Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Zigzag Expanded Navigation Plots in R: The R Package zenplots

Marius Hofert
University of Waterloo

Wayne Oldford
University of Waterloo

Abstract

We describe the features and implementation of the R package zenplots (zigzag expanded navigation plots) for displaying high-dimensional data according to the recently proposed zenplots.

By default, zenplots lay out alternating one- and two-dimensional plots in a zigzag-like pattern where adjacent axes share the same variate. Zenplots are especially useful when subsets of pairs can be identified as of particular interest by some measure, or as not meaningfully comparable, or when pairs of variates can be ordered in terms of potential interest to view, or the number of pairs is too large for more traditional layouts such as a scatterplot matrix. They also allow an essentially arbitrary layout of plots. High-dimensional space can be explored in a zenplot (`zenplot()`) by navigating through lower dimensional subspaces along a zenpath (`zenpath()`) which orders the dimensions (i.e., variates) visited according to some measure of interestingness; see Hofert and Oldford (2017) for an application to S&P 500 constituent data.

The R package **zenplots** provides compact displays for high-dimensional data via the notion of zenplots, grouping of variates, and customizable displays of zigzag layouts. It accommodates different graphical systems including the base **graphics** package of R Core Team (2017b), the package **grid** of R Core Team (2017a) (and hence packages like **ggplot2** of Wickham and Chang (2016)), and the interactive graphical package **loon** of Waddell and Oldford (2017). **zenplots** handles groups of variates, partial and fully missing data, and more. One important feature is that `zenplot()` and its auxiliary functions in **zenplots** distinguish layout from plotting which allows one to freely choose and create one- and two-dimensional plot functions; predefined functions are exported for all graphical systems.

All R plots in this paper are reproducible with the vignette `selected_features` (available in **zenplots** $\geq$ 0.0-2).

Keywords: R, data visualization, data analysis, graphics, grid, loon.

1. Introduction

Upper triangle scatterplot matrices for visualizing high-dimensional data through pairwise projections first appeared in Hartigan (1975). Tukey and Tukey (1981b) called them "generalized draftsman's views", the name "scatterplot matrix" (or, in short, "splom") is introduced in Tukey and Tukey (1983). The inherent limitations of scatterplot matrices for high-dimensional data were recognized soon thereafter:

> "For data in several dimensions we can make triangular arrays of two-coordinates-at-a-time scatter plots. However, as the dimensionality of the data increases, two problems arise, one minor, the other more serious. First, the plots must be made smaller and smaller if they are to fit on a single page; second, and more importantly, they become less and less representative of the totality of all possible views. With increasing dimensionality the need for good methods of selection becomes ever more pressing"
> Tukey and Tukey (1981b, p. 210)

In Tukey and Tukey (1981b, Table 10.2, p. 195), "several dimensions" meant six to ten; "many" meant 11–20, "lots of" meant 21–40, and "high-dimensional" was reserved for $d \geq 41$ dimensions. The scatterplot matrix works well for "data in several dimensions".

In our experience, beyond $d = 30$ to $d = 50$ dimensions, each cell in the scatterplot matrix becomes so small that, even after employing such tricks as sampling and alpha transparency, it can be challenging to extract useful information from this display; see, e.g., Hofert and Mächler (2014) or Hofert and Oldford (2017, Figure 3) (the latter only being able to show $\binom{22}{2} = 231$ out of $\binom{465}{2} = 107,880$ different pairs under consideration, so about 0.2%). In the extreme case, each cell might even be reduced to the size of a single pixel essentially rendering the scatterplot matrix as a pixel array. This (and other) limitations of scatterplot matrices have recently motivated the use of zenplots, see Hofert and Oldford (2017), for investigating high-dimensional data.

The R package **zenplots** presented here provides an implementation of zenplots (and accompanying tools) to help addressing both problems raised by Tukey and Tukey (1981b); see the quotation above.

First, by providing a more compact layout, a zenplot will either accommodate many more (different) "two-coordinates-at-a-time", or "2d", plots in the same space, or, equivalently, give each individual 2d plot more relative space within any fixed size layout. Unlike scatterplot matrices, zenplots can also be easily broken over pages.

A zenplot accomplishes this by relaxing the common coordinate feature across columns (and rows) forced by the square (or triangular) layout of a scatterplot matrix (or generalized draftsman's display). With a zenplot, comparisons across a common coordinate are still available between any two 2d plots in the layout provided they have a single coordinate, or "1d", plot appearing between them. To effect this, a zenplot follows a rectilinear "zigzag" path over the display region alternating between 1d and 2d plots, all the while ensuring that neighbours along this (zen)path share a one-coordinate-at-a-time boundary.

When the objective is to visually search for plots that contain interesting patterns, the zenplot thus accommodates many more (or larger) plots and, by following the zenpath through the display, still preserves plot to plot comparisons. Moreover, the zenpath can be constrained to include only those pairs, or plot to plot comparisons, that are of interest.

The second problem raised by Tukey and Tukey (1981b) is that there are many more possible views of a high-dimensional data cloud than simply those chosen from all possible pairs of coordinates. For example, any two randomly chosen direction vectors on the $(d-1)$-dimensional unit sphere in $\mathbb{R}^d$ define a projection plane which might reveal interesting structure in the data and that structure might be hidden from every one of the $\binom{d}{2}$ 2d plots defined by any two original coordinates. Moreover, there are exponentially many more of these planes than those defined only by pairs of coordinates (see Tukey and Tukey (1981a)).

Zenplots address this problem in two important ways.

First, it is surprising how many "two-coordinates-at-a-time" plots can now actually be examined with modern computer displays. As a proof of concept, in a visual analysis of pairwise dependence for $d = 465$ dimensional data as defined by constituents of the S&P 500 stock prices, Hofert and Oldford (2017) examined all $\binom{465}{2}$ =107,880 distinct 2d scatterplots via 164 zenplots (one per page) as a single PDF document in only 30 minutes.

Second, and more importantly, zenpaths may be constructed so that only the most interesting, or meaningful, plots are produced. Following Hurley and Oldford (2010, 2011b), imagine a graph $G = (V, E)$ whose vertex set V is the set of all coordinates (variates) in the data and whose edge set E is the set of all meaningfully paired coordinates (variates) – typically this is a complete graph, but need not be. Weights could be attached to each edge which measure the "interestingness" of the respective two coordinates. For example, such measures were proposed as "cognostics" (for computer guided diagnostics) or "scagnostics" (scatterplot diagnostics) by Tukey and Tukey (1985) and formalized more recently by Wilkinson, Anand, and Grossman (2005). No matter how each node determines a coordinate (e.g., an original variate, some randomly or purposely chosen linear combination of original variates, or any other real-valued function of the variates), and however the weights on the edges between pairs of coordinates might be determined (e.g., statistical or scagnostic measures on that pair), a zenpath is any path on G, often it is one selected according to the weights along its edges.

In this way, following a selected zenpath provides a means to construct an interesting sequence of coordinates to be displayed within the corresponding zenplot. For example, a path of maximum (minimum) total weight would correspond to a solution to the "travelling salesman problem"; following the resulting path would display all coordinates exactly once each yet show the most interesting 2d displays. The idea of a zenpath is to find those paths whose zenplot display reveals interesting structure via its coordinate sequence.

Finally, as in the original Hartigan (1975), there is no reason to restrict the 2d plots to be only scatterplots, or for the 1d plots to be, say, histograms. The `pairs(...)` function in R, e.g., has arguments (e.g., `panel`, `lower.panel`, `diag.panel`, ...) which allow the user to create essentially any display of the variates in that panel (or cell) of the scatterplot matrix. More recent authors such as Friendly (1999), Emerson, Green, Schloerke, Crowley, Cook, Hofmann, and Wickham (2013), or Im, McGuffin, and Leung (2013) have undertaken generalizations of the scatterplot matrix to accommodate different pairwise and marginal displays depending on the variates in that panel of the scatterplot matrix.

Zenplots also accomodate any 1d or 2d plot. And the plot can be written in either of the R graphics packages **graphics**, see R Core Team (2017b), **grid**, see R Core Team (2017a) or Murrell (2016) and hence **ggplot2**, see Wickham and Chang (2016) or Wickham (2016), as well as in the more recent interactive visualization package **loon**, see Waddell and Oldford (2017).

The R package **zenplots** provides an implementation of zenplots and zenpaths with the functions

`zenplot()` and `zenpath()`, respectively, as well as several other functions for exploratory data analysis and visualization. In this paper we focus on describing the functionality of the **zenplots** package, intentionally confining our illustrations to data only of "several dimensions", in the sense of Tukey and Tukey (1981b). A fuller appreciation of the package's functionality and value in visual data analysis for "high-dimensional" data is more readily had from Hofert and Oldford (2017) where the package is applied to a data example in $d = 465$ dimensions.

The paper is organized as follows. In Section 2 we present the structure, technical aspects and selected features of `zenplot()` as well as the related basic notion of a zenpath. Section 3 then focuses on zenpaths and describes how visual search can be conducted with the function `zenpath()`. Section 4 addresses the construction of customized zenplots and Section 5 presents more advanced features.

Some remarks concerning the history of high-dimensional data visualization are in order at this point. For several decades now, people have visualized high dimensional space by connecting displays of low-dimensional projections. The connections are often made temporally so that one low-dimensional view dynamically and smoothly morphs into the next. The earliest versions of these would have been 3d point cloud rotations which date back to at least Ball and Hall (1970) The more general "tour" methods date to Asimov (1985) whereby arbitrary planes connected along geodesic paths are displayed over time. The earliest implementation of this was Buja, Hurley, and McDonald (1986). A series of re-implementations of these methods followed (namely XGobi, GGobi, and **rggobi** of Temple Lang, Swayne, Wickham, and Lawrence (2018)) the most recent incarnations of which appear as **tourr** and **tourrGui**; see Huang, Cook, and Wickham (2012). Hurley and Oldford (2011b) constrained the planes to the orthogonal axes of the coordinate system defined in advance and proposed exploring high-dimensional spaces by sequences of planes temporally following navigation graphs. The first publicly available implementation of this strategy was **RnavGraph** of Waddell and Oldford (2011); it has most recently been implemented in the interactive and extendible data visualization package **loon** (see Waddell and Oldford (2017)), which **zenplots** can also utilize if required. By default, zenplots accomplish the layout spatially (rather than temporally) and are therefore a novel contribution to high-dimensional data visualization.

2. Zenplots

We start by considering the olive data set of Forina, Armanino, Lanteri, and Tiscornia (1983); see also Azzalini and Torelli (2007). For convenience, this data set is available in the R package **zenplots**. It consists of $n = 572$ measurements of $d = 10$ variates (the geographical area and the region of origin of the olive oil, and measurements of eight fatty acid components in each of the oil specimens, namely palmitic, palmitoleic, stearic, oleic, linoleic, linolenic, arachidic and eicosenoic). As with scatterplot matrices, by default a zenplot will produce a two-dimensional point cloud (scatterplot without axes) for each variate pair and the variate's name for each one-dimensional plot. Unlike a scatterplot matrix, not all pairs of variates are displayed by default. Instead, variates are paired only if they appear next to one another in the column order of the data; following column order lets variate pair sequences be identified via column indices in the `data.frame`. To keep the number of variate pairs low and the layout discussion simple, we consider only the nine pairs of variates as they appear in column order in the `olive` data; in Section 4.5, we discuss how sequences of "interesting" variate pairs might be determined and laid out for these data.

