Colors and Residual-based Shadings in the Strucplot Framework

by David Meyer, Achim Zeileis, and Kurt Hornik

1 Introduction

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 2). 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). We provide several shading functions, including support for both HSV and HCL colors, and the visualization of significance tests (see Section 4).

2 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 1):

```
> (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).

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 2):

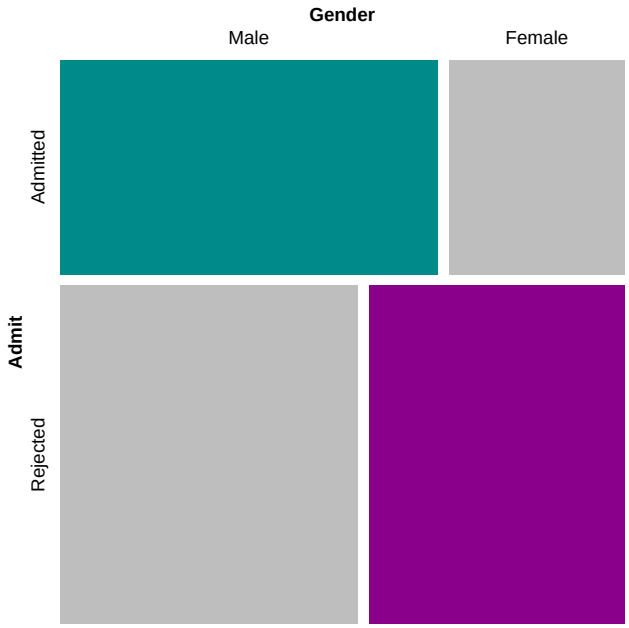

Figure 1: Mosaic plot for the '`UCBAdmissions`' data with highlighted cells.

```
> 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 the separate vignette: "Labeling in the Strucplot Framework" for more information).

3 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. Pre-computing 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.

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 taking parameters and returning a function as described in alternaive 2.

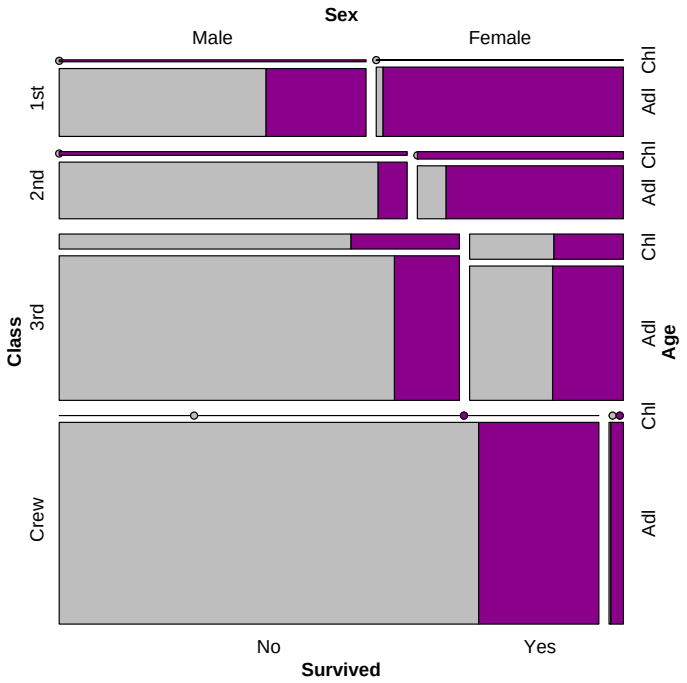

Figure 2: Recycling of parameters, used for highlighting the survived passengers in the 'Titanic' data.

