Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II. http://www.jstatsoft.org/

The Strucplot Framework: Visualizing Multi-way Contingency Tables With vcd

David Meyer
Wirtschaftsuniversität Wien

Achim Zeileis
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

Abstract

This paper describes the 'strucplot' framework for the visualization of multi-way contingency tables. Strucplot displays include hierarchical conditional plots such as mosaic, association, and sieve plots, and can be combined into more complex, specialized plots for visualizing conditional independence, GLMs, and the results of independence tests. The frameworks' modular design allows flexible customization of the plots' graphical appearance, including shading, labeling, spacing, and legend, by means of graphical appearance control ('grapcon') functions. The framework is provided by the **vcd** package freely available from the Comprehensive R Archive Network (CRAN).

Keywords: contingency tables, mosaic plots, association plots, sieve plots, R, categorical data, independence, conditional independence, HSV, HCL, residual-based shading, **grid**.

1. Introduction

In order to explain multi-dimensional categorical data, statisticians typically look for (conditional) independence structures. Whether the task is purely exploratory or model based, techniques such as mosaic and association plots offer good support for visualization. Both have been extended over the last two decades, and implementations exist in many statistical environments. Both graphical methods visualize aspects of (possibly higher-dimensional) contingency tables. A *mosaic plot* (Hartigan and Kleiner 1984) is basically an area-proportional visualization of (typically observed) frequencies, composed of tiles (corresponding to the cells) created by recursive vertical and horizontal splits of a square. Thus, the area of each tile is proportional to the corresponding cell entry *given* the dimensions of previous splits. An *association plot* (Cohen 1980) visualizes the standardized deviations of observed frequencies from those expected under a certain independence hypothesis. Each cell is represented by a rectangle that has (signed) height proportional to the residual and width proportional to the square root of the expected counts, so that the area of the box is proportional to the difference in observed and expected frequencies.

Over the years, extensions to these techniques were mainly focused on five aspects:

1. Varying the shape of bar plots and mosaic displays to yield, e.g., double-decker plots (Hofmann 2001), spine plots, or spinograms.
2. Using residual-based shadings to visualize log-linear models (Friendly 1994, 2000) and significance of statistical tests (Meyer, Zeileis, and Hornik 2003).
3. Using pairs plots and trellis-like layouts for marginal, conditional and partial views (Friendly 1999).
4. Adding direct user interaction, allowing quick exploration and modification of the visualized models (Unwin, Hawkins, Hofmann, and Siegl 1996; Theus 2003).
5. Providing a modular and flexible implementation to easily allow user extensions (Meyer *et al.* 2003).

Current implementations of mosaic displays can be found, e.g., for SAS, ViSta, MANET, Mondrian, R, and S-PLUS. Table 1 gives an overview of the available functionality in these systems.

	SAS	S-PLUS	R	ViSta	MANET/Mondrian
Basic functionality	×	×	×	×	×
Shape			×		×
Residual-based shadings	×		×	×	(×)
Conditional Views	×		×		×
Interaction				×	×
Extensible Design			×		

Table 1: Comparison of current software environments

Figures 1 to 4 illustrate some of these extensions. Figure 1 shows the results from a double-blind clinical trial investigating a new treatment for rheumatoid arthritis, using an extended mosaic plot with residual-based shading based on the maximum statistic: clearly, the new treatment is effective. Figure 2 visualizes the well-known UCB admissions data by the means of a conditional association plot. The panels show the residuals from a conditional independence model (independence of gender and admission, given department), stratified by department. Clearly, the situation in department A (more women/less men accepted than would be expected under the null) causes the rejection of the hypothesis of conditional independence. Figure 3 illustrates the conditional independence of premarital and extramarital sex, given gender and marital status. The Chi-squared test of independence rejects the null hypothesis: possibly, because too many men who had premarital sex also had extramarital sex? Finally, Figure 4 visualizes the 'Survival on the Titanic' data using a double-decker plot. Here, a binary response (survival to the disaster) shall be explained by other factors (class, gender, and age). The blue boxes represent the proportion of survived passengers in a particular stratum. The proportions of saved women and children are indeed higher than those of men, but they clearly decrease from the 1st to the 3rd class. In addition, the proportion of saved men in the 1st class is higher than in the others.

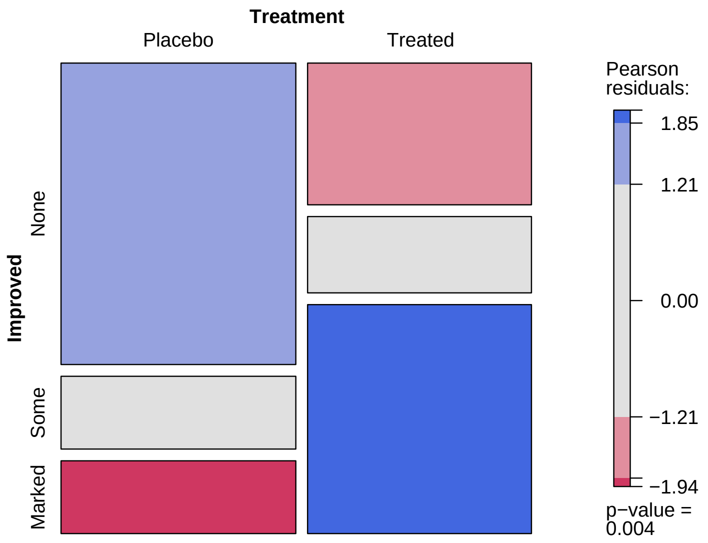


Figure 1: Mosaic plot for the arthritis data.

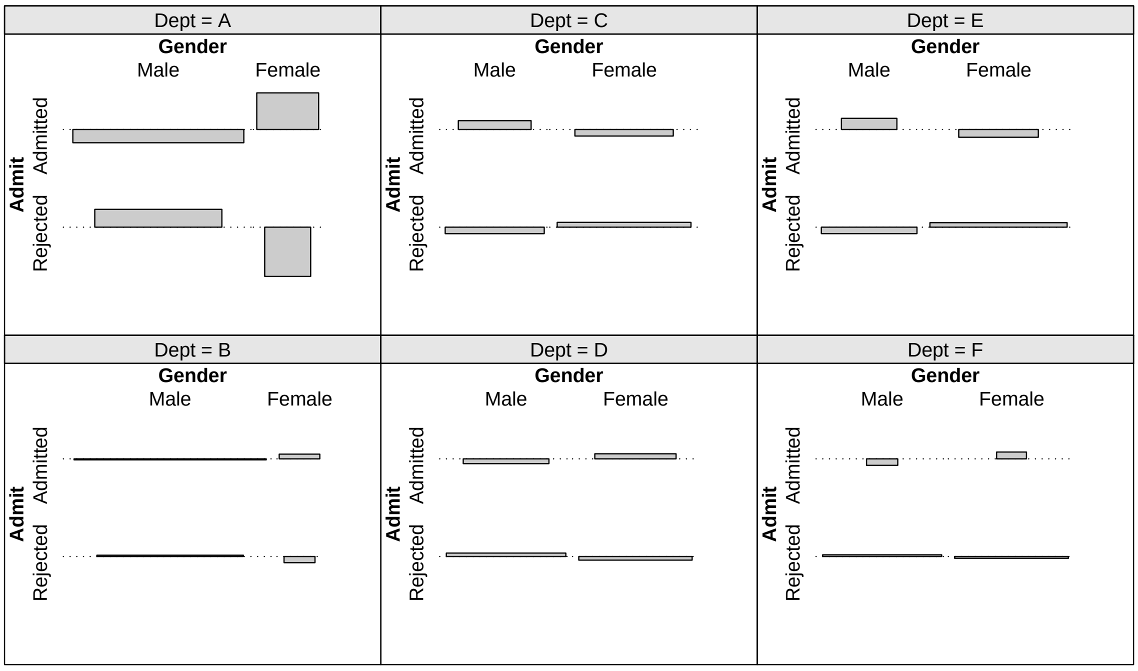


Figure 2: Conditional association plot for the UCB admissions data.

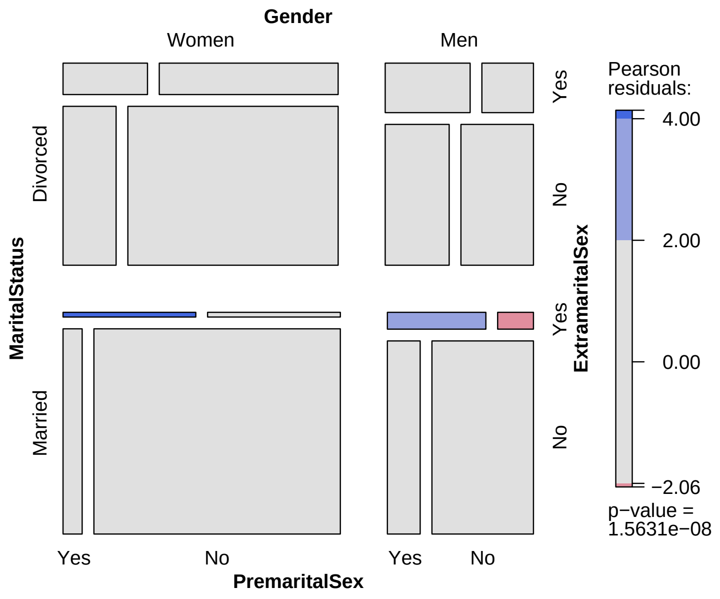


Figure 3: Mosaic plot for the pre- and extramarital sex data.

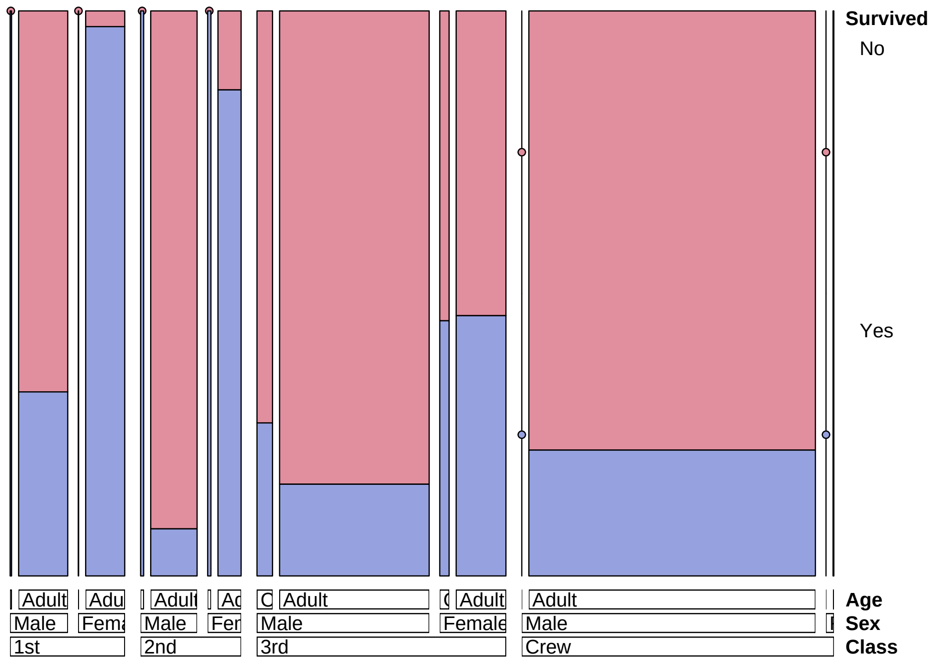


Figure 4: Double-decker plot for the survival on the Titanic data.

This paper describes the strucplot framework provided by the **vcd** package for the R system and environment for statistical computing and graphics, available from the Comprehensive R Archive Network (CRAN). The framework integrates displays such as mosaic, association, and sieve plots by their common property of being flat representations of contingency tables. These basic plots, as well as specialized displays for conditional independence, can be used both for exploratory visualization and model-based analysis. Exploratory techniques include specialized displays for the bivariate case, as well as pairs and trellis plot-like displays for higher-dimensional tables. Model-based tools include methods suitable for the visualization of conditional independence tests (including permutation tests), as well as for the visualization of particular GLMs (logistic regression, log-linear models). Additionally, two of the frameworks' further strengths are its flexibility and extensibility: graphical appearance aspects such as shading, labeling, and spacing are modularized by means of 'graphical appearance functions', allowing fine-granular customization and user-level extensions.

The reminder of the paper is organized as follows. In Section 2, we give an overview of the strucplot framework, describing the hierarchy of the main components and the basic functionality. In Section 3, we demonstrate how residual-based shadings support the visualization of log-linear models and the results of independence tests. Also, we explain step-by-step how the concepts of generating functions and graphical appearance control functions can be combined to provide a flexible customization of complex graphical displays as created by the strucplot framework. Sections 4 and 5 discuss in detail the labeling and spacing features, respectively. Section 6 finally concludes the work.

2. The Strucplot Framework

The strucplot framework in the R package **vcd**, used for visualizing multi-way contingency tables, integrates techniques such as mosaic displays, association plots, and sieve plots. The main idea is to visualize the tables' cells arranged in rectangular form. For multi-way tables, the variables are nested into rows and columns using recursive conditional splits, given the margins. The result is a 'flat' representation that can be visualized in ways similar to a two-dimensional table. This principle defines a class of conditional displays which allows for granular control of graphical appearance aspects, including:

- the content of the tiles
- the split direction for each dimension
- the graphical parameters of the tiles' content
- the spacing between the tiles
- the labeling of the tiles

The strucplot framework is highly modularized: Figure 5 shows the hierarchical relationship between the various components. On the lowest level, there are several groups of workhorse and parameter functions that directly or indirectly influence the final appearance of the plot. These are examples of 'graphical appearance control' ('grapcon') functions. They are created by generating functions ('grapcon generators'), allowing flexible parameterization and extensibility (Figure 5 only shows the generators). The first part of the generator names (`group_foo()`) reflects the group they belong to (strucplot core, labeling, legend, shading, or spacing). The workhorse functions (created by `struc_foo()`, `labeling_foo()`, and `legend_foo()`) directly produce graphical output ("add ink to the canvas"), whereas the parameter functions (created by `spacing_foo()` and `shading_foo()`) compute graphical parameters used by the others. The grapcon functions returned by `struc_foo()` implement the core functionality, creating the tiles and their content. On the second level of the framework, a suitable combination of the low-level grapcon functions (or, alternatively, corresponding generating functions) is passed as "hyperparameters" to `strucplot()`. This central function sets up the graphical layout using grid viewports (see Figure 6), and coordinates the specified core, labeling, shading, and spacing functions to produce the plot. On the third level, we provide several convenience functions such as `mosaic()`, `sieve()`, `assoc()`, and `doubledecker()` which interface `strucplot()` through sensible parameter defaults and support for model formulas. Finally, on the fourth level, there are 'related' **vcd** functions (such as `cotabplot()` and the `pairs()` methods for table objects) arranging collections of plots of the strucplot framework into more complex displays (e.g., by means of panel functions).