For (column ordered) variate pairs of the olive data, the zenplot is constructed as follows:

```
R> library("zenplots")
R> data("olive")
R> zenplot(olive)
```

and appears as the left-hand side of Figure 1.

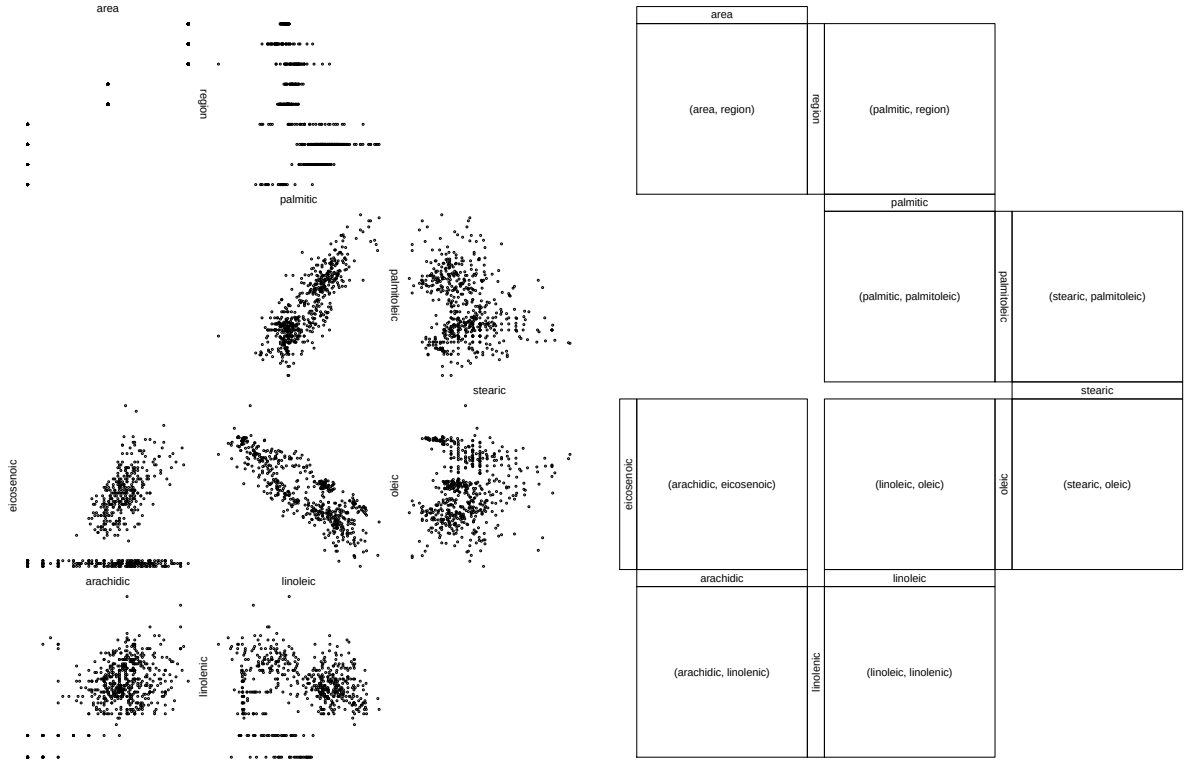

Figure 1: Zenplot of the olive data set (default, left) and again using `"layout"` (right).

The result is an alternating sequence of one-dimensional (1d) and two-dimensional (2d) plots laid out in a zigzag-like structure so that each consecutive pair of 2d plots has one of its variates (or coordinates) in common with that of the 1d plot appearing between them.

The right-hand side of Figure 1 is also a zenplot of these data but one constructed by choosing the value `"layout"` for arguments `plot1d` and `plot2d`:

```
R> zenplot(olive, plot1d = "layout", plot2d = "layout")
```

Now only a box containing the labels of the variates in the 1d and 2d plots is shown. This should help identify the variates that are available to construct each plot as well as the zigzag pattern of alternating plots in the display. Note also that in either display of Figure 1, the labels of vertical 1d plots (e.g., "region", "palmitoleic" or "oleic") are oriented to reflect the left or right direction of the layout but that for horizontal 1d plots (e.g., "area", "palmitic" or "stearic") the natural reading direction is preserved for both up or down directions.

The entire set of arguments of `zenplot()` are seen by printing its structure:

```
R> str(zenplot)

function (x, turns = NULL, first1d = TRUE, last1d = TRUE,
          n2dcols = c("letter", "square", "A4", "golden", "legal"),
          n2dplots = NULL,
          plot1d = c("label", "points", "jitter", "density", "boxplot",
                     "hist", "rug", "arrow", "rect", "lines", "layout"),
          plot2d = c("points", "density", "axes", "label", "arrow",
                     "rect", "layout"),
          zargs = c(x = TRUE, turns = TRUE, orientations = TRUE,
                    vars = TRUE, num = TRUE, lim = TRUE, labs = TRUE,
                    width1d = TRUE, width2d = TRUE,
                    ispace = match.arg(pkg) != "graphics"),
          lim = c("individual", "groupwise", "global"),
          labs = list(group = "G", var = "V", sep = ", ", group2d = FALSE),
          pkg = c("graphics", "grid", "loon"),
          method = c("tidy", "double.zigzag", "single.zigzag"),
          width1d = if (is.null(plot1d)) 0.5 else 1,
          width2d = 10,
          ospace = if (pkg == "loon") 0 else 0.02,
          ispace = if (pkg == "graphics") 0 else 0.037, draw = TRUE, ...)
```

This seemingly imposing number and variety of arguments group naturally as related to:

- data: `x`,
- plots: `plot1d`, `plot2d`, `zargs`, `pkg`, `lim`, `labs`, `draw`, `...`,
- layout: `turns`, `method`, `n2dplots`, `n2dcols`, `first1d`, `last1d`, `width1d`, `width2d`,
- spacing: `ospace`, `ispace`.

Each of these arguments will be discussed in some detail and illustrated in sections to come, but for now it is their grouping which merits attention because it helps answer the question: What is a zenplot?

A zenplot is first and foremost a *navigation plot*. By this we mean it is a sequence of low-dimensional plots which follow some trajectory through a higher dimensional data space for the purpose of revealing structure. Different trajectories and different 1d and 2d displays may reveal different features of the data. Choosing which trajectories and which displays amounts to navigating through the high-dimensional data space.

The trajectory is determined entirely by the data `x`. This single argument, `x`, provides the data and the order in which the data dimensions will be traversed in the low-dimensional trajectory. For example, in Figure 1 plots are laid out in the same numerical order in which the variates appear in the data-frame. That is, for a d-dimensional data-frame, with variates numbered $1, \ldots, d$, the variate pairs appear as $(1,2), (2,3), \ldots, (d-1, d)$; 2d plots would be constructed for each variate pair and 1d plots interspersed between them for the variate they have in common. In this way, the determination of the trajectory is separated from its visualization. The code respects this separation in that the function `zenplot()` assumes the order given by the data `x`. Another function, `zenpath()`, is available to provide a variety of different orders which might be used to arrange that the data `x` produce a given trajectory.

The second grouping of arguments, the plots, determine the 1d and 2d plots that will be used to display the lower dimensional spaces along the trajectory. A number of plot types are built

in as specific values of these arguments and, by using the information bundled as `zargs`, the user may also write their own display functions using any one of the R graphics packages given by `pkg`. Together with the trajectory, this specification of plots define the nature of the navigation through the high-dimensional space.

One could, as in Hurley and Oldford (2011b), Oldford and Waddell (2011) and Waddell and Oldford (2011, 2014, 2017) use dynamic graphics to move from one display to another thus linking the displays in time. Motion graphics very powerfully connect one low-dimensional display to the next, visually reinforcing the notion of navigation through high-dimensional space along low-dimensional trajectories. However, such temporal linking becomes increasingly burdensome on our short-term memory as the time over which the displays are transiently presented grows. And the presentation time can indeed grow quickly given that the number of variate pairs can grow quadratically with dimensionality d.

In contrast, `zenplot()` spatially links the displays along the trajectory by laying them out following a zigzag path pattern (hence the name "zigzag expanded navigation plot" or zenplot). Thus, it is in the third grouping of arguments, the layout arguments, which effectively define a zenplot and which distinguish it from other displays of high-dimensional data. The last grouping, spacing, is arguably also part of the layout but is not peculiar to the definition of a zenplot layout and so can be regarded as separate, more generic, layout arguments.

The next few sections deal with each of these groupings of arguments in some detail beginning with the layout arguments, being those which most distinguish a zenplot. Standard features are treated separately from those which are more customized so that the reader may skip the more advanced at first reading. Because of their simplicity, the spacing arguments will be mentioned and illustrated in the following section on the layout.

2.1. Layout

Nearly all arguments of `zenplot()` are concerned with the physical layout of the plots. In this section, only those basic layouts (zigzag patterns) which conceptually distinguish a zenplot from other layouts are treated.

Zigzagging layout methods

The argument `method` provides a choice between the zigzag patterns `"tidy"`, `"double.zigzag"`, `"single.zigzag"` or `"rectangular"`; the default value is `"tidy"`. To clearly contrast these options for `method`, more plots than those which appeared in Figure 1 are required. To this end, we simply double the number of columns in the olive data set so that there are now $d = 20$ variates to plot:

```
R> olive2 <- cbind(olive, olive)
```

Using `olive2` as the data `x`, the layout for the first three of the `method` values appears in Figure 2 produced as follows:

```
R> zenplot(olive2, n2dcols = 6, plot1d = "layout", plot2d = "layout",
+          method = "single.zigzag")
R> zenplot(olive2, n2dcols = 6, plot1d = "layout", plot2d = "layout",
+          method = "double.zigzag")
R> zenplot(olive2, n2dcols = 6, plot1d = "layout", plot2d = "layout",
+          method = "tidy")
```