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 3):

```
> 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))
```

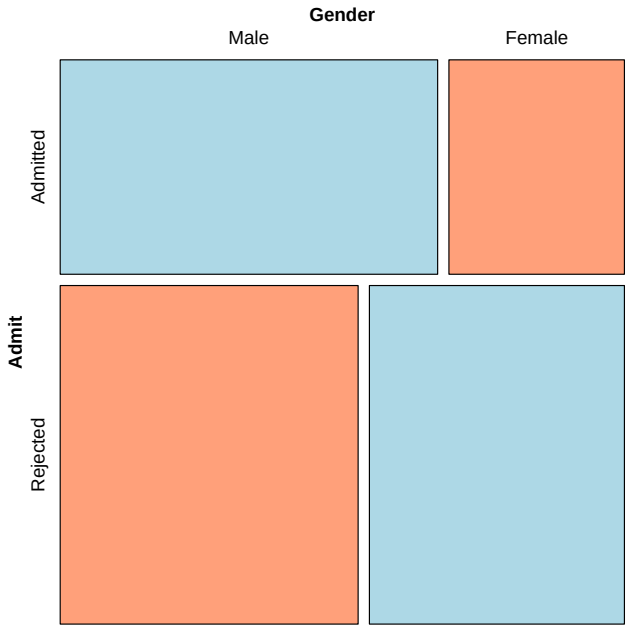

Figure 3: Binary shading visualizing the sign of the residuals.

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"))
> class(shading2_fun) <- c("grapcon", "shading")
```

Note the `class` attribute set on `shading2_fun()`. 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-generating function

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 follwing 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!")
+     ret <- function(res) gpar(fill = ifelse(res > 0, col[1], col[2]))
+     structure(ret, class = c("grapcon", "shading"))
+ }
```

In the call of `mosaic()` (figure not shown), using the new `shading3a_fun()` function, we can now simply change the colors:

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

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 shall 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 4), 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!")
+     ret <- function(res) gpar(fill = ifelse(res > 0, col[1], col[2]))
+     structure(ret, class = c("grapcon", "shading"))
+ }
> class(shading3b_fun) <- c("grapcon_generator", "shading")
```

In some sence, 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")))
```

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

4 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 *et al.*, 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 4)

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

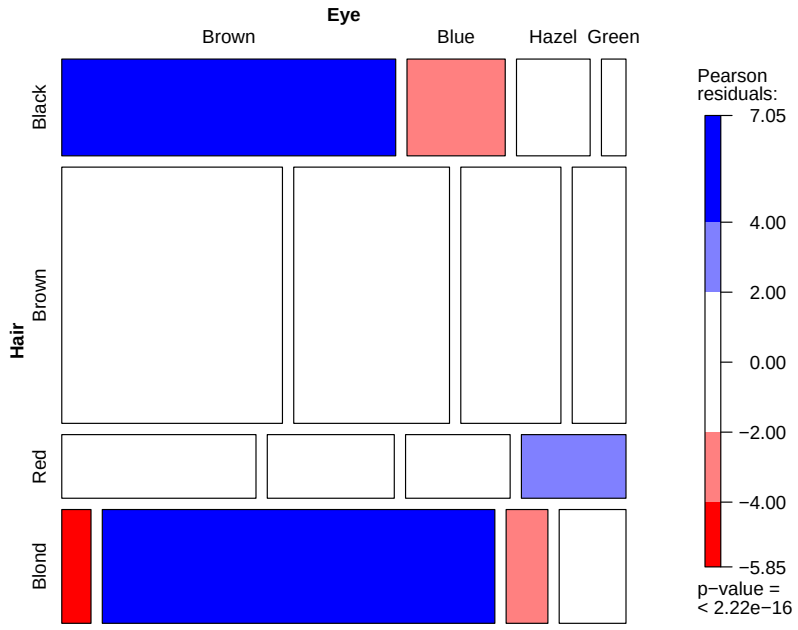

Figure 4: Shaded residuals in the '`HairEyeColor`' data set—two cut-off points.

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 5):