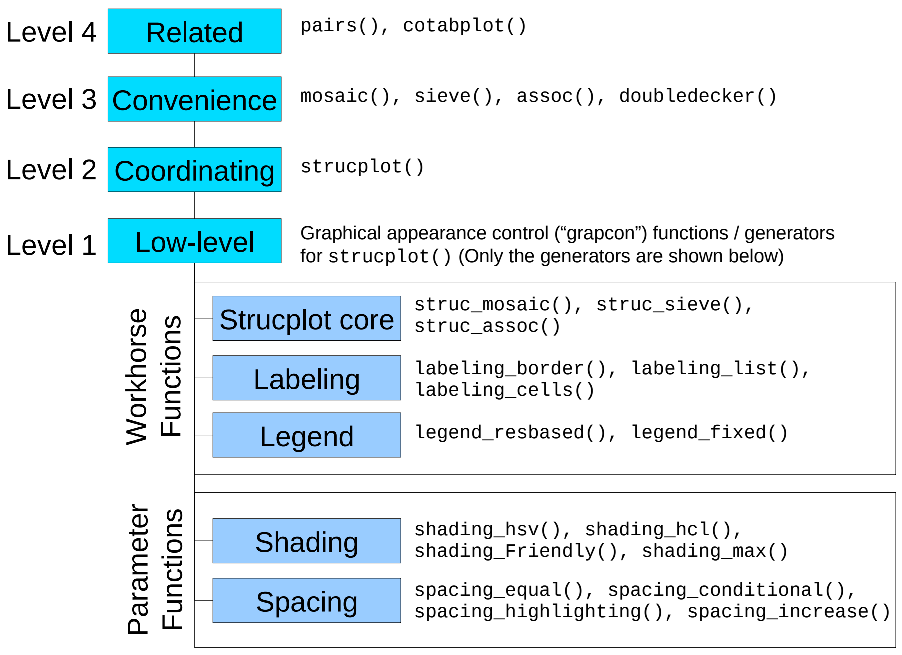

Level 4
Related
pairs(), cotabplot()
Level 3
Convenience
mosaic(), sieve(), assoc(), doubledecker()
Level 2
Coordinating
strucplot()
Level 1
Low-level
Graphical appearance control ("grapcon") functions / generators for strucplot() (Only the generators are shown below)
Workhorse Functions
Strucplot core
struc_mosaic(), struc_sieve(), struc_assoc()
Labeling
labeling_border(), labeling_list(), labeling_cells()
Legend
legend_resbased(), legend_fixed()
Parameter Functions
Shading
shading_hsv(), shading_hcl(), shading_Friendly(), shading_max()
Spacing
spacing_equal(), spacing_conditional(), spacing_highlighting(), spacing_increase()

Figure 5: Components of the strucplot framework.

2.1. Mosaic, Association, and Sieve Plots

As an example, consider the '`HairEyeColor`' data containing two polytomous variables (hair and eye color), as well as one (artificial) dichotomous variable (sex, i.e., gender). The 'flattened' contingency table can be obtained using the `structable()` function (quite similar to `ftable()` in base R, but allowing the specification of split directions):

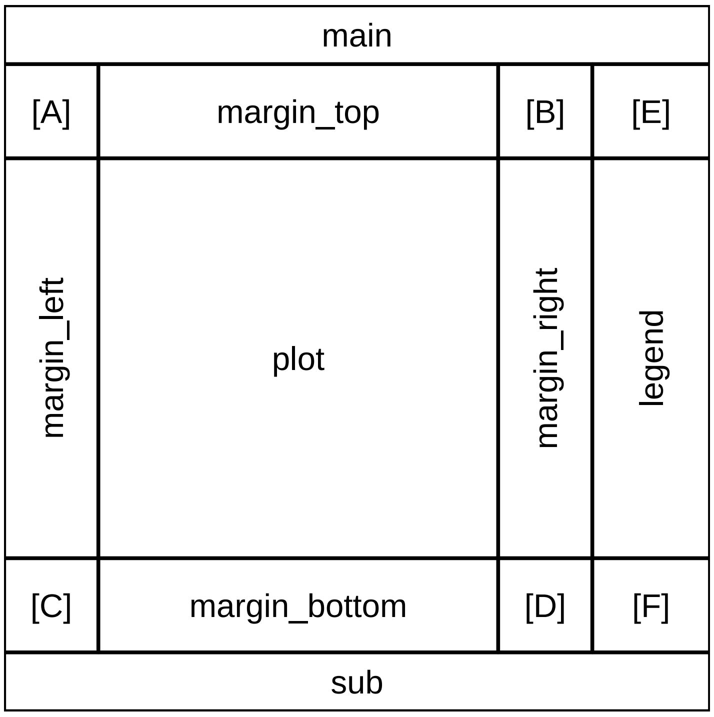


Figure 6: Viewport layout for strucplot displays with their names. [A] = "corner_top_left", [B] = "corner_top_right", [C] = "corner_bottom_left", [D] = "corner_bottom_right", [E] = "legend_top", [F] = "legend_sub".

```
> (hec <- structable(Eye ~ Sex + Hair, data = HairEyeColor))

             Eye Brown Blue Hazel Green
Sex    Hair
Male   Black        32   11    10     3
       Brown        38   50    25    15
       Red          10   10     7     7
       Blond         3   30     5     8
Female Black        36    9     5     2
       Brown        81   34    29    14
       Red          16    7     7     7
       Blond         4   64     5     8
```

Let us first visualize the contingency table by means of a mosaic plot (Hartigan and Kleiner 1984) which is basically an area-proportional visualization of (typically, observed) frequencies, composed of tiles (corresponding to the cells) created by recursive vertical and horizontal splits of a square. Thus, the area of each tile is proportional to the corresponding cell entry *given* the dimensions of previous splits. Figure 7 depicts the effect of

```
> mosaic(hec)
```

equivalent to

```
> mosaic(~Sex + Eye + Hair, data = HairEyeColor)
```

The small bullets indicate zero entries in the corresponding cell. Note that in contrast to, e.g., `mosaicplot()` in base R, the default split direction and level ordering in all strucplot displays correspond to the textual representation. It is also possible to visualize the expected values instead of the observed values (see Figure 8):

```
> mosaic(hec, type = "expected")
```

In order to compare observed and expected values, a sieve plot (Riedwyl and Schüpbach 1994) could be used (see Figure 9):

```
> sieve(hec)
```

Alternatively, we can directly inspect the residuals. The Pearson residuals (standardized deviations of observed from expected values) are preferably visualized using association plots (Cohen 1980). In contrast to `assocplot()` in base R, **vcd**'s `assoc()` function scales to more than two variables (see Figure 10):

```
> assoc(hec, compress = FALSE)
```

The `compress` argument keeps distances between tiles equal for better comparison.

For both mosaic plots and association plots, the splitting of the tiles can be controlled using the `split_vertical` argument (default: alternating splits starting with a vertical one).

```
> mosaic(hec, split_vertical = c(TRUE, FALSE, TRUE),
+     labeling_args = list(abbreviate = c(Eye = 3)))
```

For compatibility with `mosaicplot()` in base R, the `mosaic()` function also allows the use of a `"direction"` argument taking a vector of `"h"` and `"v"` characters (see Figure 11):

```
> mosaic(hec, direction = c("v", "h", "v"))
```

By a suitable combination of splitting, spacing, and labeling settings, the functions provided by the strucplot framework can be customized in a quite flexible way. For example, `doubledecker()` is simply a wrapper for `mosaic()`, setting the right defaults. Figure 12 shows a doubledecker plot of the '`Titanic`' data, explaining the probability of survival ('survived') by age, given sex, given class. It is created by:

```
> doubledecker(Titanic)
```

equivalent to:

```
> doubledecker(Survived ~ Class + Sex + Age, data = Titanic)
```

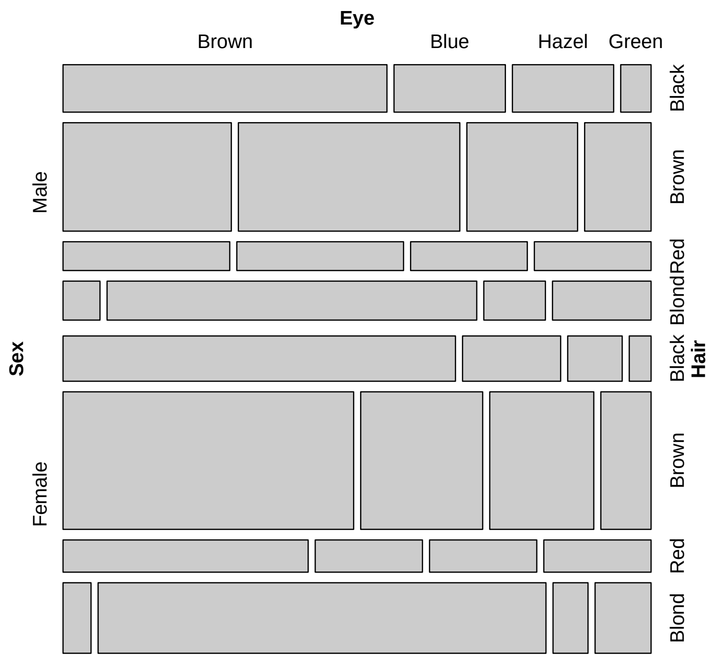


Figure 7: Mosaic plot for the ‘`HairEyeColor`’ data.

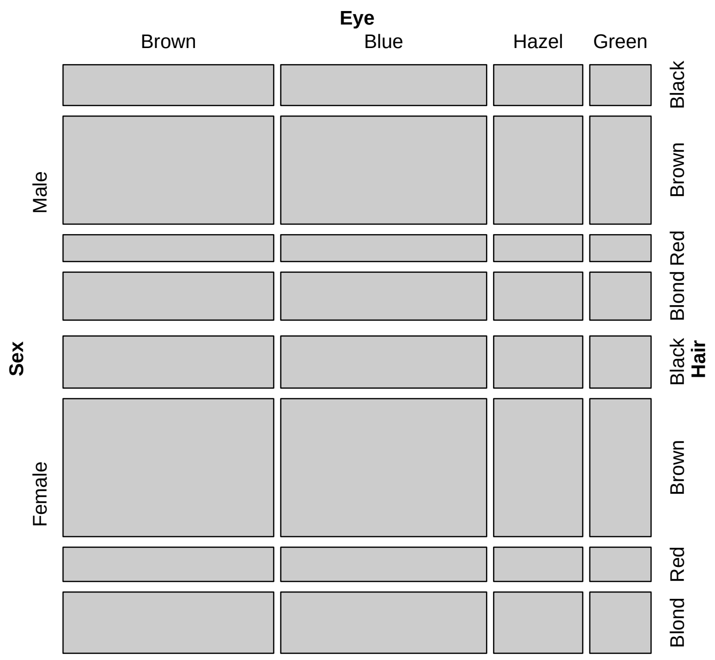


Figure 8: Mosaic plot for the ‘`HairEyeColor`’ data (expected values).

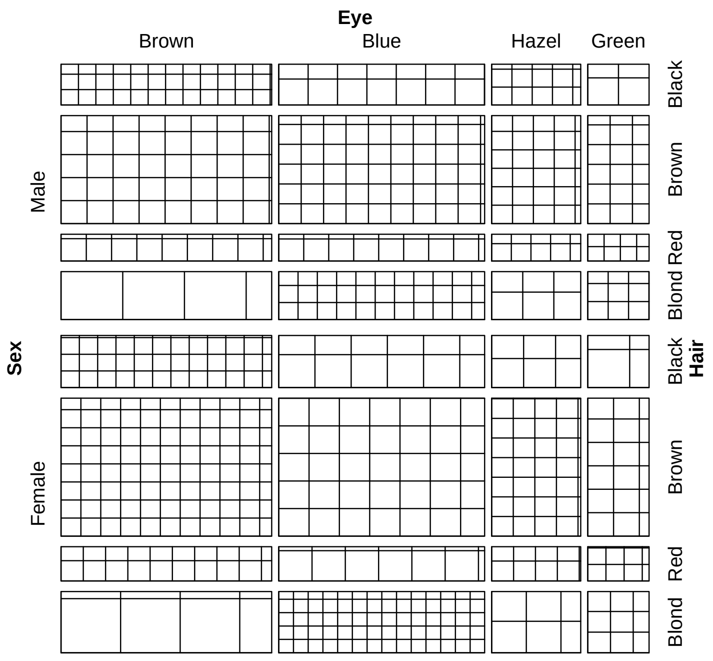


Figure 9: Sieve plot for the 'HairEyeColor' data visualizing simultaneously observed and expected values.

2.2. Conditional and partial views

So far, we have visualized full tables. For objects of class table, conditioning on levels (i.e., choosing a table subset for fixed levels of the conditioning variable(s)) is simply done by indexing. However, subsetting "structable" objects is more restrictive because of their inherent conditional structure. Since the variables on both the row and the columns side are nested, conditioning is only possible "outside-in":

```
> hec
```

```
              Eye Brown Blue Hazel Green
Sex    Hair
Male   Black         32   11    10     3
       Brown         38   50    25    15
       Red           10   10     7     7
       Blond          3   30     5     8
Female Black         36    9     5     2
       Brown         81   34    29    14
       Red           16    7     7     7
       Blond          4   64     5     8
```

```
> hec["Male", ]
```

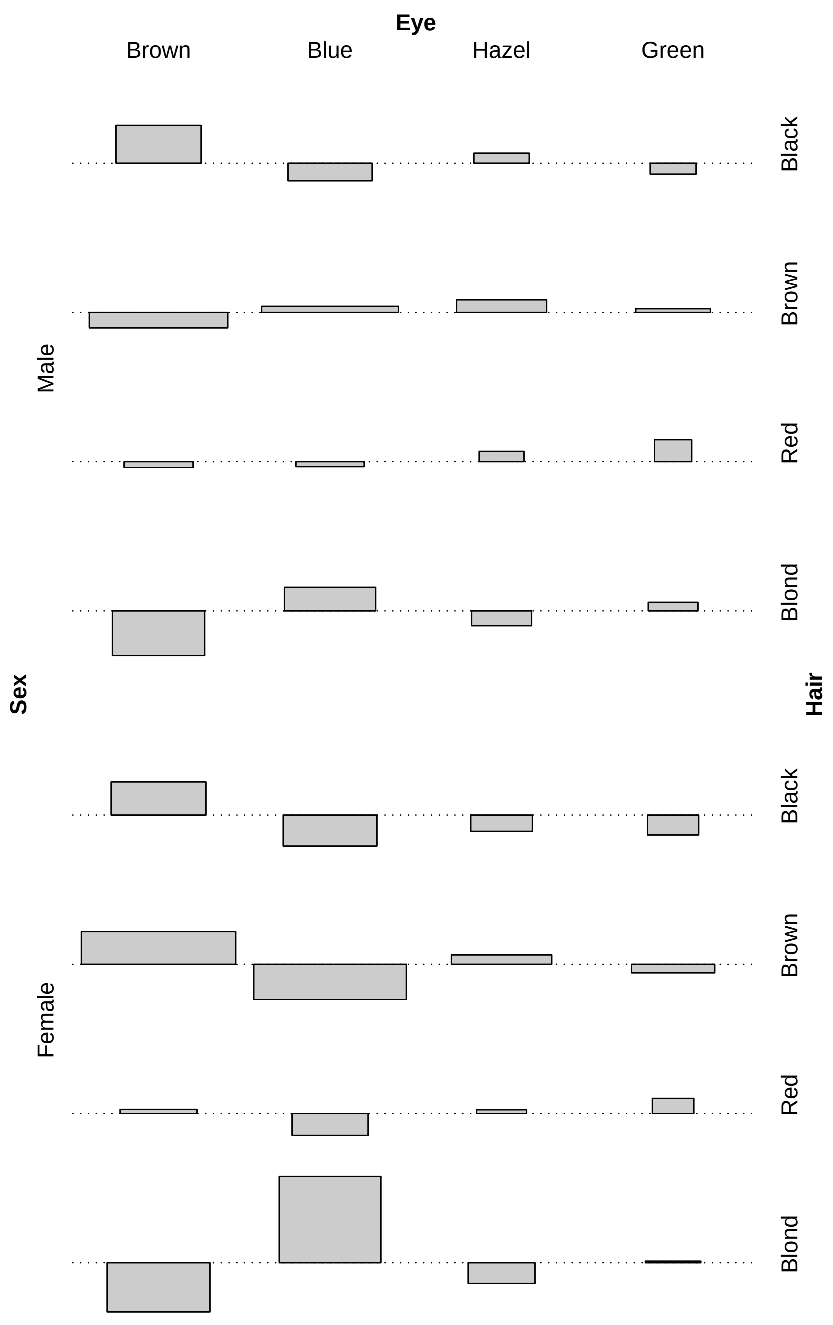


Figure 10: Association plot for the '`HairEyeColor`' data.