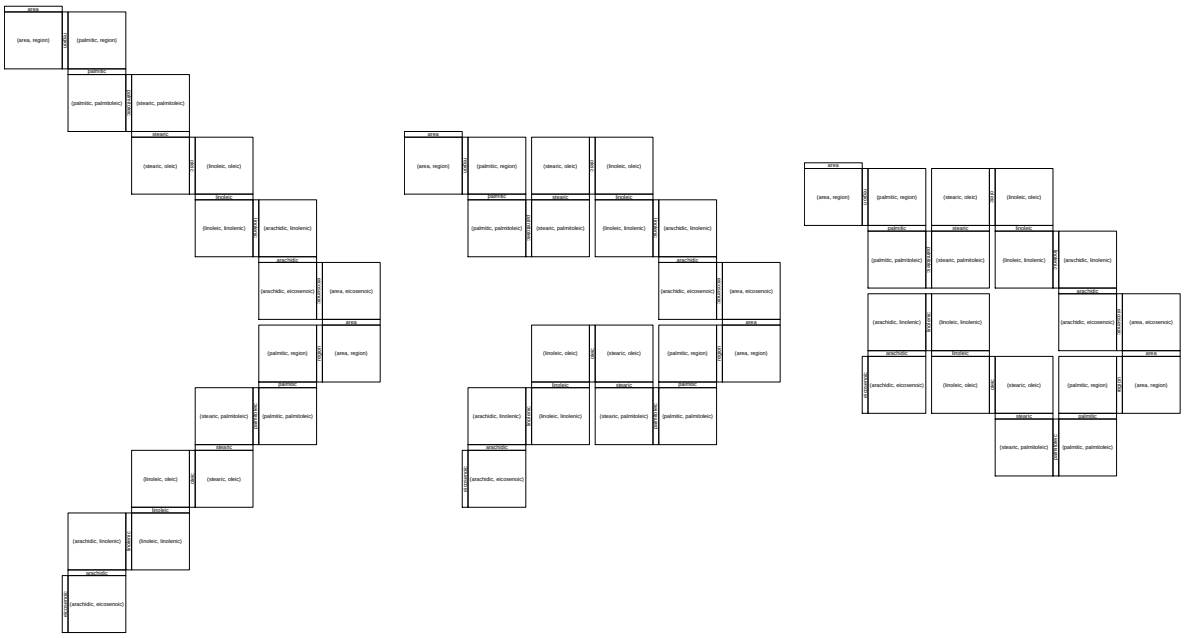

Figure 2: Zenplots of the layout of the doubled olive data set with `method = "single.zigzag"` (left), `method = "double.zigzag"` (middle), `method = "tidy"` (right) and `n2dcols = 6`.

Both `plot1d` and `plot2d` have value `"layout"`, and `n2dcols = 6` so as to exaggerate the effect; the argument `n2dcols` determines the number of columns of 2d plots and is further explained in Section 2.1.2. The leftmost layout is the `"single.zigzag"` method which follows the simplest pattern, zigzagging downwards left to right alternating 1d and 2d plots down and right as it goes. At most `n2dcols` columns can be used for 2d plots. Once this limit is reached, the right hand side is reached, and the pattern then reverses to be from right to left, continuing downwards until ultimately it reaches the corresponding left side of the display area. Were there more plots to lay out, the pattern would continue repeating itself, reversing horizontal directions as each edge of the display is reached.

Note that when the display edge is reached on the right in Figure 2, the rightmost plot is a 2d plot and the change in direction is effected by moving down through a 1d plot followed by moving out left from the 2d plot below it. It is not possible to have a vertical 1d plot as the rightmost plot (unless it were the last plot of all). Minimally, the `"single.zigzag"` method must be able to tell that the edge has been reached and the layout must turn down and reverse directions horizontally.

The `"single.zigzag"` layout is easy to follow but wastes a considerable amount of display space. Had there been only ten 2d plots to display, the plots would move diagonally down the display leaving unused the large areas on either side of the diagonal.

The `"double.zigzag"` method tries to make better use of this space by zigzagging up and down horizontally across the display, exhausting a double row of 2d plots before reaching the edge of the display, where it then moves down enough to reverse horizontal directions and continue as before.

The main challenge for the `"double.zigzag"` method is to not get "cornered" before it reverses horizontal direction. The middle display of Figure 2 shows a `"double.zigzag"` layout. Clearly

there would be room at the top-right corner of this display for three more 2d plots to appear but using this space would trap the layout in that corner; there would be no room to turn and reverse horizontal direction. This is what we mean by getting "cornered". By the time the position of the rightmost 2d plot in the second row (from the top) is reached, the zenplot must determine that it is time to turn down and effect its horizontal reversal. This requires looking ahead a little farther than was done for reversal of the `"single.zigzag"` method.

While the `"double.zigzag"` layout is much more space efficient than is the `"single.zigzag"` one, there is still room for some improvement. For example, in the middle display of Figure 2, the blank space at the left of the row containing only two 2d plots seems wasted. This unused space was generated by the turning of the corner at the right of the display. Similarly, the very last 2d plot at the bottom seems to be poorly placed. This too is a consequence of changing horizontal directions, this time entirely due to anticipating the horizontal reversal. To effect the turn (on the left this time), the `"double.zigzag"` rule is to drop down so as to not get "cornered". However, the method will only get cornered if there are more than three 2d plots left to display. In the present case there is only one 2d plot left which could be easily accommodated by moving up instead of down.

The `"tidy"` method tries to make the most efficient use of the space. This requires looking ahead a little more before each turn. The right most display of Figure 2 shows a `"tidy"` layout. The first two rows are identical to that of the `"double.zigzag"`; so too are the rightmost two in row three, the rightmost three in row four, and the rightmost two in row five. Only the last three 2d plots are positioned in different places. For the `"double.zigzag"` method these are directed down and left anticipating the next possible reversal, whereas for the `"tidy"` method they are directed up and left to fill the empty space. Note that although there were only three 2d plots left, the `"tidy"` method was not in danger of being "cornered". As the position of the very last 1d plot shows, there is room for another 2d plot and for the corner to be turned.

The `"tidy"` method clearly has the most sophisticated look ahead and consequently compact display of the three methods. This becomes more important when there are a great many variates (dimensions) to display.

Finally, `method = "rectangular"` (see the vignette `selected_features`) produces a rectangular layout filled from left to right (then right to left etc.) before moving downwards. This is an example of a method which leaves the zigzagging zenplot paradigm but can be useful for laying out 2d plots which are not necessarily connected through a variable; see **zenplots** for more details.

The number of columns containing 2d plots

As seen in the previous section, the layout of a zenplot (whichever the `method`) depends on the width of the display area available. In `zenplot()` this width is essentially determined by the number of columns there are for 2d plots. This is specified by the value of the argument `n2dcols`. In Figure 2, we specified that this be `n2dcols = 6` for illustrative purposes.

The default value of `n2dcol` is the string `"letter"` and was used, e.g., in Figure 1. The idea here is that the user imagines that the zenplot will be laid out to fit on a North American standard "letter" size display. Other possibilities corresponding to other standard formats are the strings `"square"`, `"A4"`, `"golden"` and `"legal"`; `"golden"` stands for the golden ratio, namely $(1 + \sqrt{5})/2$ (interpreted as height/width).

Provided the total number of 2d plots, say $n_{2\text{dp}}$, is known, the number of columns and rows

containing 2d plots can be determined to (approximately) respect the aspect ratio of any format. This approximation is based on the following reasoning. Let $n_{2\text{dr}}$ and $n_{2\text{dc}}$ denote the number of rows and columns of 2d plots. Since there is typically an empty 2d plot space at the end of a row in a zenplot, the total number of 2d plots displayed is about $(n_{2\text{dc}} - 1)n_{2\text{dr}}$. A standard format suggests that the ratio of the number of rows to columns (i.e., height/width) be fixed at some specified scale $s = n_{2\text{dr}}/n_{2\text{dc}}$. This gives $n_{2\text{dr}} = sn_{2\text{dc}}$ and hence, for known $n_{2\text{dp}}$ and s, $n_{2\text{dc}}$ is the solution to

$$n_{2\text{dp}} = (n_{2\text{dc}} - 1)sn_{2\text{dc}} \quad \text{or equivalently} \quad n_{2\text{dc}}^2 - n_{2\text{dc}} - n_{2\text{dp}}/s = 0.$$

Solving the latter equation gives $n_{2\text{dc}} = \frac{1+\sqrt{1+4n_{2\text{dp}}/s}}{2}$. For `n2dcols` we essentially use the numerical value

$$n_{2\text{dc}} = \max\left\{3,\ \text{round}\left(\frac{1+\sqrt{1+4n_{2\text{dp}}/s}}{2}\right)\right\}, \tag{1}$$

where $n_{2\text{dp}}$ is the number of 2d plots and s is the scaling factor (height/width) given by the relevant standard. Furthermore, if necessary, $n_{2\text{dc}}$ is increased by 1 to obtain an odd number of 2d columns in order to have a slightly more compact layout.

Alternatively, the user may supply any positive integer as the value of `n2dcols`.

The number of 2d plots displayed

In determining the layout, particularly the value of `n2dcols`, the total number of 2d plots to be displayed needs to be determined. This can be determined (by default) directly from the number of variates in the data given by the argument `x`. Alternatively, the user may wish to specify a specific number of plots to be displayed only. This number must be less than or equal to $d - 1$ where d is the number of variates appearing in `x`.

This, and the remaining basic features of a zenplot layout are illustrated by the pair of zenplots displayed in Figure 3: The right-hand side plot shows the effect of the argument `n2dplots` of `zenplot()`. It allows to adjust the number of 2d plots displayed along the zenpath (with the obvious default to show all plots). Due to our choice `n2dplots = 8`, the right-hand side of Figure 3 shows only eight instead of all nine plots.

Omitting first or last 1d plots

Sometimes, the first or last of the 1d plots might be preferred to be omitted from the zenplot display. Examples are if there is no 2d plot after the last one (see Hofert and Oldford (2017, Figure 9) for such an example) or to save space to provide a more compact plot. This is effected by the logical-valued arguments `first1d` and `last1d`. For example, the right-hand side of Figure 1 shows the last 1d plot, the left-hand side of Figure 3 does not.

Relative widths of 1d and 2d plots

The arguments `width1d` and `width2d` are non-negative integers whose ratio determine the relative space given to 1d and 2d plots. If a 2d plot occupies a square of side `width2d`, then every horizontal 1d plot will occupy a rectangle of width `width2d` and height `width1d`, and every vertical 1d plot will occupy a rectangle of width `width1d` and height `width2d`. See the right-hand side of Figure 3.