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

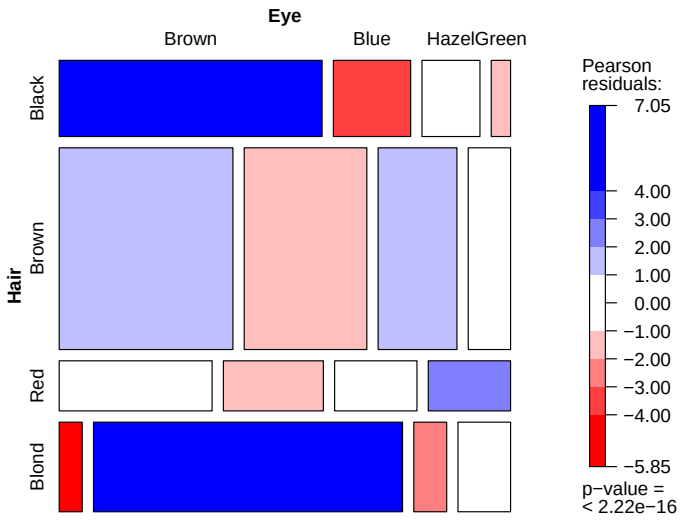

Figure 5: Shaded residuals in the '`HairEyeColor`' data set—four cut-off points.

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 6) is deceiving:

```
> 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 7):

```
> 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 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 8):

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

(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 9 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 10 depicts the default palette, and Figure 11 an alternative setting):

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

```
> 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 12):

```
> 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.

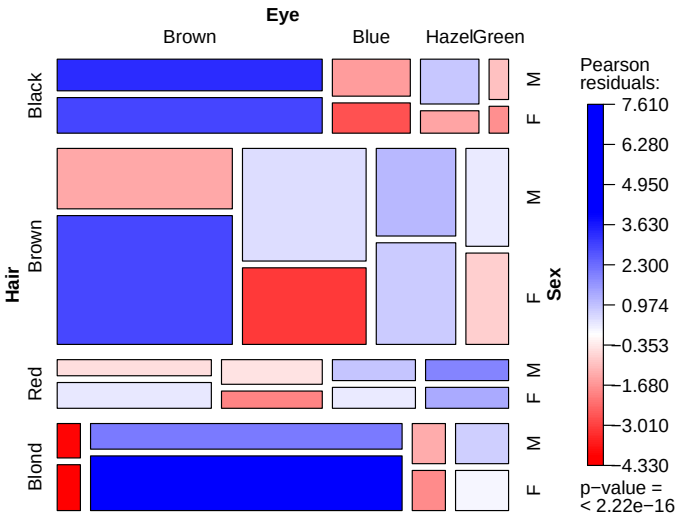

Figure 6: The '`HairEyeColor`' data with continuous shading.

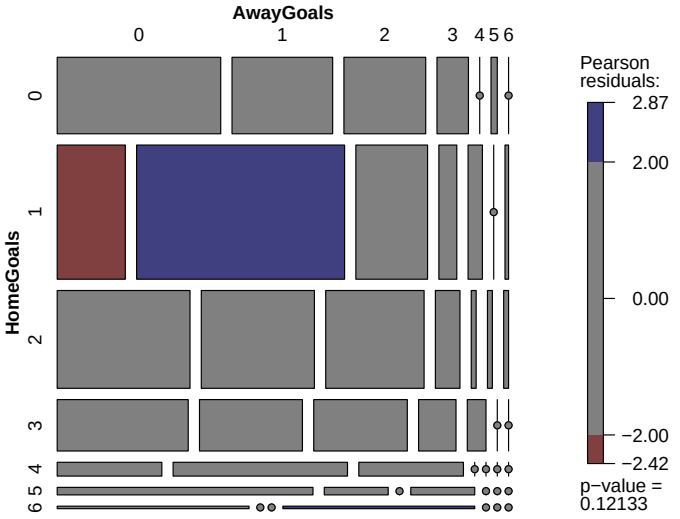

Figure 7: Non-significant χ^2 test using part of the '`Bundesliga`' data.

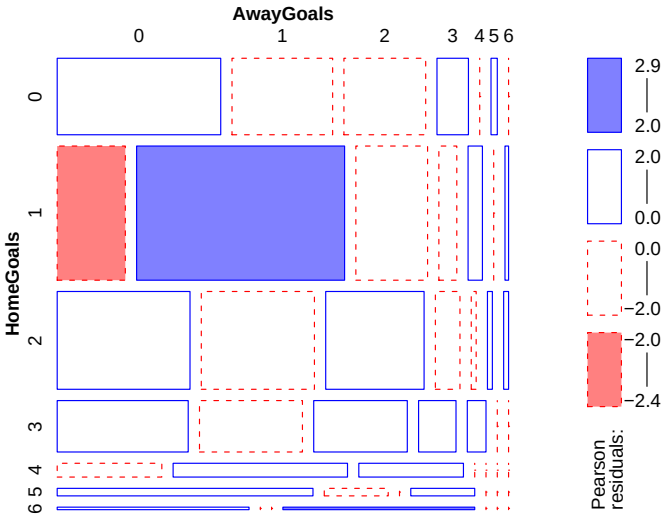

Figure 8: The '`Bundesliga`' data for 1995 using the Friendly shading and a legend with fixed bins.