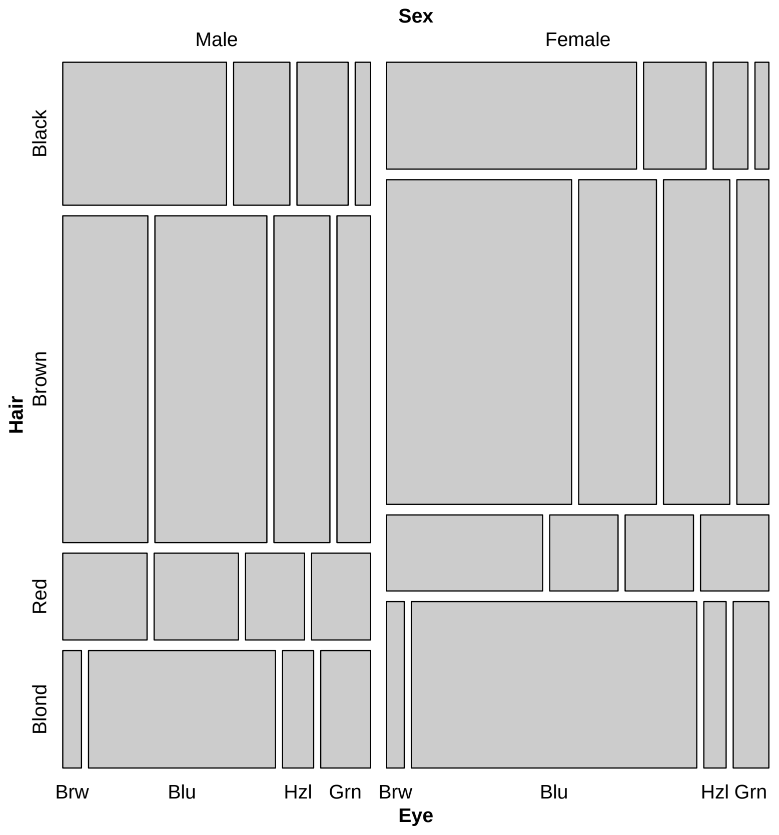


Figure 11: Mosaic plot for the ‘`HairEyeColor`’ data—alternative splitting.

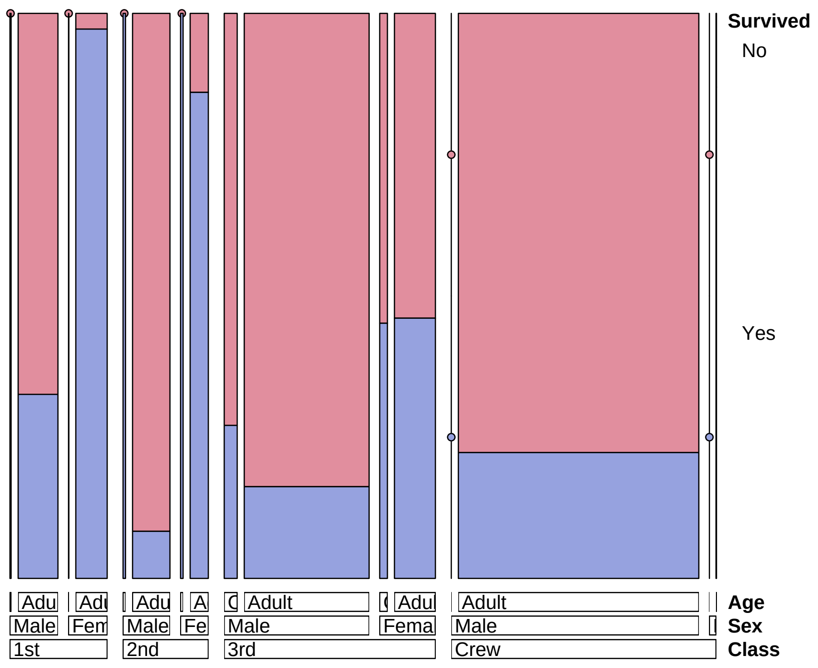


Figure 12: Doubledecker plot for the 'Titanic' data.

```
      Eye Brown Blue Hazel Green
Hair
Black         32   11    10     3
Brown         38   50    25    15
Red           10   10     7     7
Blond          3   30     5     8

> hec[c("Male", "Brown"), ]

Eye Brown Blue Hazel Green

       38    50    25    15

> hec["Male", "Green"]

Hair
Black   3
Brown  15
Red     7
Blond   8
```

Now, there are several ways for visualizing conditional independence structures. The "brute force" method is to draw separate plots for the strata. The following example compares the association between hair and eye color, given gender, by using subsetting on the flat table and **grid**'s viewport framework to visualize the two groups besides each other:

```
> pushViewport(viewport(layout = grid.layout(ncol = 2)))
> pushViewport(viewport(layout.pos.col = 1))
> mosaic(hec["Male"], margins = c(left = 2.5, top = 2.5,
+     0), sub = "Male", newpage = FALSE)
> popViewport()
> pushViewport(viewport(layout.pos.col = 2))
> mosaic(hec["Female"], margins = c(top = 2.5, 0), sub = "Female",
+     newpage = FALSE)
> popViewport(2)
```

Note the use of the `margins` argument: it takes a vector with up to four values whose unnamed components are recycled, but "overruled" by the named arguments. Thus, in the example, only the top margin is set to 2 lines, and all other to 0. This idea applies to almost all vectorized arguments in the strucplot framework (with `split_vertical` as a prominent exception).

Since mosaic displays are "conditional plots" by definition, we can also use one single mosaic for stratified plots. The formula interface of `mosaic()` allows the specification of conditioning variables (see Figure 14):

```
> mosaic(~Hair + Eye | Sex, data = hec, split_vertical = TRUE,
+     keep_aspect_ratio = FALSE)
```

The effect of using this kind of formula is that conditioning variables are permuted ahead of the the conditioned variables in the table, and that `spacing_conditional()` is used as default to better distinguish conditioning from conditioned dimensions. This spacing uses equal space between tiles of conditioned variables, and increasing space between tiles of conditioning variables. In addition, we release the fixed aspect ratio to get less distorted margins.

The `cotabplot()` function does a much better job on this task: it arranges stratified strucplot displays in a lattice-like layout, conditioning on variable *levels*. The plot in Figure 15 shows hair and eye color, given sex:

```
> cotabplot(~Hair + Eye | Sex, data = hec, panel_args = list(margins = 3),
+     labeling = labeling_left(clip = FALSE))
```

The `labeling_args` argument modifies the labels' appearance: here, to be left-aligned and unclipped (see Section 4).

Another high-level function for visualizing conditional independence models are the `pairs()` methods for table and structable objects. In contrast to `cotabplot()` which conditions on variables, the `pairs()` methods create pairwise views of the table. The function produces, by default, a plot matrix having strucplot displays in the off-diagonal panels, and the variable names (optionally, with univariate statistics) in the diagonal cells. Figure 16 shows a pairs display with mosaic plots visualizing mutual independence in the lower triangle, association plots for the same in the upper triangle, and bar charts in the diagonal.

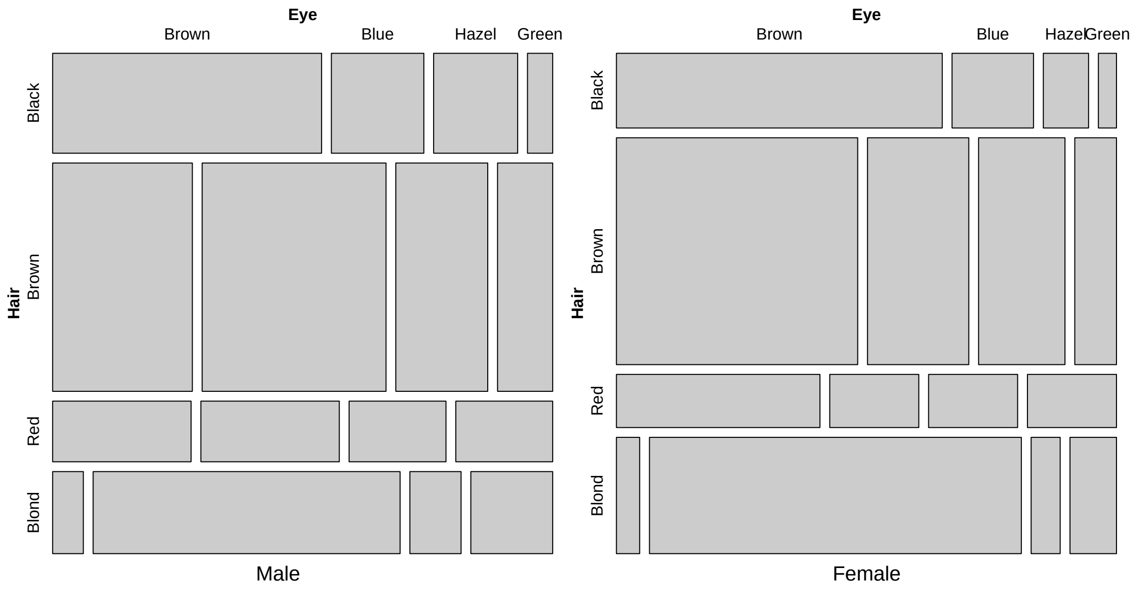


Figure 13: Distribution of hair and eye color, given gender.

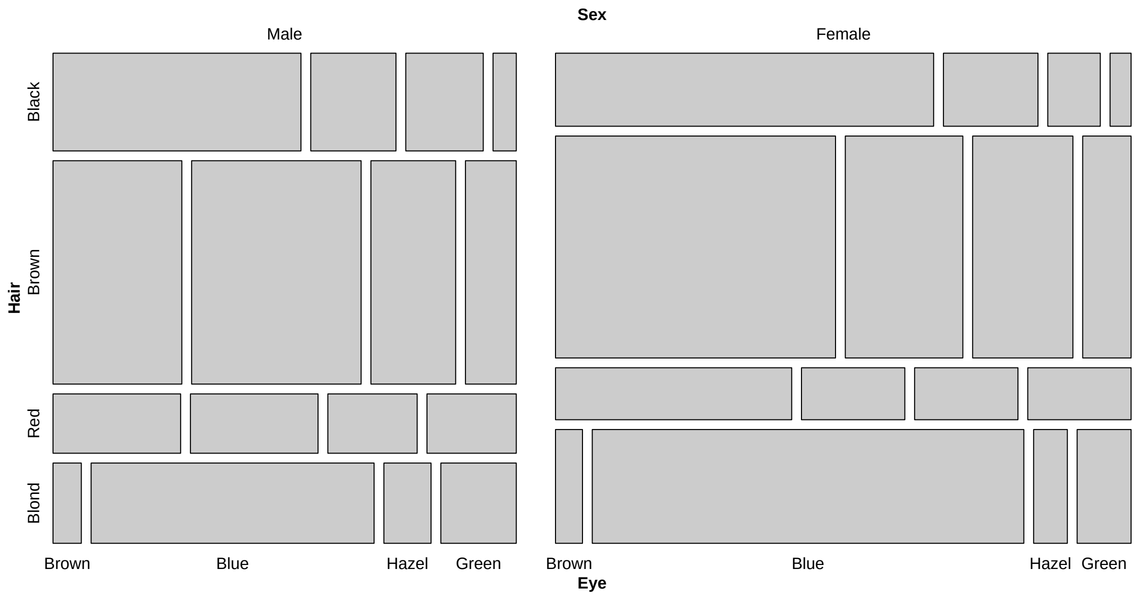


Figure 14: Mosaic plot for conditional independence structures.

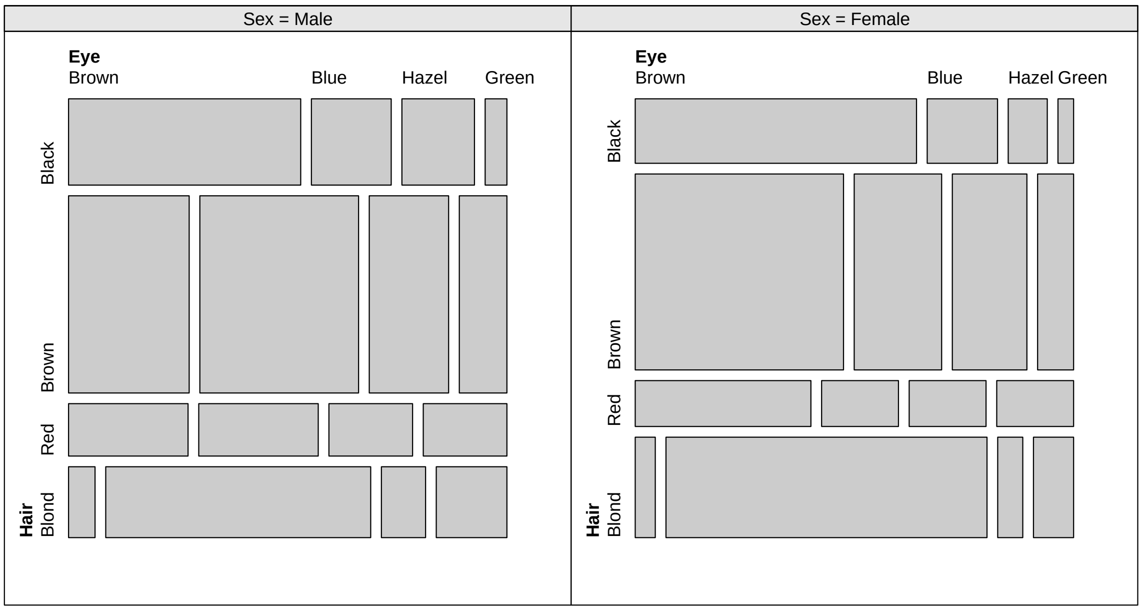


Figure 15: Conditional table plot for the 'HairEyeColor' data.

```
> pairs(hec, lower_panel = pairs_assoc, space = 0.3, diag_panel_args = list(rot = -45,
+     just_leveltext = c("left", "bottom")))
```

(The labels of the variables are to be read from left to right and from top to bottom.) In plots produced by `pairs()`, each panel's row and column define two variables X and Y used for the specification of four different types of independence: pairwise, total, conditional, and joint. The pairwise mosaic matrix shows bivariate marginal relations between X and Y, collapsed over all other variables. The total independence mosaic matrix shows mosaic plots for mutual independence, i.e., for marginal and conditional independence among all pairs of variables. The conditional independence mosaic matrix shows mosaic plots for marginal independence of X and Y, given all other variables. The joint independence mosaic matrix shows mosaic plots for joint independence of all pairs (X, Y) of variables from the others.

Since the matrix is symmetric, the upper and lower parts can independently be used to display different types of independence models, or different strucplots displays (mosaic, association, or sieve plots). The available panel functions (`pairs_assoc()`, `pairs_mosaic()`, and `pairs_sieve()`) are simple wrappers to `assoc()`, `mosaic()`, and `sieve()`, respectively. Obviously, seeing patterns in strucplot matrices becomes increasingly difficult with higher dimensionality. Therefore, this plot is typically used with a suitable residual-based shading (see Section 3).