Spacings between 1d or 2d plots and around whole zenplots

Users will also want to sometimes control the spacing around the individual 1d and 2d plots as well as the spacing around the whole zenplot layout. To illustrate this, as well as some of the arguments of the previous sections, consider the two zenplots produced as follows and displayed as Figure 3:

```
R> zenplot(olive, plot1d = "layout", plot2d = "layout",
+          method = "double.zigzag", last1d = FALSE, ispace = 0.1)
R> zenplot(olive, plot1d = "layout", plot2d = "layout", n2dcol = 4,
+          n2dplots = 8, width1d = 2, width2d = 4)
```

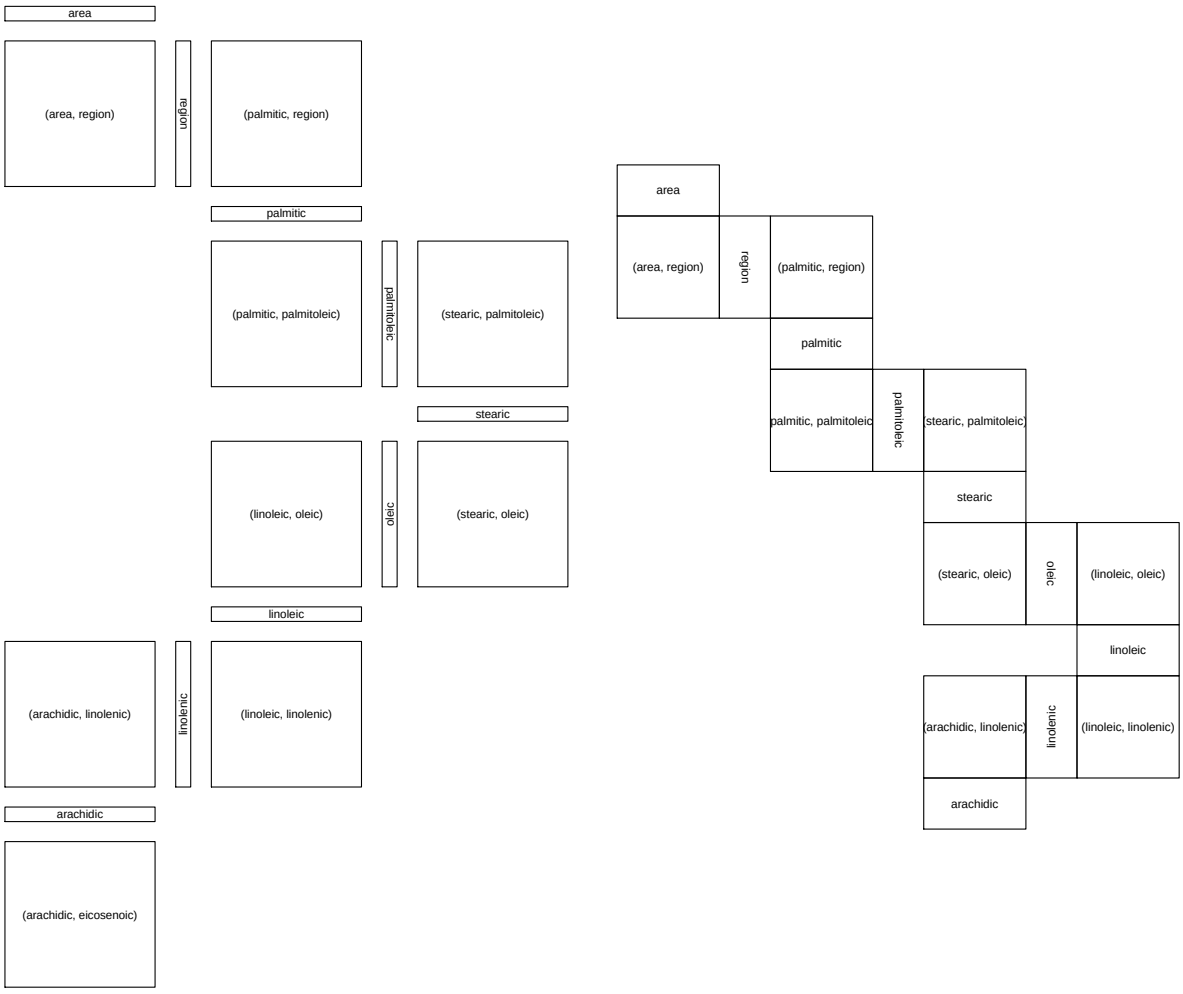

Figure 3: Zenplots of the layout of the olive data set with `method = "double.zigzag"`, `last1d = FALSE` and `ispace = 0.1` (left) and `n2dcol = 4`, `n2dplots = 8`, `width1d = 2`, and `width2d = 4` (right).

The left-hand side of Figure 3 shows the same layout as the right-hand side of Figure 1, except for the changes described in the preceding subsubsections and below.

The left-hand side of Figure 3 has a larger gap between all plots. This follows from `ispace = 0.1`, which sets the inner space (ispace) (a number in $[0, 1]$). For **graphics** (the default; more on that later), `ispace` defaults to 0, otherwise to some small positive fraction. (This difference in behaviour is because other graphics systems (like **grid** or **loon**) may not have a default extension of the plot region and so would lead to clipping of plot symbols near the margins of the plot region.) Note that `ispace` can also be a vector of length four, in which case it provides the bottom, left, top and right inner space. To adjust the space around the whole zenplot, an analagous argument `ospace` is used to determine the zenplot's outer space (ospace).

Note that for **graphics** plots, `ispace` and `ospace` are internally set with the `par` arguments `plt` and `omd`, respectively; for **grid**-based zenplots they are set with respective viewports.

2.2. Plots

A large number of plot functions for each of the arguments `plot1d` and `plot2d` can be specified via character strings. For 1d plots, these include:

`"label"`: to show the label of the current plot variable,

`"points"`: scatter plots (of the data against their index),

`"jitter"`: jittered version of `"points"`,

`"density"`: density plot based on `density()`,

`"boxplot"`: box plots,

`"hist"`: histograms,

`"rug"`: rug plots,

`"arrow"`: an arrow head indicating the direction of the zenpath,

`"rect"`: a rectangle indicating the plot region,

`"lines"`: line spanning the plot region,

`"layout"`: showing the current plot variable and a box around it.

For 2d plots, the following are available:

`"points"`: scatter plots (of the two data columns under consideration),

`"density"`: density plot based on **MASS**'s `kde2d()` (see Ripley (2017)),

`"axes"`: coordinate axes,

`"label"`: to show the labels of the two current plot variables,

`"arrow"`: as for 1d plots (see above),

`"rect"`: as for 1d plots (see above),

`"layout"`: showing the pair of current plot variables and a box around it.

It is also possible for the user to supply their own plot functions; this will be covered in more detail in Section 4.

Enforcing common plot limits

By construction, neighbouring plots along a trajectory share a variate (on their common dimension/axis) and hence have common data limits on that variate. This allows the viewer to visually track the same location across adjacent plots that share that variate. By default, the limits of these shared dimensions are determined individually for each variate; this is effected by the `lim` argument's default value `"individual"`.

Should the viewer wish that all dimensions have the same extent across all plots, then `lim = "global"` should be used. Alternatively, when the data argument `x` is a list (indicating groups of data and variates; see Section 2.3 and 4) setting `lim = "groupwise"` ensures that within each group all plots have identical data limit extents.

The graphical system

By default, the R package **graphics** is used for drawing zenplots; see `pkg = "graphics"`. The underlying rectangular layout with rows and columns containing the 1d and 2d plots is constructed internally with `layout()` based on `width1d` and `width2d`; the same applies to `pkg = "loon"`. For `pkg = "grid"`, the layout is implemented with the more sophisticated `grid.layout()`. The 1d and 2d plots are placed (with `placeGrob()`) in a frame grob (with `frameGrob()`) which is returned invisibly. This grid object can further be modified, if required. The support of `pkg = "grid"` opens the door for other **grid**-based graphics systems; see, e.g., Figure 5 which shows a zenplot based on **ggplot2** (the special layout will be discussed in Section 4.3). If `pkg = "loon"`, then an interactive zenplot (e.g., with brushing, zooming, panning, and so on) is constructed via the R package **loon**. The resulting zenplot can then be linked to any other interactive plot in **loon**.

Note that plotting is done if `draw = TRUE` (the default). Either way, information about the underlying zenpath and layout is returned; more on this later.

Ellipsis arguments

All plots produced using the basic argument values described above for `plot1d` and `plot2d` are implemented using standard plot functions from the relevant package. The ellipsis arguments are passed on as extra arguments to these standard plot functions. So, e.g., if a zenplot is using `pkg = "graphics"` (the default), then plot arguments such as `col`, `cex`, `pch` and so on could reasonably be given to `zenplot()` whenever `plot1d = "points"` or `plot2d = "points"` were given as well; the extra arguments and their values would be passed on.

2.3. Data

The data argument `x` of `zenplot()` is typically a `matrix` or a `data.frame`. In either case, each column/variate of `x` is taken to be a dimension and the trajectory moves through these dimensions in order as described earlier.

However, the value of `x` can also be a `list` of objects of these types. Then each element of the list is interpreted as a group of variables which belong together. As such, each group should be visually separated from one another in the resulting zenplot. This case is covered in more detail in Section 4.

3. Zenpaths

Suppose the `zenplot()` argument `x` is a matrix named `dataMat` having $d = 5$ columns. By default, the zenplot will follow the trajectory through the data given by the path $1, 2, 3, 4, 5$ as in

```
R> (path <- 1:5)

[1] 1 2 3 4 5
```

and the sequence of dimension pairs $(1, 2), (2, 3), (3, 4), (4, 5)$ would determine the order and dimensions of the 2d plots.

This default sequence is not perhaps the best one to reveal interesting structure in the data. We might, e.g., wish to look at all pairs of dimensions at once in the zenplot. This could be the sequence

```
R> zenpath(5)

 [1] 5 1 2 3 1 4 2 5 3 4 5
```

for example since every number appears beside every other number once. Or, if we had some way to measure the interestingness of any pair of dimensions, we might choose a path which visited only the most interesting pairs.

In any case, if `path` were a vector which contained the desired sequence then

```
R> zenplot(x = dataMat[,path])
```

would produce the zenplot that followed the navigation path of interest. We call such a navigation path a zenpath.

To construct zenpaths, the R package **zenplots** provides a function of the same name, `zenpath()`, which takes a variety of arguments and returns a sequence of dimensions which can be used for the path.

```
R> str(zenpath)

function (x, pairs = NULL,
          method = c("front.loaded", "back.loaded", "balanced",
                     "eulerian.cross", "greedy.weighted", "strictly.weighted"),
          decreasing = TRUE)
```

Some of the methods to construct zenpaths simply depend upon the number of dimensions that are involved and ensure that all possible pairs of dimensions occur in the sequence. Since there are a great many possible sequences a few approaches are offered:

- `"front.loaded"`: If `x` is an integer and `method = "front.loaded"` (the default), `zenpath()` provides a sequence of numbers which, considering two consecutive numbers at a time, contains all pairs of the variables 1 to `x` sorted in such a way that the first variables appear the most frequently early in the sequence. Note that this sequence of numbers is typically not exactly of length $\binom{d}{2}$ as the pairs have to be "connected" along a zenpath in the sense of consecutive numbers building pairs along the zenpath.

```
R> zenpath(5, method = "front.loaded")

 [1] 5 1 2 3 1 4 2 5 3 4 5
```

- `"back.loaded"`: Similar to `method = "front.loaded"` but with later variables appearing the most frequently later in the sequence.

```
R> zenpath(5, method = "back.loaded")

 [1] 1 2 3 1 4 2 5 3 4 5 1
```

- `"balanced"`: Similar to `method = "front.loaded"` but with variables appearing in balanced blocks throughout the sequence (a so-called Hamiltonian Decomposition, see Hurley and Oldford (2011b)).

```
R> zenpath(5, method = "balanced")

 [1] 1 2 3 5 4 1 3 4 2 5 1
```

- `"eulerian.cross"`: For this method, two integers representing the sizes of two groups of variables must be passed to `zenpath()`. `zenpath()` returns a sequence of numbers, which, when interpreted two consecutive numbers at a time, sorts all pairs of variables such that each pair is formed with one variable from each group.

```
R> zenpath(c(3,5), method = "eulerian.cross")

 [1] 1 4 2 5 1 6 2 7 1 8 3 4 3 6 7 3 5 8 2
```

As an example, Figure 4 displays a zenplot of all pairs of `olive` acids only (no `"region"` or `"area"`) based on an Eulerian zenpath showing the (default `"front.loaded"`) ordering of all pairs of variables. Such a display would be of particular interest for very large data sets where simple scatter plots would have so many points overstriking one another as to obfuscate patterns in the density easily seen via contour plots. The code to generate the figure is as follows:

```
R> oliveAcids <- olive[, !names(olive) %in% c("area", "region")]
R> zpath <- zenpath(ncol(oliveAcids))
R> zenplot(oliveAcids[, zpath], plot1d = "hist", plot2d = "density")
```

The histograms point in the direction of the zigzag layout. Since `plot1d = "hist"` by default only shows histograms , the labelling of the variates known from the default `plot1d = "label"` is lost; a more sophisticated version of this plot using `pkg = "grid"` will be given in Section 4.1 to illustrate how more complex displays via simple `plot1d` and `plot2d` functions can be written.

