ggalign: Bridging the Grammar of Graphics and Complex layout

Yun Peng

2024-11-24

Table of contents

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.

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 layout while still using 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 principles into advanced visualization, achieved through its object-oriented `Layout` system. Two basic `Layout` classes are available:

- 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.

Both `Layout` classes support the alignment of observations (ordinal variable). Observations refer to data points or samples, allowing for consistent alignment of corresponding data across multiple plots when using the same axis values. Depending on whether you want to align observations across multiple plots within the layout, the following variants are available:

For `StackLayout`:

- `stack_align()`: Align the observations along the stack.
- `stack_free()`: Does not align the observations.

For `QuadLayout`:

- `quad_free`/`ggside`: Never align observations.
- `quad_alignh`: Align observations in the horizontal direction.
- `quad_alignv`: Align observations in the vertical direction.
- `quad_alignb`: Align observations in both horizontal and vertical directions.

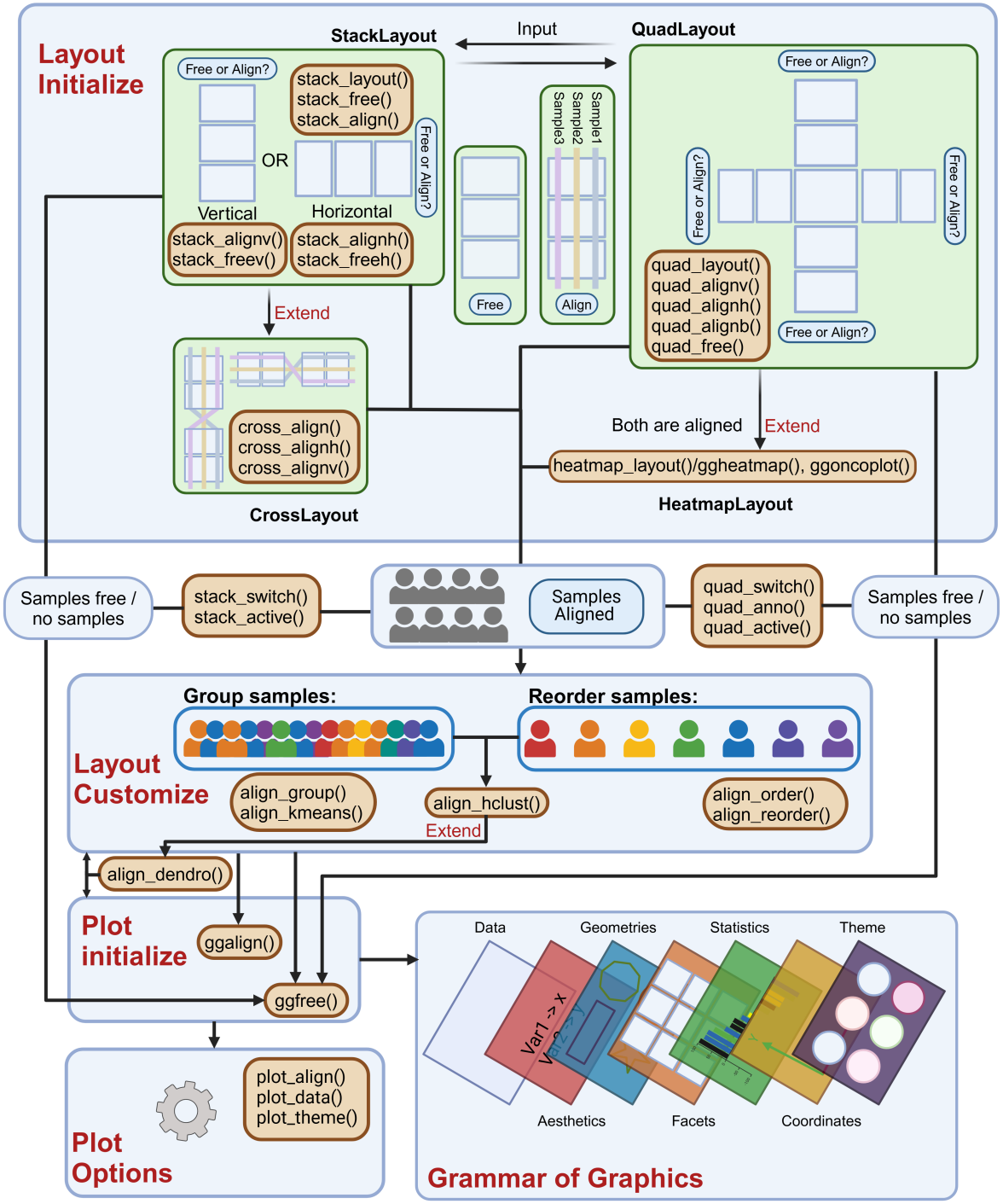


Figure 1.1: General design of ggalign

1.3 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 using:

- `stack_layout()`: Arrange Plots Horizontally or Vertically
  - `cross_align`: Arrange Plots Crosswise Horizontally or Vertically
- `quad_layout()`: Arrange Plots in the Quad-Side of a main plot
  - `ggheatmap()`: Create a Complex Heatmap.
  - `ggoncoplot()`: Create `OncoPrint` Visualizations from Genetic Alteration Data

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_hclust()`/`align_dendro()`: Reorder or group observations based on hierarchical clustering.

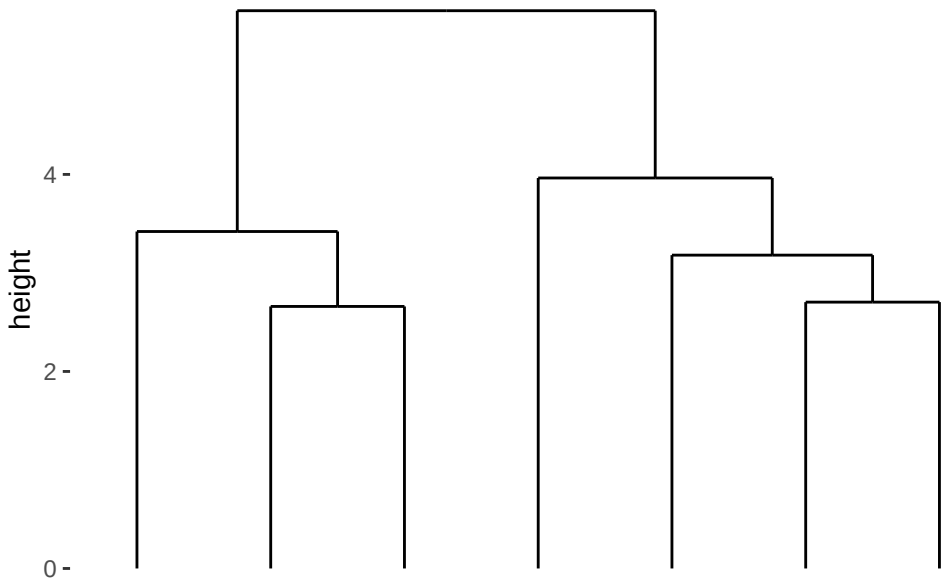
3. Adding plots with `ggalign()` or `ggfree()`, and then layer additional ggplot2 elements such as geoms, stats, or scales.

```
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)))
```

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

Here's a simple example:

```
stack_alignv(small_mat) +
    align_dendro() +
    theme(axis.text.y = element_text())
```



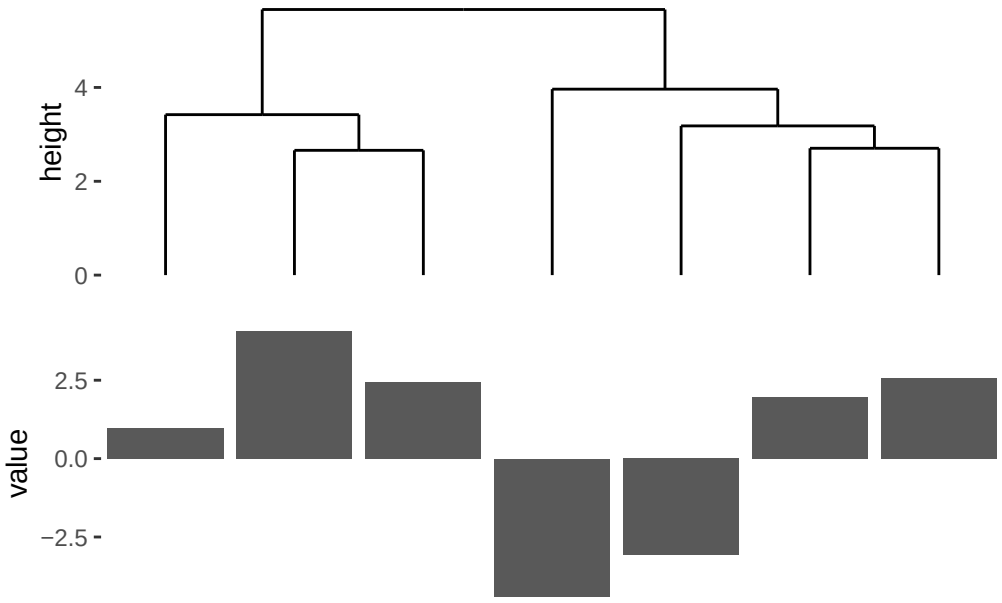
In this example:

1. We initialize a vertical stack (`stack_alignv(small_mat)`).
2. Reorder the observations based on hierarchical clustering and add a dendrogram tree (`align_dendro()`).
3. Add y-axis text (`theme(axis.text.y = element_text())`).

This produces a simple dendrogram. By default, `stack_alignv()` removes the axis text on the axis used for aligning observations. 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_alignv(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(.names, value), stat = "identity") +
    theme(axis.text.y = element_text())
```



In this example:

1. We initialize a vertical stack (`stack_alignv(small_mat)`).
2. Reorder the observations based on hierarchical clustering and add a dendrogram tree (`align_dendro()`).
3. Create a new ggplot in the layout, and use data based on the sum of the layout data (`ggalign(data = rowSums)`).
4. Add a bar layer (`geom_bar(aes(.names, value), stat = "identity")`).
5. Add y-axis text (`theme(axis.text.y = element_text())`).

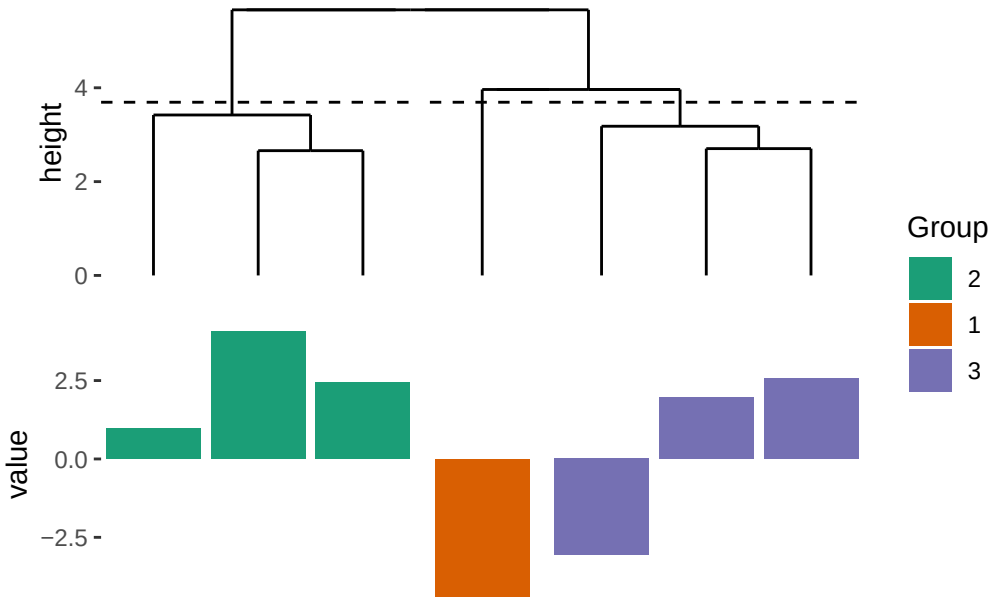
The data in the underlying `ggplot` object of `ggalign` function contains at least following columns (more details will be introduced in the later chapter):

- `.panel`: the group panel for the aligned axis. It means `x-axis` for vertical stack layout (including top and bottom annotation), `y-axis` for horizontal stack layout (including left and right annotation).
- `.x` or `.y`: the `x` or `y` coordinates
- `.names` and `.index`: A factor of the 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 `.names` 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(.names, value, fill = .panel), stat = "identity") +
    scale_fill_brewer(palette = "Dark2", name = "Group") +
    theme(axis.text.y = element_text())
```



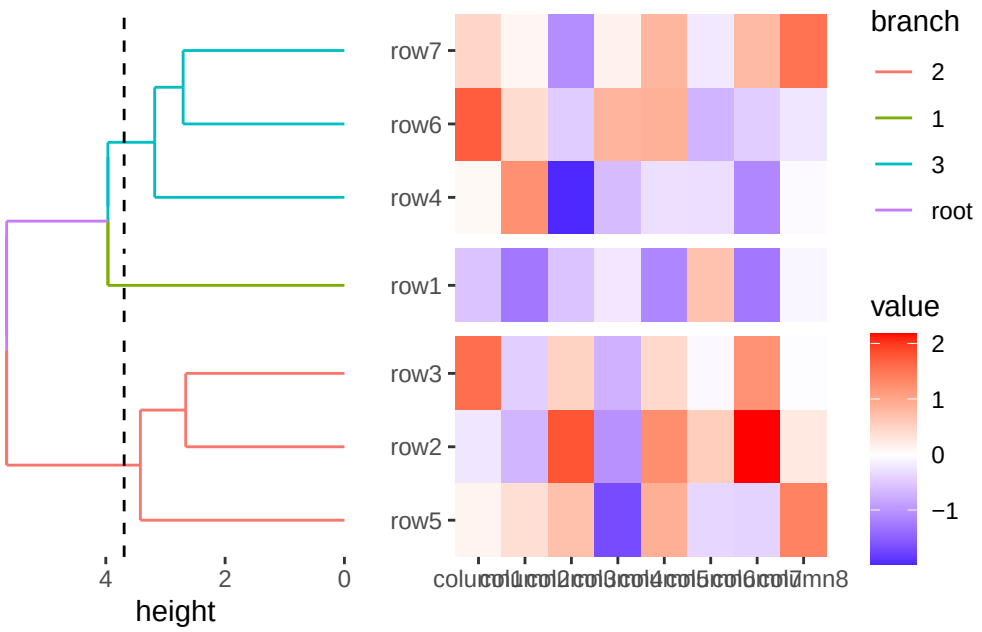
In this example:

1. We initialize a vertical stack (`stack_alignv(small_mat)`).
2. Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree (`align_dendro(k = 3)`).
3. Create a new ggplot in the layout, and use data based on the sum of the layout data (`ggalign(data = rowSums)`).
4. Add a bar layer (`geom_bar(aes(.names, value), stat = "identity")`).
5. Add fill mapping scale (`scale_fill_brewer(palette = "Dark2", name = "Group")`).
6. Add y-axis text (`theme(axis.text.y = element_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) +
    anno_left() +
    align_dendro(aes(color = branch), k = 3) +
    scale_fill_brewer(palette = "Dark2")
```

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



In this example:

1. We initialize a heatmap layout (`ggheatmap(small_mat)`).
2. 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. (`anno_left()`)
3. Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree, coloring the tree by `branch` (`align_dendro(k = 3)`).
4. Add fill mapping scale (`scale_fill_brewer(palette = "Dark2")`).

`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())`.

2 stack layout

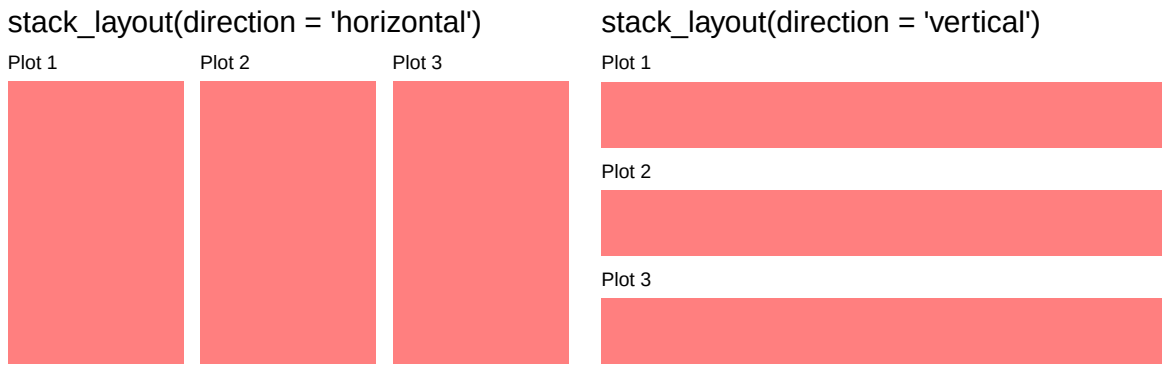
`stack_layout()` arranges plots either horizontally or vertically, and we can also use the alias `ggstack()`. Based on whether we want to align the observations, there are two types of stack layouts:

- `stack_align()`: align the observations along the stack.
- `stack_free()`: don't align the observations.

Several aliases are available for convenience:

- `stack_alignv`: Aligns the stack vertically (special case of `stack_align()`).
- `stack_alignh`: Aligns the stack horizontally (special case of `stack_align()`).
- `stack_freev`: A vertical version of `stack_free()`.
- `stack_freeh`: A horizontal version of `stack_free()`.