2.3. Interactive plot modifications

All strucplot core functions are supposed to produce conditional hierarchical plots by the means of nested viewports, corresponding to the provided splitting information. Thus, at the end of the plotting, each tile is associated with a particular viewport. Each of those viewports

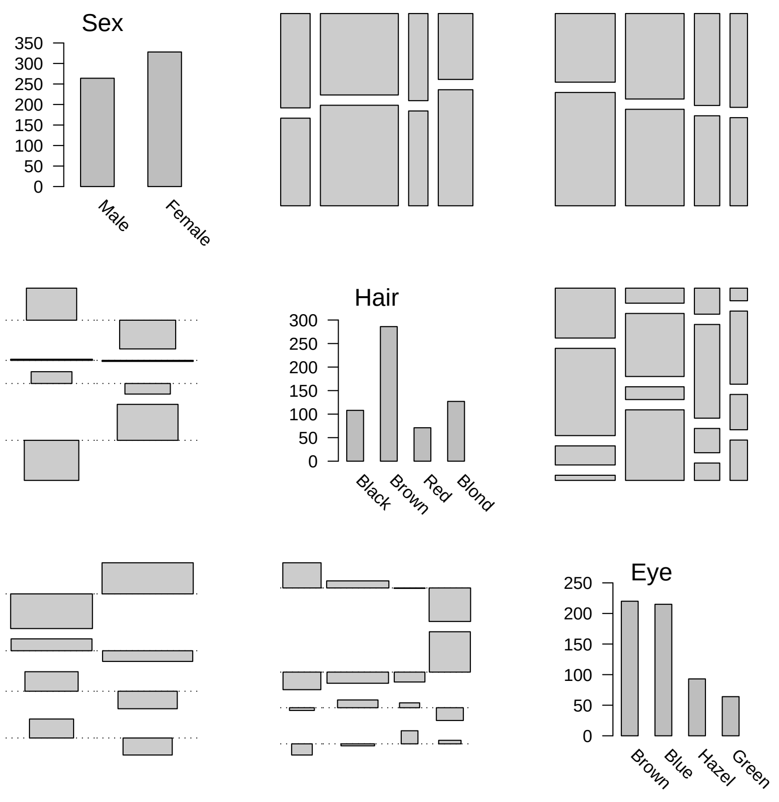


Figure 16: Pairs plot for the ‘`HairEyeColor`’ data.

has to be conventionally named, enabling other strucplot modules, in particular the labeling functions, to access specific tiles after they have been plotted. The naming convention for the viewports is:

`cell:`*`Variable1`*`=`*`Level1`*`,`*`Variable2`*`=`*`Level2`* ...

Clearly, these names depend on the splitting. The following example shows how to access parts of the plot after it has been drawn (see Figure 17):

```
> mosaic(~Hair + Eye, data = hec, pop = FALSE)
> seekViewport("cell:Hair=Blond")
> grid.rect(gp = gpar(col = "red", lwd = 4))
> seekViewport("cell:Hair=Blond,Eye=Blue")
> grid.circle(r = 0.2, gp = gpar(fill = "cyan"))
```

Note that the viewport tree is removed by default. Therefore, the `pop` argument has to be set to `FALSE` when viewports shall be accessed.

In addition to the viewports, the main graphical elements get names following a similar construction method. This allows to change graphical parameters of plot elements *after* the plotting (see Figure 18):

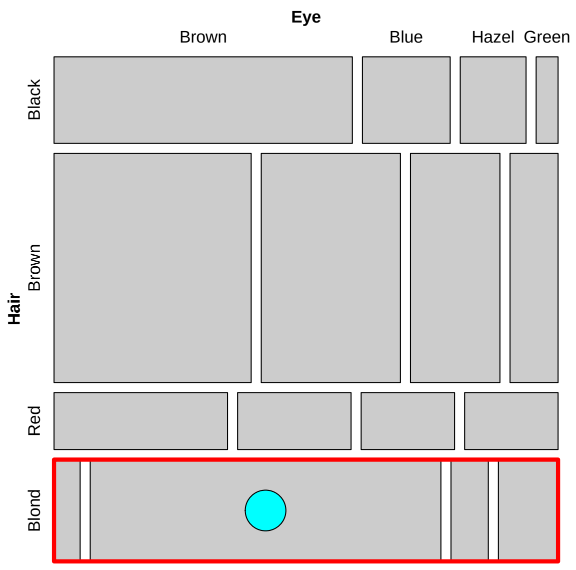


Figure 17: Adding elements to a mosaic plot after drawing.

```
> assoc(Eye ~ Hair, data = hec, pop = FALSE)
> getNames()[1:6]

[1] "GRID.GROB.2320"            "rect:Hair=Black,Eye=Brown"
[3] "GRID.GROB.2321"            "rect:Hair=Brown,Eye=Brown"
[5] "GRID.GROB.2322"            "rect:Hair=Red,Eye=Brown"

> grid.edit("rect:Hair=Blond,Eye=Blue", gp = gpar(fill = "red"))
```

3. Residual-based Shadings

Unlike other graphics functions in base R, the strucplot framework allows almost full control over the graphical parameters of all plot elements. In particular, in association plots, mosaic plots, and sieve plots, the user can modify the graphical appearance of each tile individually. Built on top of this functionality, the framework supplies a set of shading functions choosing colors appropriate for the visualization of log-linear models. The tiles' graphical parameters are set using the `gp` argument of the functions of the strucplot framework. This argument basically expects an object of class `gpar` whose components are arrays of the same shape (length and dimensionality) as the data table (see Section 3.1). For convenience, however, the user can also supply a specialized graphical appearance control ("grapcon") function that computes such an object given a vector of residuals, or, alternatively, a generating function that takes certain arguments and returns such a grapcon function (see Section 3.2). We provide several shading functions, including support for both HSV and HCL colors, and the visualization of significance tests (see Section 3.3).

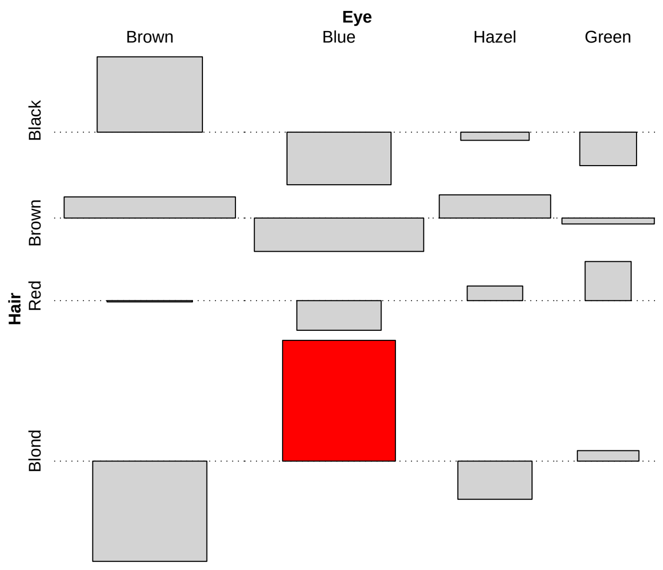


Figure 18: Changing graphical parameters of elements after drawing.

3.1. Specifying graphical parameters of strucplot displays

As an example, consider the ‘`UCBAdmissions`’ data. In the table aggregated over departments, we would like to highlight the (incidentally wrong) impression that there were too many male students accepted compared to the presumably discriminated female students (see Figure 19):

```
> (ucb <- margin.table(UCBAdmissions, 1:2))

          Gender
Admit      Male Female
  Admitted 1198    557
  Rejected 1493   1278

> (fill_colors <- matrix(c("dark cyan", "gray", "gray",
+     "dark magenta"), ncol = 2))

     [,1]        [,2]
[1,] "dark cyan" "gray"
[2,] "gray"      "dark magenta"

> mosaic(ucb, gp = gpar(fill = fill_colors, col = 0))
```

As the example shows, we create a fourfold table with appropriate colors (dark cyan for admitted male students and dark magenta for rejected female students) and supply them to the `fill` component of the `gpar` object passed to the `gp` argument of `mosaic()`. For visual clarity, we additionally hide the tiles’ borders by setting the `col` component to 0 (white).

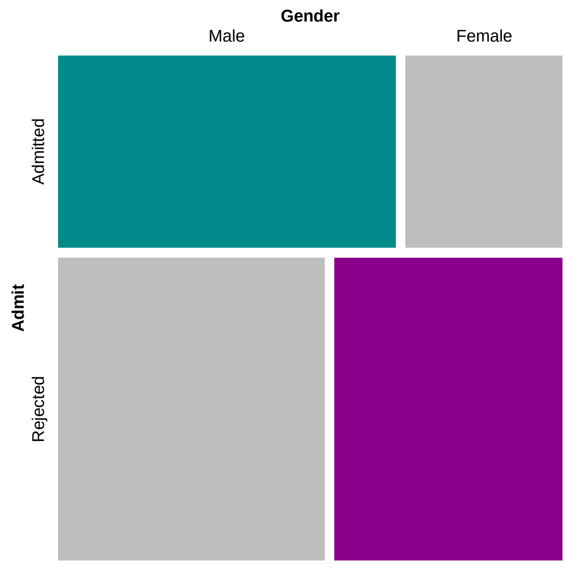


Figure 19: Mosaic plot for the ‘`UCBAdmissions`’ data with highlighted cells.

If the parameters specified in the `gpar` object are “incomplete”, they will be recycled along the last splitting dimension. In the following example based on the ‘`Titanic`’ data, we will highlight all cells corresponding to survived passengers (see Figure 20):

```
> mosaic(Titanic, gp = gpar(fill = c("gray", "dark magenta")),
+     spacing = spacing_highlighting, labeling_args = list(abbreviate = c(Age = 3),
+         rep = c(Survived = FALSE)))
```

Note that `spacing_highlighting()` sets the spaces between tiles in the last dimension to 0. The `labeling_args` argument ensures that labels do not overlap (see Section 4).

3.2. Customizing residual-based shadings

This flexible way of specifying graphical parameters is the basis for a suite of shading functions that modify the tiles’ appearance with respect to a vector of residuals, resulting from deviations of observed from expected values under a given log-linear model. The idea is to visualize at least sign and absolute size of the residuals, but some shadings, additionally, indicate overall significance. One particular shading, the maximum shading, even allows to identify those cells that cause the rejection of the null hypothesis.

Conceptually, the strucplot framework offers three alternatives to add residual-based shading to plots:

1. Precomputing the graphical parameters (e.g., fill colors), encapsulating them into an object of class `gpar` as demonstrated in the previous section, and passing this object to the `gp` argument.

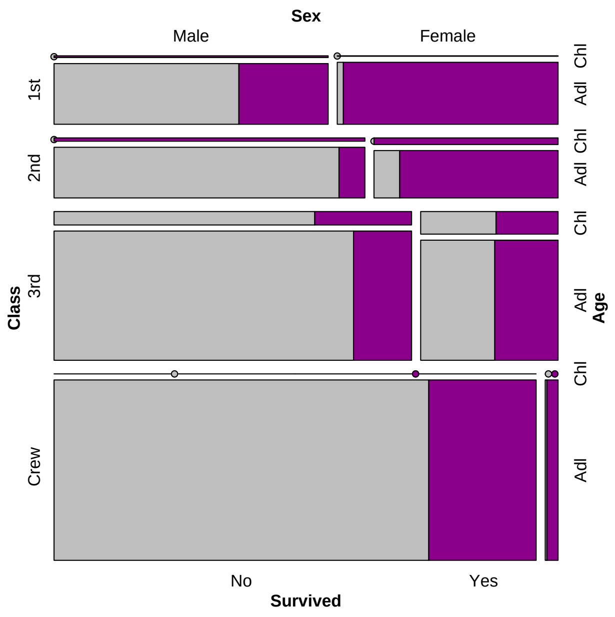


Figure 20: Recycling of parameters, used for highlighting the survived passengers in the ‘`Titanic`’ data.

2. Providing a grapcon function to the `gp` argument that takes residuals as input and returns an object as described in alternative 1.
3. Providing a grapcon *generating* function (‘grapcon generator’) taking parameters and returning a function as described in alternative 2.

For each of these approaches, we will demonstrate the necessary steps to obtain a binary shading that visualizes the sign of the residuals by a corresponding fill color (for simplicity, we will treat 0 as positive).

Alternative 1: Precomputed gpar *object*

The first method is precomputing the graphical parameters “by hand”. We will use ‘light blue’ color for positive and ‘light salmon’ color for negative residuals (see Figure 21):

```
> expected <- independence_table(ucb)
> (residuals <- (ucb - expected)/sqrt(expected))
```

```
          Gender
Admit           Male    Female
  Admitted  4.784093 -5.793466
  Rejected -3.807325  4.610614
```

```
> (shading1_obj <- ifelse(residuals > 0, "lightblue", "lightsalmon"))
```

```
          Gender
Admit      Male          Female
  Admitted "lightblue"   "lightsalmon"
  Rejected "lightsalmon" "lightblue"
```

```
> mosaic(ucb, gp = gpar(fill = shading1_obj))
```

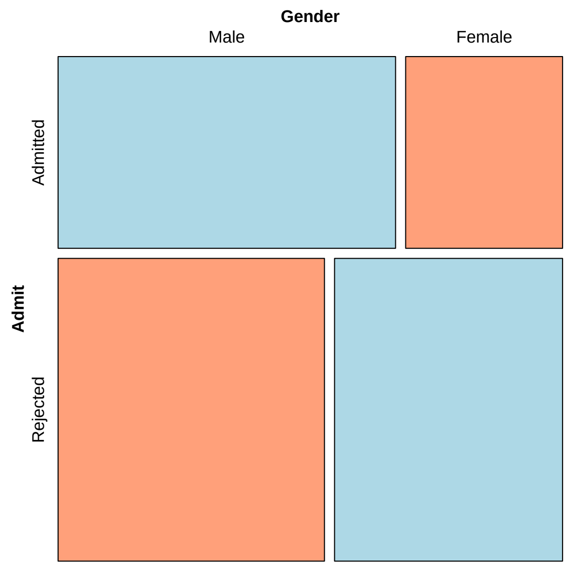


Figure 21: Binary shading visualizing the sign of the residuals.

Alternative 2: Grapcon function

For implementing alternative 2, we need to create a 'shading function' that computes `gpar` objects from residuals. For that, we can just reuse the code from the previous step:

```
> shading2_fun <- function(res) gpar(fill = ifelse(res >
+     0, "lightblue", "lightsalmon"))
```

To create a mosaic display with binary shading, it now suffices to specify the data table along with `shading2_fun()`:

```
> mosaic(ucb, gp = shading2_fun)
```

`mosaic()` internally calls `strucplot()` which computes the residuals from the specified independence model (total independence by default), passes them to `shading2_fun()`, and uses the `gpar` object returned to finally create the plot.

Our `shading2_fun()` function might be useful, but can still be improved: the hard-wired colors should be customizable. We cannot simply extend the argument list to include, e.g., a `fill = c("lightblue", "lightsalmon")` argument because `strucplot()` will neither know how to handle it, nor let us change the defaults. In fact, the interface of shading functions is fixed, they are expected to take exactly one argument: a table of residuals. This is where generating functions (alternative 3) come into play.

