3
#>
                        s1
                                   s2
#>
             g1 snv; indel
                                           indel
#>
                           snv;indel
             g2
             g3
                       snv
                                      indel; snv
```

A basic oncoprint can be generated as follows:

ggoncoplot(mat)



14.2 oncoPrint Customization

By default, all alterations are represented with tiles of equal width and height, which may lead to overlapping. You can control the width and height of the tiles using the map_width and map_height arguments (we will introduce another more effective ways to handle this in the Specialized Geoms section):



By default, all empty string will be converted to NA value and ggplot2 will translate the NA values and render it in the legend. To prevent this, you can turn off the translation using na.translate = FALSE:

```
ggoncoplot(mat, map_width = c(snv = 0.5), map_height = c(indel = 0.9)) +
scale_fill_brewer(palette = "Dark2", na.translate = FALSE)
```



The function automatically reorders rows and columns based on alteration frequency. Disable this with reorder_row = FALSE and reorder_column = FALSE.

You can further customize layouts using methods from vignette("layout-customize").

```
ggoncoplot(mat,
    map_width = c(snv = 0.5), map_height = c(indel = 0.9),
    reorder_row = FALSE, reorder_column = FALSE
) +
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE)
```



14.3 Advanced Data Handling

ggoncoplot() is built on ggheatmap() with a default scheme_data() (see vignette("plot-options") for details), which splits alterations into separate entries for visualization. (See ggplot2 specification section in ggheatmap() for the input data of scheme_data()):



By default, the scheme_data() is inherited from the parent layout if the data of the plot is inherited from the layout. You can apply the parent scheme_data() first and then apply another transformation by setting inherit = TRUE. This functionality is especially useful when working with ggoncoplot().

```
ggoncoplot(mat, map_width = c(snv = 0.5), map_height = c(indel = 0.9)) +
    anno_top(size = 0.2) +
    ggalign() +
    # by setting `inherti = TRUE`, we apply the parent layout `scheme_data()`
    # (from the `ggoncoplot()` layout) firstly, which will split the alteration
    # string and unnested the columns.
    # Here: We then remove `NA` value
    scheme_data(~ subset(.x, !is.na(value)), inherit = TRUE) +
    geom_bar(aes(.x, after_stat(count), fill = value)) +
    # note: `ggoncoplot()` use `geom_tile()` to draw the oncoPrint,
    # the guide is different from `geom_bar()`, though both looks
    # like the same, we remove the guide of `geom_bar()` plot
    guides(fill = "none") &
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE)
```



14.4 Integration with maftools

The ggoncoplot() function supports MAF objects from the maftools package using the fortify_matrix.MAF() method. It includes options to control data generation for ggoncoplot(), such as drawing the top n_top genes.

```
# load data from `maftools`
laml.maf <- system.file("extdata", "tcga_laml.maf.gz", package = "maftools")
# clinical information containing survival information and histology. This is optional
laml.clin <- system.file("extdata", "tcga_laml_annot.tsv", package = "maftools")
laml <- maftools::read.maf(
    maf = laml.maf,
    clinicalData = laml.clin,
    verbose = FALSE
)
# Visualizing the Top 20 Genes
ggoncoplot(laml, n_top = 20) +
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE) +
    theme_no_axes("x")</pre>
```



By default, samples without alterations in the selected genes are removed. To include all samples, set remove_empty_samples = FALSE:

```
align_plots(
    ggoncoplot(laml, n_top = 20L, remove_empty_samples = FALSE) +
        scale_fill_brewer(palette = "Dark2", na.translate = FALSE) +
        theme_no_axes("x") +
        ggtitle("Keep empty samples"),
    ggoncoplot(laml, n_top = 20L, remove_empty_samples = TRUE) +
        scale_fill_brewer(palette = "Dark2", na.translate = FALSE) +
        theme_no_axes("x") +
        ggtitle("Remove empty samples"),
    ncol = 1L,
    guides = "tlbr"
)
```



14.5 Specialized Geoms

14.5.1 geom_subtile()

When multiple alterations occur in the same sample and gene, they are combined into a single value, "Multi_Hit", by default. To visualize these alterations separately, you can set collapse_vars = FALSE. However, doing so can lead to overlapping alterations within the same cell, making the visualization cluttered and hard to interpret.

In such cases, disabling the default filling and defining a custom heatmap layer with <code>geom_subtile()</code> is more effective. This function subdivides each cell into smaller rectangles, allowing the distinct alterations to be clearly displayed.