References

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

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/.

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/.

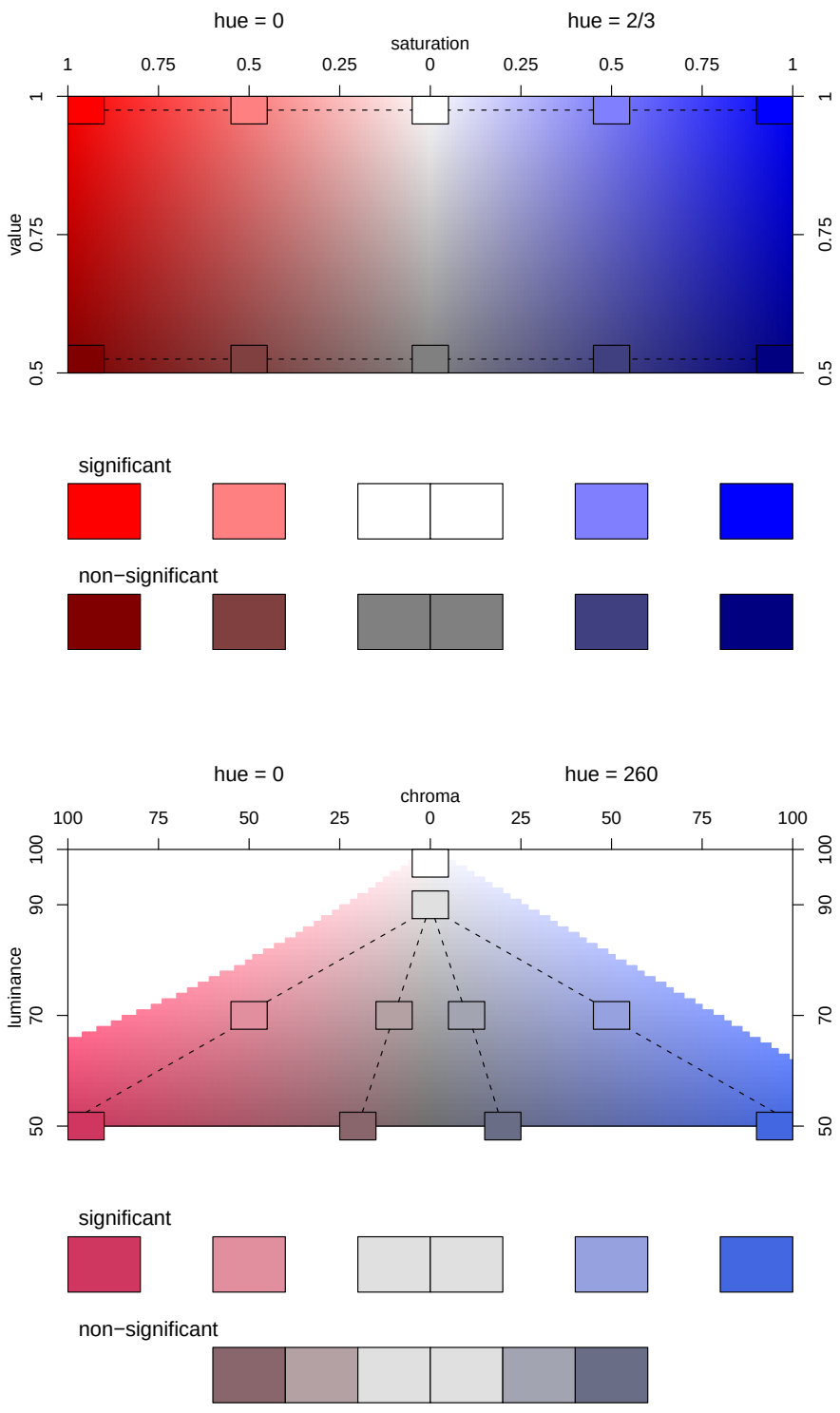

Figure 9: Residual-based shadings in HSV (upper) and HCL space (lower).