```
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)))
```



2.1 Input data

When aligning observations, we typically use a matrix, as it is easy to melt the matrix into a long-formatted data frame. Additionally, matrices are used to fit the observation concept, as they can be transposed (rows to columns, columns to rows), which is necessary for use in functions like `quad_layout()` and `ggheatmap()`, where observations may be aligned in both directions simultaneously.

- For `stack_free()`, a data frame is required, and the input will be automatically converted using `fortify_data_frame()` if needed.
- For `stack_align()`, 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.

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

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

In this example, we use `layout_annotation()` to insert a plot background in the entire layout.

2.2 Layout Customization

When we use `stack_align()`, it aligns the observations across multiple plots along the specified direction:

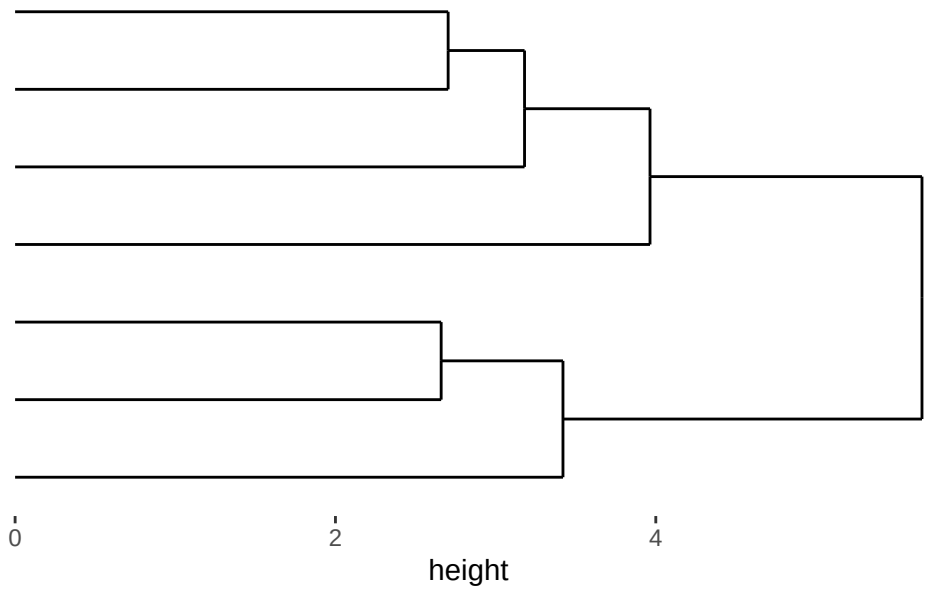
- For `stack_alignh()`: Alignment occurs along the horizontal direction (y-axis).
- For `stack_alignv()`: Alignment occurs along the vertical direction (x-axis).

The package offers a suite of `align_*` functions designed to give you precise control over the layout. These functions enable you to reorder the observations or partition the observations into multiple groups. Instead of detailing each `align_*` function individually, we will focus on the general usage and how to combine them with `stack_align()`.

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

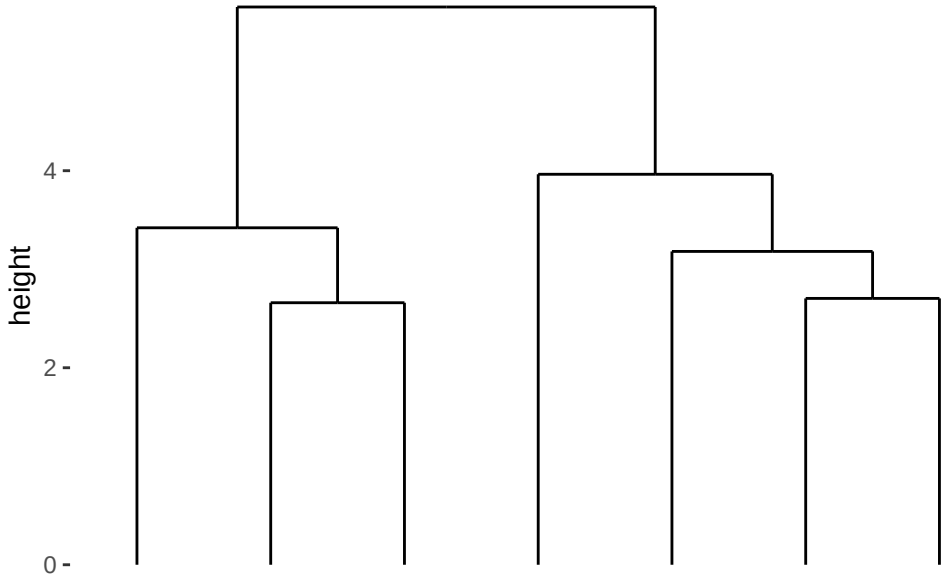
When used for `stack_alignh()`, the observations are aligned along the `y-axis`:

```
stack_alignh(small_mat) +
    align_dendro()
```



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

```
stack_alignv(small_mat) +
    align_dendro()
```

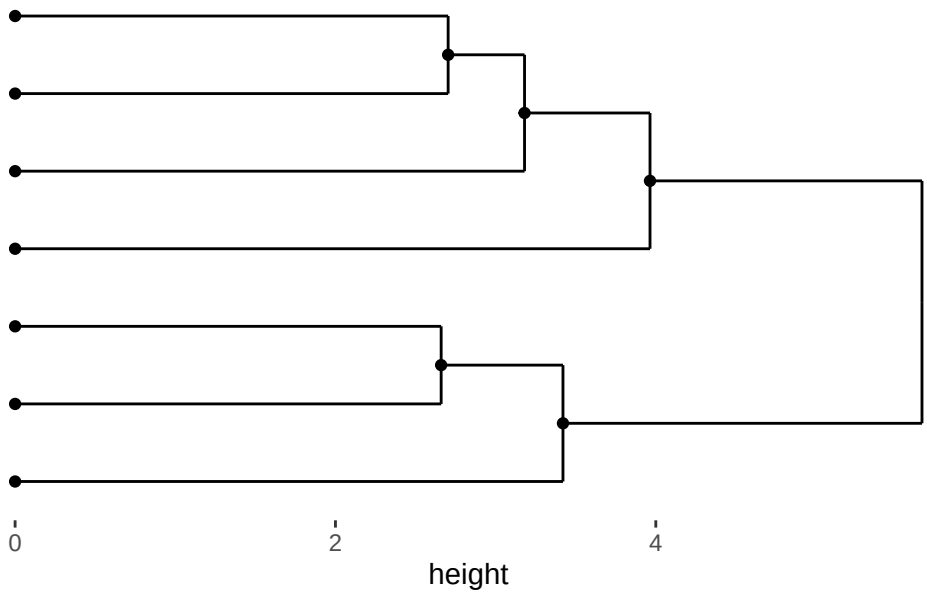


When `align_dendro()` is added to the layout, it performs following actions:

```
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 a later chapter.

```
stack_alignh(small_mat) +
    align_dendro() +
    geom_point()
```



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

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

```
Error in `stack_layout_add()`:
! Cannot add `geom_point()` to `stack_align()`
i No active plot component
i Did you forget to initialize a <ggplot> object with `ggalign()` or
  `ggfree()`?
```

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()
```

```
`align_dendro()` object:
  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)
```

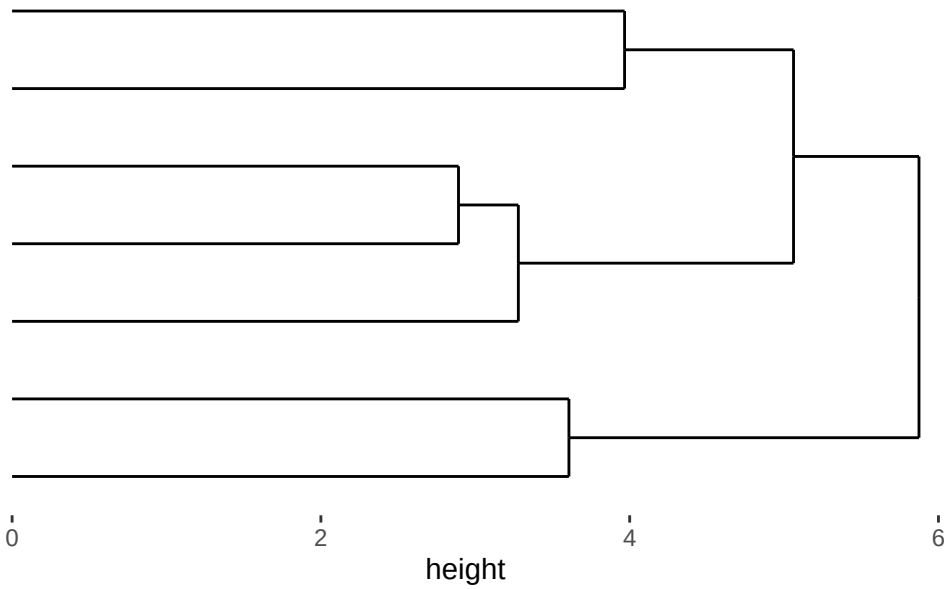
```
`align_dendro()` object:
  plot: yes
  reorder: yes
  split: yes
```

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.

```
set.seed(123)
stack_alignh(small_mat) +
    align_dendro(data = matrix(rnorm(56), nrow = 8)) +
    theme(axis.text.y = element_text())
```

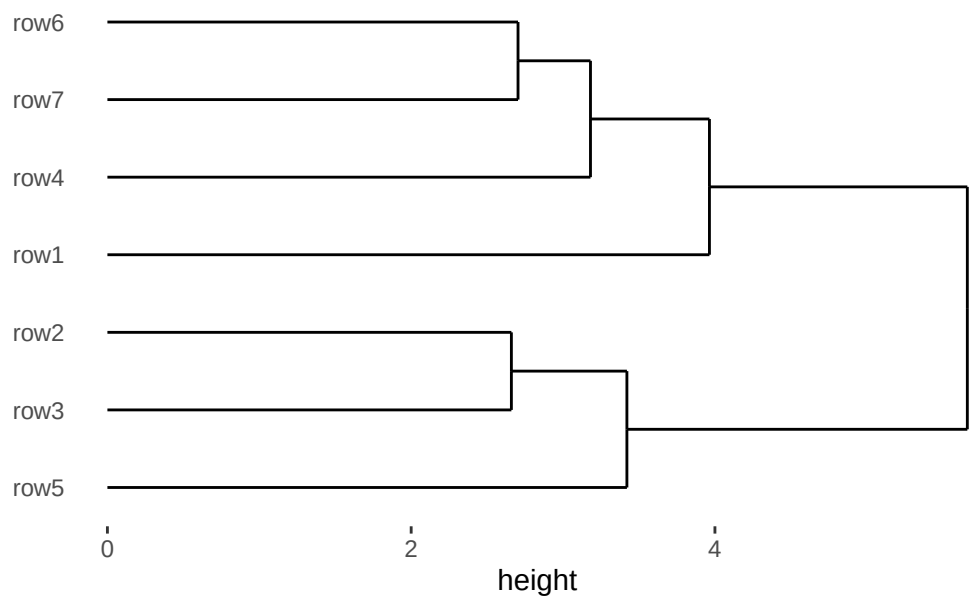
```
Error in `align()`:
! `align_dendro(data = matrix(rnorm(56), nrow = 8))` (nobs: 8) is not
  compatible with the `stack_align()` (nobs: 7)
```

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



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

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



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

2.3 Plot initialize

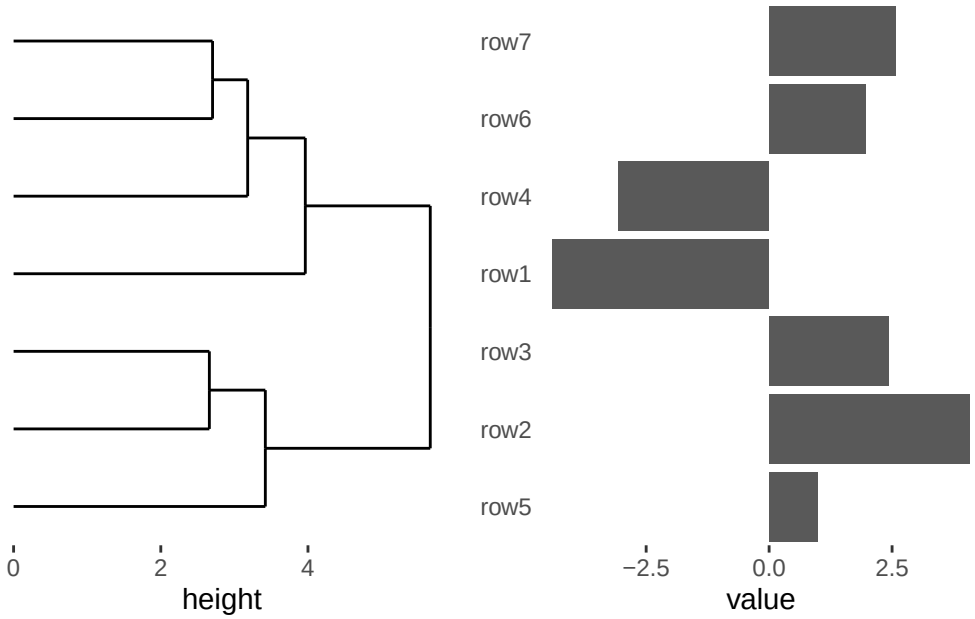
There are two primary functions for adding plots:

- `align_gg()`/`ggalign()`: Create a ggplot object and align with the layout.
- `free_gg()`/`ggfree()`: Create a ggplot object without aligning.

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

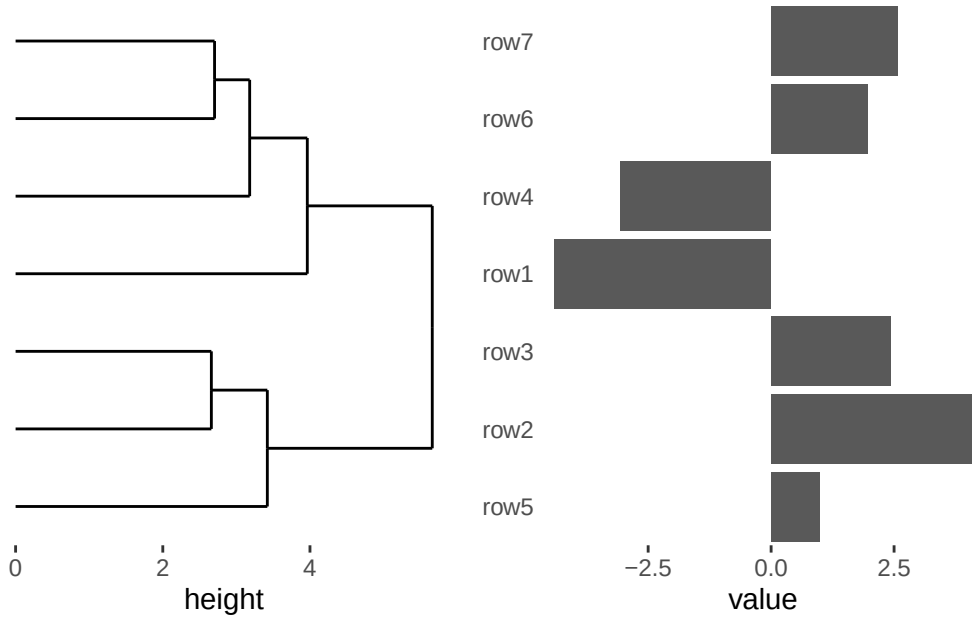
For `stack_align()`, plots can be added regardless of whether they need to align observations.

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



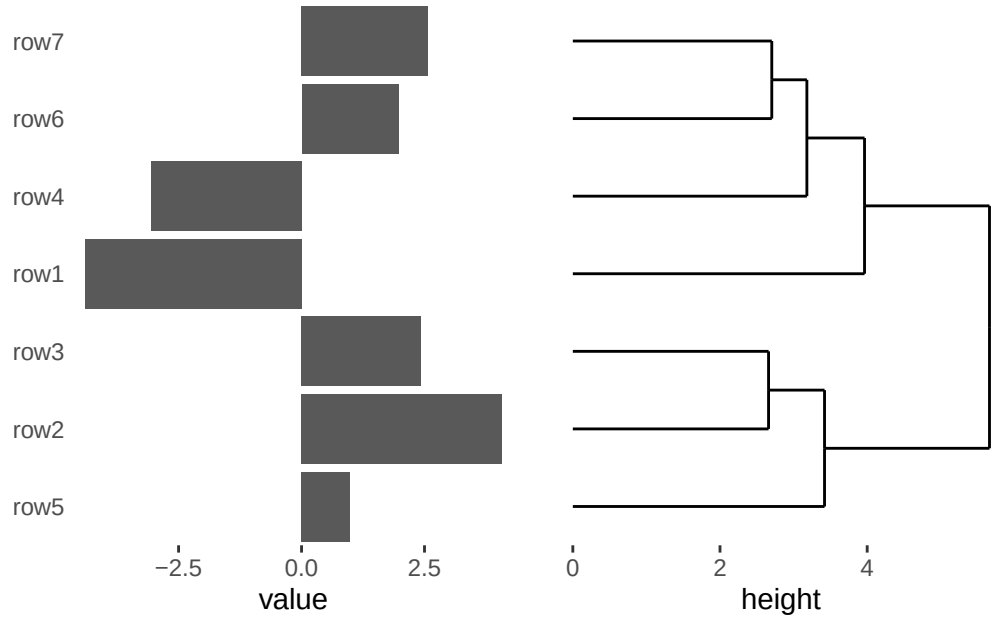
You can build the plot layer separately and then add it to the layout:

```
my_bar <- ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    theme(axis.text.y = element_text())
stack_alignh(small_mat) +
    align_dendro() +
    my_bar
```



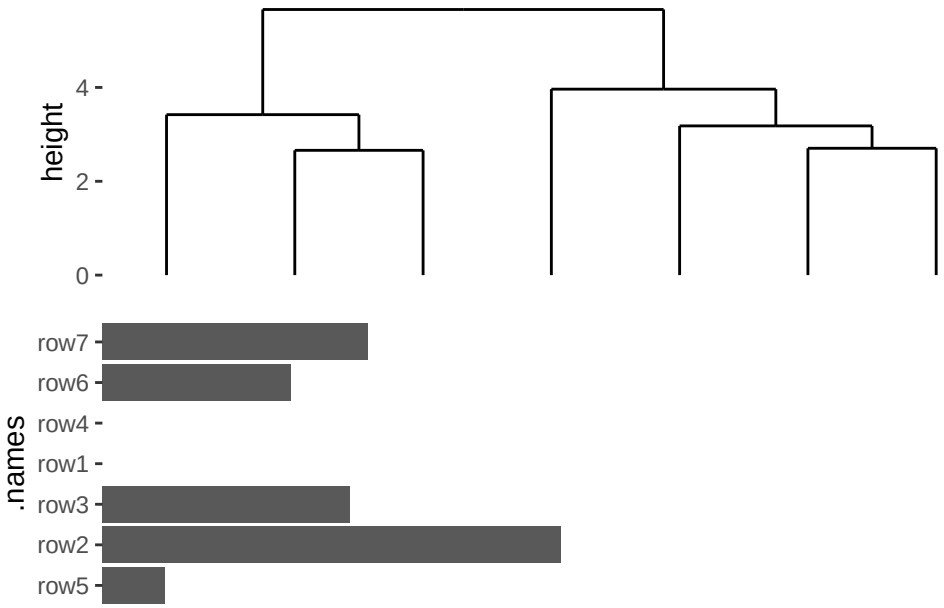
The `active` argument can also control the place of the plot area to be added. It accepts an `active()` object with the `order` argument to specify the order of the plot area.

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



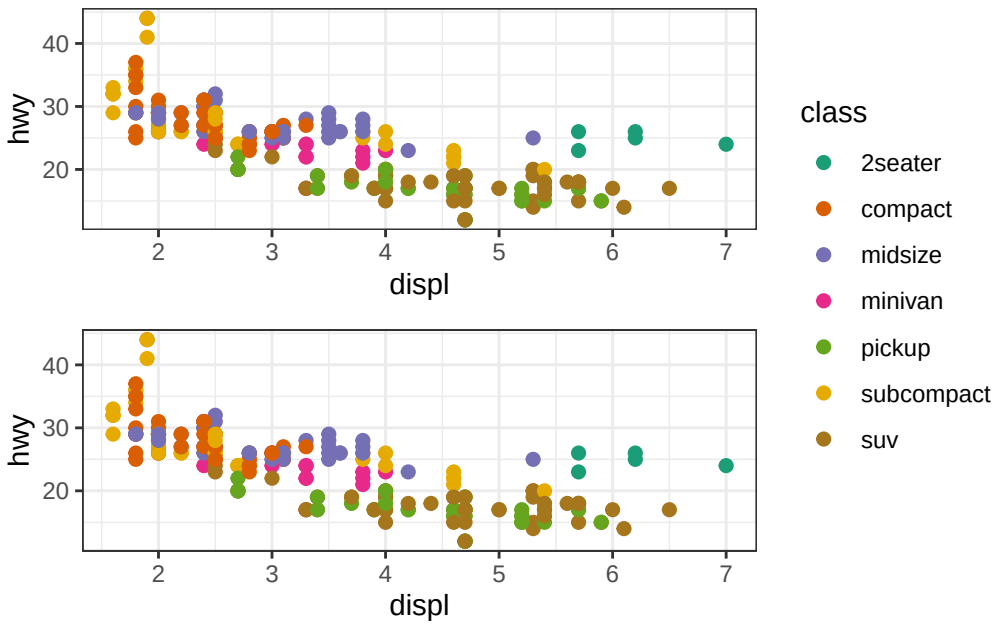
You can also stack plots vertically using `stack_alignv()`:

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



`stack_align()` can also add plot without aligning observations. `free_gg()` focuses on layout integration without enforcing strict axis alignment. `ggfree()` is an alias for `free_gg`.

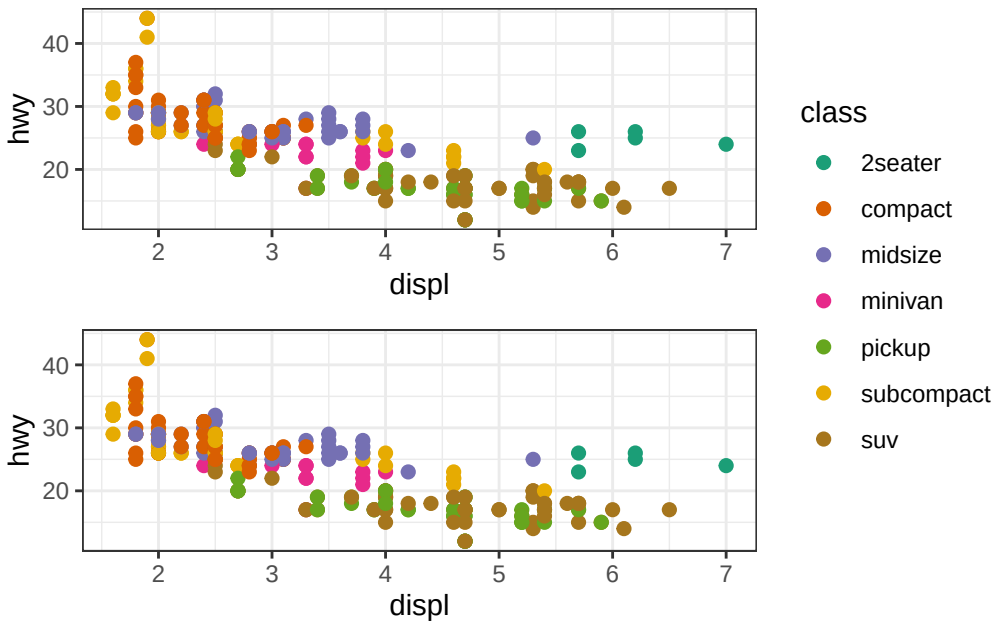
```
stack_alignv() +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```



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

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

```
stack_freev(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()
```

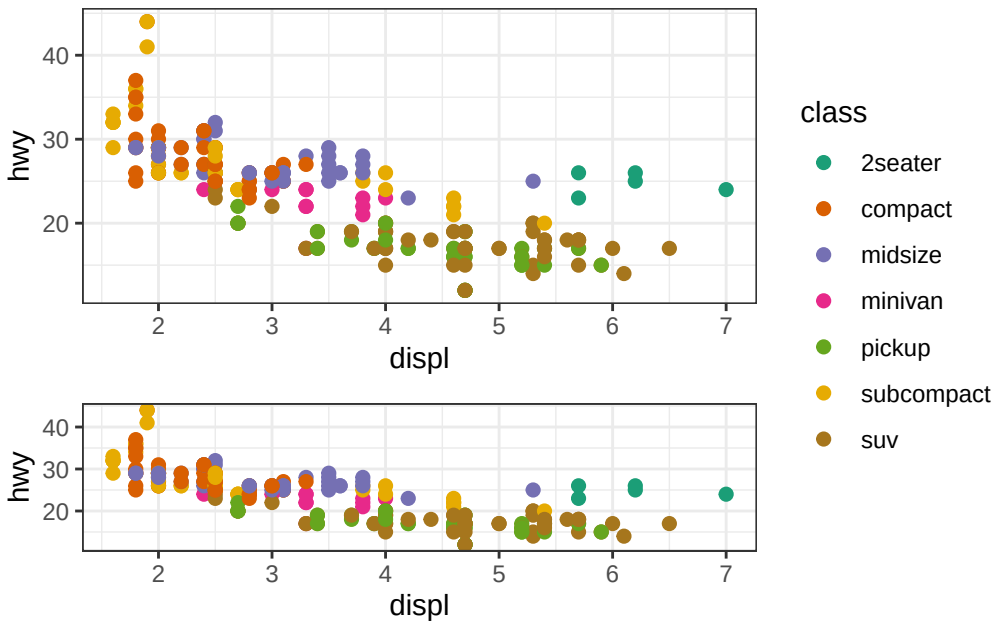


By default, `ggfree()` will also inherit data from the layout and call `fortify_data_frame()` to convert the data to a data frame. So, note that if the layout data is a matrix, it will be converted into a long-formatted data frame.

2.4 Plot Size

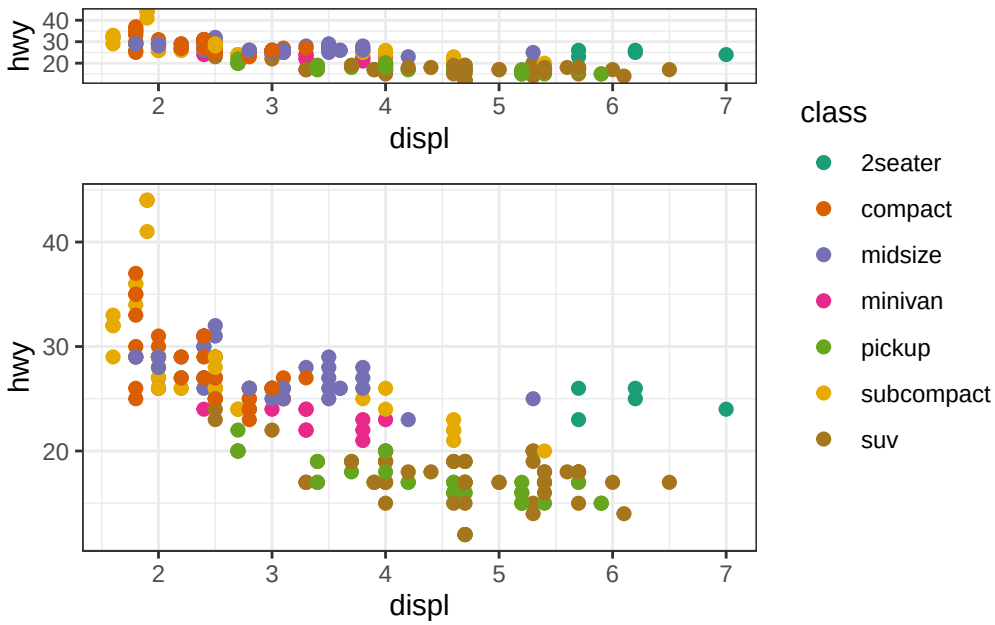
Both `ggalign()` and `ggfree()` functions have a `size` argument to control the relative `width` (for horizontal stack layout) or `height` (for vertical stack layout) of the plot's panel area.

```
stack_freev(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_freev(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.

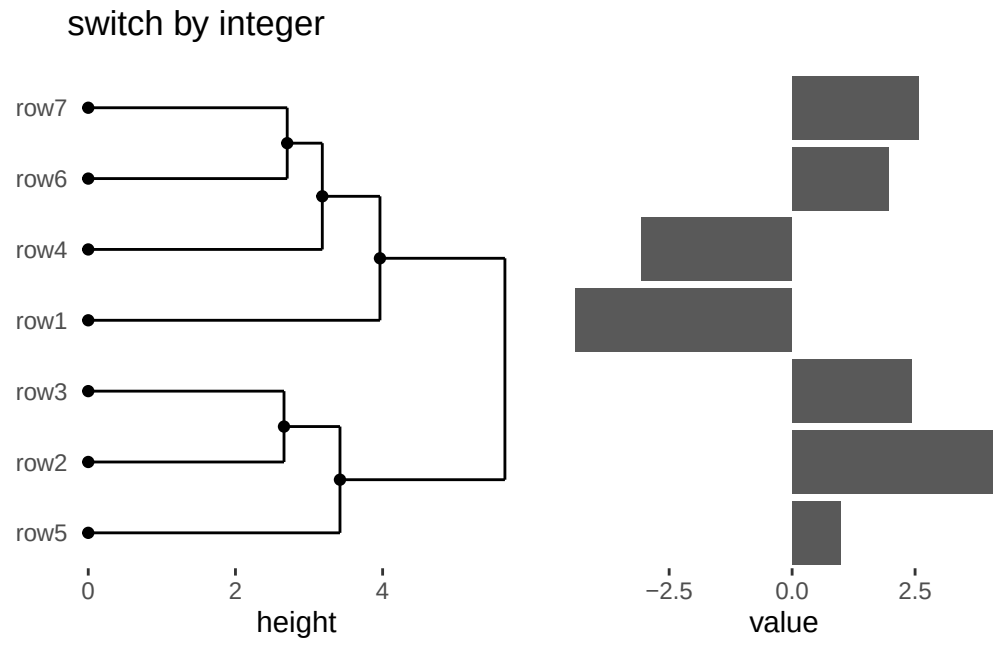
- `stack_switch()`: switch the active context
- `stack_active`: An alias for `stack_switch()`, which sets `what = NULL`

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.

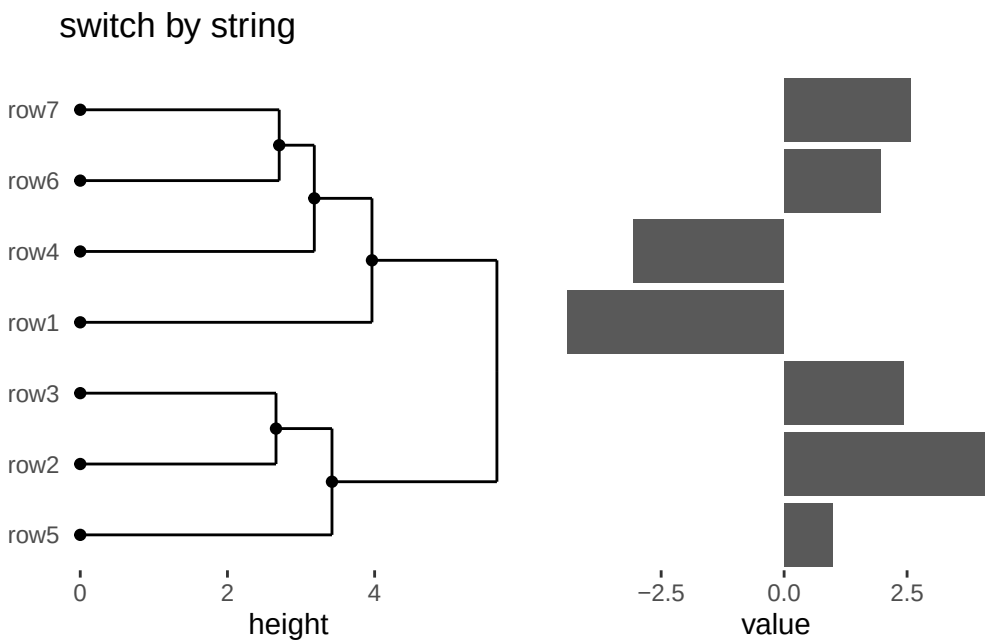
Note that the `what` argument must be explicitly named, as it is placed second in the function signature. This is because, in most cases, we don't need to switch the active plot manually—adjusting the order of plot additions typically suffices.

```
stack_alignh(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    # switch to the `align_dendro()` plot area
    stack_switch(what = 1) +
```

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




```
stack_alignh(small_mat) +
    align_dendro(active = active(name = "tree")) +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    # switch to the `align_dendro()` plot area
    stack_switch(what = "tree") +
    geom_point() +
    theme(axis.text.y = element_text()) +
    layout_title(title = "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 directly to the `StackLayout`.

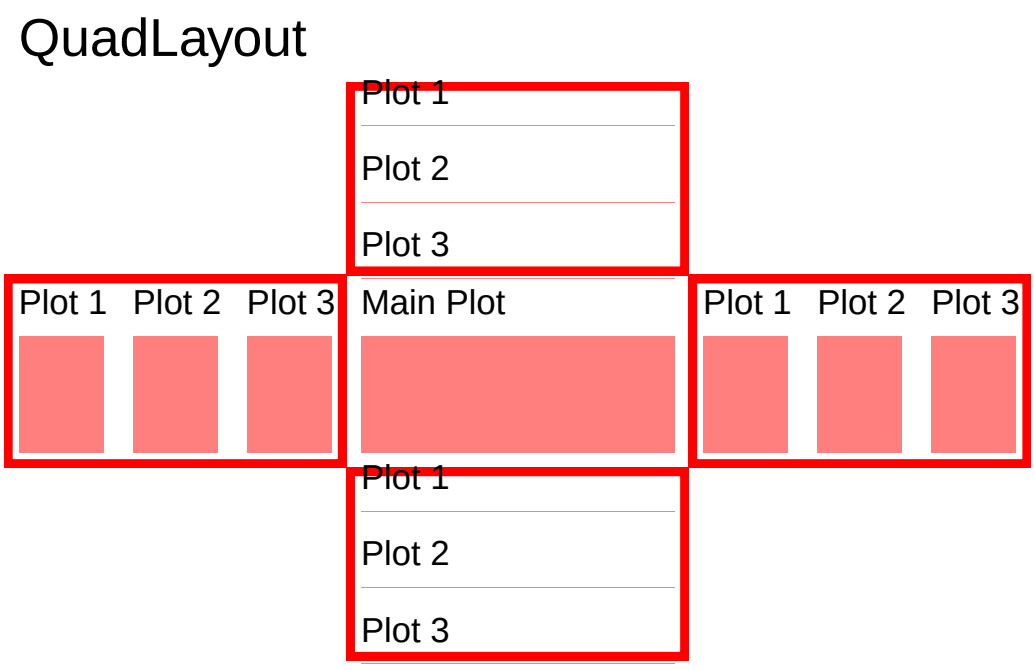
Now, let's move on to the next chapter where I'll introduce a new `Layout`.

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_alignb()`, which itself is a specific variant of `QuadLayout` (`quad_layout()`) designed to align observations 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`.

```
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)))
```



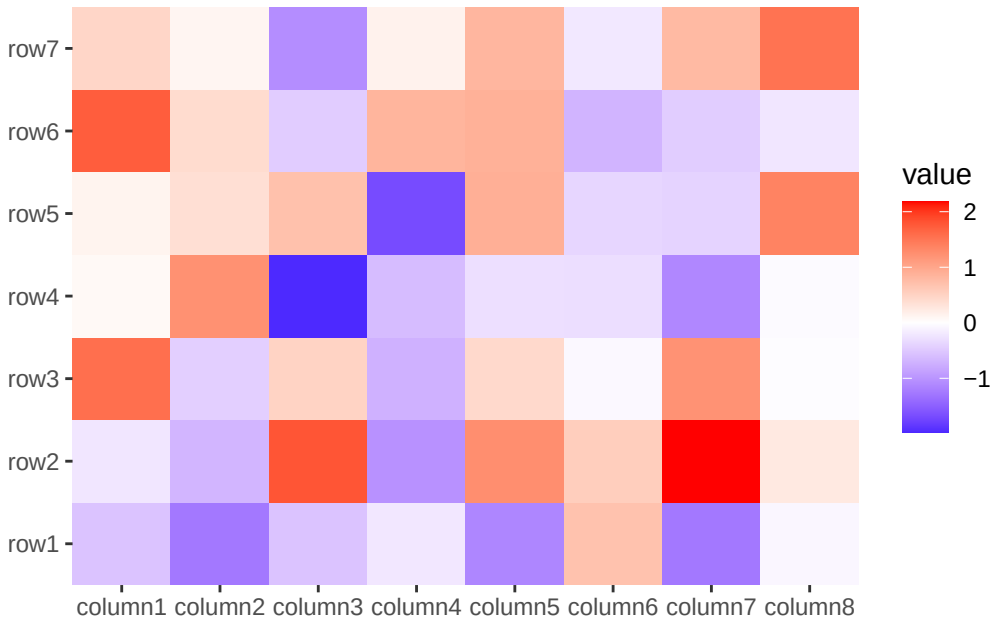
`heatmap_layout()` simplifies the creation of heatmap plots by integrating essential elements for a standard heatmap layout, ensuring that the appropriate data mapping and visualization layers are automatically applied. `ggheatmap()` is an alias for `heatmap_layout()`.