```
ggoncoplot(laml, n_top = 20, collapse_vars = FALSE, filling = FALSE) +
    geom_subtile(aes(fill = value), direction = "v") +
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE) +
    theme_no_axes("x")
#> `geom_subtile()` subdivide tile into a maximal of 3 rectangles
```



We focus exclusively on cells with multiple alterations to highlight the differences, by filtering the data before applying geom_subtile():

```
ggoncoplot(laml, n_top = 20, collapse_vars = FALSE, filling = FALSE) +
    scheme_data(function(data) {
        dplyr::filter(data, dplyr::n() > 1L, .by = c(".x", ".y"))
        # we apply the parent layout `scheme_data()` first (`inherit = TRUE`),
        # which will split the alteration string and unnested the columns.
    }, inherit = TRUE) +
    geom_subtile(aes(fill = value), direction = "v") +
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE) +
    theme_no_axes("x")

#> `geom_subtile()` subdivide tile into a maximal of 3 rectangles
```



14.5.2 geom_draw2()

geom_subtile() often suffices for most scenarios. However, if you require a strategy similar to that of ComplexHeatmap, consider using geom_draw2(), which offers greater flexibility for complex customizations.

Key Features of geom_draw2():

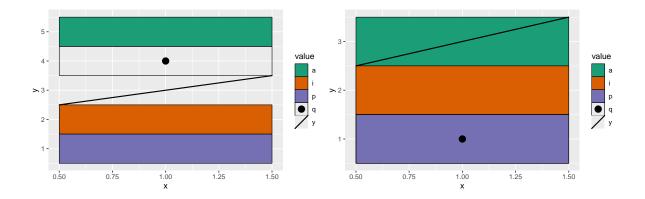
- Custom Drawing Logic: Accepts a draw aesthetic, allowing each cell to be rendered as a specific graphical object (grob).
- Manual Scaling: Requires scale_draw_manual() to map draw values to corresponding drawing functions.
- Flexible Aesthetic Mapping: Functions mapped through scale_draw_manual() can utilize any number of ggplot2 aesthetics and output custom graphical objects (grob). Beyond the x, y, xmin, xmax, width, and height aesthetics, you may want to rely solely on mapped aesthetics (aes()). Unmapped aesthetics will default to ggplot2's behavior, which might not produce the desired outcome. And always use native unit.
- :) Sorry, I cannot deal with a pretty example for this, but the function provided in values argument of scale_draw_manual operates similarly to alter_fun in ComplexHeatmap. Currently, geom_subtile() performs well enough for most cases, making geom_draw2() somewhat cumbersome to use. Managing alteration types, especially in the case of overlapping

alterations, requires meticulous design to ensure that the visual elements do not interfere with each other.

```
library(grid)
draw_mapping <- list(</pre>
    function(x, y, width, height, fill) {
        rectGrob(x, y,
            width = width, height = height,
            gp = gpar(fill = fill),
            default.units = "native"
        )
    },
    function(x, y, width, height, fill) {
        rectGrob(x, y,
            width = width, height = height,
            gp = gpar(fill = fill),
            default.units = "native"
    },
    function(x, y, width, height, fill) {
        rectGrob(x, y,
            width = width, height = height,
            gp = gpar(fill = fill),
            default.units = "native"
        )
    },
    function(x, y, width, height, shape) {
        gList(
            pointsGrob(x, y, pch = shape),
            # To ensure the rectangle color is shown in the legends, you
            # must explicitly provide a color argument and include it in
            # the `gpar()` of the graphical object
            rectGrob(x, y, width, height,
                gp = gpar(col = "black", fill = NA)
        )
    },
    function(xmin, xmax, ymin, ymax) {
        segmentsGrob(
            xmin, ymin,
            xmax, ymax,
            gp = gpar(lwd = 2)
```

```
}
)
```

```
value <- sample(letters, 5L)
ggplot(data.frame(value = value, y = seq_len(5))) +
    geom_draw2(aes(x = 1, y = y, draw = value, fill = value)) +
    scale_draw_manual(values = draw_mapping) +
    scale_fill_brewer(palette = "Dark2")
ggplot(data.frame(value = c(value, value[5L]), y = c(1, 2, 3, 1, 2, 3))) +
    geom_draw2(aes(x = 1, y = y, draw = value, fill = value)) +
    scale_draw_manual(values = draw_mapping) +
    scale_fill_brewer(palette = "Dark2")</pre>
```



14.6 ggalign attributes

Except for the data used for the main plot, fortify_matrix.MAF() also attaches several useful attributes.

- gene_summary: gene summary informations. See maftools::getGeneSummary() for details.
- sample_summary: sample summary informations. See maftools::getSampleSummary() for details.
- sample_anno: sample clinical informations. See maftools::getClinicalData() for details.
- n_genes: Total of genes.
- n_samples: Total of samples.
- titv: A list of data.frames with Transitions and Transversions summary. See maftools::titv() for details.

You can extract these attributes using the ggalign_attr() function. Below is an example of how to extract the sample summary data (e.g., TMB) using ggalign_attr() and add it to the top annotation of an oncoplot:

```
ggoncoplot(laml, n_top = 20, collapse_vars = FALSE, filling = FALSE) +
    geom_subtile(aes(fill = value), direction = "v") +
    theme_no_axes("x") +
    anno_top(size = 0.2) +
    ggalign(data = function(data) {
        data <- ggalign_attr(data, "sample_summary")
        # matrix input will be automatically melted into a long foramted data
        # frame in `ggalign()` function.
        as.matrix(data[2:(ncol(data) - 1L)])
    }) +
    geom_bar(aes(.x, value, fill = .column_names),
        stat = "identity"
    ) +
    ylab("TMB") &
    scale_fill_brewer(palette = "Dark2", na.translate = FALSE)

** `geom_subtile()` subdivide tile into a maximal of 3 rectangles</pre>
```



We can draw the example from maftools vignette.