Alternative 3: Grapcon generator

We simply wrap our grapcon shading function in another function that takes all additional arguments it needs to use, possibly preprocesses them, and returns the actual shading function. This returned function will have access to the parameters since in R, nested functions are lexically scoped. Thus, the grapcon generator returns ('creates') a 'parameterized' shading function with the minimal standard interface `strucplot()` requires. The following example shows the necessary extensions for our running example:

```
> shading3a_fun <- function(col = c("lightblue", "lightsalmon")) {
+     if (length(col) != 2)
+         stop("Need exactly two colors!")
+     function(res) gpar(fill = ifelse(res > 0, col[1],
+         col[2]))
+ }
```

In the call to `mosaic()`, using the new `shading3a_fun()` function, we can now simply change the colors:

```
> mosaic(ucb, gp = shading3a_fun(c("red", "blue")))
```

(figure not shown). The procedure described so far is a rather general concept, applicable to a wide family of user-level **grid** graphics. Indeed, the customization of other components of the strucplot framework (labeling, spacing, legend, and core functions) follows the same idea. Now for the shading functions, more customization is needed. Note that `shading3a_fun()` needs to be evaluated by the user, even if the defaults are to be used. It is a better idea to let `strucplot()` call the generating function, which, in particular, allows the passing of arguments that are computed by `strucplot()`. Since shading functions can be used for visualizing significance (see Section 3.3), it makes sense for generating functions to have access to the model, i.e., observed and expected values, residuals, and degrees of freedom. For example, the `shading_max()` generating function computes a permutation distribution of the maximum statistic and p values for specified significance levels based on the observed table to create data-driven cut-off points. If this was done in the shading function itself, the permutation statistic would be recomputed every time the shading function is called, resulting in possibly severe performance loss and numerical inconsistencies. Therefore, generating functions for shadings are required to take at least the parameters `observed`, `expected`, `residuals`, and `df` (these are provided by the strucplot framework), followed by other parameters controlling the shading appearance (to be specified by the user):

```
> shading3b_fun <- function(observed = NULL, residuals = NULL,
+     expected = NULL, df = NULL, col = c("lightblue",
+         "lightsalmon")) {
```

```
+     if (length(col) != 2)
+         stop("Need exactly two colors!")
+     function(res) gpar(fill = ifelse(res > 0, col[1],
+         col[2]))
+ }
> class(shading3b_fun) <- "grapcon_generator"
```

In some sense, generating functions for shadings are parameterized both by the user and the strucplot framework. For shading functions that require model information, the user-specified parameters are to be passed to the `gp_args` argument instead, and for this to work, the generating function needs a class attribute to be distinguishable from the "normal" shading functions. For others (like our simple `shading3b_fun()`) this is optional, but recommended for consistency:

```
> mosaic(ucb, gp = shading3b_fun, gp_args = list(col = c("red",
+     "blue")))
```

The final `shading3b_fun()` pretty much resembles `shading_binary()`, one of the standard shading functions provided by the **vcd** package.

3.3. An overview of the shading functions in vcd

Friendly (1994) suggested a residual-based shading for the mosaic tiles that can also be applied to the rectangles in association plots (Meyer *et al.* 2003). Apart from `shading_binary()`, there are currently two basic shadings available in **vcd**: `shading_hcl()` and `shading_hsv()`, as well as two derived functions: `shading_Friendly()` building upon `shading_hsv()`, and `shading_max()` building upon `shading_hcl()`. `shading_hsv()` and `shading_hcl()` provide the same conceptual tools, but use different color spaces: the Hue-Saturation-Value (HSV) and the Hue-Chroma-Luminance (HCL) scheme, respectively. We will first expose the basic concept of these shading functions using the HSV space, and then briefly explain the differences to the HCL space (a detailed discussion can be found in Zeileis, Meyer, and Hornik 2005). The HCL space is trickier to use, but preferable to the HSV space from a perceptual point of view.

In the HSV space, colors are specified in three dimensions: Hue, Saturation ('colorfulness'), and Value ('lightness', amount of gray). These three dimensions are used by `shading_hsv()` to visualize information about the residuals and the underlying independence model. The hue indicates the residuals' sign: by default, blue for positive, and red for negative residuals. The saturation of a residual is set according to its size: high saturation for large, and low saturation for small residuals. Finally, the overall lightness is used to indicate the significance of a test statistic: light colors for significant, and dark colors for non-significant results.

As an example, we will visualize the association of hair and eye color in the '`HairEyeColor`' data set (see Figure 22)

```
> haireye <- margin.table(HairEyeColor, 1:2)
> mosaic(haireye, gp = shading_hsv)
```

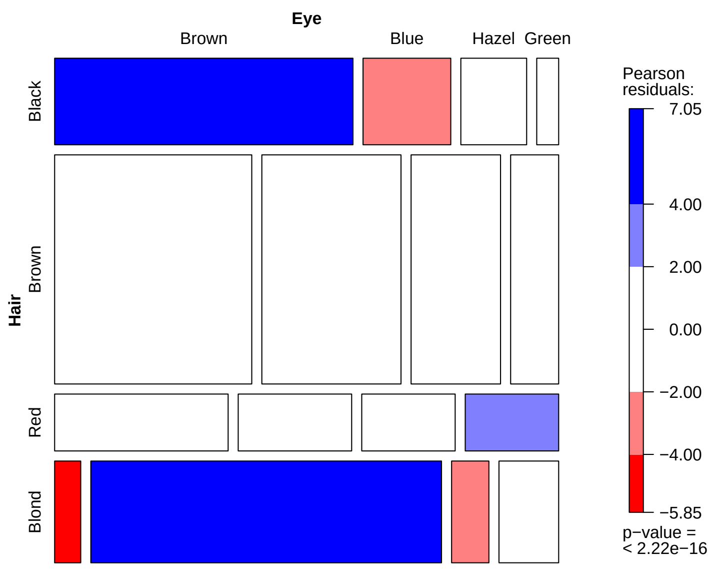


Figure 22: Shaded residuals in the ‘`HairEyeColor`’ data set—two cut-off points.

Large positive residuals (greater than 4) can be found for brown eyes/black hair and blue eyes/blond hair, and are colored in saturated blue. On the other hand, there is a large negative residual (less than -4) for brown eyes/blond hair, colored deep red. There are also three medium-sized positive (negative) residuals between 2 and 4 (-2 and -4): the colors for them are less saturated. Residuals between -2 and 2 are shaded in white. The heuristic for choosing the cut-off points 2 and 4 is that the Pearson residuals are approximately standard normal which implies that the highlighted cells are those with residuals *individually* significant at approximately the $\alpha = 0.05$ and $\alpha = 0.0001$ levels, respectively. These default cut-off points can be changed to alternative values using the `interpolate` argument (see Figure 23):

```
> mosaic(haireye, gp = shading_hsv, gp_args = list(interpolate = 1:4))
```

The elements of the numeric vector passed to `interpolate` define the knots of an interpolating step function used to map the absolute residuals to saturation levels. The `interpolate` argument also accepts a user-defined function, which then is called with the absolute residuals to get a vector of cut-off points. Thus, it is possible to automatically choose the cut-off points in a data-driven way. For example, one might think that the extension from four cut-off points to a continuous shading—visualizing the whole range of residuals—could be useful. We simply need a one-to-one mapping from the residuals to the saturation values:

```
> ipol <- function(x) pmin(x/4, 1)
```

Note that this `ipol()` function maps residuals greater than 4 to a saturation level of 1. However, the resulting plot (Figure 24) is deceiving:

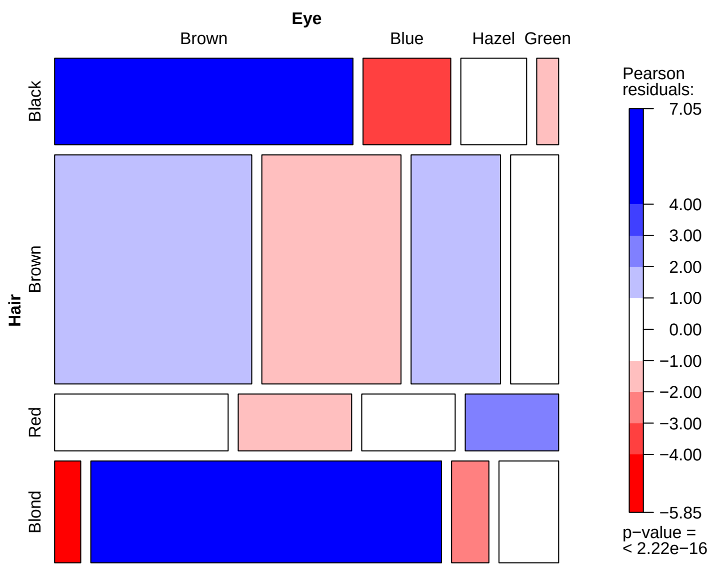


Figure 23: Shaded residuals in the '`HairEyeColor`' data set—four cut-off points.

```
> mosaic(HairEyeColor, gp = shading_hsv, gp_args = list(interpolate = ipol),
+     labeling_args = list(abbreviate = c(Sex = TRUE)))
```

Too much color makes it difficult to interpret the image, and the subtle color differences are hard to catch. Therefore, we only included shadings with discrete cut-off points.

The third remaining dimension, the value, is used for visualizing the significance of a test statistic. The user can either directly specify the p value, or, alternatively, a function that computes it, to the `p.value` argument. Such a function must take observed and expected values, residuals, and degrees of freedom (used by the independence model) as arguments. If nothing is specified, the p value is computed from a χ^2 distribution with `df` degrees of freedom. The `level` argument is used to specify the confidence level: if `p.value` is smaller than `1 - level`, light colors are used, otherwise dark colors are employed. The following example using the '`Bundesliga`' data shows the relationship of home goals and away goals of Germany's premier soccer league in 1995: although there are two "larger" residuals (one greater than 2, one less then -2), the χ^2 test does not reject the null hypothesis of independence. Consequently, the colors appear dark (see Figure 25):

```
> bl <- xtabs(~HomeGoals + AwayGoals, data = Bundesliga,
+     subset = Year == 1995)
> mosaic(bl, gp = shading_hsv)
```

A shading function building upon `shading_hsv()` is `shading_Friendly()`, implementing the shading introduced by Friendly (1994). In addition to the defaults of the HSV shading, it

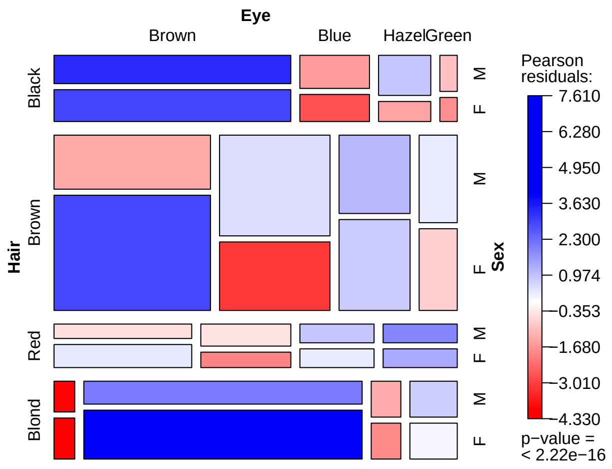


Figure 24: The ‘`HairEyeColor`’ data with continuous shading.

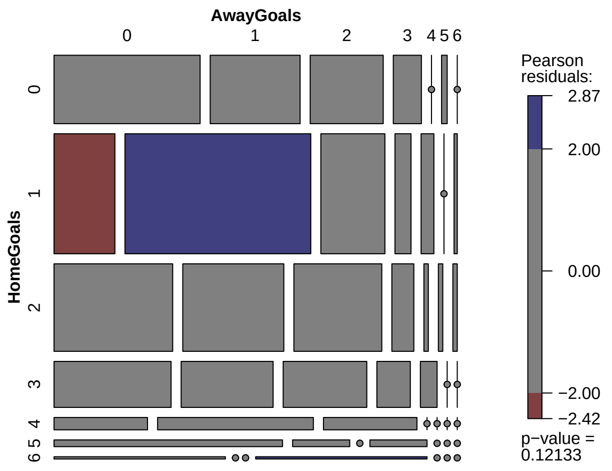


Figure 25: Non-significant χ^2 test using part of the ‘`Bundesliga`’ data.

uses the border color and line type to redundantly code the residuals' sign. The following example again uses the '`Bundesliga`' data from above, this time using the Friendly scheme and, in addition, an alternative legend (see Figure 26):

```
> mosaic(bl, gp = shading_Friendly, legend = legend_fixed,
+     zero_size = 0)
```

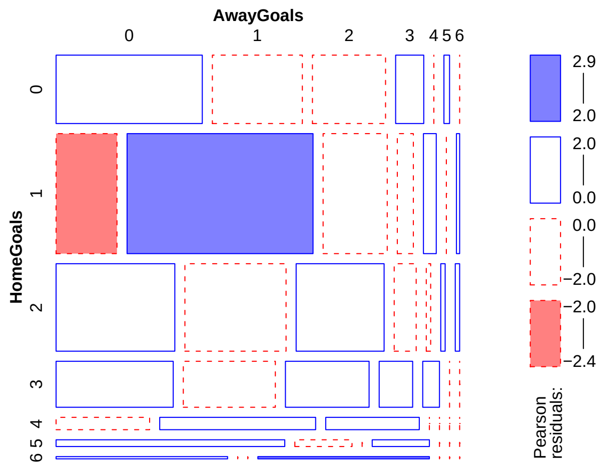
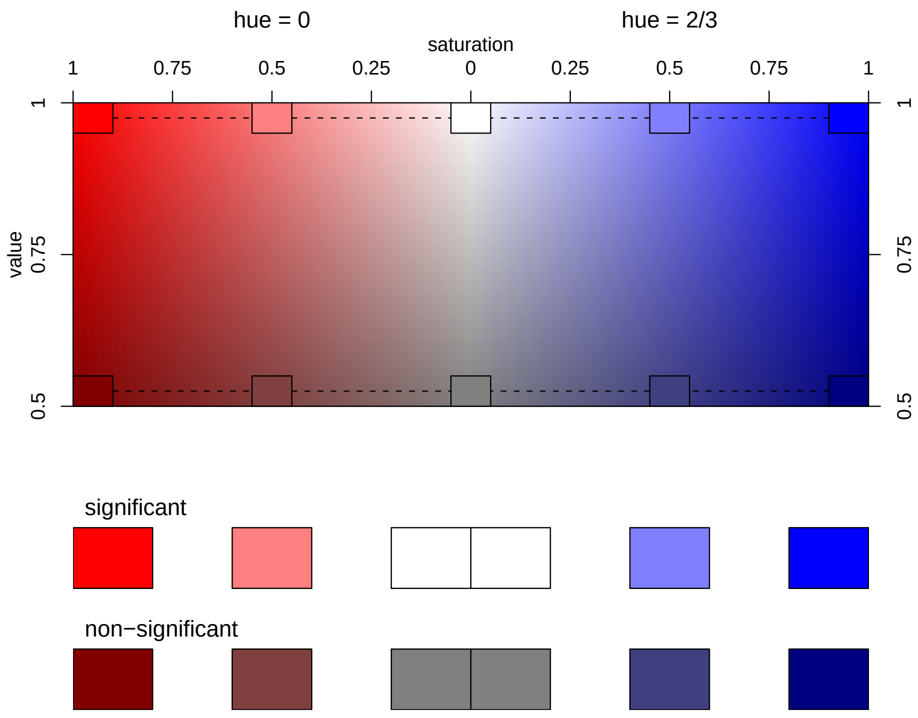


Figure 26: The '`Bundesliga`' data for 1995 using the Friendly shading and a legend with fixed bins.

(The `zero_size = 0` argument removes the bullets indicating zero observed values. This feature is not provided in the original SAS implementation of the Friendly mosaic plots.)