3.1 input data

As mentioned in Section 2.1, we typically require a matrix for the `Layout` which need align observations. 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 heatmap body

A `ggplot` object will be automatically created for the heatmap body, the matrix input will be converted into a long formated data frame when drawing. The data in the underlying `ggplot` object contains following columns:

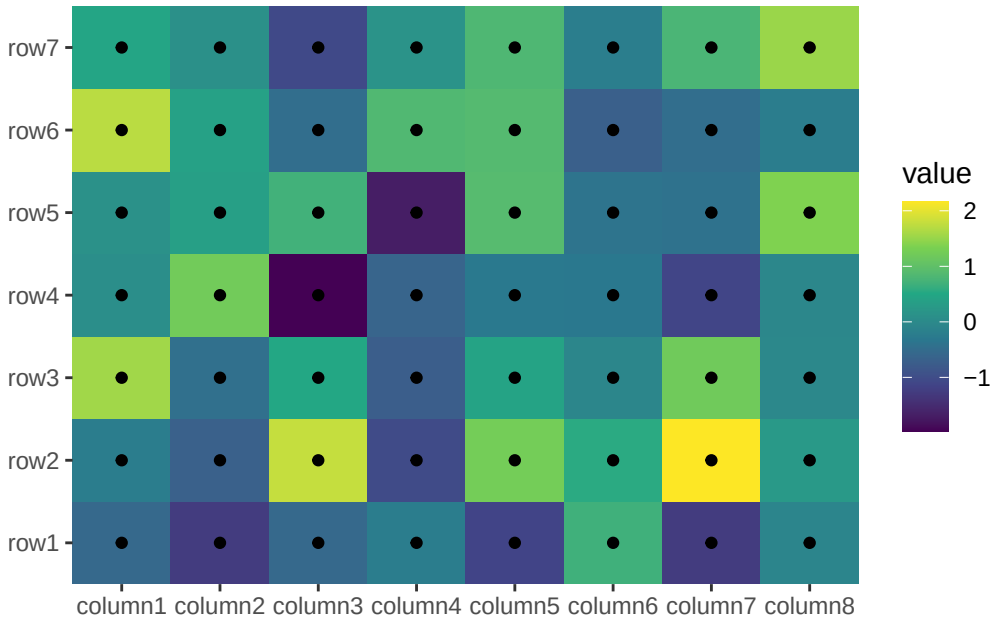
- `.xpanel` and `.ypanel`: the column and row panel
- `.x` and `.y`: the `x` and `y` coordinates
- `.row_names` and `.column_names`: A factor of the row and column names of the original matrix (only applicable when names exist).
- `.row_index` and `.column_index`: the row and column index of the original 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 heatmap body is regarded as the active plot, meaning you can add ggplot2 elements directly to the heatmap body.

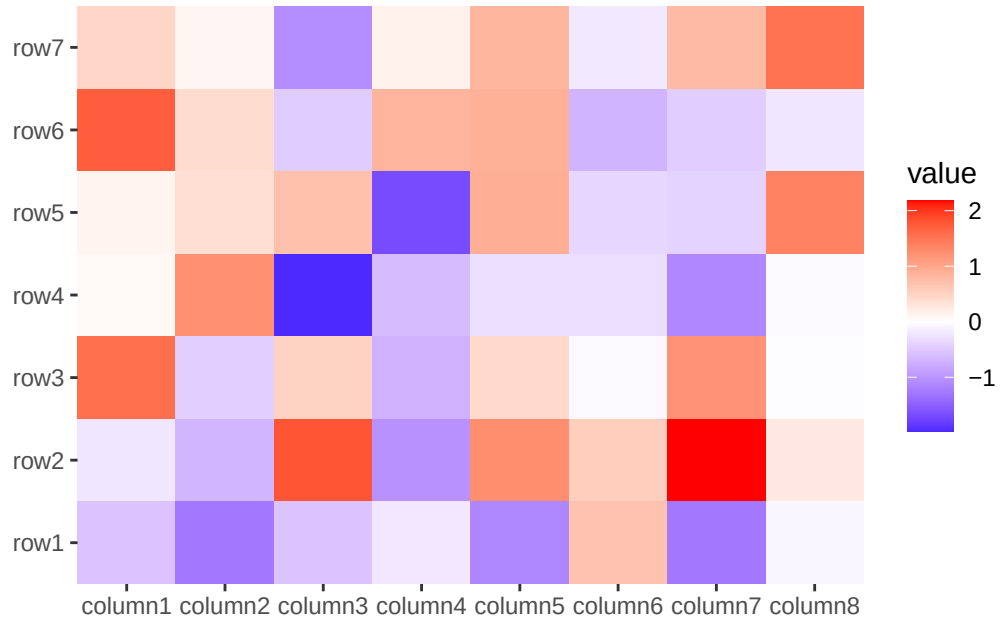
```
ggheatmap(small_mat) + geom_point() + scale_fill_viridis_c()
```

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

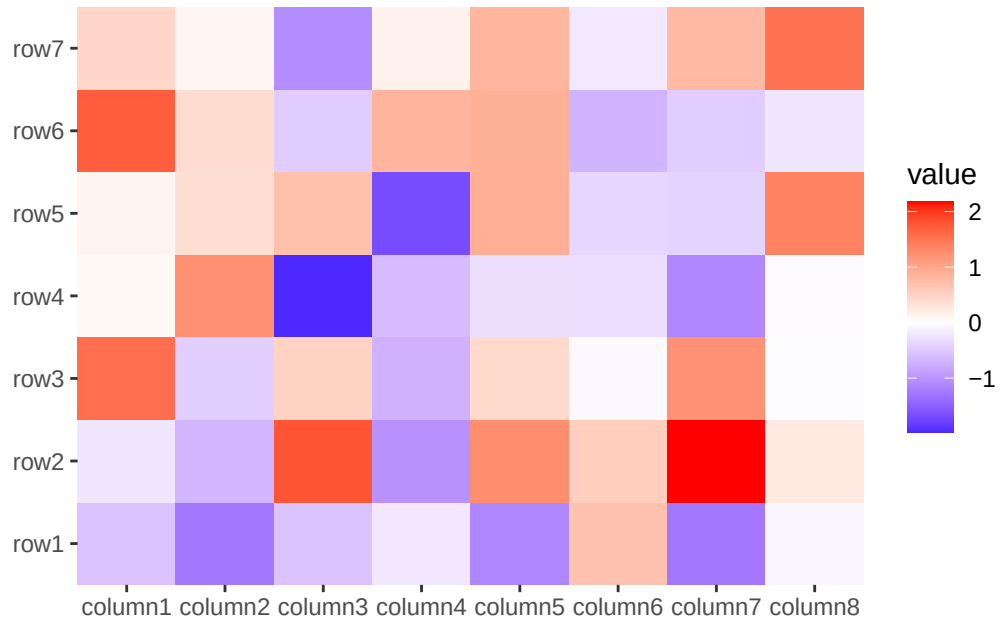


By default, `ggheatmap()`/`heatmap_layout()` adds a heatmap layer. If the matrix has more than 20,000 cells (`nrow * ncol > 20000`), it uses `geom_raster()` for performance efficiency; for smaller matrices, `geom_tile()` is used. You can explicitly choose the layer by providing a single string (`"raster"`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

```
ggheatmap(small_mat, filling = "raster")
```

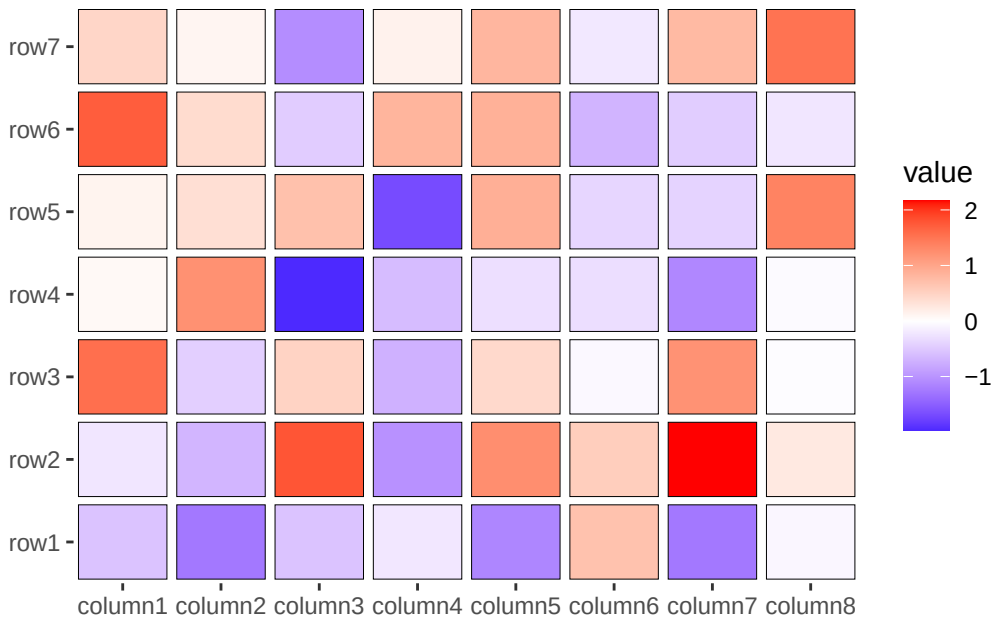



```
ggheatmap(small_mat, filling = "tile")
```



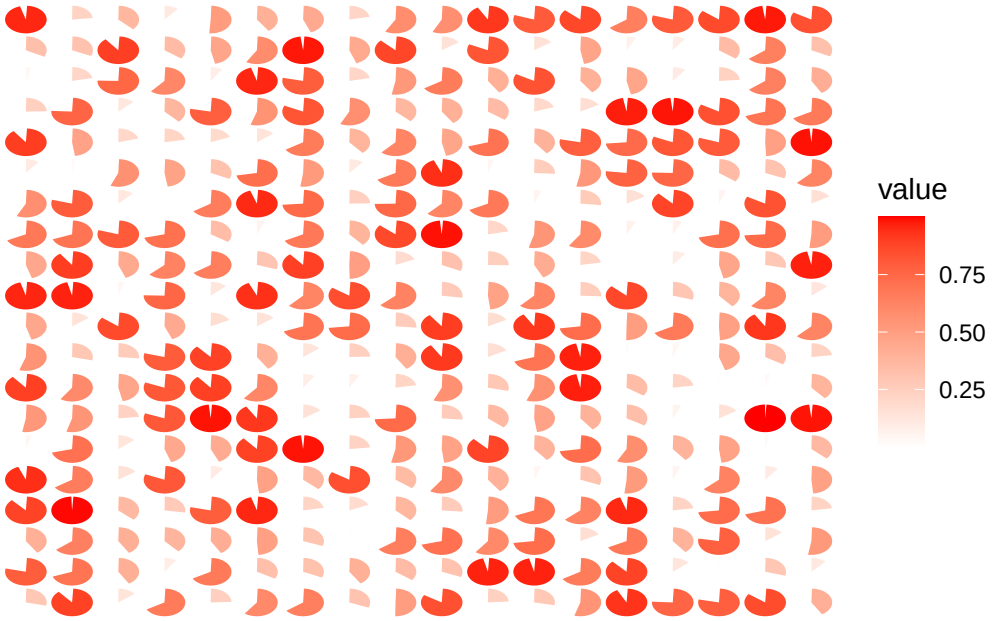
Note, the filling layer will always use `mapping` of `aes(.data$.x, .data$.y)`, if you want to customize filling, you can set `filling = NULL`, which will remove the filling layer and allow you to add custom filling geoms.

```
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:

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

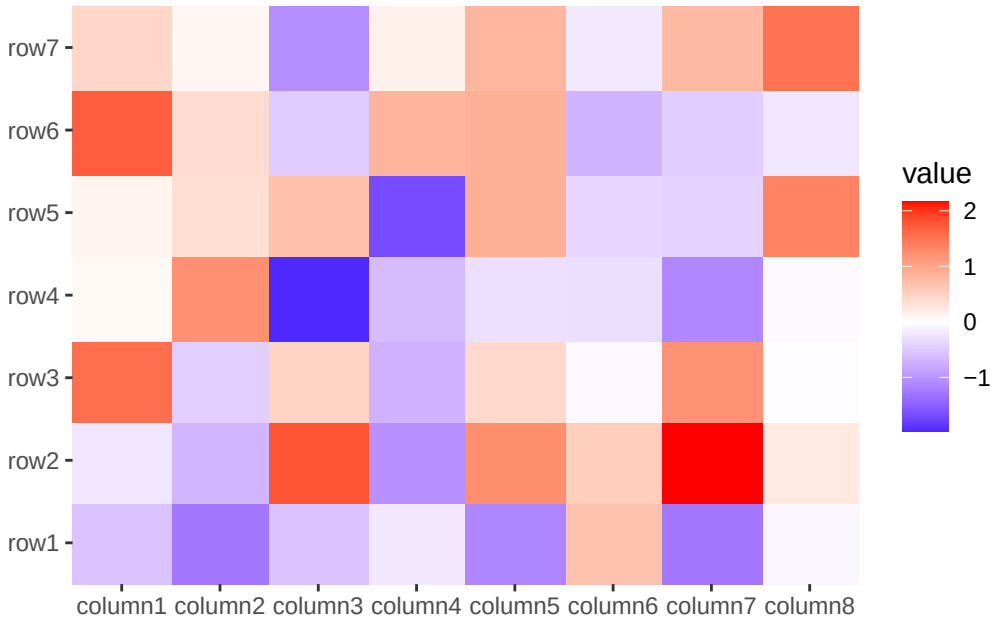


For more complex customizations of pie charts, you can try using `ggforce::geom_arc_bar()` instead.

3.3 rasterization

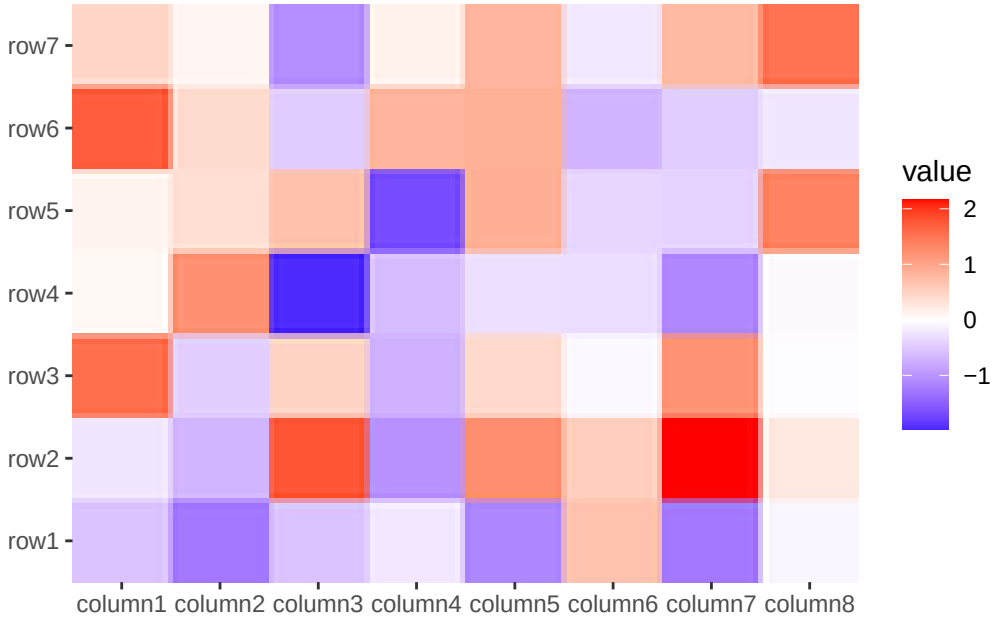
When working with large heatmaps, it's often beneficial to rasterize the heatmap body layer. You can achieve this by using the `raster_magick()` function. The `res` argument controls the resolution of the raster image. By default, the `res` argument matches the resolution of the current device, but specifying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

```
ggheatmap(small_mat, 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.

```
ggheatmap(small_mat, filling = NULL) +
    # Use `magick::filter_types()` to check available `filter` arguments
    raster_magick(geom_raster(aes(fill = value)),
        magick = function(image) {
            magick::image_resize(image,
                # we resize to the 50% of width
                geometry = "50%x", filter = "Lanczos"
            )
        }
    )
```

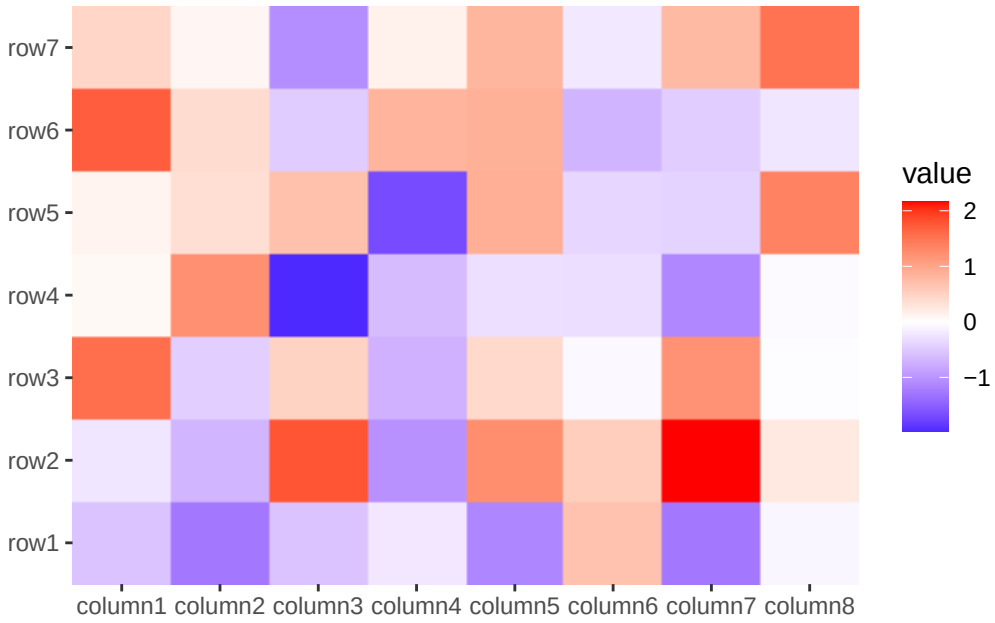


Note: When using `magick::image_resize()`, you should specify the `geometry` argument to resize the image. If only the `filter` is specified, it will only distort the image data (though subtle). For more information on image resizing, refer to ImageMagick's resize documentation.