```
ggoncoplot(laml, n top = 20, collapse vars = FALSE, filling = FALSE) +
    geom subtile(aes(fill = value), direction = "v") +
   theme no axes("x") +
   # since legends from geom_tile (oncoPrint body) and `geom_bar`
    # is different, though both looks like the same, the internal
    # won't merge the legends. we remove the legends of oncoPrint body
    guides(fill = "none") +
    # add top annotation
    anno_top(size = 0.2) +
    ggalign(data = function(data) {
        data <- ggalign_attr(data, "sample_summary")</pre>
        # matrix input will be automatically melted into a long foramted data
        # frame in `ggalign()` function.
        as.matrix(data[2:(ncol(data) - 1L)])
   }) +
    geom_bar(aes(.x, value, fill = .column_names),
        stat = "identity"
   ylab("TMB") +
   # add right annotation
   anno right(size = 0.2) -
    # remove bottom spaces of the right annotation when aligning
    scheme align(free spaces = "b") +
    # add the text percent for the alterated samples in the right annotation
    ggalign(data = function(data) {
        # Atomic vector will be put in the `value` column of the data frame.
        ggalign_attr(data, "gene_summary")$AlteredSamples /
            ggalign_attr(data, "n_samples")
    }) +
    geom_text(aes(1, label = scales::label_percent()(value)), hjust = 1) +
    scale_x_continuous(
        expand = expansion(),
        name = NULL, breaks = NULL,
        limits = c(0, 1)
    theme(plot.margin = margin()) +
    # add the bar plot in the right annotation
    ggalign(data = function(data) {
        data <- ggalign_attr(data, "gene_summary")</pre>
        # matrix input will be automatically melted into a long foramted data
        # frame in `ggalign()` function.
        as.matrix(data[2:8])
```

```
geom_bar(aes(value, fill = .column_names),
        stat = "identity",
        orientation = "y"
    ) +
    xlab("No. of samples") -
    # we apply the scale mapping to the top and right annotation: `position = "tr"`
    # and the main plot: `main = TRUE`
    with_quad(
        scale_fill_brewer("Mutations",
            palette = "Dark2", na.translate = FALSE
        ),
        position = "tr",
        main = TRUE
    ) +
    # add bottom annotation
    anno bottom(size = 0.2) +
    # add bar plot in the bottom annotation
    ggalign(data = function(data) {
        data <- ggalign_attr(data, "titv")$fraction.contribution</pre>
        # matrix input will be automatically melted into a long foramted data
        # frame in `ggalign()` function.
        as.matrix(data[2:7])
    }) +
    geom_bar(aes(y = value, fill = .column_names), stat = "identity") +
    ylab("Ti/Tv") +
    scale_fill_brewer("Ti/Tv", palette = "Set2")
#> `geom_subtile()` subdivide tile into a maximal of 3 rectangles
#> Warning: Removed 24 rows containing missing values or values outside the scale range
#> (`geom_bar()`).
```



14.7 Integration with GISTIC results from maftools

The package also includes a fortify_matrix.GISTIC() method designed to handle GISTIC objects from the maftools package. This allows you to seamlessly apply the same operations to visualize GISTIC results. The following ggalign attributes are generated as part of the analysis:

- sample anno: sample clinical informations provided in sample anno argument.
- sample_summary: sample copy number summary informations. See data@cnv.summary for details.
- cytoband_summary: cytoband summary informations. See data@cytoband.summary for details.
- gene summary: gene summary informations. See data@gene.summary for details.
- summary: A data frame of summary information. See data@summary for details.

```
# Ensure the maftools package is installed and load the example GISTIC data
all.lesions <- system.file("extdata", "all_lesions.conf_99.txt", package = "maftools")
amp.genes <- system.file("extdata", "amp_genes.conf_99.txt", package = "maftools")
del.genes <- system.file("extdata", "del_genes.conf_99.txt", package = "maftools")
scores.gistic <- system.file("extdata", "scores.gistic", package = "maftools")
laml.gistic <- maftools::readGistic(
    gisticAllLesionsFile = all.lesions, gisticAmpGenesFile = amp.genes,</pre>
```

```
gisticDelGenesFile = del.genes, gisticScoresFile = scores.gistic
)
#> -Processing Gistic files..
#> --Processing amp_genes.conf_99.txt
#> --Processing del_genes.conf_99.txt
#> --Processing scores.gistic
#> --Summarizing by samples
ggoncoplot(laml.gistic) +
    scale_fill_brewer("CNV", palette = "Dark2", na.translate = FALSE) +
    theme_no_axes("x")
```



Part III

Cases

15 Complete examples

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))</pre>
```