Although it is possible to lay out hundreds of thousands of plots with a zenplot, see Hofert and Oldford (2017), one is often not interested in all of them. For example, Hofert and Oldford (2017, Figure 6) shows a zenplot containing only those 10 pairs of variables with largest and those 10 pairs with smallest pairwise tail dependence among all $\binom{465}{2}$ =107,880 pairs of stocks in the S&P 500. Given a (numerical) measure of "interestingness" (such as correlation, tail

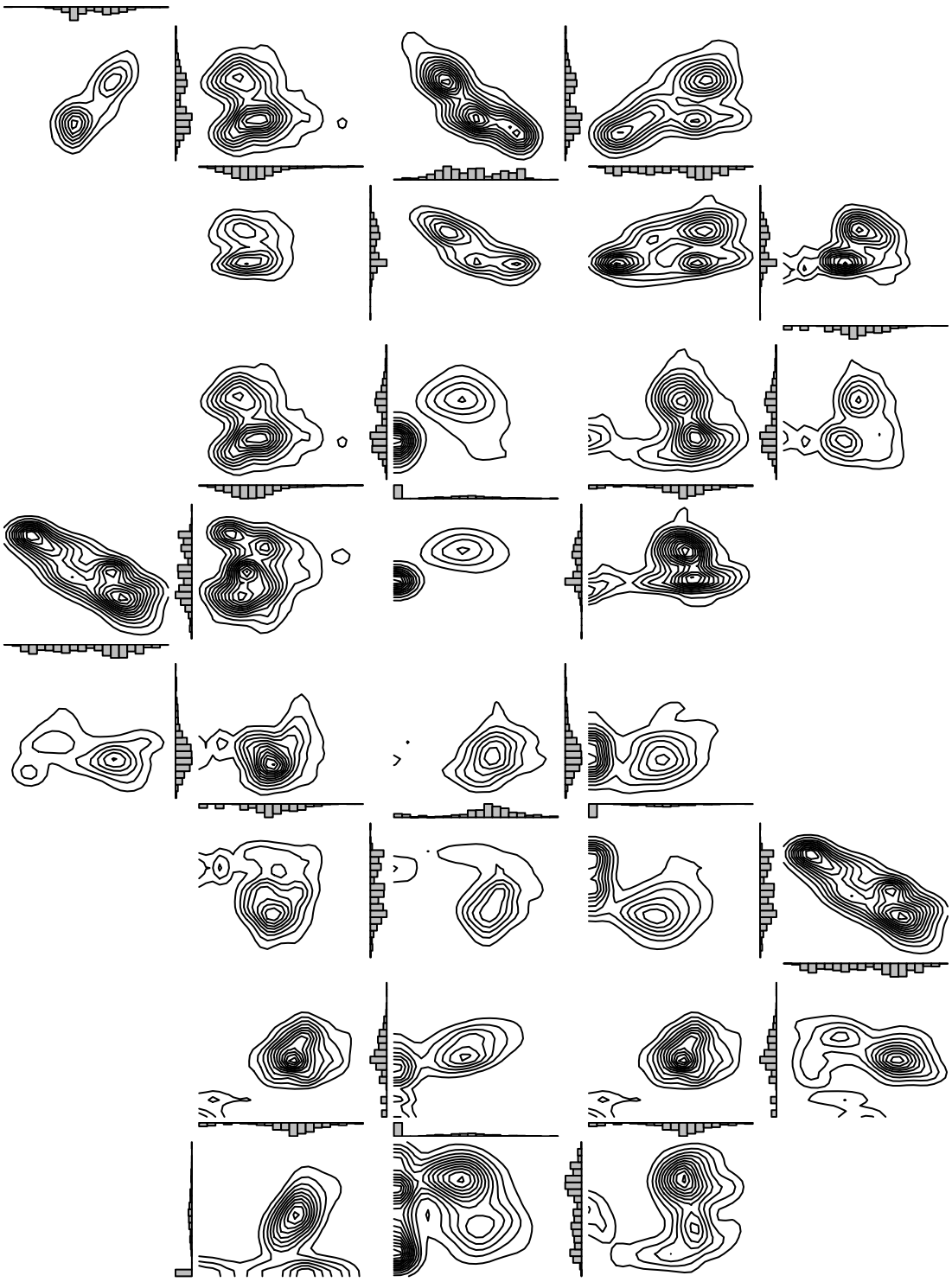

Figure 4: Zenplot of an Eulerian zenpath showing histograms (as 1d plots) and densities (as 2d plots) of all pairs of the acids of the olive data set.

dependence etc.) assigned to all pairs of variables, one can sort all pairs according to this measure and plot the most (or also the least) interesting ones.

The `zenpath()` methods `"greedy.weighted"` and `"strictly.weighted"` provide the means for doing this, given that `x` is a numeric vector or matrix or distance matrix of weights which encodes the interestingness of each pair. The argument `pairs` of `zenpath()` contains a two-column matrix, that, row-wise, contains the connected pairs of variables to be sorted according to the weights `x`; if `pairs = NULL`, a default is constructed in lexicographical order. For `method = "greedy.weighted"`, `zenpath()` returns a sequence of numbers which sorts all pairs according to a greedy (heuristic) Euler path visiting each edge at least once (some edges may be visited twice since when the graph is not even, some number of edges will be duplicated to make it so and an Eulerian visiting each edge on the new graph precisely once will be constructed, see Hurley and Oldford (2011a)); note that this method internally uses the function `ftM2graphNEL()` from the R package **graph** of Gentleman, Whalen, Huber, and Falcon (2019), available from Bioconductor. For `method = "strictly.weighted"`, `zenpath()` returns a sequence of numbers strictly respecting the order given by the weights, so the first, second, third, etc. adjacent pair of numbers of the output of `zenpath()` corresponds to the pair with largest, second-largest, third-largest, etc. weight; should there be a disjunction in the path then plots will be grouped by contiguous sub-path (see Section 4.5 for an example). If the argument `decreasing` of `zenpath()` is `TRUE` (the default), then the sorting by weight is done in decreasing order.

4. Build your own zenplots

The `zenplot()` arguments `plot1d`, `plot2d` and `method` provide sufficient functionality to satisfy a variety of common uses. Spacing arguments provide some simple aesthetic control over the layout and the use of `zenpath()` allows the analyst to select a trajectory through the high-dimensional data space. In this section, we describe some extended features of `zenplot()` which allow the user to begin to tailor a `zenplot()` to their particular analysis needs. This will require some familiarity with one of the graphics packages allowed by `zenplot()`.

4.1. Custom plot functions

Each named value of either argument `plot1d` or `plot2d` is implemented by calling a **zenplots** package function whose name has that argument value as prefix and the drawing package name as suffix. Between prefix and suffix the plot type `1d` or `2d` appears. For example, for arguments `plot1d = "hist"`, `plot1d = "points"` and `pkg = "grid"`, 1d plots are drawn by the function `hist_1d_grid()` and 2d plots by the function `points_2d_grid()`.

That is, the plotting functions called within `zenplot()` are of the form `*_1d_graphics()`, `*_2d_graphics()` (for `pkg = "graphics"`), `*_1d_grid()`, `*_2d_grid()` (for `pkg = "grid"`) and `*_1d_loon()`, `*_2d_loon()` for `pkg = "loon"`; here the `*` stands for the respective string as given above. All these functions are exported from the **zenplots** package and so are also available to the user for direct use.

This means, e.g., that the effect of the call

```
R> zenplot(oliveAcids, plot1d = "hist", plot2d = "density", pkg = "graphics")
```

would be identically produced by

```
R> zenplot(oliveAcids, plot1d = hist_1d_graphics, plot2d = density_2d_graphics,
+          pkg = "graphics")
```

Note that `hist_1d_graphics` and `density_2d_graphics` each end in "`_graphics`" indicating that all of their drawing is done using only functions from the **graphics** package.

As the second case suggests, the user may also provide their own plot function using functionality exclusively from the named package (e.g., `pkg = "graphics"`). However, because it will be called within `zenplot()`, any user supplied plot function must also accept `zargs` as its first argument, the value of which will be constructed by `zenplot()`. The built-in plot functions named above may be helpful as templates and can also be called within any other function provided it passes on the `zargs` argument.

The `zargs` argument consists of a `list` of further named arguments; which of these named arguments must appear in this list is determined by the `zenplot()` call and in particular by the value given to its argument `zargs`. For `zenplot()`, the `zargs` argument is merely a `logical` vector indicating which of the named set of arguments are to be passed to every 1d and 2d plot function as argument `zargs`.

For example, the default value of `zargs` (in the call to `zenplot()` is `c(x = TRUE, turns = TRUE, orientations = TRUE, vars = TRUE, num = TRUE, lim = TRUE, labs = TRUE, width1d = TRUE, width2d = TRUE, ispace = match.arg(pkg)!= "graphics")`. If `TRUE`, the value of each argument (constructed within `zenplot()` itself), is passed to the every 1d and 2d plot functions (in a named `list`) as the value of the plot function's argument `zargs`. The value and meaning of each argument contained in this `zargs` is as follows:

`x`: the original data object `x` as provided by the user. Virtually any object can thus be passed to the underlying 1d and 2d plot functions as long as the latter two take care of appropriate plotting. As such, `zenplot()` completely distinguishes between layout and plotting, which provides great flexibility for data visualization. Although certainly a rare use case, setting `x = FALSE` in `zargs` avoids `x` being passed on. With appropriate `plot1d` and `plot2d` arguments, one can then even plot independently of the provided data object.