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.

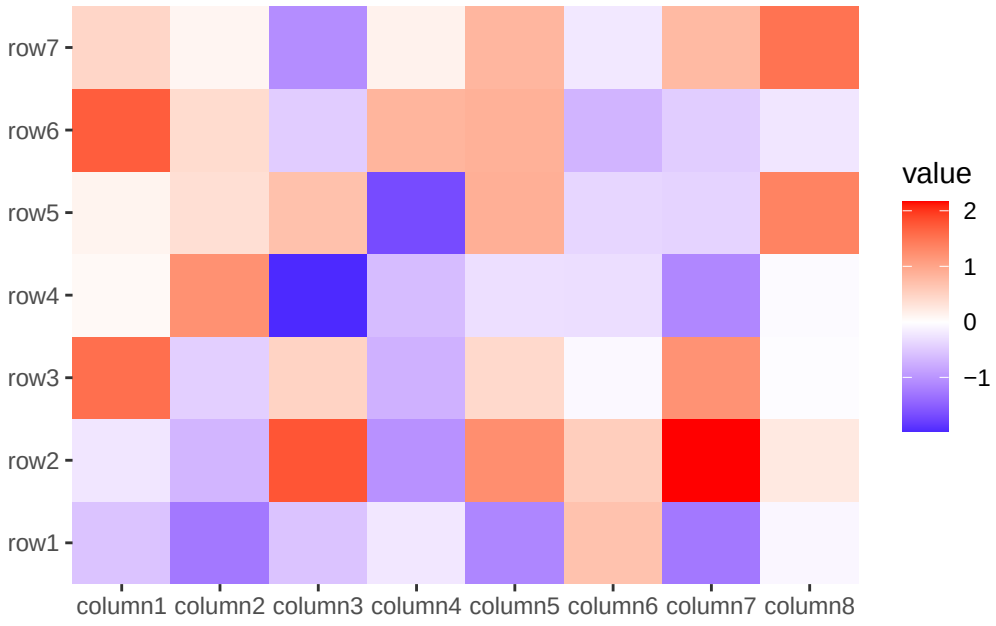
```
ggheatmap(small_mat, filling = FALSE) +
    ggrastr::rasterise(geom_tile(aes(fill = value)), dev = "ragg")
```



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).

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:

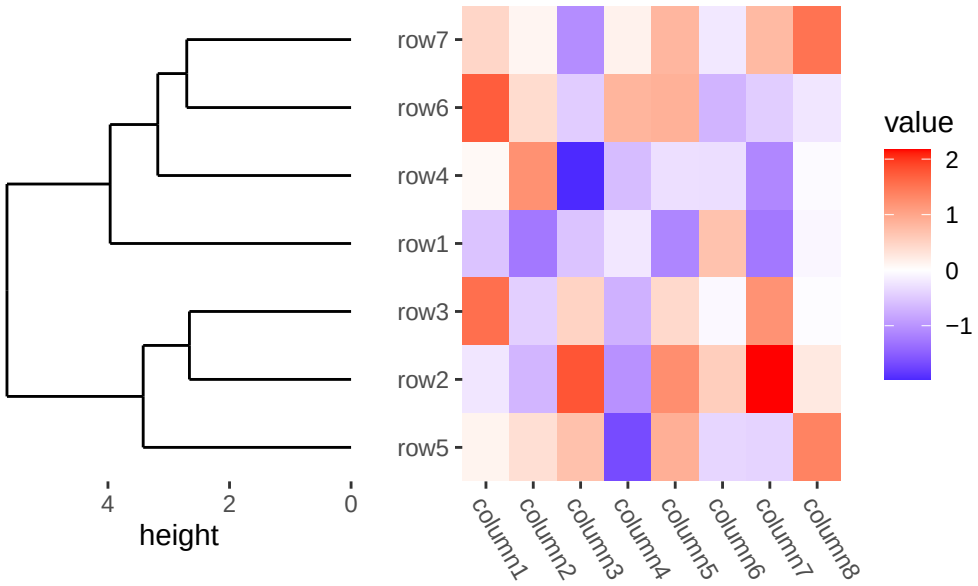
- `anno_top`: A special case of `quad_anno()` with `position = "top"`.
- `anno_left`: A special case of `quad_anno()` with `position = "left"`.
- `anno_bottom`: A special case of `quad_anno()` with `position = "bottom"`.
- `anno_right`: A special case of `quad_anno()` with `position = "right"`.

When `quad_anno()` is added to a `ggheatmap()`/`quad_layout()`, it will try to automatically create a new `stack_layout()` (either `stack_align()` or `stack_free()`) depending on whether you want to align observations in the specified direction. For top and bottom annotations, `stack_alignv()` or `stack_freev()` will be used; for left and right annotations, `stack_alignh()` or `stack_freeh()` 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 described in Chapter 2), since the annotation is a `Layout` object.

```
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    # we set the active context to the left annotation
    anno_left() +
    align_dendro()
```

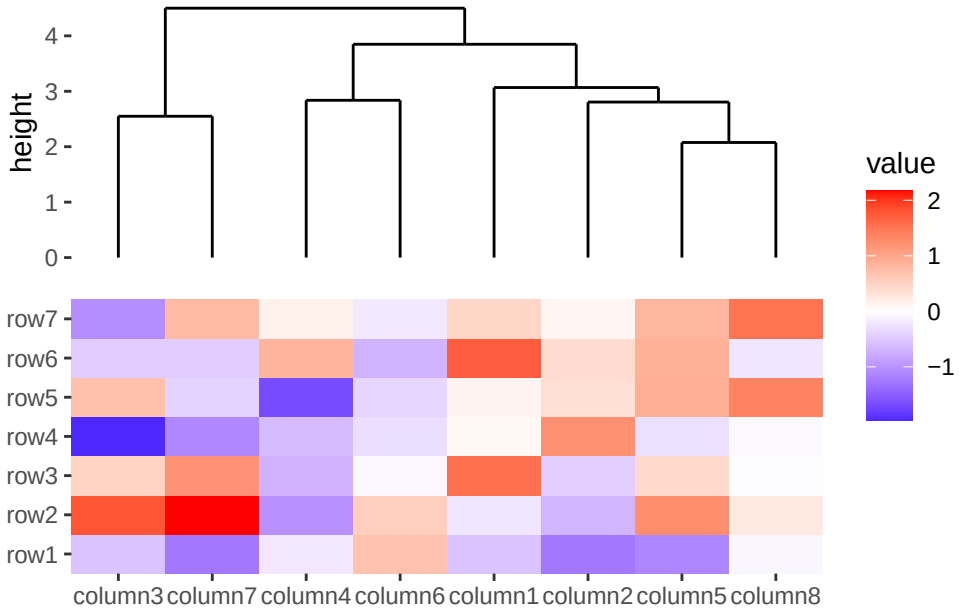
```
> heatmap built with `geom_tile()`
```



By default, the annotation `stack_layout()` will inherit data from `ggheatmap()`/`quad_layout()`. If the observations require alignment 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()
```

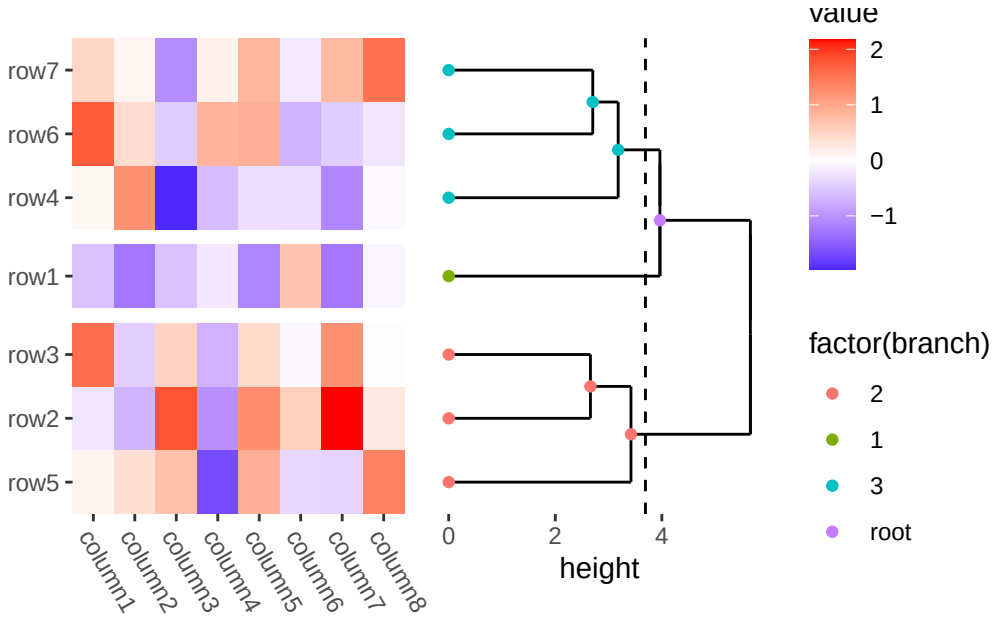
```
> heatmap built with `geom_tile()`
```



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 = factor(branch)))
```

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



In this example:

- `anno_right()` initialize the right annotation stack, and change the active context to the right 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 = factor(branch)))` is then added to this active plot within the annotation stack, here, it means the `align_dendro()` plot.

`ggheatmap()` aligns observations 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_alignh()` and `quad_alignv()`, which only align observations in one direction, the data in the layout may not fit the data for the annotation (when the layout requires alignment of observations, we typically use a matrix, regardless of whether alignment is needed in one or two directions) - `quad_alignh()`: aligning observations in horizontal direction, for column annotations, we ll need a data frame for `stack_free()`. - `quad_alignv()`: aligning observations in vertical direction, for row annotations, we ll need a data frame for `stack_free()`.

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

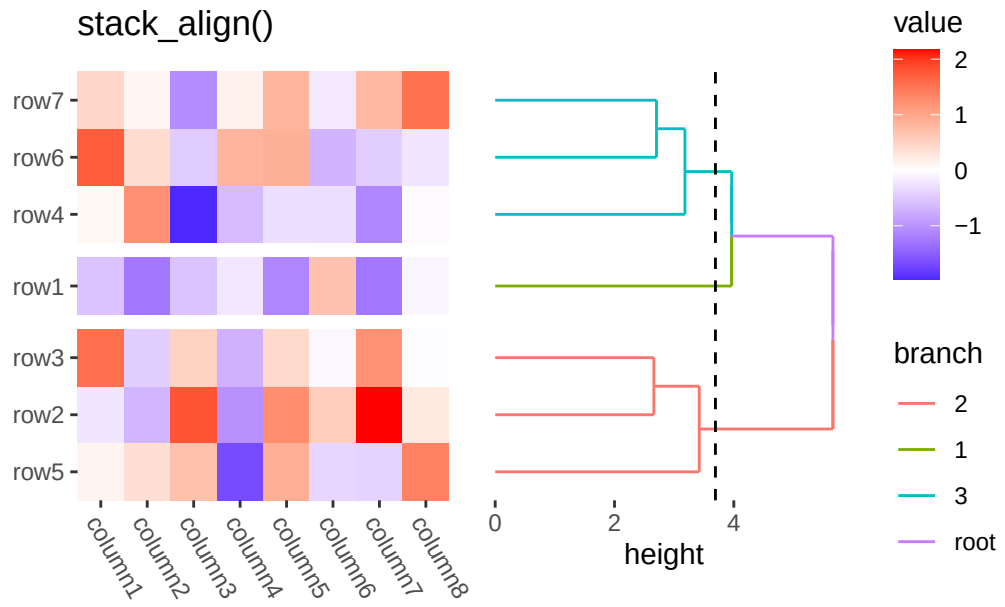
3.5 Adding stack layout

Like adding plot in `stack_layout()` (Chapter 2), when the direction neeed alignment, you can add a `stack_layout()` regardless of whether it need to align observations.

To add a `stack_layout()` to the `ggheatmap()`, we must prevent the automatical creation of annotation by `quad_anno()` by setting `initialize = FALSE`

```
my_stack_align <- stack_alignh(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_align +
    layout_title("stack_align()")
```

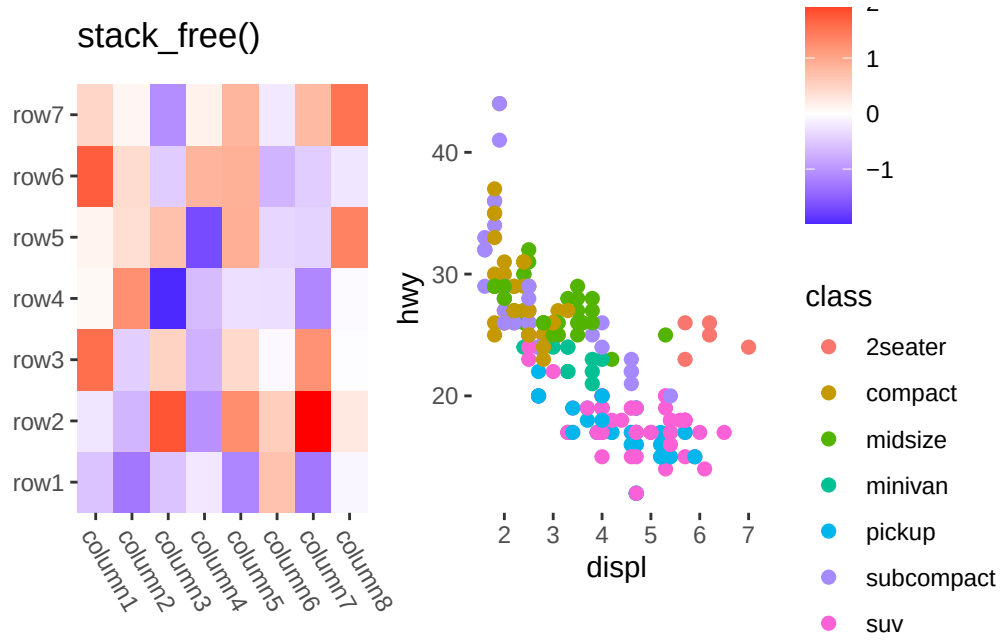
```
> heatmap built with `geom_tile()`
```




```
my_stack_free <- stack_freeh(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2)
ggheatmap(small_mat) +
```

```
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(initialize = FALSE) +
    my_stack_free +
    layout_title("stack_free()")
```

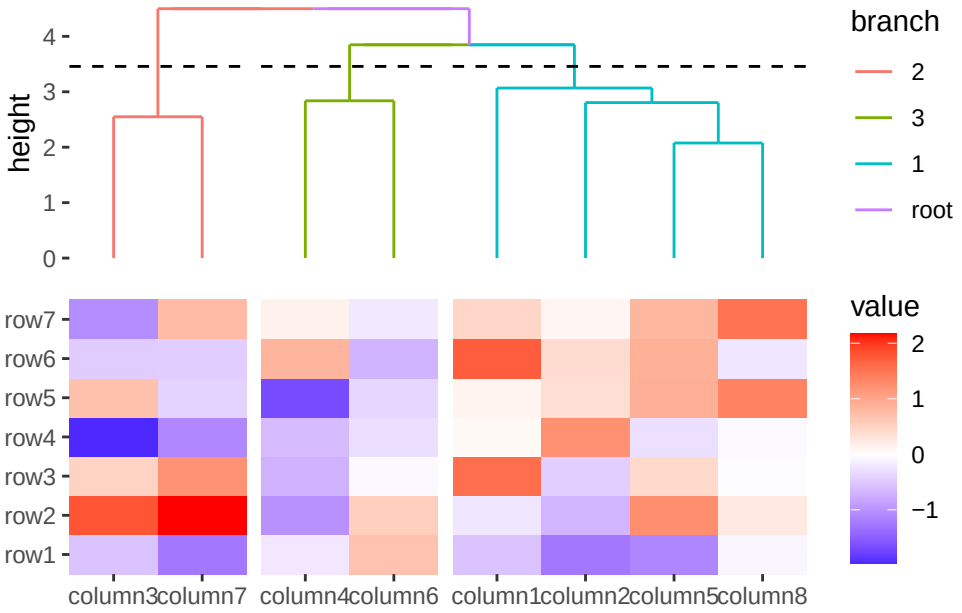
```
> heatmap built with `geom_tile()`
```



Note when aligning the observations, 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_alignv(t(small_mat)) +
    align_dendro(aes(color = branch), k = 3L)
ggheatmap(small_mat) +
    anno_top(initialize = FALSE) +
    my_stack
```

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

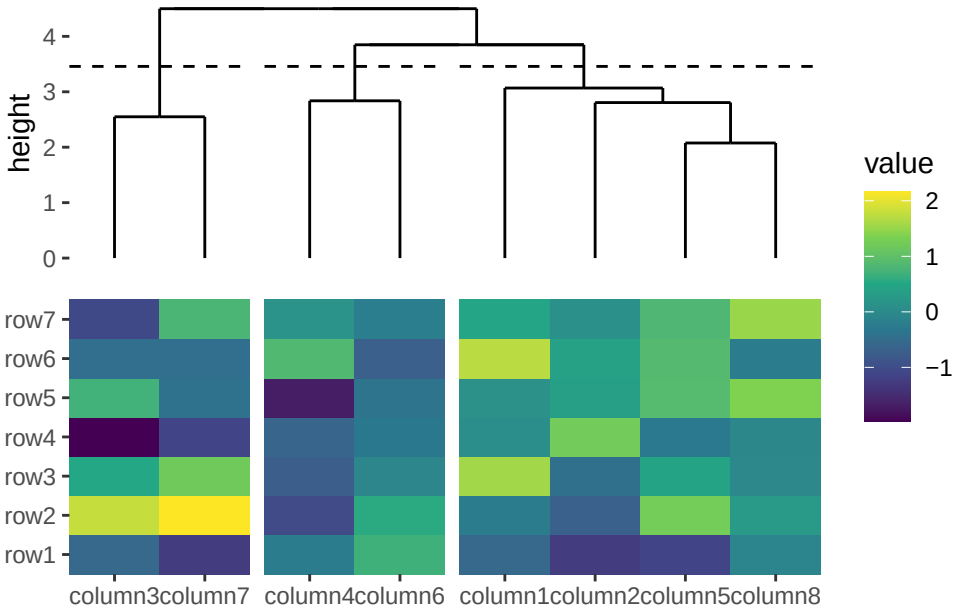


3.6 `quad_active()`

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()`
```

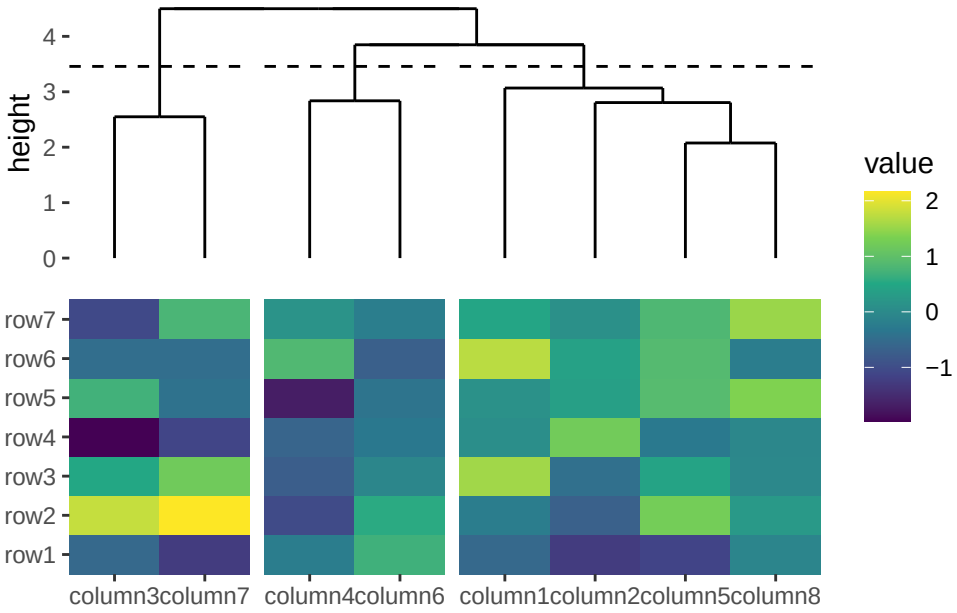


3.7 quad_switch()/hmanno()

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.