15.1 Simple heatmap

```
ggheatmap(small_mat)
#> > heatmap built with `geom_tile()`
```



15.2 heatmap layout customize

15.2.1 Based on dendrogram

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(aes(color = branch), k = 3) +
   geom_point(aes(color = branch, y = y)) +
   scale_color_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
```



15.2.2 Based on kmeans

```
ggheatmap(small_mat) +
   anno_top() +
   align_kmeans(3L)
#> > heatmap built with `geom_tile()`
```



15.2.3 Based on a group variable

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE))
#> > heatmap built with `geom_tile()`
```



15.2.4 Based on an ordering weights

Here, we ordered the heatmap rows based on the row means.

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans)
#> > heatmap built with `geom_tile()`
```



15.3 Heatmap annotation plot

```
ggheatmap(small_mat) +
   anno_top() +
   align_dendro(aes(color = branch), k = 3) +
   geom_point(aes(color = branch, y = y)) +
   scale_color_brewer(palette = "Dark2") +
   ggalign(mapping = aes(y = value)) +
   geom_boxplot(aes(factor(.x), fill = .panel)) +
   scale_fill_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(small_mat) +
   anno_top(size = 0.5) +
   align_dendro(aes(color = branch), k = 3L) +
   ggalign(rowSums, aes(y = value)) +
   geom_bar(stat = "identity", aes(fill = factor(.panel))) +
   scale_fill_brewer(name = NULL, palette = "Dark2") +
   anno_left(size = 0.5) +
   align_dendro(aes(color = branch), size = 0.5, k = 4L) +
   ggalign(rowSums, aes(x = value)) +
   geom_bar(
        aes(y = .y, fill = factor(.y)),
        stat = "identity",
        orientation = "y"
   ) +
   scale_fill_brewer(name = NULL, palette = "Paired", guide = "none")
#> > heatmap built with `geom_tile()`
```



15.4 Multiple heatmaps

15.4.1 Horizontal layout

```
(stack_alignh(small_mat) +
    ggheatmap() +
    ggheatmap() &
    theme(axis.text.x = element_text(angle = -60, hjust = 0))) +
    stack_active() +
    align_dendro(aes(color = branch), k = 4L, size = 0.2) +
    scale_color_brewer(palette = "Dark2")
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



15.4.2 Vertical layout

```
stack_alignv(small_mat) -
    scheme_theme(
        axis.text.x = element_blank(),
        axis.ticks.x = element_blank()
) +
    align_dendro(aes(color = branch), k = 4L, size = 0.2) +
    scale_color_brewer(palette = "Dark2") +
    ggheatmap() +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0))
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



15.5 marginal plots

```
ggside(mpg, aes(displ, hwy, colour = class)) -
    # set default theme for all plots in the layout
    scheme_theme(theme_bw()) +
    geom_point(size = 2) +
    # add top annotation
```

```
anno_top(size = 0.3) -
# set default theme for the top annotation
scheme_theme(theme_no_axes("tb")) +
# add a plot in the top annotation
ggalign() +
geom_density(aes(displ, y = after_stat(density), colour = class), position = "stack") +
anno_right(size = 0.3) -
# set default theme for the right annotation
scheme_theme(theme_no_axes("lr")) +
# add a plot in the right annotation
ggalign() +
geom_density(aes(x = after_stat(density), hwy, colour = class),
    position = "stack"
) +
theme(axis.text.x = element_text(angle = 90, vjust = .5)) &
scale_color_brewer(palette = "Dark2")
```



Part IV ComplexHeatmap

In this chapter, we'll use ggalign to draw all the heatmap in https://jokergoo.github.io/ComplexHeatmap-reference/book/index.html.

The chapter is divided into several sections, following the structure of the original book.

16 A Single Heatmap

In this thread, we'll use ggalign to draw all the heatmap in https://jokergoo.github.io/ComplexHeatmap-reference/book/a-single-heatmap.html