`turns`: the `character` vector of turns (either computed or user-provided). This has the advantage of each 1d and 2d plot knowing all turns before plotting is done and thus the layout of the zenplot. Typically most helpful are the turns into and out of the current plot position. For example, one could have all plots colored blue (red) which turn down (up) out of the current position along the zenpath.

`orientations`: the `character` vector consisting of the plot orientations ("s" for square (so 2d) plots, "h" for horizontal and "v" for vertical plots) as determined internally based on the turns. This information can be used, e.g., to determine the orientation of plot labels in a zenplot.

`vars`: a two-column `matrix` containing, for each 1d and 2d plot (so each row), the (pairs of) variable(s) which is (are) to be plotted in the current plot; for 1d plots, the respective row simply shows the same variable twice.

`num`: the current plot number among all 1d or 2d plots along the zenpath. This is especially helpful for indexing, say, a row in `vars` to find out which variable(s) was/is (were/are) to be plotted previously, currently and next.

`lim`, `labs`, `width1d`, `width2d`, `ispace`: (already discussed) arguments of `zenplots()` that are passed on to the underlying 1d and 2d plot functions.

Any user defined 1d or 2d plot function can then access any of these values from its `zargs` argument provided they were marked for use by the logical values of the argument `zargs` of the `zenplots()` call. This gives the user access to all of the functionality used by the built-in plotting functions.

Furthermore, `plot1d` or `plot2d` can also be `NULL`, in which case an "empty plot" is generated, so no visible plotting is done.

Besides `zargs`, the ellipsis arguments ("`...`") are also passed to the underlying 1d and 2d plot functions. If only one of the two functions shall obtain additional graphical parameters, say, one can proceed as already seen in the code for the right-hand side of Figure 6, namely by providing a corresponding 1d or 2d plot function.

4.2. Custom layouts – as the plot turns

The defining layout of a zenplot is the "zigzag" pattern determined by the value of the `zenplot()` argument `method`. The various choices were described and illustrated in Section 2.1 and all zenplots so far considered have followed one of these zigzag layout methods. The attentive reader may have noticed, however, that one of the layout arguments mentioned in Section 2, namely the `turns` argument, did not appear in the basic layout discussion of Section 2.1. This is because the `turns` argument, if specified, will override the `method` argument and so can be used to produce a zenplot which need not follow a zigzag pattern; normally, the argument `turns` is not given but is determined within `zenplot()` based on the value of its `method` argument.

What is meant by `turns`? These are directional instructions "d", "l", "u" and "r", being short for the directions "down", "left", "up" and "right", which describe the next direction that the layout path follows upon completion of that plot. Each 1d or 2d plot has a directional instruction associated with it which describes the direction to be taken from that plot to the next. In this way, the entire layout path can be thought of as a sequence of directional instructions; this sequence is the value of `turns`.

For example, consider the most recent zenplot given in Figure 4. The zenplot begins with the histogram (a 1d plot) in the top-left corner. The next plot in the layout is immediately below the histogram so that the directional instruction from the histogram is "d", meaning move down. Moving down leads to the 2d density contour plot. From this plot the layout moves right to the next histogram so this first density contour has exiting directional instruction "r". This leads to the next histogram from which again we move "right" to the next density plot from which we move down to the next histogram, and so on. The value of `turns` associated with this plot is the sequence `"d", "r", "r", "d", "d", "r", "r", "u", "u", "r", "r", "d", "d", "r", "r", "d", "d", "l", "l", "d", "d", "l", "l", "u", "u",...`, `"d", "d", "l"`. Note that there is no turn associated with the last histogram at the lower-left of the Figure 4 because no plot follows this histogram in the layout.

Note the zigzag pattern apparent in both these turns and in the display layout. In the next section, we consider a layout which does not follow any of the zigzag patterns considered so far, though it is a zigzag spiral. In general, `turns` need not follow a zigzag of any sort.

4.3. Custom layout and plots – a spiral of ggplots example

For most applications, the sequence of turns provided by `zenplot()` via the `method` argument

will be all that is needed for a compact layout. The value of the `turns` argument is that it provides the user a means to construct new layouts which may be more appropriate to their analysis.

To illustrate this, and to also show how plots from `ggplot2` can be incorporated, we consider how plots might be laid out in a spiral. Though perhaps a somewhat fanciful layout, it will serve to show how much control the user has over layout should they desire it or their application warrant it. Even over-plotting of one plot on top of another could be possible (e.g. via the `grid` package) though would not be generally recommended.

Because the zenpath is laid out according to the given `turns`, the length of `turns` must match that of the zenpath. For the `oliveAcids` we might therefore choose the following path and sequence of turns:

```
R> path <- c(1,2,3,1,4,2,5,1,6,2,7,1,8,2,3,4,5,3,6,4,7,3,8,4,5,6,7,5,8,6,7,8)
R> turns <- c("l",
+             "d","d","r","r","d","d","r","r","u","u","r","r","u","u","r","r",
+             "u","u","l","l","u","u","l","l","u","u","l","l","d","d","l","l",
+             "u","u","l","l","d","d","l","l","d","d","l","l","d","d","r","r",
+             "d","d","r","r","d","d","r","r","d","d","r","r","d","d")
```

which already begins in a different way in that the layout moves "left" out from the first plot. The resulting layout starts at the centre of the display area and moves in a zigzag pattern but now in a (counter-clockwise) spiral instead of any of the row-filling patterns seen earlier.

Note that there is no restriction on `turns` so that the layout is limited only to following a continuous path of alternating 1d and 2d plots. This flexibility allows analysts to tailor the layout to the needs of the problem.

As an example, we might prefer to use the plotting paradigm given by the R package **ggplot2**. In this case, we need to write custom 1d and 2d plotting functions. Here, we write a custom 2d plot function.

```
R> library("ggplot2")
R> stopifnot(packageVersion("ggplot2") >= "2.2.1")
R> ggplot2d <- function(zargs) {
+    r <- extract_2d(zargs)
+    num2d <- zargs$num/2
+    df <- data.frame(x = unlist(r$x), y = unlist(r$y))
+    p <- ggplot() +
+      geom_point(data = df, aes(x = x, y = y), cex = 0.1) +
+      theme(axis.line = element_blank(),
+            axis.ticks = element_blank(),
+            axis.text.x = element_blank(),
+            axis.text.y = element_blank(),
+            axis.title.x = element_blank(),
+            axis.title.y = element_blank())
+    if(num2d == 1) p <- p +
+      theme(panel.background = element_rect(fill = 'royalblue3'))
+    if(num2d == (length(zargs$turns)-1)/2) p <- p +
+      theme(panel.background = element_rect(fill = 'maroon3'))
+    ggplot_gtable(ggplot_build(p))
+  }
```

Note the use of `extract_2d(zargs)` to get the x and y coordinates. The principal additions to `ggplot()` are the removal of several default plot elements. Note also the use of `num2d <- zargs$num/2` to get the total number of 2d plots in the zenplot so that the first and last ones can be distinguished by background colour (respectively, `"royalblue3"` and `"maroon3"`). Finally, because `ggplot()` is written using the **grid** package, a suitable grid object must be returned.

The spiral zenplot, beginning at the center (`"royalblue3"`) and zigzagging counter-clockwise to end at the bottom-right (`"maroon3"`) and using `ggplot()`, is now simply constructed as

```
R> zenplot(oliveAcids[,path], turns = turns, pkg = "grid",
+          plot2d = function(zargs) ggplot2d(zargs))
```

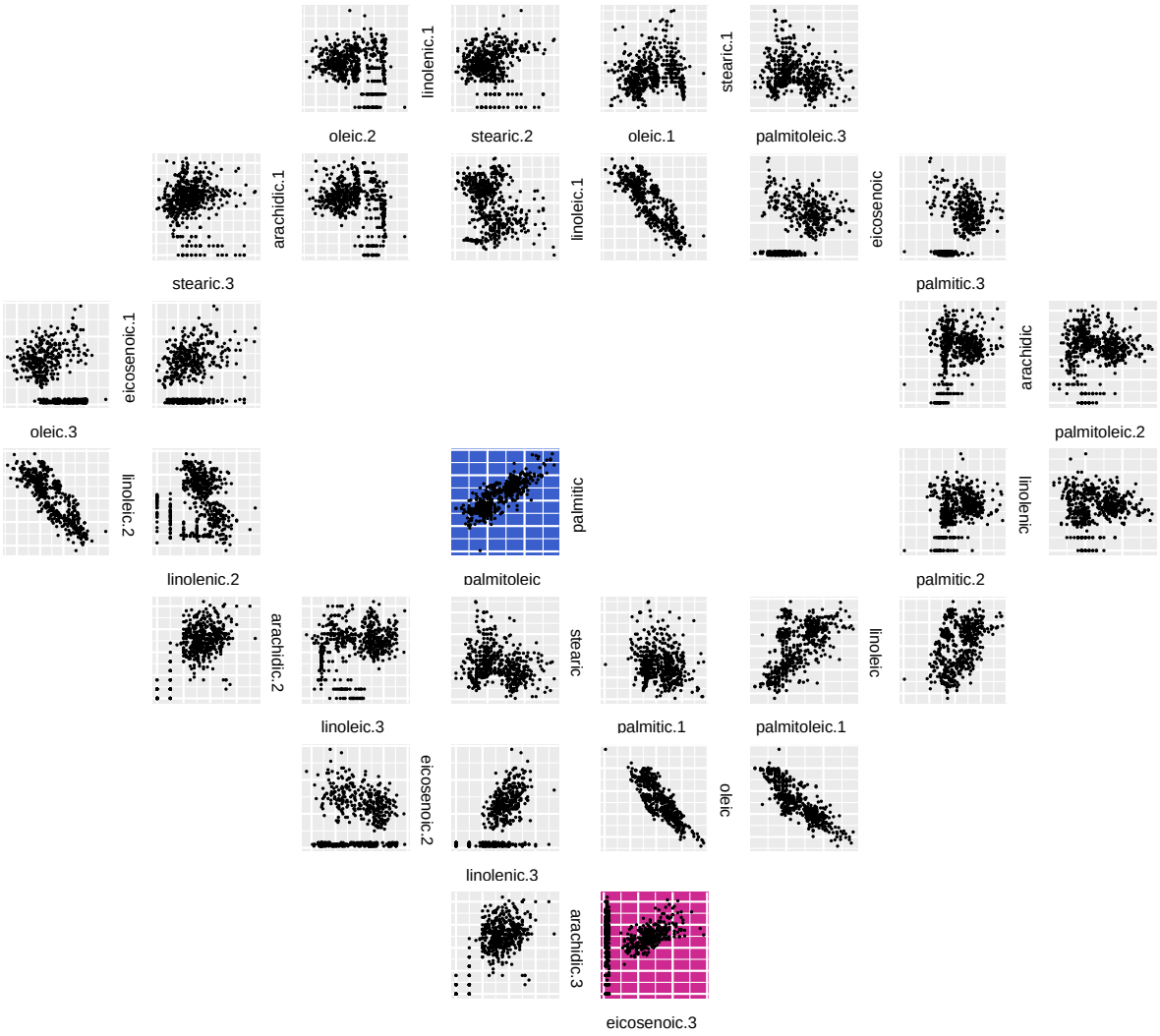

Figure 5: A zenplot displaying all pairs of acid components of the olive data at least once. In particular, this zenplot shows that one can use one's own turns (provided by the argument `turns`) and that plotting can also be done with **ggplot2**.