As introduced before, the default shading scheme is not `shading_hsv()` but `shading_hcl()` due to the better perceptual characteristics of the HCL color space. Figure 27 depicts the HSV space in the upper panel and the HCL space in the lower panel. On the left (right) side, we see the color scales for red (blue) hue, respectively. The x-axis represents the colorfulness, and the y-axis the brightness. The boxes represent the diverging color palettes used for the shadings. For the HSV space, we can see that the effect of changing the level of brightness ('value') is not the same for different levels of saturation, and again not the same for the two different hues. In fact, in the HSV space all dimensions are confounded, which obviously is problematic for coding information. In contrast, the HCL color space offers perceptually uniform colors: as can be seen from the lower panel, the chroma is homogeneous for different levels of luminance. Unfortunately, this comes at the price of the space being irregularly shaped, making it difficult to automatically select diverging color palettes. The following example again illustrates the '`HairEyeColor`' data, this time with HCL colors (Figure 28 depicts the default palette, and Figure 29 an alternative setting):

```
> mosaic(haireye, gp = shading_hcl)
```



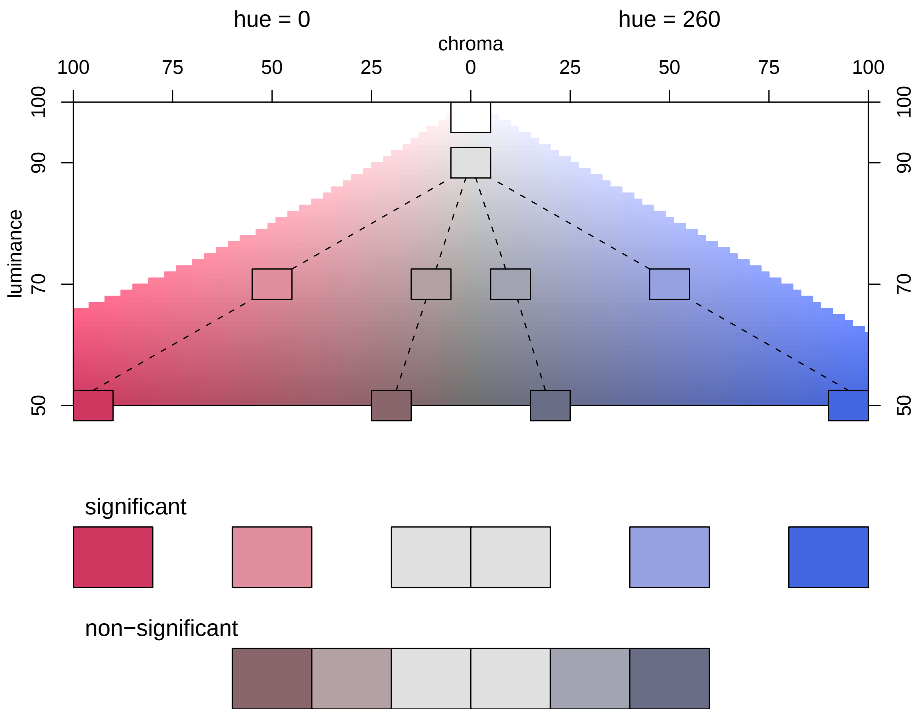


Figure 27: Residual-based shadings in HSV (upper) and HCL space (lower).

```
> mosaic(haireye, gp = shading_hcl, gp_args = list(h = c(130,
+     43), c = 100, l = c(90, 70)))
```

A more 'advanced' function building upon `shading_hcl()` is `shading_max()`, using the maximum statistic both to conduct the independence test and to visualize significant *cells* causing the rejection of the independence hypothesis (Meyer *et al.* 2003). The `level` argument of `shading_max()` then can be used to specify several confidence levels from which the corresponding cut-off points are computed. By default, two cut-off points are computed corresponding to confidence levels of 90% and 99%, respectively. In the following example, we investigate the effect of a new treatment for rheumatoid arthritis on a group of female patients using the maximum shading (see Figure 30):

```
> mosaic(~Treatment + Improved, data = Arthritis, subset = Sex ==
+     "Female", gp = shading_max)
```

The maximum test is significant although the residuals are all in the $[-2, 2]$ interval. The `shading_hcl()` function with default cut-off points would not have shown any color. In addition, since the test statistic is the maximum of the absolute Pearson residuals, *each* colored residual violates the null hypotheses of independence, and thus, the 'culprits' can immediately be identified.

4. Flexible Labeling

One of the major enhancements in package **vcd** compared to `mosaicplot()` and `assocplot()` in base R is the labeling in the strucplot framework which offers many more features and flexibility. Like shading, spacing, and drawing of legend and core plot, labeling is now carried out by grapcon functions, rendering labeling completely modular. The user supplies either a labeling function, or, alternatively, a generating function that parameterizes a labeling function, to `strucplot()` which then draws the labels. Labeling is well-separated from the actual plotting that occurs in the low-level core functions. It only relies on the viewport tree produced by them, and the '`dimnames`' attribute of the visualized table. Labeling functions are grapcons that "add ink to the canvas": the drawing of the labels happens after the actual plot has been drawn by the core function. Thus, it is possible to supply one's own labeling function, or to combine some of the basic functions to produce a more complex labeling. In the following, we describe the three basic modules (`labeling_text()`, `labeling_list()`, and `labeling_cells()`) and derived functions that build upon them.

4.1. Labels in the borders: `labeling_text()`

`labeling_text()` is the default for all strucplot displays. It plots labels in the borders similar to the `mosaicplot()` function in base R, but is much more flexible: it is not limited to 4 dimensions, and the positioning and graphical parameters of levels and variable names are customizable. In addition, the problem of overlapping labels can be handled in several ways.

As an example, consider the '`Titanic`' data, consisting of 4 categorical variables: survival ('survived'), gender (i.e., 'sex'), age, and crew. By default, the variable names and levels are plotted 'around' the plot in a counter-clockwise way (see Figure 31):

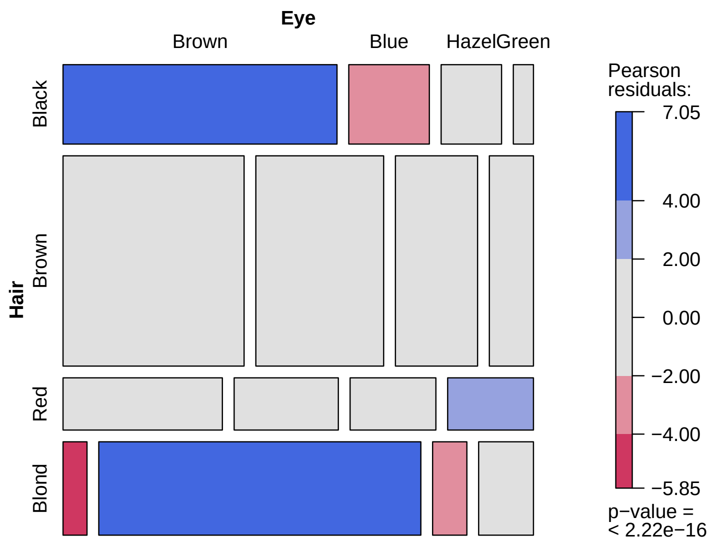


Figure 28: The ‘HairEyeColor’ data, using default HCL color palette.

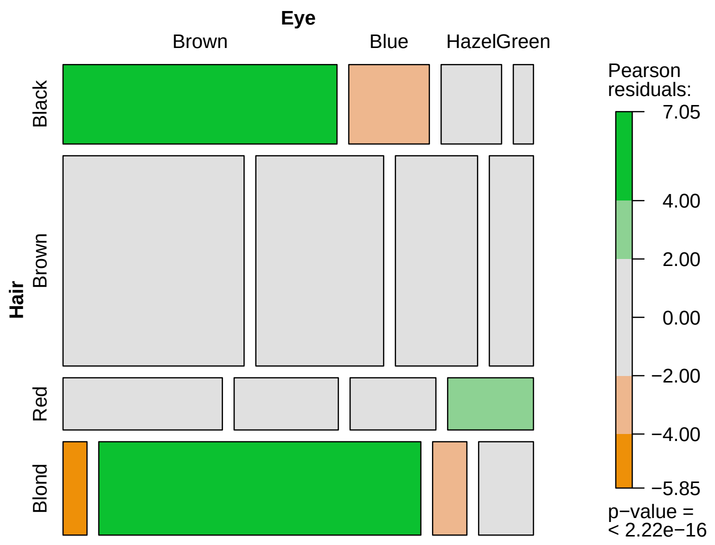


Figure 29: The ‘HairEyeColor’ data, using a custom HCL color palette.

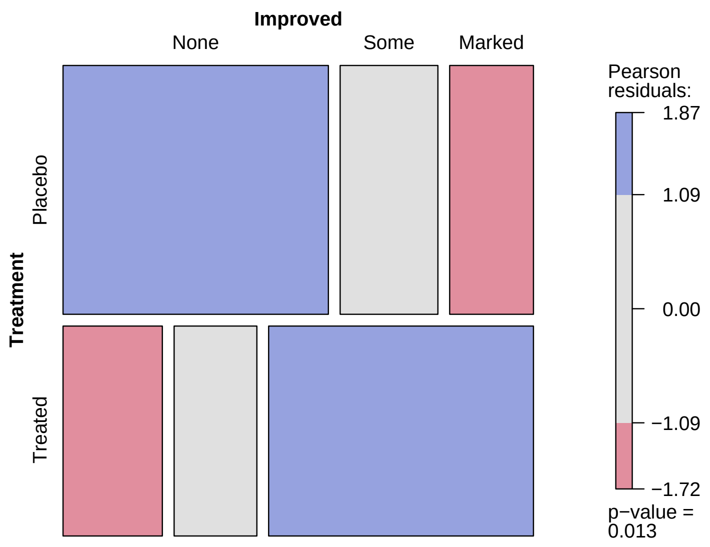


Figure 30: Significant maximum test on female patients of the ‘`Arthritis`’ data.

```
> mosaic(Titanic)
```

Note that the last two levels of the ‘survived’ variable do overlap, as well as some adult and child labels of the ‘age’ Variable. This issue can be addressed in several ways. The ‘brute force’ method is to enable clipping for these dimensions (see Figure 32):

```
> mosaic(Titanic, labeling_args = list(clip = c(Survived = TRUE,
+     Age = TRUE)))
```

The `clip` parameter is passed to the labeling function via the `labeling_args` argument which takes a list of parameters. `clip` itself takes a vector of logicals (one for each dimension). Almost all vectorized arguments in the strucplot framework can be abbreviated in the following way: unnamed components (or the defaults, if there are none) are recycled as needed, but overridden by the named components. Here, the default is `FALSE`, and therefore clipping is enabled only for the ‘survived’ and ‘age’ variables. A more sensible solution to the overlap problem is to abbreviate the levels (see Figure 33):

```
> mosaic(Titanic, labeling_args = list(abbreviate = c(Survived = TRUE,
+     Age = 3)))
```

The `abbreviate` argument takes a vector of integers indicating the number of significant characters the levels should be abbreviated to (`TRUE` is interpreted as 1, obviously). Abbreviation is performed using the `abbreviate()` function in base R. Another possibility is to rotate the levels (see Figure 34):

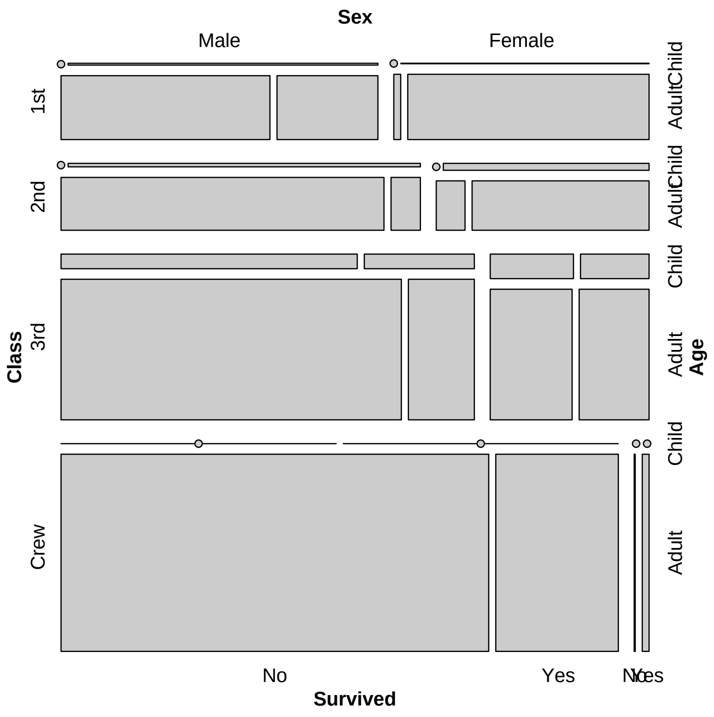


Figure 31: Mosaic plot for the ‘Titanic’ data with default settings for labeling.

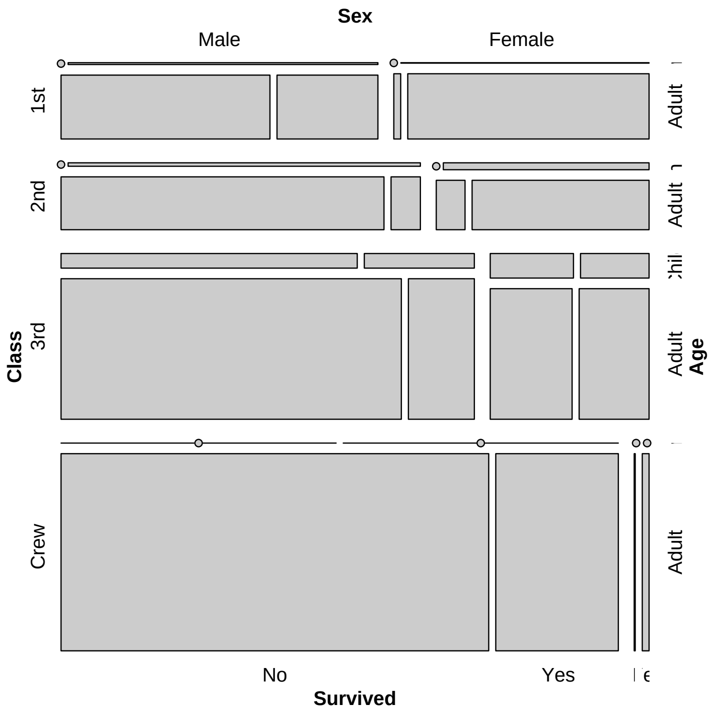


Figure 32: The effect of clipping.