```
library(ggalign)
#> Loading required package: ggplot2
```

```
set.seed(123)
nr1 <- 4
nr2 <- 8
nr3 <- 6
nr \leftarrow nr1 + nr2 + nr3
nc1 <- 6
nc2 <- 8
nc3 <- 10
nc \leftarrow nc1 + nc2 + nc3
mat <- cbind(</pre>
    rbind(
        matrix(rnorm(nr1 * nc1, mean = 1, sd = 0.5), nrow = nr1),
        matrix(rnorm(nr2 * nc1, mean = 0, sd = 0.5), nrow = nr2),
        matrix(rnorm(nr3 * nc1, mean = 0, sd = 0.5), nrow = nr3)
    ),
    rbind(
        matrix(rnorm(nr1 * nc2, mean = 0, sd = 0.5), nrow = nr1),
        matrix(rnorm(nr2 * nc2, mean = 1, sd = 0.5), nrow = nr2),
        matrix(rnorm(nr3 * nc2, mean = 0, sd = 0.5), nrow = nr3)
    ),
    rbind(
        matrix(rnorm(nr1 * nc3, mean = 0.5, sd = 0.5), nrow = nr1),
        matrix(rnorm(nr2 * nc3, mean = 0.5, sd = 0.5), nrow = nr2),
        matrix(rnorm(nr3 * nc3, mean = 1, sd = 0.5), nrow = nr3)
    )
mat <- mat[sample(nr, nr), sample(nc, nc)]</pre>
rownames(mat) <- paste0("row", seq_len(nr))
colnames(mat) <- paste0("column", seq_len(nc))</pre>
```

16.1 Colors

It is important to note that the ComplexHeatmap package reorders the dendrogram by default, while align_dendro() in ggalign does not modify the tree layout.

Another key difference is in how the two packages treat the starting point. ggalign considers the left-bottom as the starting point, whereas ComplexHeatmap starts from the left-top. When reordering the dendrogram, ComplexHeatmap does so in decreasing order, while ggalign uses an ascending order.

To modify colors in the heatmap, you can use the scale_fill_*() function from ggplot2, which provides a flexible way and enriched pallete to adjust color schemes.

```
dim(mat)
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
# ComplexHeatmap::Heatmap(mat)
#> [1] 18 24
```

```
ggheatmap(mat) +
    scale_fill_gradient2(low = "green", high = "red") +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro() +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



oob argument in the scale_fill_* function can be used to deal with the outliers.

```
mat2 <- mat
mat2[1, 1] <- 100000
ggheatmap(mat2) +
    scale_fill_gradient2(
    low = "green", high = "red",
    limits = c(-2, 2),</pre>
```

```
oob = scales::squish
) +
theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
anno_right(size = unit(15, "mm")) +
align_dendro(reorder_dendrogram = TRUE) +
theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
anno_top(size = unit(15, "mm")) +
align_dendro(reorder_dendrogram = TRUE) +
quad_active() &
theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



We can use align_plots() to arrange them.

```
h1 <- ggheatmap(mat) +
    scale_fill_gradient2(name = "mat", low = "green", high = "red") +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &</pre>
```

```
theme(plot.margin = margin())
h2 \leftarrow ggheatmap(mat / 4) +
    scale_fill_gradient2(
        name = mat/4, limits = c(-2, 2L),
        oob = scales::squish,
        low = "green", high = "red"
    ) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
h3 <- ggheatmap(abs(mat)) +
    scale_fill_gradient2(name = "abs(mat)", low = "green", high = "red") +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
align_plots(h1, h2, h3, ncol = 2L)
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    scale_fill_gradientn(colors = rev(rainbow(10))) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
```



For character matrix, you can use ggplot2 discrete filling scales.

```
discrete_mat <- matrix(sample(1:4, 100, replace = TRUE), 10, 10)
colors <- structure(1:4, names = c("1", "2", "3", "4")) # black, red, green, blue
ggheatmap(discrete_mat, aes(fill = factor(value))) +
    scale_fill_manual(values = colors) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
discrete_mat <- matrix(sample(letters[1:4], 100, replace = TRUE), 10, 10)
colors <- structure(1:4, names = letters[1:4])
ggheatmap(discrete_mat) +
    scale_fill_manual(values = colors)
#> > heatmap built with `geom_tile()`
```



```
mat_with_na <- mat</pre>
na_index <- sample(c(TRUE, FALSE),</pre>
    nrow(mat) * ncol(mat),
    replace = TRUE, prob = c(1, 9)
)
mat_with_na[na_index] <- NA</pre>
ggheatmap(mat_with_na) +
    scale_fill_gradient2(low = "blue", high = "red", na.value = "black") +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



We won't compare the LAB and RGB space. If you want to convert color between different color space, try to use farver pacakge.

In ggplot2, you can use panel.border argument in theme() function to control the Heatmap body border.

```
ggheatmap(mat) +
    theme(
        axis.text.x = element_text(angle = -60, hjust = 0),
        panel.border = element_rect(linetype = "dashed", fill = NA)
    ) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



You can use the filling argument to turn off the heatmap cell filling, allowing you to customize the heatmap body geoms. Use the color aesthetic to specify the cell border color and the linewidth aesthetic to set the border width.