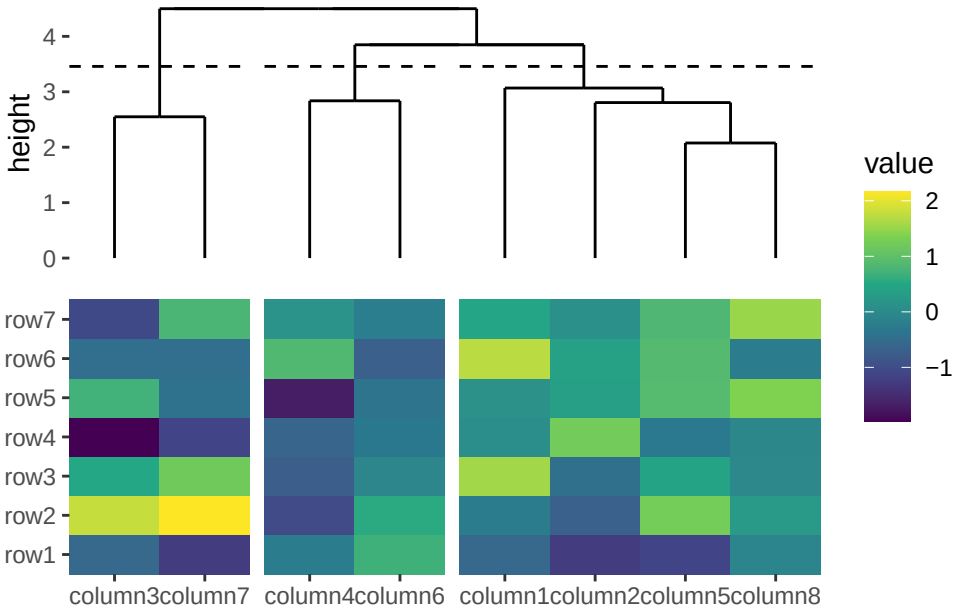
```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    quad_switch("t") +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    quad_switch() +
    # set fill color scale for the heatmap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

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




```
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()
```

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



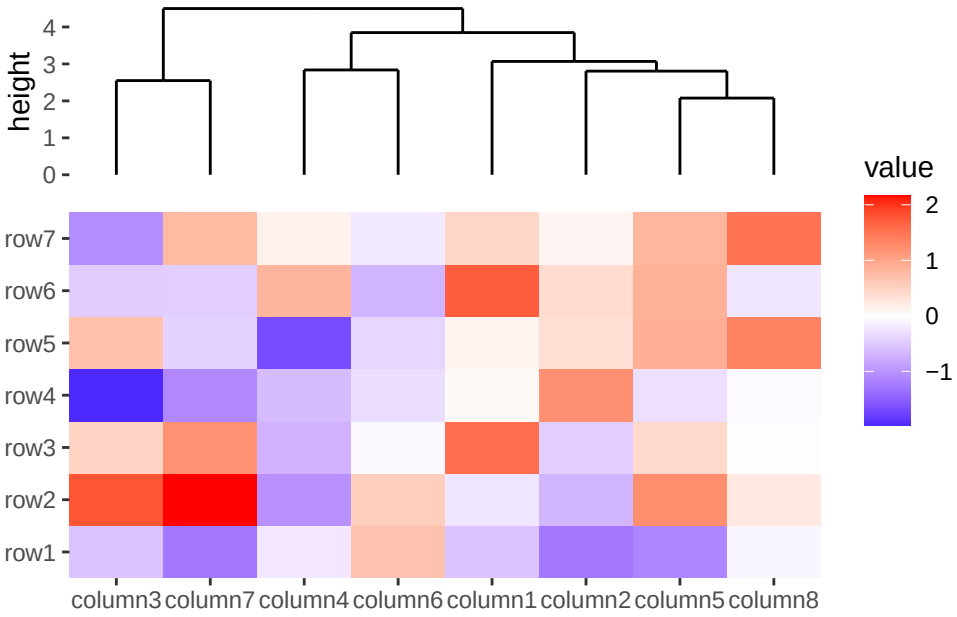
3.8 Plot Size

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()
```

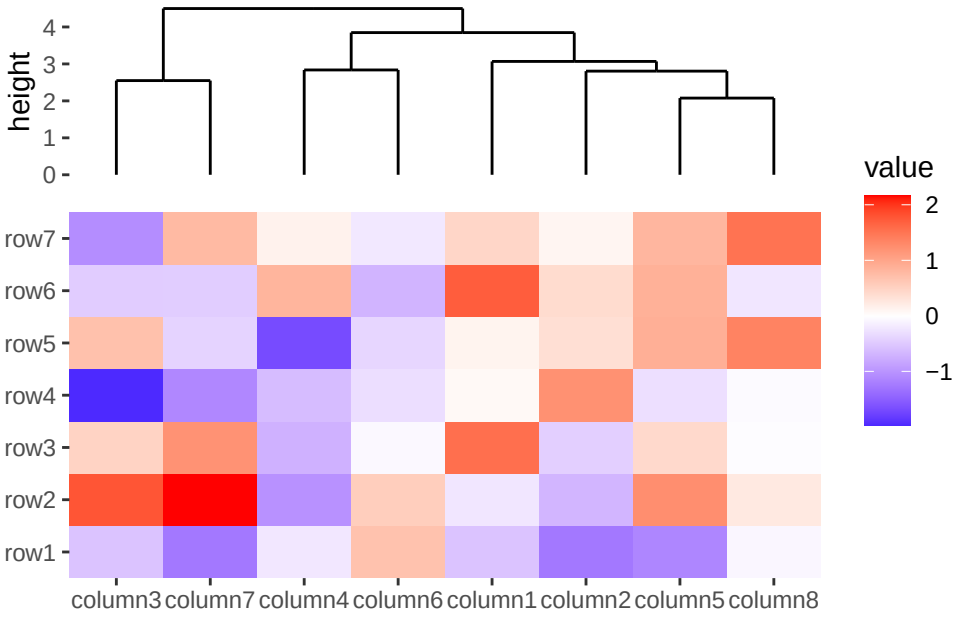
```
> heatmap built with `geom_tile()`
```



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()`
```

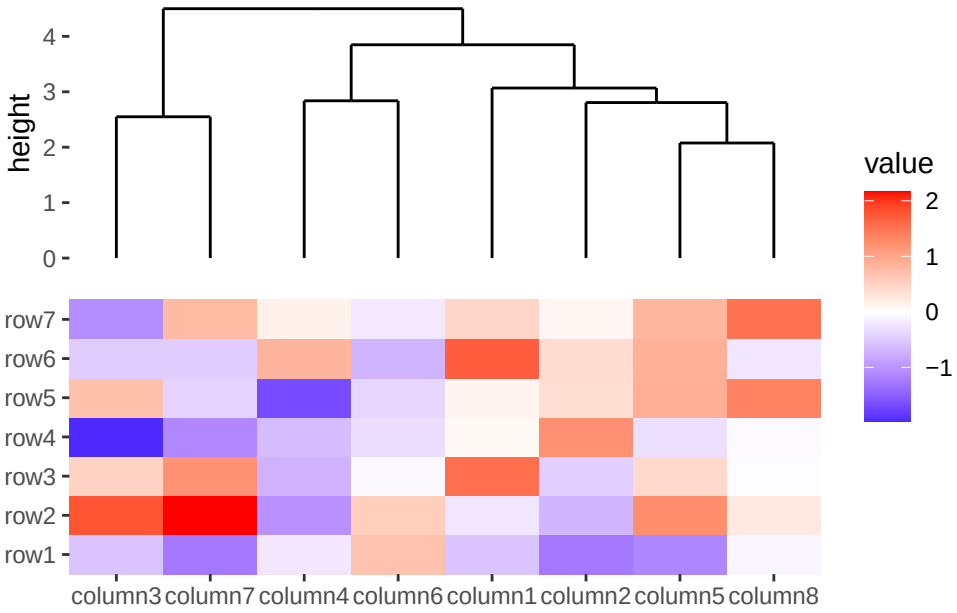


3.8.2 Annotation Stack Size

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()
```

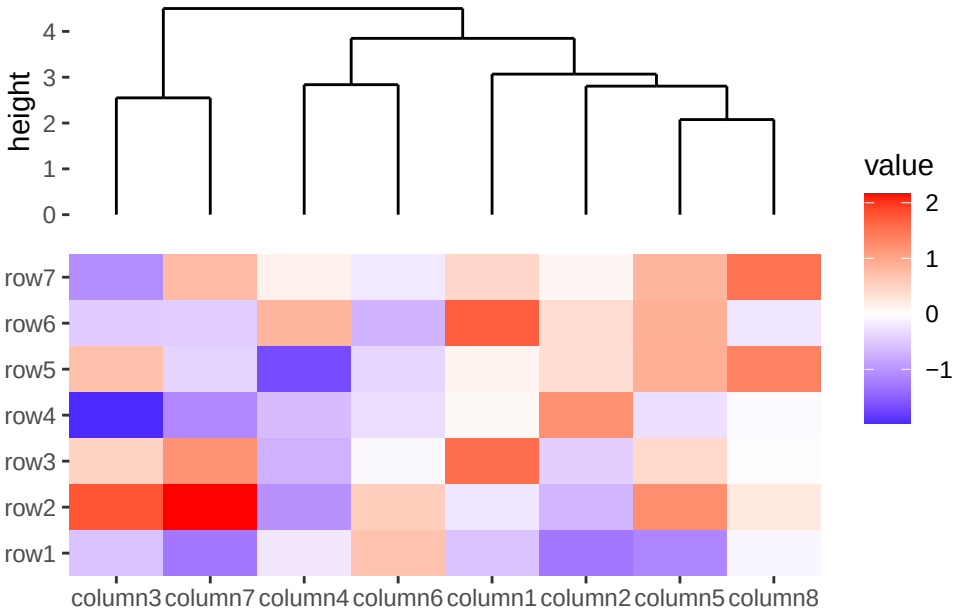
```
> heatmap built with `geom_tile()`
```



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()`
```


height
4
3
2
1
0
row7
row6
row5
row4
row3
row2
row1
column3
column7
column4
column6
column1
column2
column5
column8
value
2
1
0
−1

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 layout. 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_order`**: Reorder layout observations based on statistical weights or allows for manual reordering based on user-defined ordering index.
- **`align_kmeans`**: Group observations by k-means clustering results.
- **`align_hclust`**: Reorder or group observations based on hierarchical clustering.

Note that none of these functions add a plot area or set the active context, meaning you cannot add ggplot2 elements to these objects.

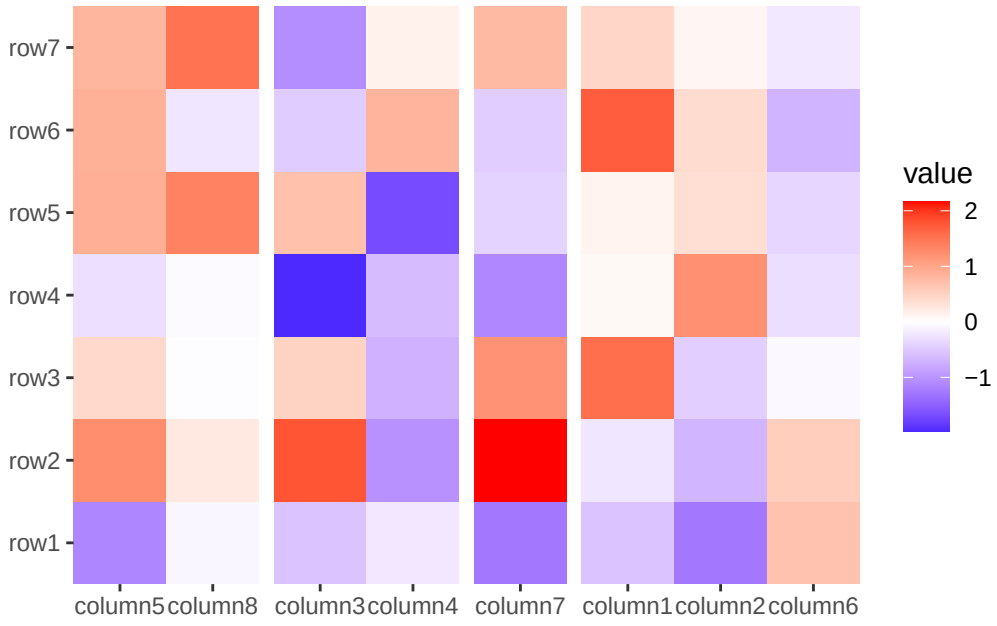
```
library(ggalign)
## Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

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 `align()`:
! align_group(sample(letters[1:5], ncol(small_mat), replace = TRUE))
  disrupt the previously established panel groups of the top annotation
  `stack_align()`
```

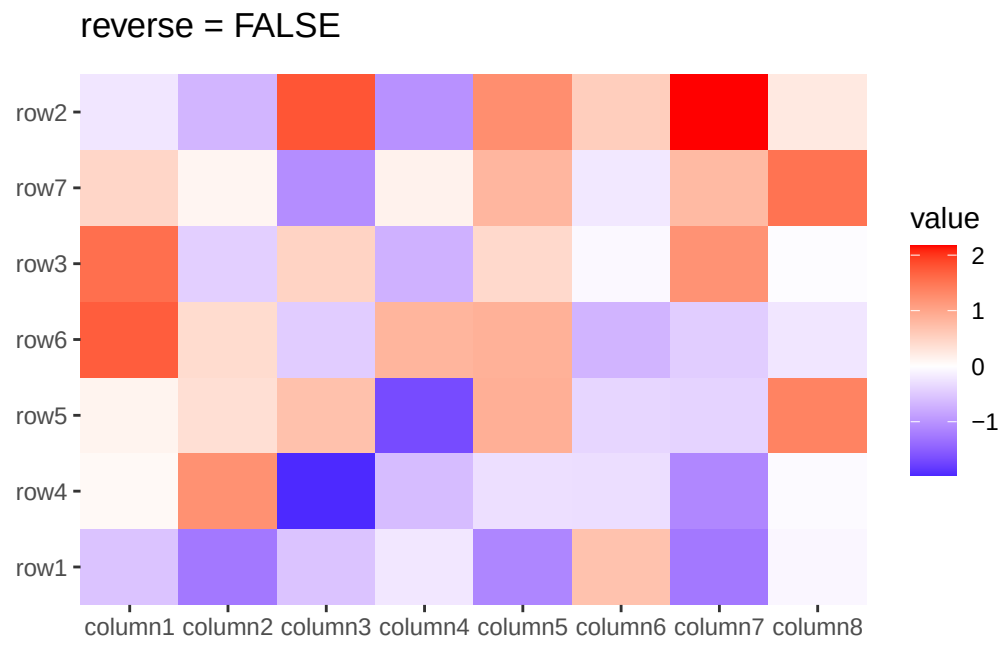
4.2 `align_order()`

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`.

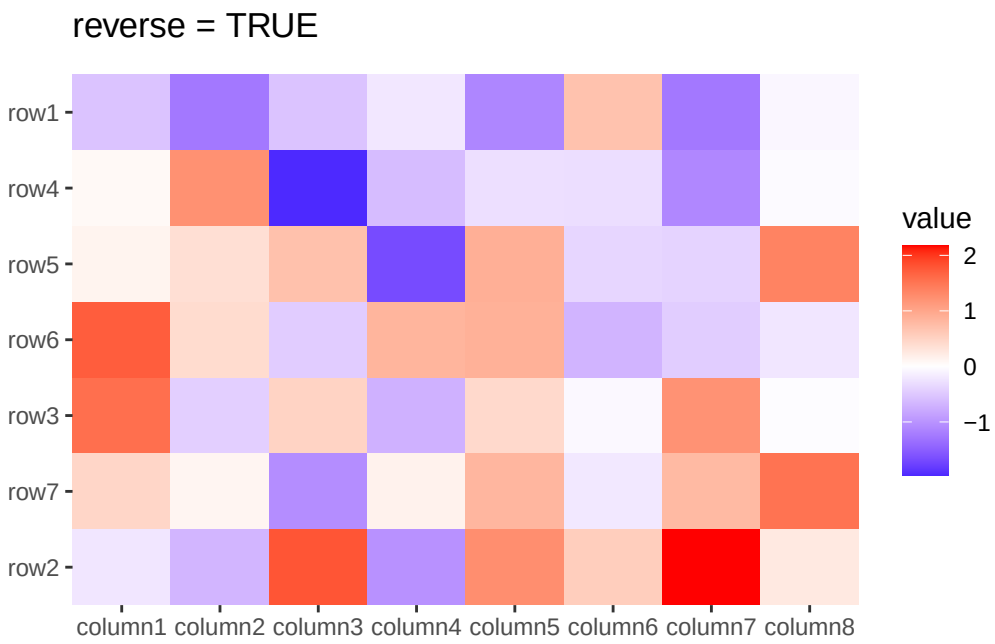
```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans) +
    layout_title(title = "reverse = FALSE")
```

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