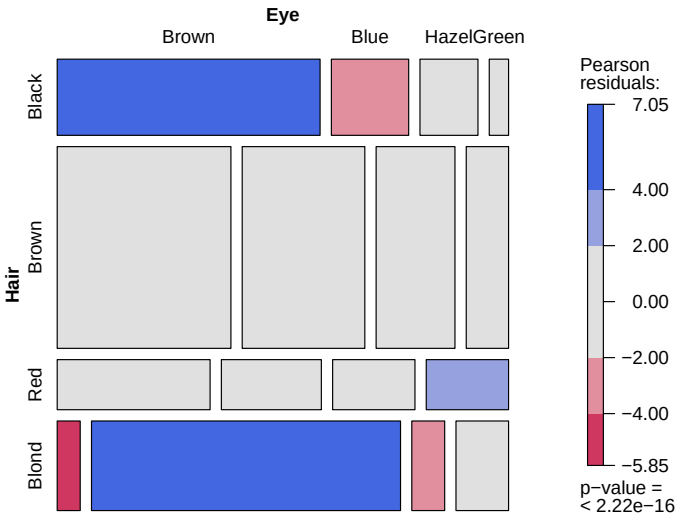

Figure 10: The '`HairEyeColor`' data, using default HCL color palette.

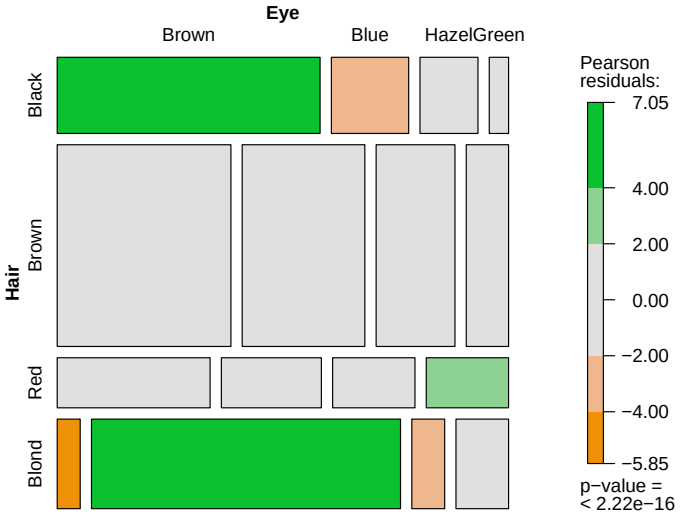

Figure 11: The '`HairEyeColor`' data, using a custom HCL color palette.

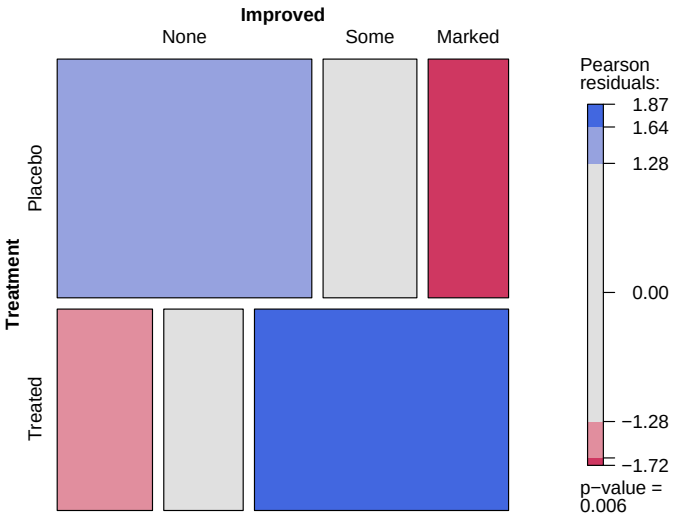

Figure 12: Significant maximum test on female patients of the '`Arthritis`' data.