```
ggheatmap(mat, filling = NULL) +
    geom_tile(aes(fill = value), color = "white") +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) &
    theme(plot.margin = margin())
```



To draw a blank heatmap body:

```
ggheatmap(mat, filling = NULL) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) &
    theme(plot.margin = margin())
```



16.2 Titles

We can use patch_titles() to add titles around each border of the plot. You can use theme() to control the text appearance.

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    patch_titles(right = "I am a row title") +
    theme(plot.patch_title.right = element_text(face = "bold", size = 16)) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    patch_titles(top = "I am a column title") +
    theme(plot.patch_title.top = element_text(face = "bold", size = 16)) &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.3 Clustering

16.3.1 Distance methods

```
# ComplexHeatmap::Heatmap(mat,
# name = "mat", clustering_distance_rows = "pearson",
# column_title = "pre-defined distance method (1 - pearson)"
# )

ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_dendro(distance = "pearson", reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    patch_titles(top = "pre-defined distance method (1 - pearson)") +
    theme(plot.patch_title.top = element_text(face = "bold", size = 16)) &
    theme(plot.margin = margin())

#> > heatmap built with `geom_tile()`
```

pre-defined distance method (1 - pearson)



```
# ComplexHeatmap::Heatmap(mat,
# name = "mat", clustering_distance_rows = function(m) dist(m),
# column_title = "a function that calculates distance matrix"
# )

ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_dendro(distance = dist, reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    patch_titles(top = "a function that calculates distance matrix") +
    theme(plot.patch_title.top = element_text(face = "bold", size = 16)) &
    theme(plot.margin = margin())

#> > heatmap built with `geom_tile()`
```

a function that calculates distance matrix



16.3.2 Clustering methods

Method to perform hierarchical clustering can be specified by method argument, Possible methods are those supported in hclust() function.



Use distance = NULL if you don't want to calculate the distance.

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_dendro(
        distance = NULL, method = cluster::diana,
        reorder_dendrogram = TRUE
) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(
        distance = NULL, method = cluster::agnes,
        reorder_dendrogram = TRUE
) &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.3.3 Render dendrograms

It's easy for ggalign to color the branches by setting the color mapping, since ggalign will add the cutree() results into the underlying data.

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_dendro(aes(color = branch), k = 2L, reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.4 Set row and column orders

We can use align_order() to set the order.

```
# ComplexHeatmap::Heatmap(mat,
# name = "mat",
# row_order = order(as.numeric(gsub("row", "", rownames(mat)))),
# column_order = order(as.numeric(gsub("column", "", colnames(mat)))),
# column_title = "reorder matrix"
# )

ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_order(as.numeric(gsub("row", "", rownames(mat)))) +
    anno_top(size = unit(15, "mm")) +
    align_order(as.numeric(gsub("column", "", colnames(mat)))) &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.5 Seriation

align_reorder() can directly take the seriate() function as the input and extract the ordering information.

```
mat2 <- max(mat) - mat
ggheatmap(mat2) +
    scale_fill_gradient2(low = "blue", high = "red", midpoint = 2L) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_reorder(seriation::seriate, method = "BEA_TSP") +
    anno_top(size = unit(15, "mm")) +
    align_reorder(seriation::seriate, method = "BEA_TSP") &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



The above code will execute seriate() twice—once for each dimension. However, since a single run of seriate() can provide the ordering for both dimensions, we can manually extract the ordering indices to avoid redundancy.

```
o <- seriation::seriate(mat2, method = "BEA_TSP")
ggheatmap(mat2) +
    scale_fill_gradient2(low = "blue", high = "red", midpoint = 2L) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(20, "mm")) +
    align_order(seriation::get_order(o, 1L)) +
    anno_top(size = unit(15, "mm")) +
    align_order(seriation::get_order(o, 2L)) &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



For more use of the seriate() function, please refer to the seriation package.

16.6 Dimension labels

ggplot2 use scales and theme to control the axis labels, Please see chapter for more details.

```
# ComplexHeatmap::Heatmap(mat,
      name = "mat", row_names_side = "left", row_dend_side = "right",
      column_names_side = "top", column_dend_side = "bottom"
#
# )
ggheatmap(mat) +
    scale_x_continuous(position = "top") +
    scale_y_continuous(position = "right") +
    theme(axis.text.x = element text(angle = 60, hjust = 0)) +
    anno_left(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    scale_x_continuous(position = "top") +
    anno_bottom(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    scale_y_continuous(position = "right") +
    quad_active() &
```

```
theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    scale_y_continuous(breaks = NULL) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(
        axis.text.x = element_text(angle = -60, hjust = 0),
        axis.text.y = element_text(face = "bold", size = 16)
) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(
        axis.text.x = element_text(angle = -60, hjust = 0),
        axis.text.y = element_text(
            face = "bold", size = 16,
            colour = c(rep("red", 10), rep("blue", 8))
        )
    ) +
   anno_right(size = unit(15, "mm")) +
   align_dendro(reorder_dendrogram = TRUE) +
   anno_top(size = unit(15, "mm")) +
   align_dendro(reorder_dendrogram = TRUE) +
   quad_active() &
   theme(plot.margin = margin())
#> Warning: Vectorized input to `element_text()` is not officially supported.
#> i Results may be unexpected or may change in future versions of ggplot2.
#> > heatmap built with `geom_tile()`
```



16.7 Heatmap split

16.7.1 Split by k-means clustering

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_kmeans(2L) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_kmeans(3L) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_kmeans(2L) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_kmeans(3L) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



The dendrogram was calculated in each group defined by kmeans.