```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans, reverse = TRUE) +
    layout_title(title = "reverse = TRUE")
```

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



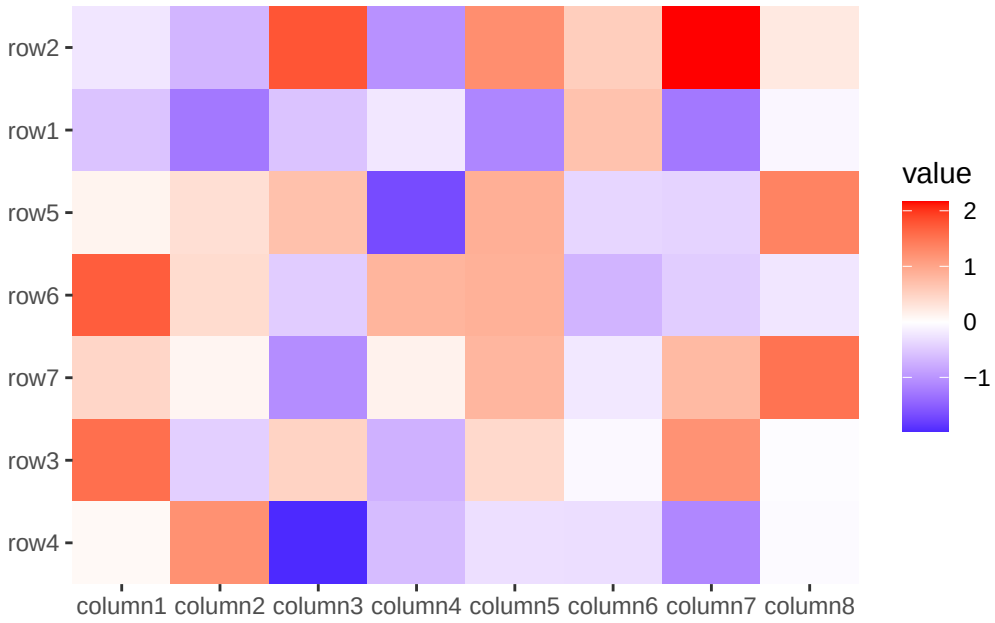
Additionally, you can provide the ordering integer index or character index directly:

```
my_order <- sample(nrow(small_mat))
print(rownames(small_mat)[my_order])
```

```
[1] "row4" "row3" "row7" "row6" "row5" "row1" "row2"
```

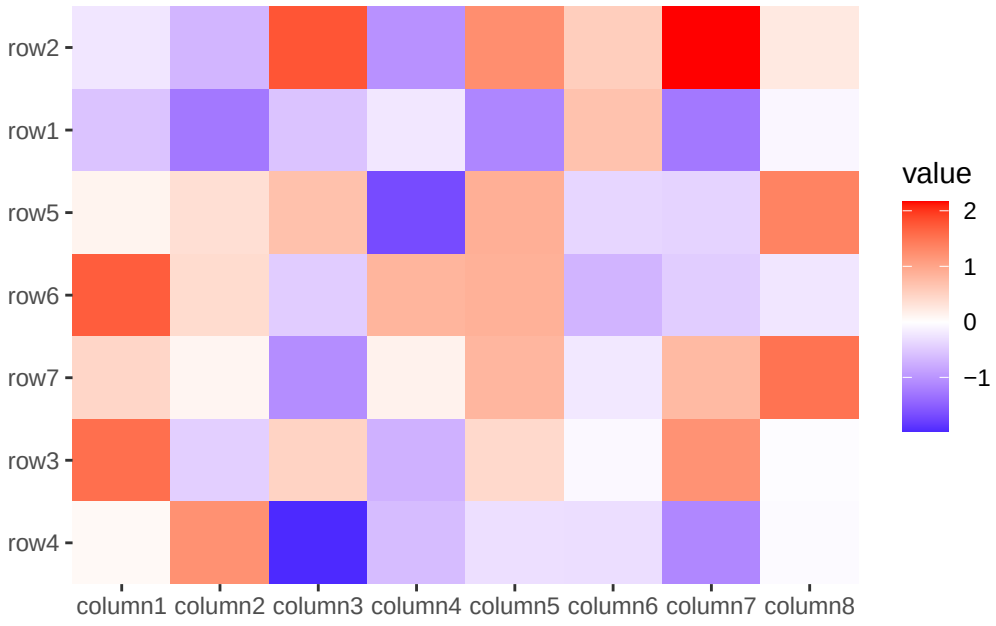
```
ggheatmap(small_mat) +
    anno_left() +
    align_order(my_order)
```

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




```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rownames(small_mat)[my_order])
```

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

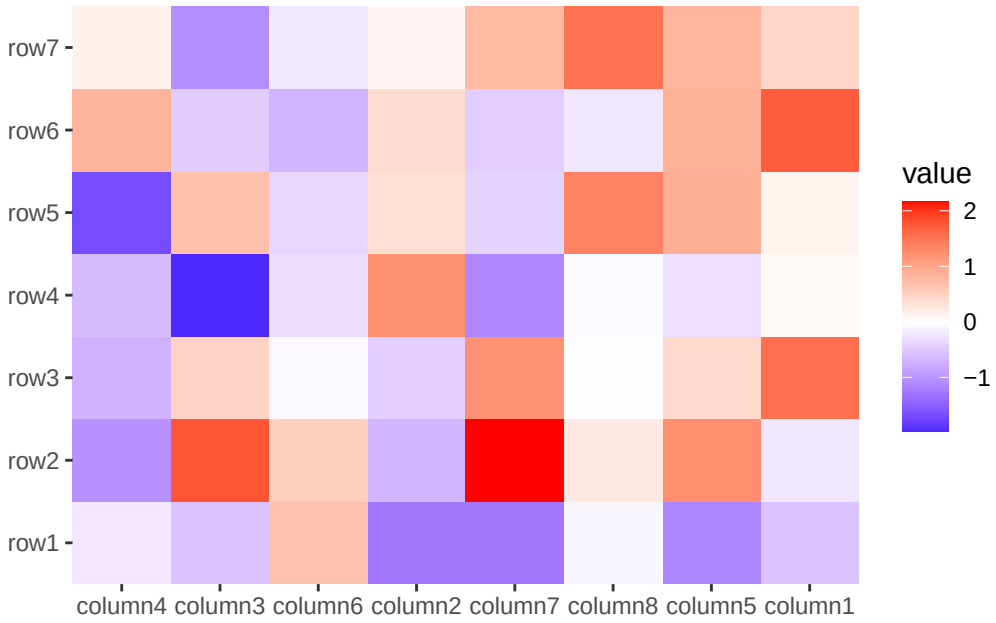


Some `align_*` functions also accept a `data` argument. It's important to note that all `align_*` functions treat rows as the observations. This means `NROW(data)` must match the number of observations along the axis used for alignment. The `data` argument can also accept a function (supporting purrr-like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

As mentioned in Chapter 3, 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)
```

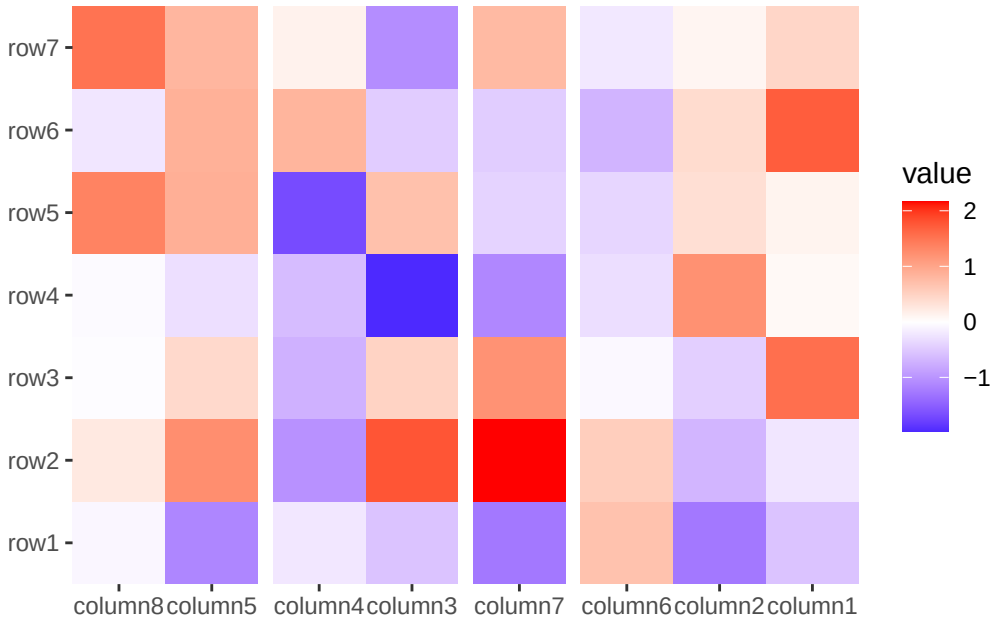
```
> heatmap built with `geom_tile()`
```



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()`
```

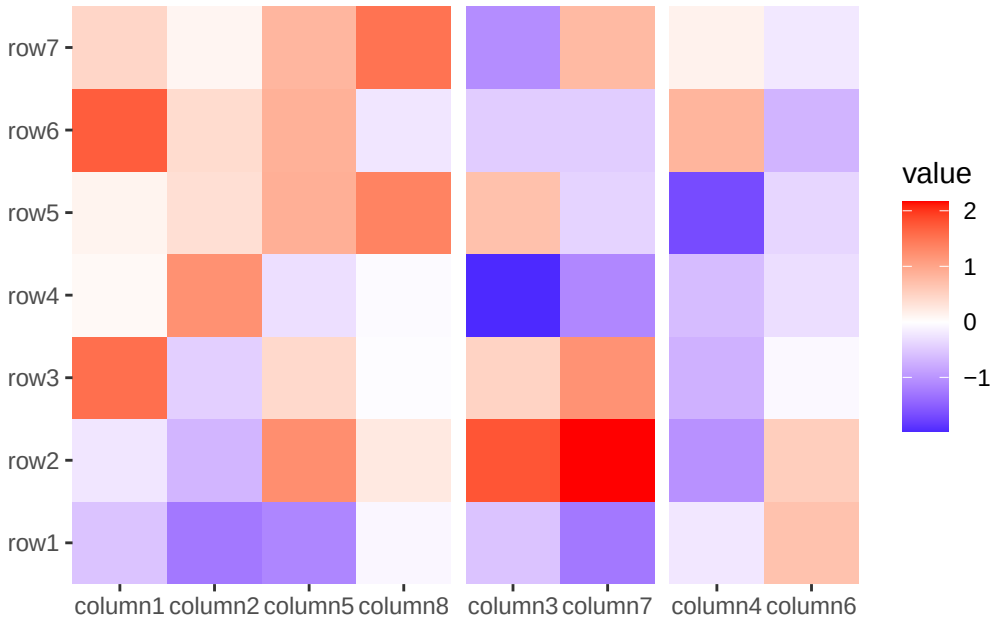


4.3 `align_kmeans()`

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

```
ggheatmap(small_mat) +
    anno_top() +
    align_kmeans(3L)
```

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

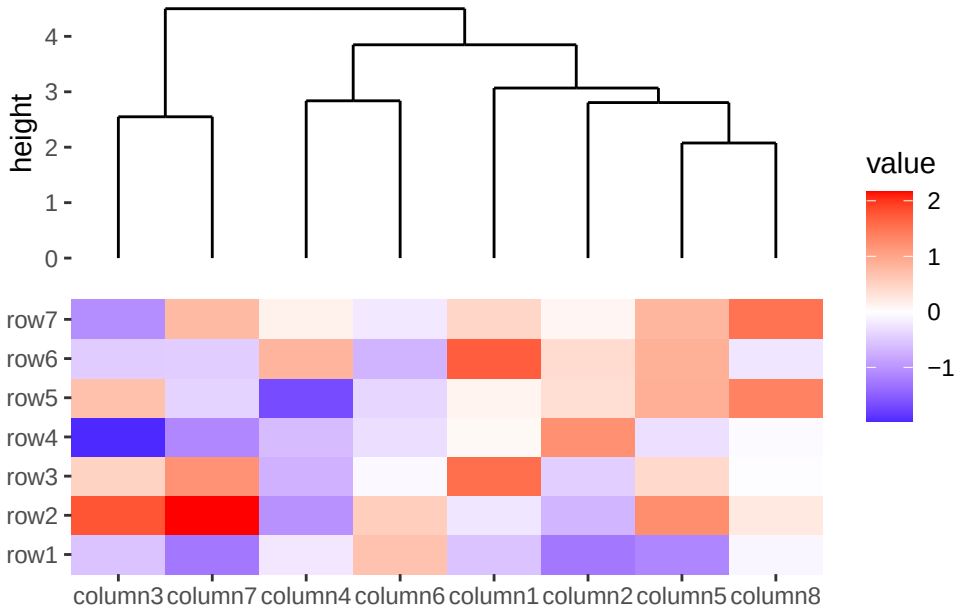


4.4 `align_hclust()`

The `align_dendro()` function adds a dendrogram to the layout and can also reorder or split the layout based on hierarchical clustering. This is particularly useful for working with heatmap plots.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro()
```

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



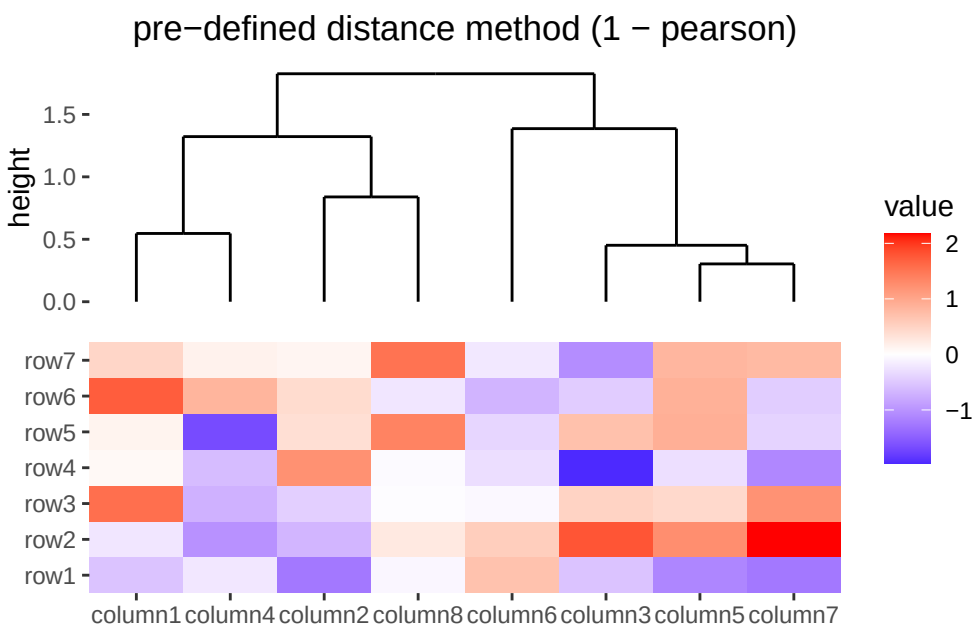
Hierarchical clustering is performed in two steps: calculate the distance matrix and apply clustering. You can use the `distance` and `method` argument to control the dendrogram builind process.

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. The function should only contain one argument. Please note for clustering on columns, the matrix will be transposed automatically.

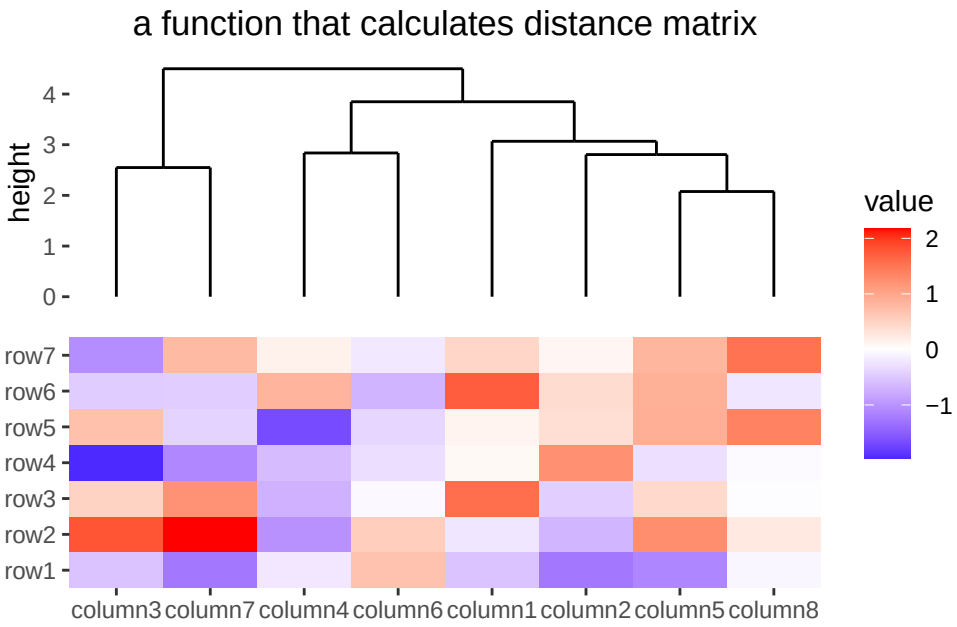
```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(distance = "pearson") +
    patch_titles(top = "pre-defined distance method (1 - pearson)")
```

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




```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(distance = function(m) dist(m)) +
    patch_titles(top = "a function that calculates distance matrix")
```