4.4. Data groups

In Section 2.3 it was noted that the data argument `x` might also be a list. In such cases, `zenplot()` assumes that each element of the list is itself a data set comprised of its own set of variates (dimensions).

The Italian growing region (e.g. `Umbria`, `Sicily`) is known for each row (i.e., oil) of the olive data and these regions have been grouped into three broad geographic areas: `South`, `Sardinia`, and `Centre.North`. It would be of interest then to study the relationships between the variates for olive oils within each area and also to see how these relationships might differ between areas. Zenplots provides a layout for observing relationships by group.

To this end, we take the `oliveAcids` data and split it into groups according to geographic area as follows:

```
R> oliveAcids.by.area <- split(oliveAcids, f = olive$area)
R> names(oliveAcids.by.area)[3] <- gsub("\\.", " ", names(oliveAcids.by.area)[3])
R> names(oliveAcids.by.area)

[1] "South"        "Sardinia"     "Centre North"
```

The data `oliveAcids.by.area` are a list having three components, the first, second and third containing 323, 98 and 151 measurements of the eight acid components, respectively. Zenplots of these data groups can now be produced as

```
R> zenplot(oliveAcids.by.area, labs = list(group = NULL))
R> zenplot(oliveAcids.by.area, lim = "groupwise", labs = list(sep = " - "),
+          plot1d = function(zargs) label_1d_graphics(zargs, cex = 0.8),
+          plot2d = function(zargs)
+              points_2d_graphics(zargs, group... = list(sep = "\n - \n")))
```

which appear as the left- and right-hand sides of Figure 6.

Following the zigzag in the left zenplot of Figure 6, we can see the groups separate first at the 2d plot containing only the geographic area names `South` and `Sardinia` and then farther on at the 2d plot containing only area names `Sardinia` and `Centre North`. These two 2d plots mark the boundaries between the groups, which in order are `South`, `Sardinia`, and `Centre North`. At a glance, we can see the greater number of observations in the `South` area, and the coarse granularity of the measurements apparent in much of the data from the `Centre North` (suggesting rounding or some other source of measurement error). Comparing relationships, one can also see for example that the acids `palmitic` and `palmitoleic` appear positively correlated in the `South` but possibly uncorrelated in `Sardinia` amd even `Centre North`. Other similarities and differences (e.g. measures on `eicosenoic`) between groups can also be readily observed.

In both zenplots of Figure 6, the displays follow the zigzag pattern for the variates that appear within each data group (each of which can have its own number of observations); the zigzag pattern continues on from one group to the next with a visually separating 2d plot showing the names of the groups to either side. This visual separator plot is constructed as a 2d plot function, `group_2d_graphics()` (and, for the other graphical systems, `group_2d_grid()` and `group_2d_loon()`, following the naming convention of Section 4.1), that simply displays the

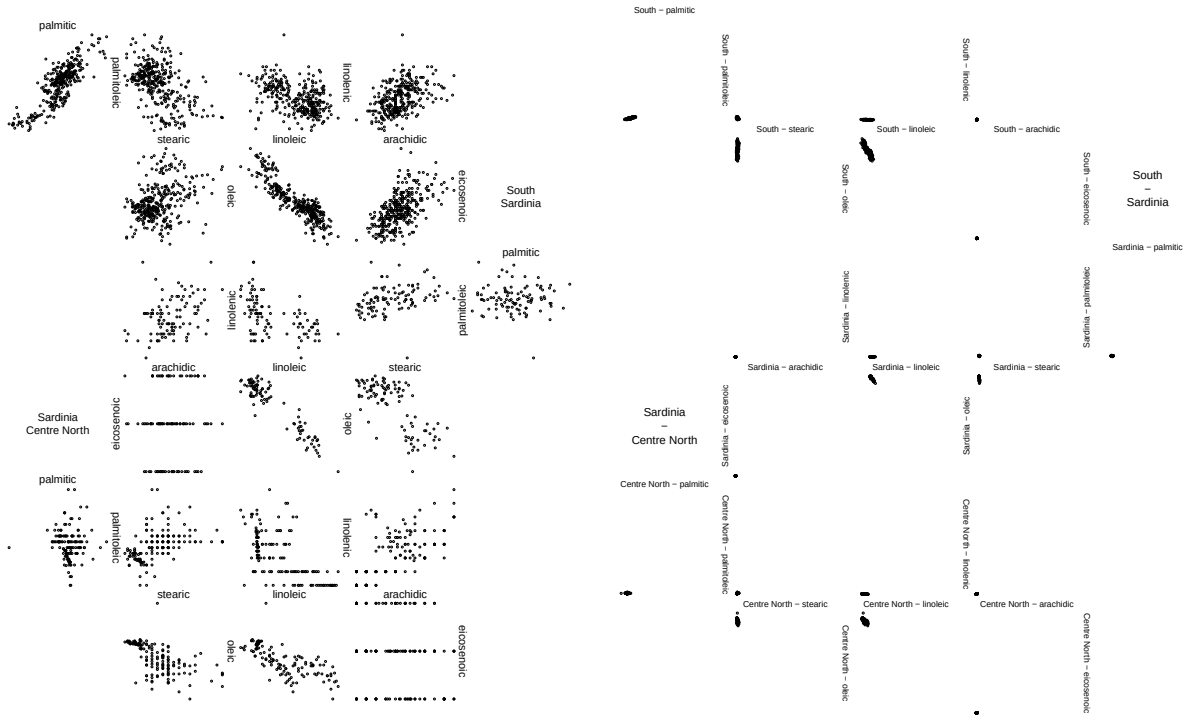

Figure 6: Zenplots showing the grouping feature (both left and right) and label feature (left: no group labels in 1d plots; right: groupwise scaling, label separators, downsizing labels).

group labels in the order in which they appear indicating the transition from one group to the other (see "South" and "Sardinia" as well as "Sardinia" and "Centre North"; the first/top label indicates the preceding group along the zenpath, the second/bottom label indicates the next group). In the right-hand side zenplot, we use a more sophisticated option, namely the possibility to pass arguments to `group_2d_graphics()` with the argument `group...`. We use this to visually separate the group labels a bit better in this case. To this end, note that we need to provide a user-defined function (which just calls `points_2d_graphics()` with one argument different from the default).

In the right-hand side zenplot, every 1d plot has the group name appear before that variate name (separated by a hyphen). This was effected by the argument `labs = list(sep = " - ")`. In contrast, the left-hand side zenplot omitted the group labels from its 1d plots by specifying `labs = list(group = NULL)`. Typically, `labs` is a list containing the components `group` (group label basename or labels for the groups; can be `NULL` in which case group labels are omitted), `var` (variable label basename or labels for the variables; can be `NULL` in which case variable labels are omitted), `sep` (separator between group and variable labels) and `group2d` (a logical indicating whether the `group_2d_*()` functions should still plot group labels even if `group = NULL`).

Also note that because the resulting 1d plot labels would be too large when the group name was prepended, a user-defined function was provided as the argument of `plot1d` so that the size of this label could be reduced via the (ellipsis) argument `cex = 0.8` passed on to the built-in plotting function `label_1d_graphics()` (see Section 4.1 for more complex examples).

Each displayed 2d plot also has its own limits (`lim = "individual"`, the default) as seen in the left-hand zenplot of Figure 6. In the right-hand zenplot, common limits are used within each group (`lim = "groupwise"`). The effect is that for this data the relationships between acid concentrations within any group have been obscured by the common scale across variates within groups (especially compared to when `lim = "individual"` as in the leftmost zenplot). Even though all proportions of olive acids must sum to 1 for each oil (or to 10,000 on the $100 \times \%$ recorded values in the data), the oils are seen to contain far more `oleic` than any other fatty acid in the data. In all areas, `oleic` has so much higher concentration than the rest that on a common scale the others are compressed into the lower end of their axes. A third choice for the `lim` argument would be to have identical limits across all plots regardless of groups (`lim = "global"`). For the olive data, the effect would be nearly identical to that of `lim = "groupwise"`.

4.5. Custom zenpaths

The R package **zenplots** provides auxiliary functions for connecting and extracting pairs along a zenpath or for obtaining a list of matrices for plotting a zenplot with grouping feature. We will now briefly present these functions.

Say our measure of "interestingness" is "convexity" according to scatter-plot diagnostics (scagnostics) and we are interested in the zenpath containing the six most convex pairs. We could proceed as follows. In a first step, we build a (d, d)-matrix, which, in the (i, j)th entry, contains the convexity measurement of the pair (i, j); this is the matrix `M` below.

```
R> library("scagnostics")
R> Y <- scagnostics(oliveAcids)
R> X <- Y["Convex",]
R> d <- ncol(oliveAcids)
R> M <- matrix(, nrow = d, ncol = d)
R> M[upper.tri(M)] <- X
R> M[lower.tri(M)] <- t(M)[lower.tri(M)]
R> round(M, 5)
```

```
        [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]
[1,]      NA 0.48952 0.46343 0.45887 0.43914 0.34583 0.31259 0.28413
[2,] 0.48952      NA 0.42276 0.50499 0.44591 0.35855 0.35846 0.31729
[3,] 0.46343 0.42276      NA 0.39700 0.36394 0.31316 0.29534 0.33709
[4,] 0.45887 0.50499 0.39700      NA 0.46454 0.36616 0.29451 0.34888
[5,] 0.43914 0.44591 0.36394 0.46454      NA 0.31977 0.31443 0.36750
[6,] 0.34583 0.35855 0.31316 0.36616 0.31977      NA 0.53726 0.34001
[7,] 0.31259 0.35846 0.29534 0.29451 0.31443 0.53726      NA 0.22231
[8,] 0.28413 0.31729 0.33709 0.34888 0.36750 0.34001 0.22231      NA
```

Next, we use the entries in `M` to determine the zenpath of strictly ordered pairs (according to the ordering with respect to "convexity"); this can be done with `zenpath(, method = "strictly.weighted")` as described in Section 3.

```
R> zpath <- zenpath(M, method = "strictly.weighted")
R> head(M[do.call(rbind, zpath)])
```

```
[1] 0.5372599 0.5049945 0.4895179 0.4645377 0.4634277 0.4588675
```

Note that `method = "strictly.weighted"` returns a list of vectors of length two since there is no guarantee to obtain connected pairs when given weights have to be strictly respected. To obtain connected pairs along this list, we provide the function `connect_pairs()`; see the last part of this section.