16.7.2 Split by categorical variables

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_group(rep(c("A", "B"), 9)) +
    align_dendro(reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_group(rep(c("C", "D"), 12)) +
    align_dendro(reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.7.3 Split by dendrogram

When you splitted by a dendrogram, the cutted height will be indicated with a dashed line.

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(k = 3L, reorder_dendrogram = TRUE) +
    anno_top(size = unit(15, "mm")) +
    align_dendro(k = 2L, reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_dendro(aes(color = branch), k = 3L, reorder_dendrogram = TRUE) +
    scale_color_brewer(palette = "Dark2") +
    anno_top(size = unit(15, "mm")) +
    align_dendro(k = 2L, reorder_dendrogram = TRUE) +
    quad_active() &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.7.4 Order of slices (panels)

The order of the panels always follow the factor level. Note: the merging of dendrogram between ComplexHeatmap and ggalign is a little different.

```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_group(rep(LETTERS[1:3], 6)) +
    align_dendro(aes(color = branch),
        reorder_dendrogram = TRUE,
        reorder_group = TRUE,
        merge_dendrogram = TRUE
    ) +
    scale_color_brewer(palette = "Dark2") +
    anno_top(size = unit(15, "mm")) +
    align_group(rep(letters[1:6], 4)) +
    align_dendro(aes(color = branch),
        reorder_dendrogram = TRUE,
        reorder_group = TRUE,
        merge_dendrogram = TRUE
```

```
quad_active() -
with_quad(theme(strip.text = element_text()), "tr") &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



```
ggheatmap(mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(size = unit(15, "mm")) +
    align_group(rep(LETTERS[1:3], 6)) +
    align_dendro(aes(color = branch), reorder_dendrogram = TRUE) +
    scale_color_brewer(palette = "Dark2") +
    anno_top(size = unit(15, "mm")) +
    align_group(rep(letters[1:6], 4)) +
    align_dendro(aes(color = branch), reorder_dendrogram = TRUE) +
    quad_active() -
    with_quad(theme(strip.text = element_text()), "tr") &
    theme(plot.margin = margin())
#> > heatmap built with `geom_tile()`
```



16.7.5 Titles for splitting (facet strip text)

By default, the facet strip text is removed. You can override this behavior with theme(strip.text = element_text()). Since align_group() does not create a new plot, the panel title can only be added to the heatmap plot.

waiting for complete ...

16.7.6 Graphic parameters for splitting

```
ggh4x::facet_grid2(strip = ggh4x::strip_themed(
    background_x = list(
        element_rect(fill = "red"),
        element_rect(fill = "blue"),
        element_rect(fill = "green")
    )
))

#> <ggproto object: Class FacetGrid2, FacetGrid, Facet, gg>
#> attach_axes: function
#> compute_layout: function
#> draw_back: function
```

```
#>
       draw_front: function
#>
       draw_labels: function
       draw_panels: function
#>
       finish_data: function
#>
#>
       finish_panels: function
       init_scales: function
#>
#>
       map_data: function
#>
       params: list
#>
       setup_aspect_ratio: function
       setup_axes: function
#>
#>
       setup_data: function
#>
       setup_panel_table: function
#>
       setup_params: function
       shrink: TRUE
#>
#>
       strip: <ggproto object: Class StripElemental, Strip, gg>
           assemble_strip: function
#>
#>
           build_strip: function
#>
           clip: inherit
           draw_labels: function
#>
#>
           elements: list
           finish_strip: function
#>
#>
           get_strips: function
           given_elements: list
#>
#>
           incorporate_grid: function
#>
           incorporate_wrap: function
           init_strip: function
#>
#>
           params: list
#>
           setup: function
#>
           setup_elements: function
#>
           strips: list
#>
           super: <ggproto object: Class StripElemental, Strip, gg>
#>
       train_scales: function
       vars: function
#>
#>
       vars combine: function
#>
       super: <ggproto object: Class FacetGrid2, FacetGrid, Facet, gg>
```

17 More examples

```
library(ggalign)
#> Loading required package: ggplot2
```

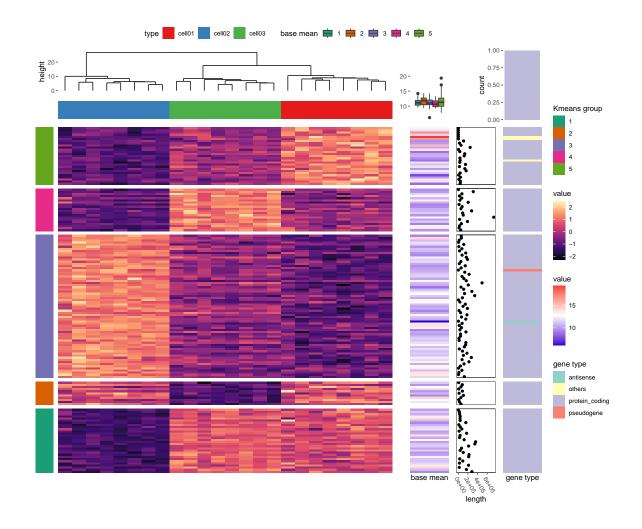
In this section, we'll use ggalign to draw all the heatmap in https://jokergoo.github.io/ComplexHeatmap-reference/book/more-examples.html.

17.1 Add more information for gene expression matrix