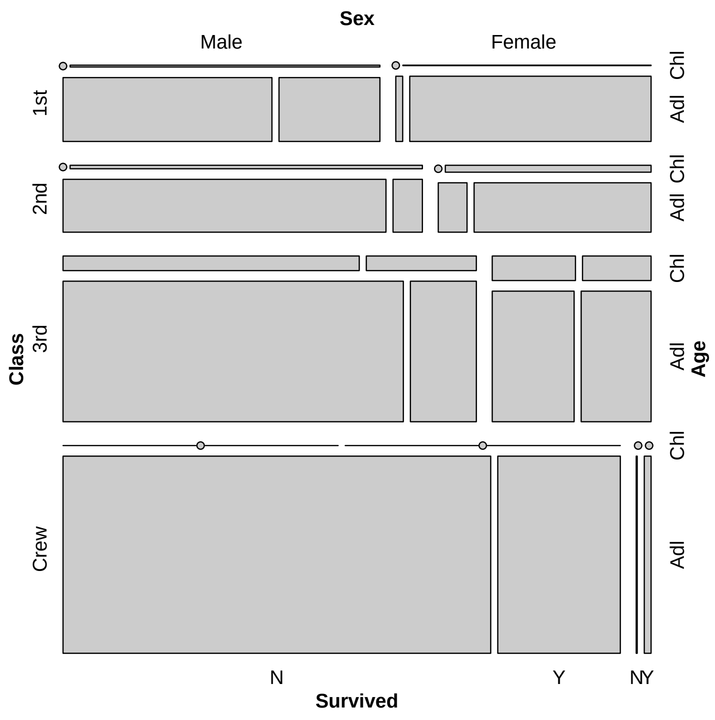


Figure 33: Abbreviating.

```
> mosaic(Titanic, labeling_args = list(rot_labels = c(bottom = 90,
+     right = 0), offset_varnames = c(right = 1), offset_labels = c(right = 0.3)),
+     margins = c(right = 4, bottom = 3))
```

Finally, we could also inhibit the output of repeated levels (see Figure 35):

```
> mosaic(Titanic, labeling_args = list(rep = c(Survived = FALSE,
+     Age = FALSE)))
```

We now proceed with a few more 'cosmetic' features (which do not all produce satisfactory results for our sample data). A first simple, but effectful modification is to position all labels and variables left-aligned: (see Figure 36):

```
> mosaic(Titanic, labeling_args = list(pos_varnames = "left",
+     pos_labels = "left", just_labels = "left", rep = FALSE))
```

Note that obviously we need to change the justification to `"left"` as well. We can achieve the same effect by using the convenience function `labeling_left()`:

```
> mosaic(Titanic, labeling = labeling_left)
```

Next, we show how to put all levels to the bottom and right margins, and all variable names to the top and left margins (see Figure 37):

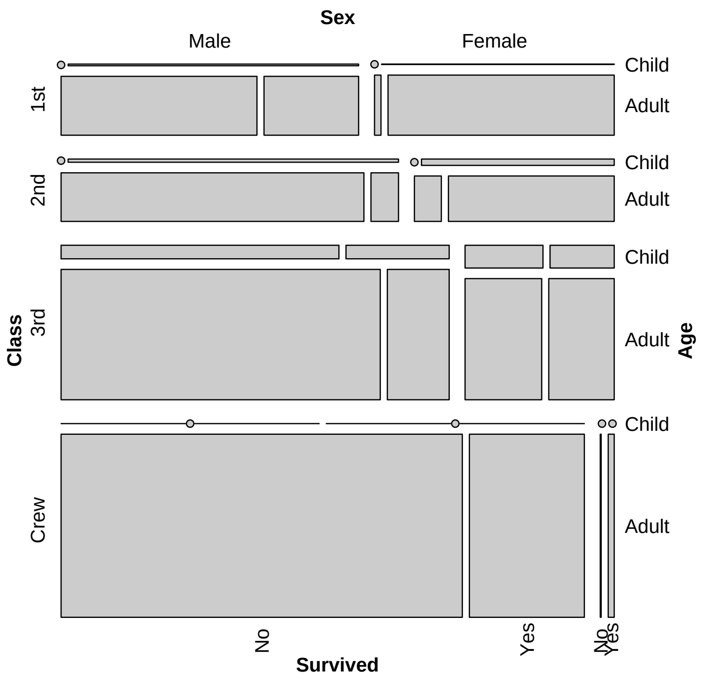


Figure 34: Rotating labels.

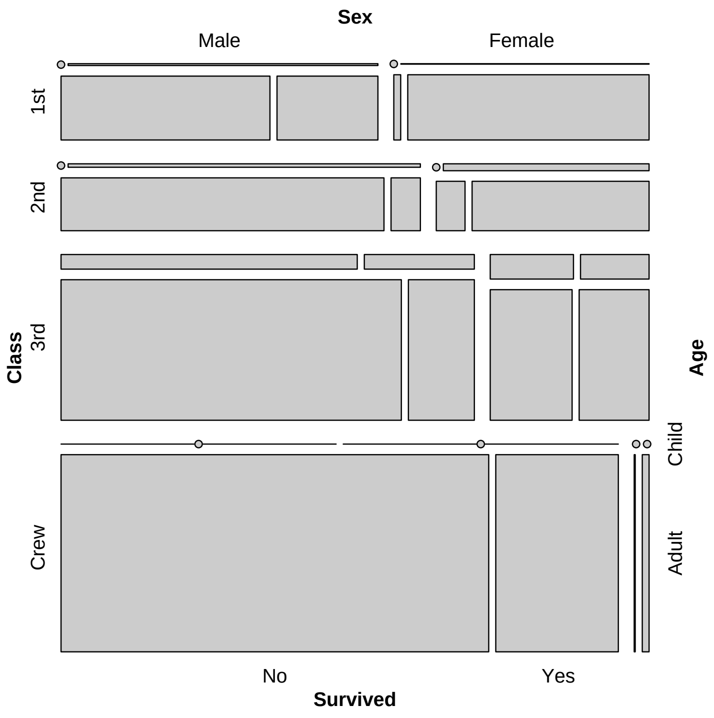


Figure 35: Inhibiting the repetition of levels.

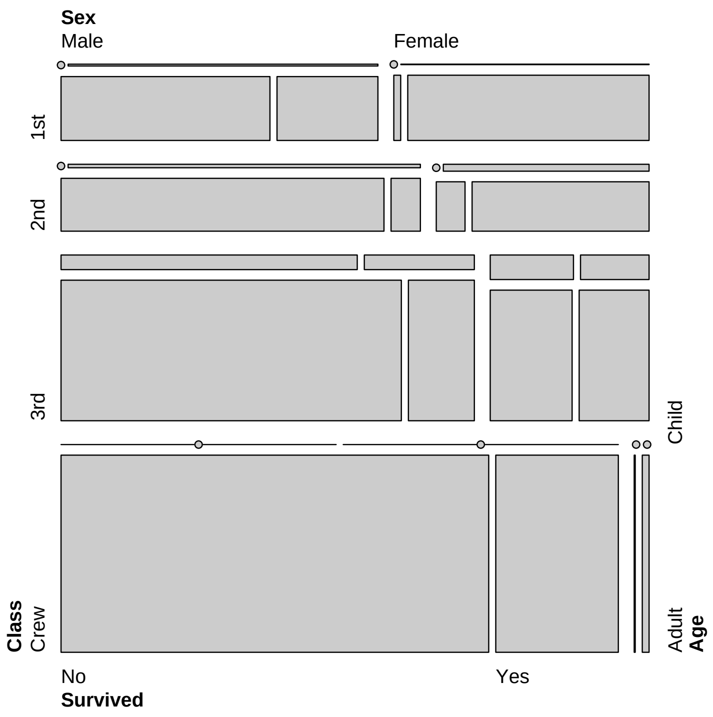


Figure 36: Left-aligning.

```
> mosaic(Titanic, labeling_args = list(tl_labels = FALSE,
+     tl_varnames = TRUE, abbreviate = c(Survived = 1,
+         Age = 3)))
```

The `tl_foo` ("top left") arguments are `TRUE` by default. Now, we will add boxes to the labels and additionally enable clipping (see Figure 38):

```
> mosaic(Titanic, labeling_args = list(tl_labels = FALSE,
+     tl_varnames = TRUE, boxes = TRUE, clip = TRUE))
```

The values to `boxes` and `clip` are recycled for all dimensions. The result is pretty close to what calling `mosaic()` with the `labeling_cboxed()` wrapper does, except that variables and levels, by default, are put to the top and to the left of the plot:

```
> mosaic(Titanic, labeling = labeling_cboxed)
```

Another variant is to put the variable names into the same line as the levels (see Figure 39):

```
> mosaic(Titanic, labeling_args = list(tl_labels = TRUE,
+     boxes = TRUE, clip = c(Survived = FALSE, TRUE), labbl_varnames = TRUE),
+     margins = c(left = 4, right = 1, 3))
```

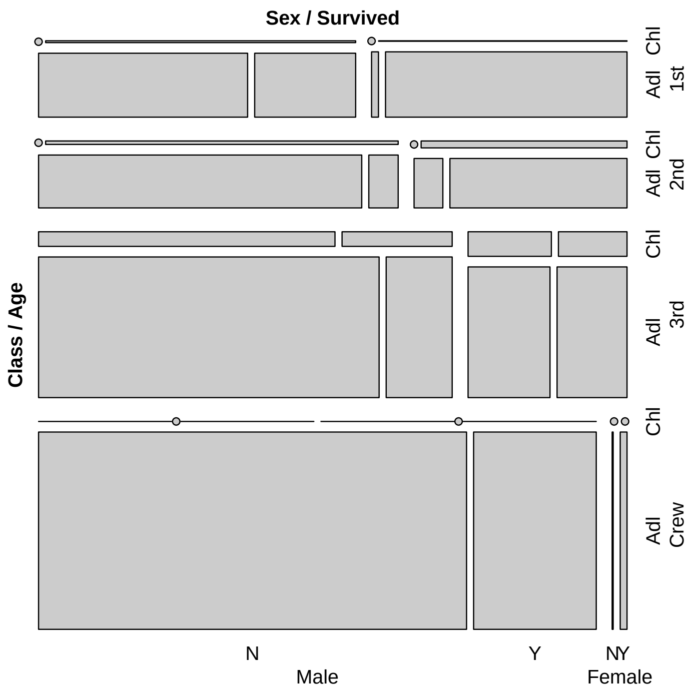


Figure 37: Changes in the margins.

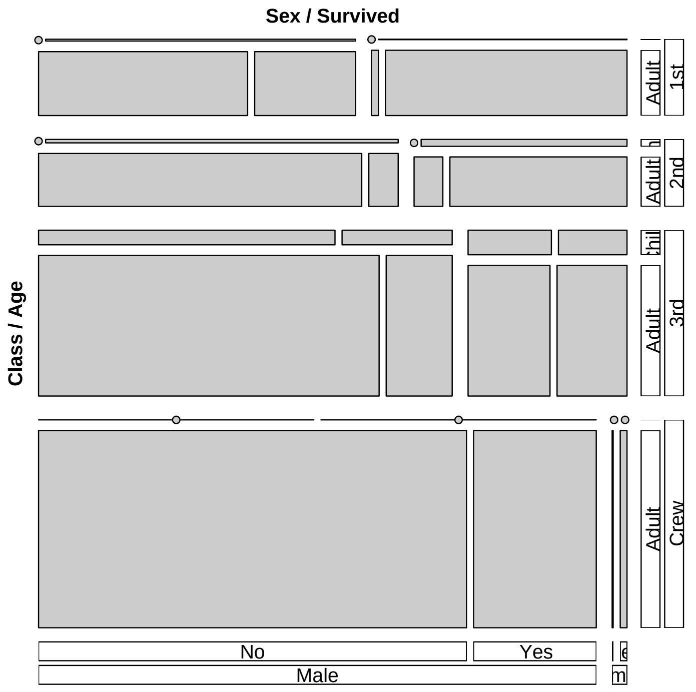


Figure 38: Boxes and Clipping.

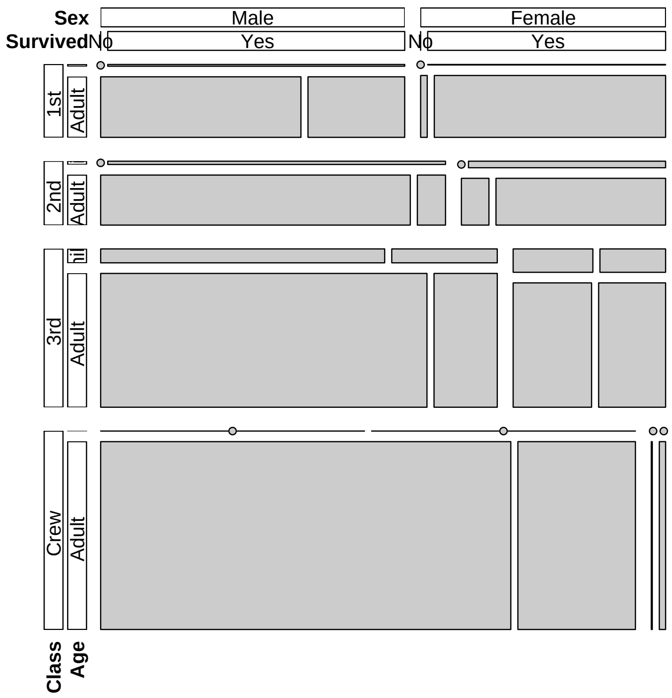


Figure 39: Variable names beneath levels.

`labbl_varnames` ("variable names to the bottom/left of the labels") is a vector of logicals indicating the side for the variable names. The resulting layout is close to what `labeling_lboxed()` produces, except that variables and levels, by default, are left-aligned and put to the bottom and to the right of the plot:

```
> mosaic(Titanic, labeling = labeling_lboxed, margins = c(right = 4,
+     left = 1, 3))
```

A similar design is used by the `doubledecker()` function.

4.2. Labels in the cells: `labeling_cells()`

This labeling draws both variable names and levels in the cells. As an example, we use the '`PreSex`' data on pre- and extramarital sex and divorce (see Figure 40):

```
> mosaic(~MaritalStatus + Gender, data = PreSex, labeling = labeling_cells)
```

In the case of narrow cells, it might be useful to abbreviate labels and/or variable names and turn off clipping (see Figure 41):

```
> mosaic(~PremaritalSex + ExtramaritalSex, data = PreSex,
+     labeling = labeling_cells(abbreviate_labels = TRUE,
+         abbreviate_varnames = TRUE, clip = FALSE))
```

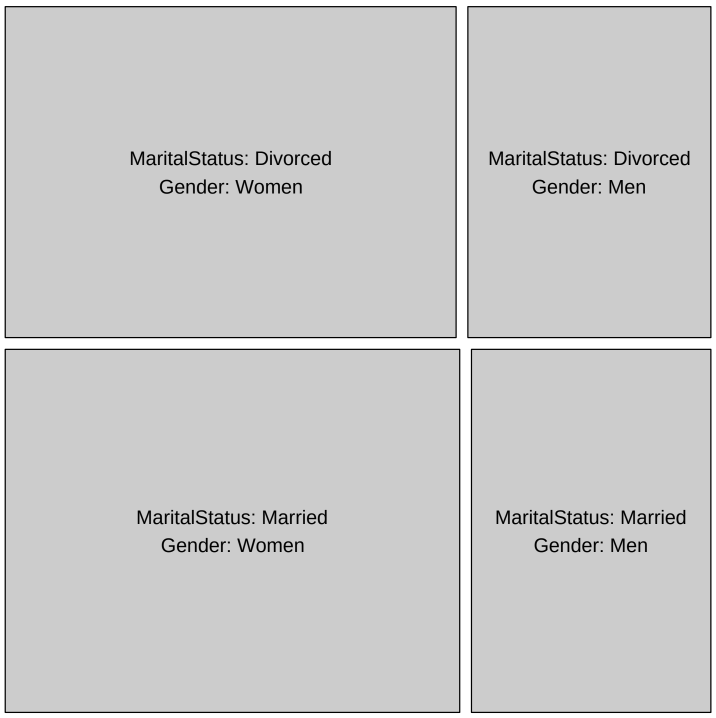


Figure 40: Cell labeling for the 'PreSex' data.