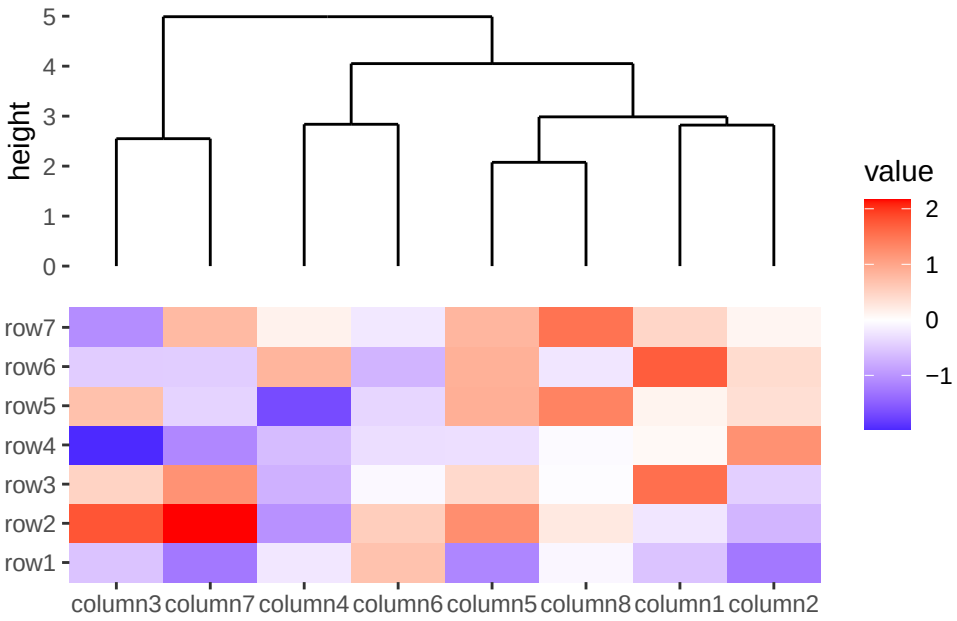
```
> heatmap built with `geom_tile()`
```



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_dendro(method = "ward.D2")
```

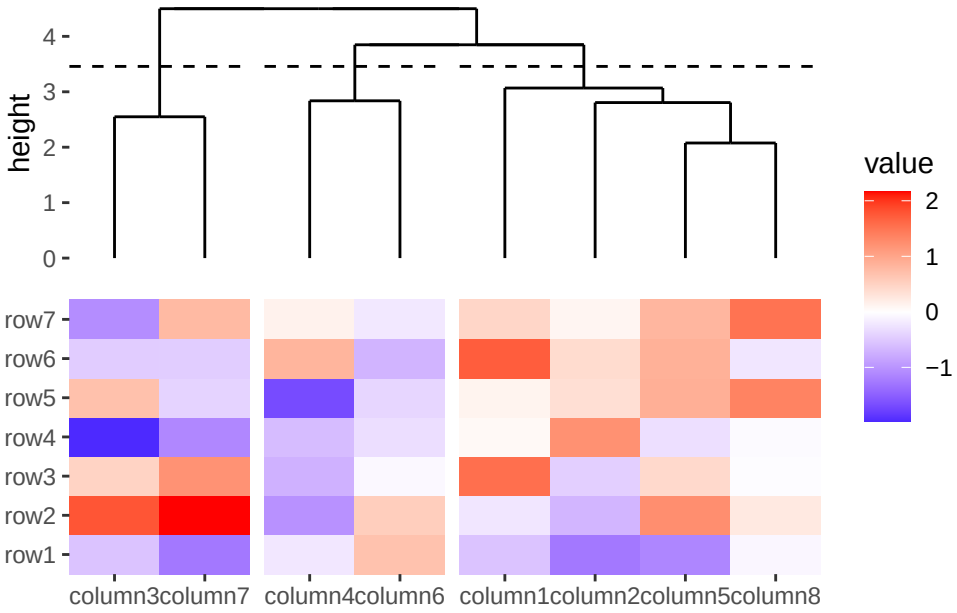
```
> heatmap built with `geom_tile()`
```



The dendrogram can also be used to cut the columns/rows into groups. You can specify `k` or `h`, which work similarly to `cutree()`:

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(k = 3L)
```

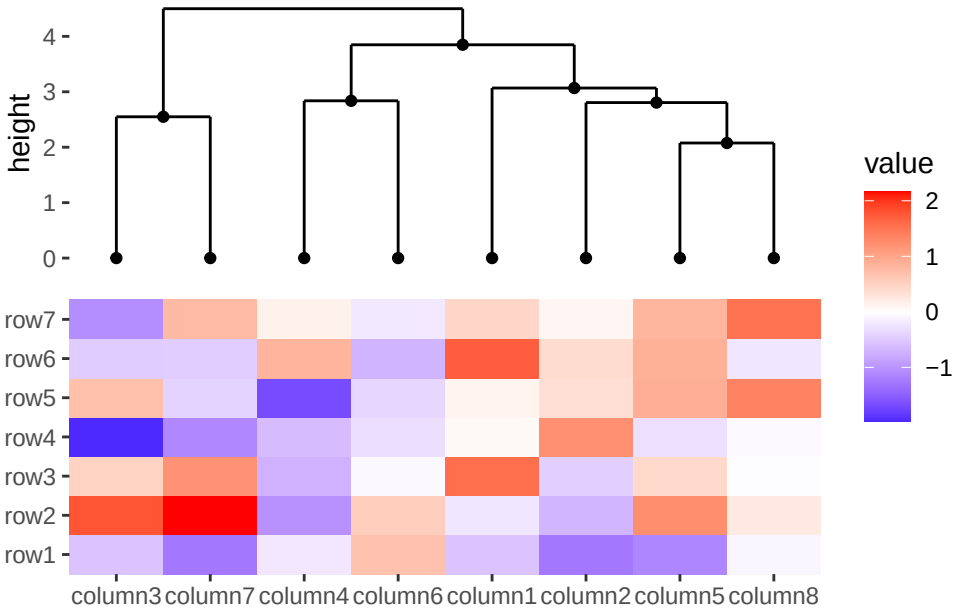
```
> heatmap built with `geom_tile()`
```



In contrast to `align_group()`, `align_kmeans()`, and `align_order()`, `align_dendro()` is capable of drawing plot components. So it has a default `set_context` value of `TRUE`, meaning it will set the active context of the annotation stack layout. In this way, we can add any ggplot elements to this plot area.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro() +
    geom_point(aes(y = y))
```

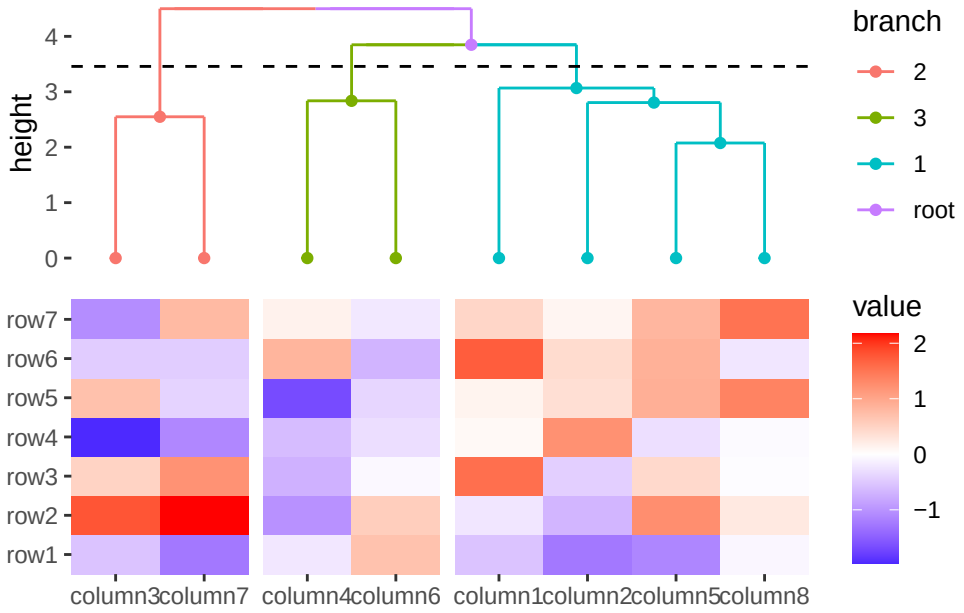
```
> heatmap built with `geom_tile()`
```



The `align_dendro()` function creates default `node` data for the ggplot. See `ggplot2 specification` in `?align_dendro` for details. Additionally, `edge` data is added to the `ggplote::geom_segment()` layer directly, used to draw the dendrogram tree. One useful variable in both `node` and `edge` data is the `branch` column, corresponding to the `cutree` result:

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    geom_point(aes(color = branch, y = y))
```

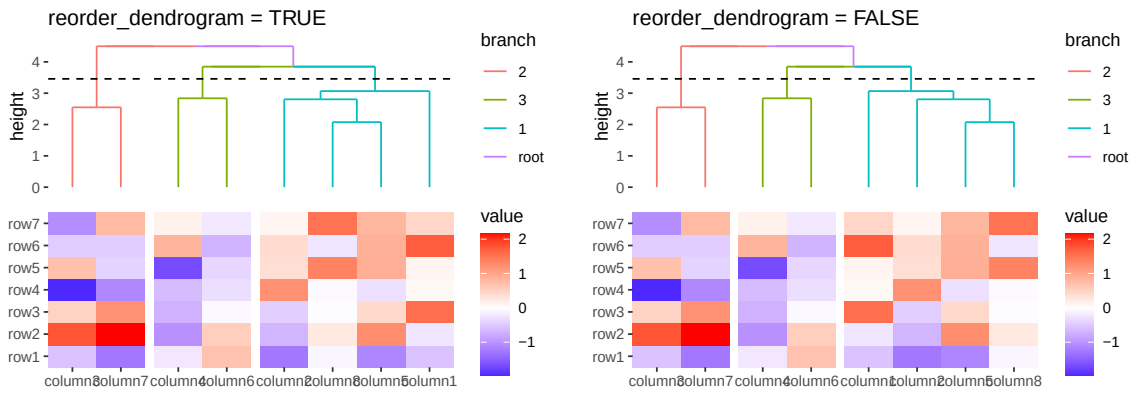
```
> heatmap built with `geom_tile()`
```



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

```
h1 <- ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3, reorder_dendrogram = TRUE) +
    ggtitle("reorder_dendrogram = TRUE")
h2 <- ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    ggtitle("reorder_dendrogram = FALSE")
align_plots(h1, h2)
```

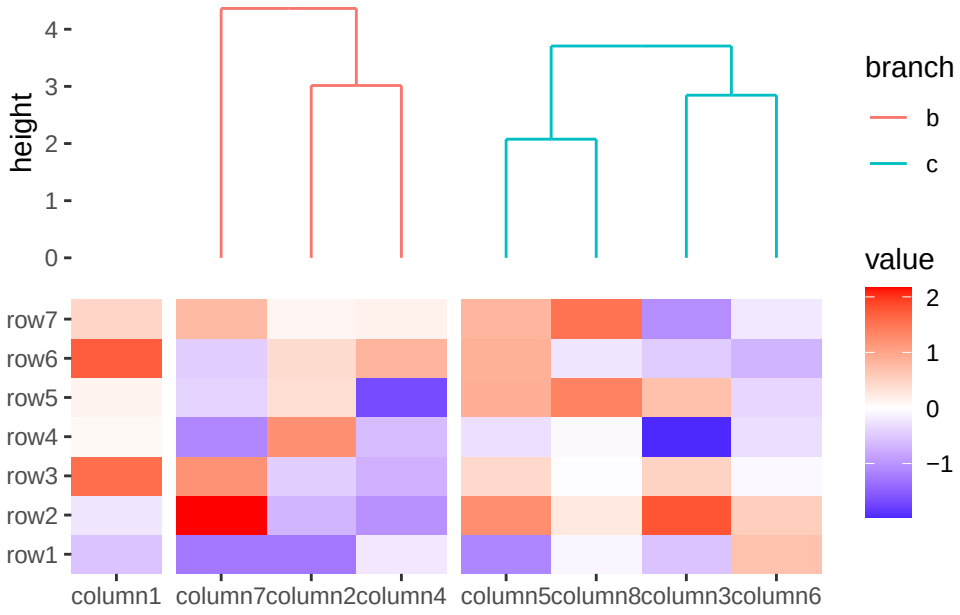
```
> heatmap built with `geom_tile()`
> heatmap built with `geom_tile()`
```



`align_dendro()` 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_dendro(aes(color = branch))
```

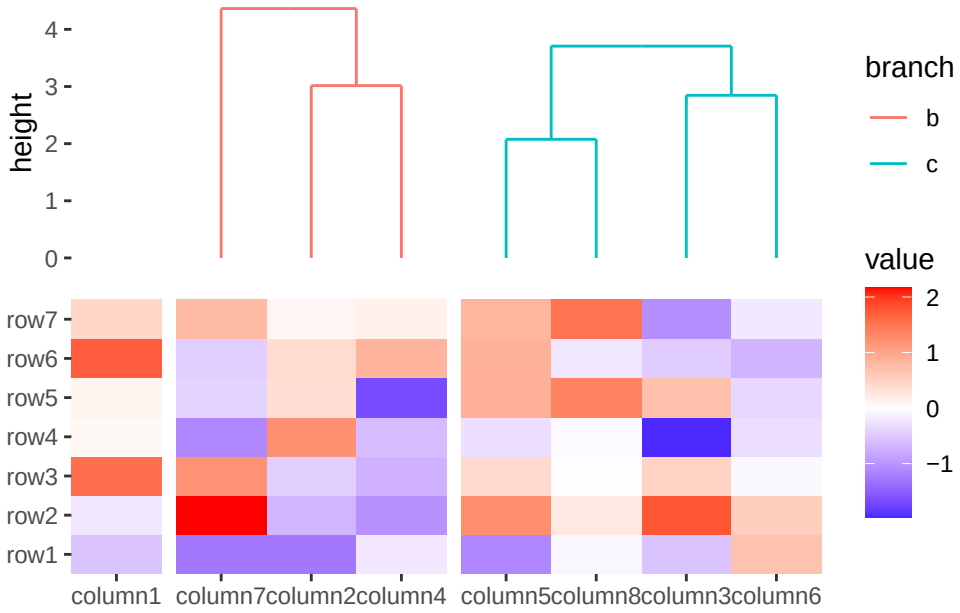
```
> heatmap built with `geom_tile()`
```



You can reorder the groups by setting `reorder_group = TRUE`.

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_dendro(aes(color = branch), reorder_group = TRUE)
```

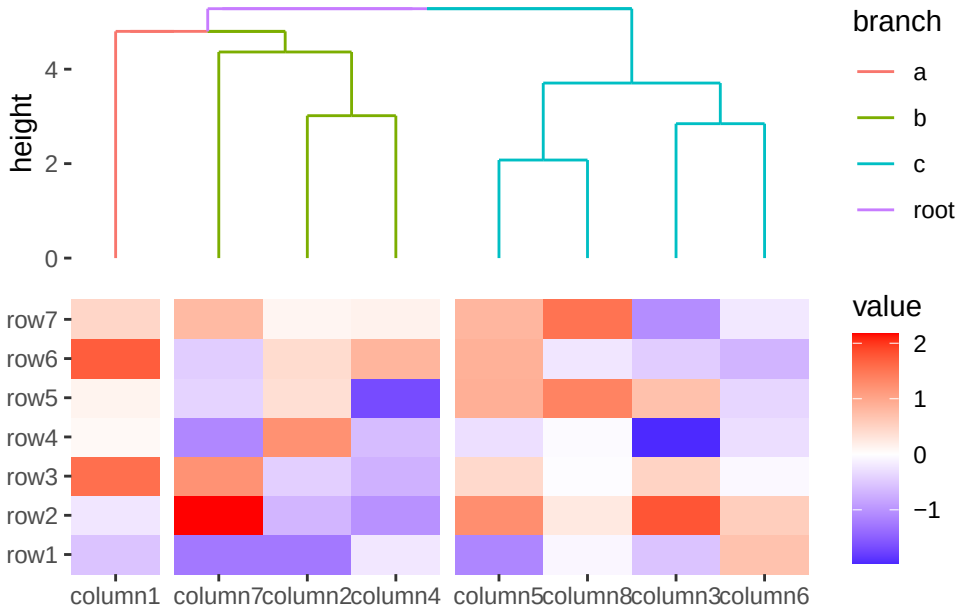
```
> heatmap built with `geom_tile()`
```



You can merge the sub-tree in each group by settting `merge_dendrogram = TRUE`.

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_dendro(aes(color = branch), merge_dendrogram = TRUE)
```

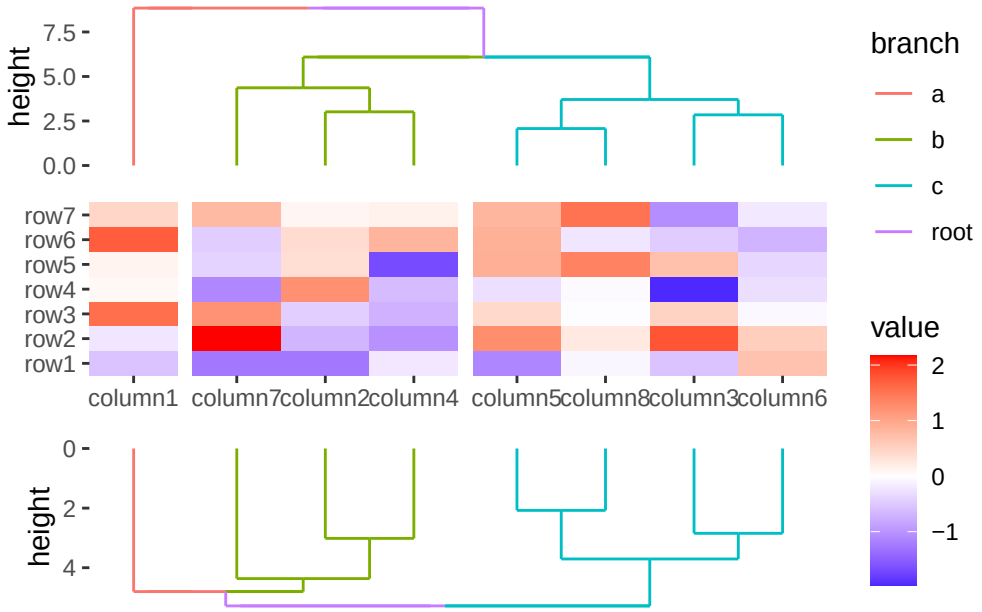
```
> heatmap built with `geom_tile()`
```



You can reorder the dendrogram and merge simutaneously.

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_dendro(aes(color = branch),
        reorder_group = TRUE,
        merge_dendrogram = TRUE
    ) +
    anno_bottom() +
    align_dendro(aes(color = branch),
        reorder_group = FALSE,
        merge_dendrogram = 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_dendro()`, where new groups must be nested within the previously established groups.

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

```
Error in `align()`:
! align_dendro(k = 2L) disrupt the previously established panel groups
  of the top annotation `stack_align()`
```