```
expr <- read_example("gene_expression.rds")</pre>
mat <- as.matrix(expr[, grep("cell", colnames(expr))])</pre>
base_mean <- rowMeans(mat)</pre>
mat_scaled <- t(apply(mat, 1, scale))</pre>
type <- gsub("s\\d+_", "", colnames(mat))</pre>
heat1 <- ggheatmap(mat_scaled) -</pre>
    scheme_align(free_spaces = "1") +
    scale_y_continuous(breaks = NULL) +
    scale_fill_viridis_c(option = "magma") +
    # add dendrogram for this heatmap
    anno_top() +
    align_dendro() +
    # add a block for the heatmap column
    ggalign(data = type, size = unit(1, "cm")) +
    geom_tile(aes(y = 1, fill = factor(value))) +
    scale_y_continuous(breaks = NULL, name = NULL) +
    scale_fill_brewer(
        palette = "Set1", name = "type",
        guide = guide_legend(position = "top")
    )
heat2 <- ggheatmap(base_mean, width = unit(2, "cm")) +</pre>
```

```
scale v continuous(breaks = NULL) +
    scale_x_continuous(name = "base mean", breaks = FALSE) +
    scale_fill_gradientn(colours = c("#2600D1FF", "white", "#EE3F3FFF")) +
    # set the active context of the heatmap to the top
    # and set the size of the top stack
    anno_top(size = unit(4, "cm")) +
    # add box plot in the heatmap top
    ggalign() +
    geom_boxplot(aes(y = value, fill = factor(.extra_panel))) +
    scale_x_continuous(expand = expansion(), breaks = NULL) +
    scale_fill_brewer(
        palette = "Dark2", name = "base mean",
        guide = guide_legend(position = "top")
    ) +
    theme(axis.title.y = element_blank())
heat3 <- ggheatmap(expr$type, width = unit(2, "cm")) +
    scale fill brewer(palette = "Set3", name = "gene type") +
    scale_x_continuous(breaks = NULL, name = "gene type") +
    # add barplot in the top annotation, and remove the spaces in the y-axis
    anno top() -
    scheme_align(free_spaces = "lr") +
    ggalign() +
    geom_bar(
        aes(.extra_panel, fill = factor(value)),
        position = position_fill()
    ) +
    scale_y_continuous(expand = expansion()) +
    scale fill brewer(palette = "Set3", name = "gene type", guide = "none") -
    scheme_theme(plot.margin = margin())
stack_alignh(mat_scaled) +
    stack active(sizes = c(0.2, 1, 1)) +
    # group stack rows into 5 groups
    align_kmeans(centers = 5L) +
    # add a block plot for each group in the stack
    ggalign(size = unit(1, "cm"), data = NULL) +
    geom tile(aes(x = 1, fill = factor(.panel))) +
   scale_fill_brewer(palette = "Dark2", name = "Kmeans group") +
    scale_x_continuous(breaks = NULL, name = NULL) +
    # add a heatmap plot in the stack
   heat1 +
```

```
# add another heatmap in the stack
   heat2 +
   # we move into the stack layout
   stack_active() +
   # add a point plot
    ggalign(data = expr$length, size = unit(2, "cm")) +
   geom_point(aes(x = value)) +
   labs(x = "length") +
    theme(
        panel.border = element_rect(fill = NA),
        axis.text.x = element_text(angle = -60, hjust = 0)
    # add another heatmap
   heat3 &
    theme(
        plot.background = element_blank(),
        panel.background = element_blank(),
        legend.background = element_blank()
    )
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```



17.2 The measles vaccine heatmap

```
mat <- read_example("measles.rds")
ggheatmap(mat, filling = FALSE) +
    geom_tile(aes(fill = value), color = "white") +
    scale_fill_gradientn(
        colours = c("white", "cornflowerblue", "yellow", "red"),
        values = scales::rescale(c(0, 800, 1000, 127000), c(0, 1))
    ) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right() +
    align_dendro(plot_dendrogram = FALSE) +</pre>
```

```
anno_top(size = unit(2, "cm")) +
ggalign(data = rowSums) +
geom_bar(aes(y = value), fill = "#FFE200", stat = "identity") +
scale_y_continuous(expand = expansion()) +
ggtitle("Measles cases in US states 1930-2001\nVaccine introduced 1961") +
theme(plot.title = element_text(hjust = 0.5)) +
anno_right(size = unit(2, "cm")) +
ggalign(data = rowSums) +
geom_bar(aes(x = value),
    fill = "#FFE200", stat = "identity",
    orientation = "y"
) +
scale_x_continuous(expand = expansion()) +
theme(axis.text.x = element_text(angle = -60, hjust = 0))
```