For some data, it might be convenient to combine cell labeling with border labeling as done by `labels_conditional()` (see Figure 42):

```
> mosaic(~PremaritalSex + ExtramaritalSex | MaritalStatus +
+     Gender, data = PreSex, labeling = labeling_conditional(abbreviate_varnames = TRUE,
+     abbreviate_labels = TRUE, clip = FALSE))
```

Additionally, the cell labeling allows the user to add arbitrary text to the cells by supplying a character array in the same shape than the data array to the `text` argument (cells with missing values are ignored). In the following example using the 'Titanic' data, this is used to add all observed values greater than 5 to the cells after the mosaic has been plotted (see Figure 43):

```
> mosaic(Titanic, labeling_args = list(abbreviate = c(Survived = 1,
+     Age = 4)), pop = FALSE)
> tab <- ifelse(Titanic < 6, NA, Titanic)
> labeling_cells(text = tab, clip = FALSE)(Titanic)
```

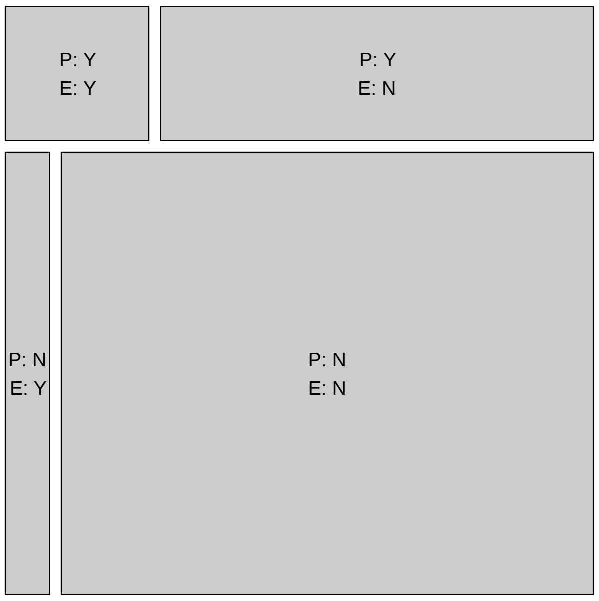


Figure 41: Cell labeling for the 'PreSex' data, labels abbreviated.

4.3. A simple list of labels: `labeling_list()`

If problems with overlapping labels cannot satisfactorily resolved, the last remedy could be to simply list the levels below the plot (see Figure 44):

```
> mosaic(Titanic, labeling = labeling_list, margins = c(bottom = 5))
```

The number of columns can be specified.

5. Customizable Spacing

Spacing of strucplot displays is customizable in a similar way than shading. The `spacing` argument of the `strucplot()` function takes a list of `unit` vectors, one for each dimension, specifying the space between the tiles corresponding to the levels. As an example, consider the introductory example of the '`arthritis`' data (Figure 1). Since we are interested in the effect of the medicament in the placebo and treatment groups, a mosaic plot is certainly appropriate to visualize the three levels of 'Improved' in the two 'Treatment' strata. However, following a slightly different approach, we might as well want to *highlight* patients with 'Marked' improvement in both groups, that is, produce a spine plot. To obtain such a display within the strucplot framework[1], it suffices to set the space between the 'Improved' tiles to 0 (see Figure 45):

[1]The **vcd** package also provides the `spine()` function for spine plots and spinograms.

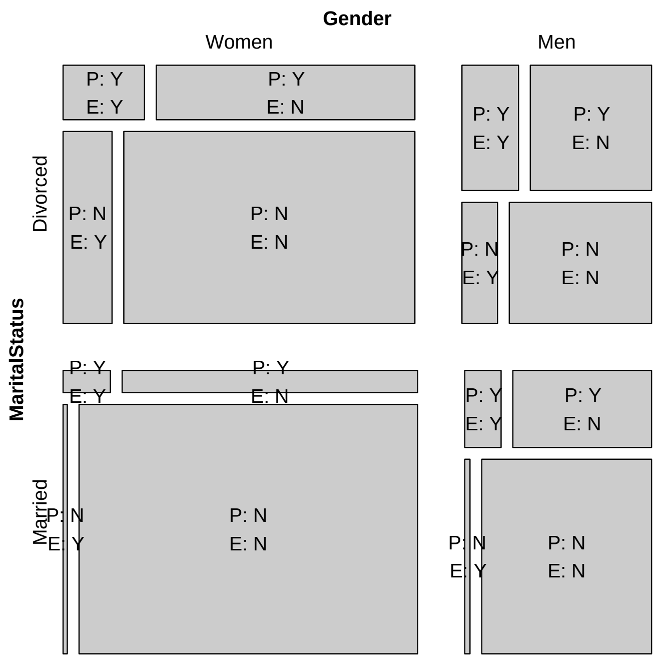


Figure 42: Conditional labeling for the 'PreSex', labels abbreviated.

```
> (art <- structable(~Treatment + Improved, data = Arthritis,
+     split_vertical = TRUE))

         Treatment Placebo Treated
Improved
None                    29      13
Some                     7       7
Marked                   7      21

> (my_spacing <- list(unit(0.5, "lines"), unit(c(0, 0),
+     "lines")))

[[1]]
[1] 0.5lines

[[2]]
[1] 0lines 0lines

> my_colors <- gpar(fill = c("lightgray", "darkgray", "black"))
> mosaic(art, spacing = my_spacing, gp = my_colors)
```

The strucplot framework also provides a set of spacing grapcons (and corresponding grapcon generators) that compute suitable spacing objects for typical applications. The simplest spacing is `spacing_equal()` that uses the same space between all tiles:

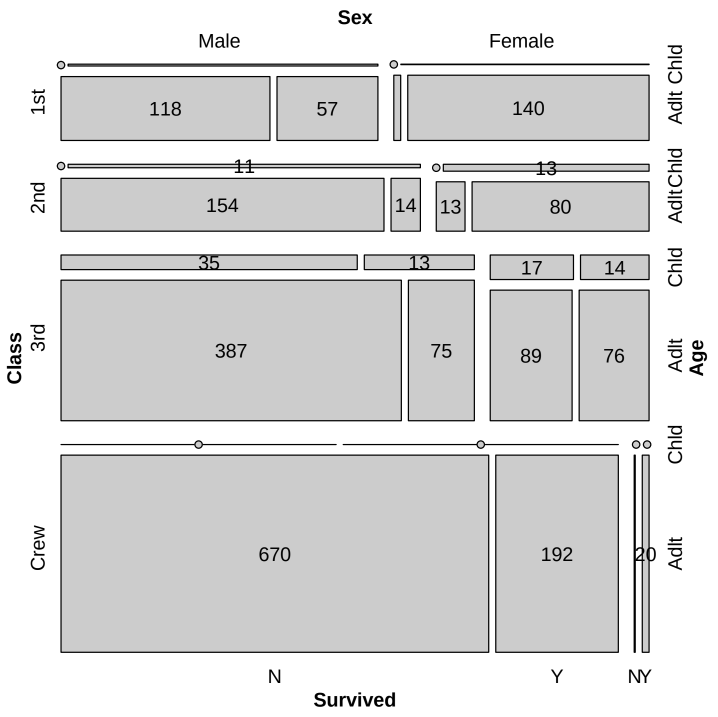


Figure 43: User-supplied Text added to a mosaic display of the ‘`Titanic`’ data.

```
> mosaic(art, spacing = spacing_equal(unit(2, "lines")))
```

`spacing_equal()` is the default grapcon generator for two-dimensional tables. Slightly more flexible is `spacing_dimequal()` that allows an individual setting for each dimension:

```
> mosaic(art, spacing = spacing_dimequal(unit(1:2, "lines")))
```

The default for multi-way contingency tables is `spacing_increase()` that uses increasing spaces for the dimensions. The user can specify a start value and the increase factor:

```
> mosaic(art, spacing = spacing_increase(start = unit(0.5,
+     "lines"), rate = 1.5))
```

For the arthritis example above, we could as well have used `spacing_highlighting()` that is similar to `spacing_increase()` but sets the spacing in the last splitting dimension to 0:

```
> mosaic(art, spacing = spacing_highlighting, gp = my_colors)
```

Finally, `spacing_conditional()` can be used for visualizing conditional independence: it combines `spacing_equal()` (for the conditioned dimensions) and `spacing_increase()` (for the conditioning dimensions). As an example, consider Figure 3: the spacing clearly allows to better distinguish the conditioning variables (‘Gender’ and ‘MaritalStatus’) from the conditioned variables (‘PremaritalSex’ and ‘ExtramaritalSex’). This spacing is the default when conditional variables are specified for a strucplot display (see Section 2).

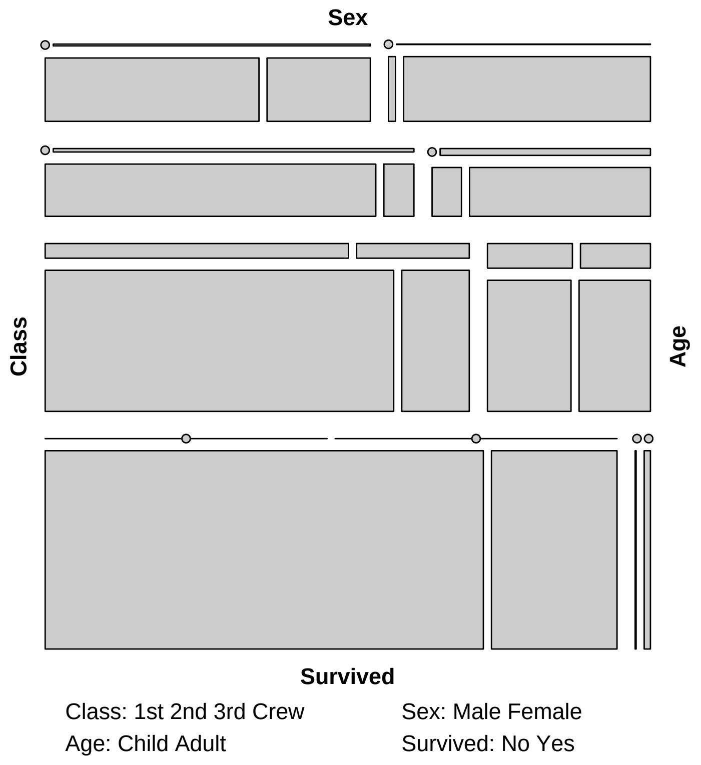


Figure 44: Labels indicated below the plot.

6. Conclusion

In this paper, we describe the 'strucplot' framework for the visualization of multi-way contingency tables. Strucplot displays include popular basic plots such as mosaic, association, and sieve plots, integrated in a unified approach: all can be seen as graphical visualizations of hierarchical conditional flat tables. Additionally, these core strucplot displays can be combined into more complex, specialized plots, such as pairs and trellis-like displays for visualizing conditional independence. Residual-based shadings permit the visualization of log-linear models and the results of independence tests. The frameworks' modular design allows flexible customization of the plots' graphical appearance, including shading, labeling, spacing, and legend, by means of graphical appearance control ('grapcon') functions. These 'graphical hyperparameters' are customized and created by generating functions. Our work includes a set of predefined grapcon generators for typical analysis tasks, and user-level extensions can easily be added. The framework is provided by the **vcd** package freely available from the Comprehensive R Archive Network (CRAN).

References

Cohen A (1980). "On the Graphical Display of the Significant Components in a Two-Way

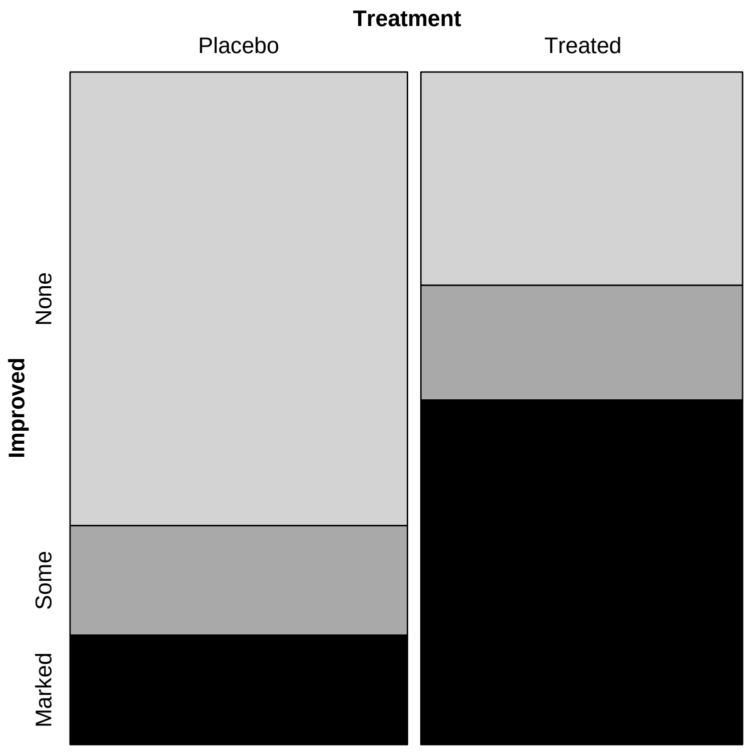


Figure 45: Spine plot for the ‘`arthritis`’ data using the strucplot framework.

Contingency Table." *Communications in Statistics—Theory and Methods*, **A9**, 1025–1041.

Friendly M (1994). "Mosaic Displays for Multi-Way Contingency Tables." *Journal of the American Statistical Association*, **89**, 190–200.

Friendly M (1999). "Extending Mosaic Displays: Marginal, Conditional, and Partial Views of Categorical Data." *Journal of Computational and Graphical Statistics*, **8**(3), 373–395.

Friendly M (2000). *Visualizing Categorical Data*. SAS Insitute, Carey, NC. URL http://www.math.yorku.ca/SCS/vcd/.

Hartigan JA, Kleiner B (1984). "A Mosaic of Television Ratings." *The American Statistician*, **38**, 32–35.

Hofmann H (2001). "Generalized Odds Ratios for Visual Modelling." *Journal of Computational and Graphical Statistics*, **10**, 1–13.

Meyer D, Zeileis A, Hornik K (2003). "Visualizing Independence Using Extended Association Plots." In K Hornik, F Leisch, A Zeileis (eds.), "Proceedings of the 3rd International Workshop on Distributed Statistical Computing, Vienna, Austria," ISSN 1609-395X, URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

Riedwyl H, Schüpbach M (1994). "Parquet diagram to plot contingency tables." In F Faulbaum (ed.), "Softstat '93: Advances in Statistical Software," pp. 293–299. Gustav Fischer, New York.

Theus M (2003). "Interactive Data Visualization using Mondrian." *Journal of Statistical Software*, **7**(11), 1–9. ISSN 1548-7660. URL http://www.jstatsoft.org/v07/i11;http://www.jstatsoft.org/v07/i11/MondrianJSSV2.pdf.

Unwin AR, Hawkins G, Hofmann H, Siegl B (1996). "Interactive Graphics for Data Sets with Missing Values - MANET." *Journal of Computational and Graphical Statistics*, **4**(6).

Zeileis A, Meyer D, Hornik K (2005). "Residual-based Shadings for Visualizing (Conditional) Independence." *Report 20*, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series. URL http://epub.wu-wien.ac.at/.

Affiliation:

David Meyer
Department of Information Systems and Process Management
E-mail: David.Meyer@wu-wien.ac.at
URL: http://wi.wu-wien.ac.at/~meyer/

Achim Zeileis
Department of Statistics & Mathematics
E-mail: Achim.Zeileis@wu-wien.ac.at
URL: http://www.ci.tuwien.ac.at/~zeileis/

Kurt Hornik
Department of Statistics & Mathematics
E-mail: Kurt.Hornik@wu-wien.ac.at
URL: http://www.wu-wien.ac.at/cstat/hornik/

Wirtschaftsuniversität Wien
1090 Wien, Austria

Journal of Statistical Software *Submitted:* yyyy-mm-dd
MMMMMM YYYY, Volume VV, Issue II. *Accepted:* yyyy-mm-dd
http://www.jstatsoft.org/