We can now conveniently use the function `extract_pairs()` as follows to extract the pairs we are most interested in, namely the first six along the zenpath.

```
R> (ezpath <- extract_pairs(zpath, n = c(6, 0)))

[[1]]
[1] 7 6

[[2]]
[1] 4 2

[[3]]
[1] 2 1

[[4]]
[1] 5 4

[[5]]
[1] 3 1

[[6]]
[1] 4 1
```

The function `graph_pairs()` (also depending on `ftM2graphNEL()` from the R package **graph** on Bioconductor) can be used to depict these six pairs in a graph, where an edge indicates that the adjacent nodes build a pair; see Figure 7.

```
R> library("graph")
R> plot(graph_pairs(ezpath))
```

As we can see, some of the pairs are naturally connected: For example, the pairs with second- and third-largest convexity measure share the variable 2 and thus could be (zen)plotted in the same group of variables $(4, 2, 1)$ (so plotting the pairs (4,2) and (2,1) with the variable 2 along the joint axis). How can we find connected pairs along a zenpath which still strictly respect the order given by "convexity"? This can be achieved with the function `connect_pairs()`, which provides us with a "compactified" zenpath which then also leads to a more compact zenplot.

```
R> (cezpath <- connect_pairs(ezpath))

[[1]]
[1] 7 6

[[2]]
[1] 4 2 1
```

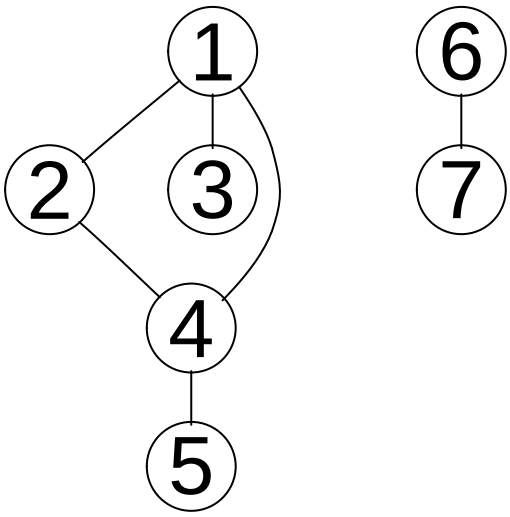

Figure 7: Graph representing those six pairs along the zenpath which exhibit largest convexity; an edge corresponds to a pair between the adjacent variables.

```
[[3]]
[1] 5 4

[[4]]
[1] 3 1 4
```

We can use the function `groupData()` to group our data into the groups determined by `cezpath`.

```
R> oliveAcids.grouped <- groupData(oliveAcids, indices = cezpath)
```

Lastly, we can plot the corresponding (grouped) zenplot, see Figure 8.

```
R> zenplot(oliveAcids.grouped)
```

5. Advanced features

In the previous section, a variety of ways in which zenplots might be customized to suit the analysis were described and illustrated. There could be many more such illustrations given. In fact, an immense amount of control over the zenplot display is available to the user. This can be used to construct nearly arbitrary displays beyond any we have imagined here. To facilitate such creativity, a number of auxiliary software tools are provided in the **zenplots** package. This section provides detail on these tools so that the user can create their own novel layouts.

Section 5.1 describes the structure of a zenplot and highlights some of its important elements. In particular, the occupancy matrix is introduced which provides a potentially useful rectangular array description of the positions occupied by every plot in the display.

In Section 5.2 several important functions useful to create plots are described. Particularly important are those functions which extract the data so that it can be accessed by the

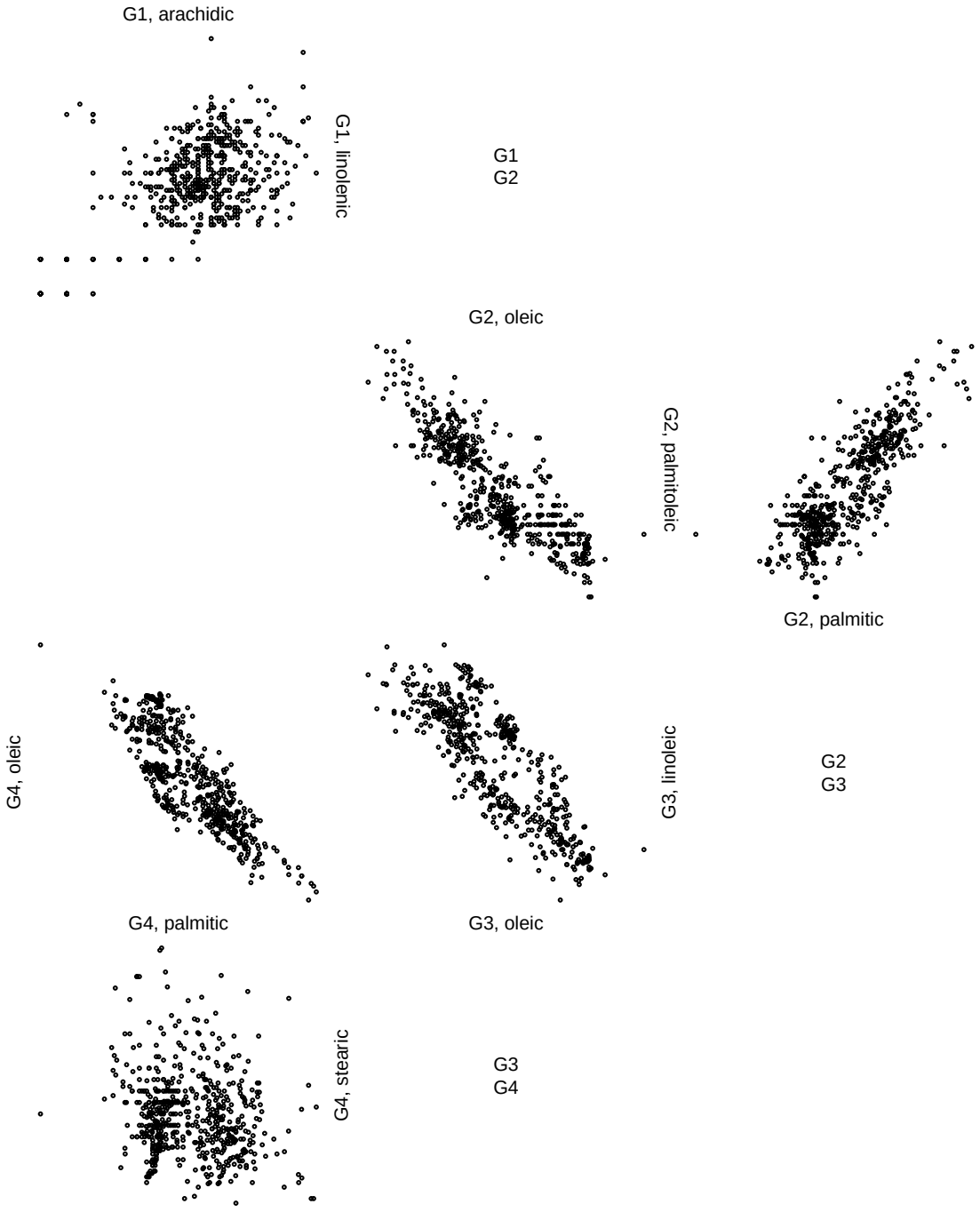

Figure 8: Zenplot showing the six pairs of acids which exhibit largest convexity; also note the U-turn at the end.

plot functions. These include the function `extract_2d()` seen earlier, its 1d counterpart `extract_1d()`, as well as all of the components which each function returns. Two important working functions are `burst()` and `unfold()`. The first, as its name suggests, takes any of the data structures acceptable to `zenplot()` and "bursts" this data into an easy to use common form. The second, `unfold()`, is the major function underpinning `zenplot()`. This name is intended to suggest its functionality which is to take a high-dimensional hypercube (all coordinate pairs of the high-dimensional space) and "unfold" it so that it may be laid flat in the two-dimensional plane. The unfolding determines the layout of the 2d plots. The algorithmic details of this layout (and other functions on which it depends) are described in Appendix A. In that appendix, the entire topdown structure of `zenplot()` and the functions on which it depends are given. The greatest detail, however, will be found in the source code itself.

5.1. The structure of a zenplot

As mentioned before, `zenplot()`, besides plotting, invisibly returns a list containing information about the zenpath and layout. For Figure 1, e.g., we obtain:

```
R> res <- zenplot(olive, plot1d = "layout", plot2d = "layout", draw = FALSE)
R> str(res)

List of 2
 $ path  :List of 3
  ..$ turns    : chr [1:19] "d" "r" "r" "d" ...
  ..$ positions: num [1:19, 1:2] 1 2 2 2 3 4 4 4 5 6 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "x" "y"
  ..$ occupancy: chr [1:8, 1:6] "" "" "" "" ...
 $ layout:List of 6
  ..$ orientations : chr [1:19] "h" "s" "v" "s" ...
  ..$ dimensions   : num [1:19] 1 2 1 2 1 2 1 2 1 2 ...
  ..$ vars         : num [1:19, 1:2] 1 1 2 3 3 3 4 5 5 5 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "x" "y"
  ..$ layoutWidth  : num 33
  ..$ layoutHeight : num 44
  ..$ boundingBoxes: num [1:19, 1:4] 0 0 10 11 11 11 21 22 22 22 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:4] "left" "right" "bottom" "top"
```

Besides the components already discussed before (such as `turns`, `orientations` and `vars`), the return object of `zenplot()`, a list of length two, also contains the plot dimensions (as part of the `layout` component). Furthermore, it contains `occupancy` and `positions`:

```
R> res[["path"]][["occupancy"]]

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] ""   "d"  ""   ""   ""   ""
```

```
[2,] ""   "r"  "r"  "d"  ""   ""
[3,] ""   ""   ""   "d"  ""   ""
[4,] ""   ""   ""   "r"  "r"  "d"
[5,] ""   ""   ""   ""   ""   "d"
[6,] "l"  "l"  ""   "d"  "l"  "l"
[7,] ""   "u"  ""   "d"  ""   ""
[8,] ""   "u"  "l"  "l"  ""   ""
```

```
R> head(res[["path"]][["positions"]])
```

```
     x y
[1,] 1 2
[2,] 2 2
[3,] 2 3
[4,] 2 4
[5,] 3 4
[6,] 4 4
```

The occupancy matrix (component `occupancy`) is a matrix which reflects the layout containing all 1d and 2d plots underlying a zenplot. Its elements are `""` (for positions not occupied by a 1d or 2d plot), `"l"`, `"r"`, `"d"` or `"u"` for left, right, downward or upward turns out of the current position in the occupancy matrix, respectively. For example, the entry $(2, 3)$ in the occupancy matrix is `"r"`, so the turn out of the position $(2, 3)$ is to the right. Note that **zenplots** also provides the auxiliary function `convert_occupancy()` which allows to convert the occupancy matrix to contain different characters for the five possible states (with a default resembling arrow heads to easily spot the zenpath).

The ith row in the 2-column matrix `positions` provides the position of the ith plot in the occupancy matrix. For example, the fifth row is $(3, 4)$, meaning that the fifth plot will be plotted in the position $(3, 4)$ of the layout corresponding to the occupancy matrix.

Finally, let us remark that for specific graphical systems, more information might be returned. For example, for `pkg = "grid"`, the whole plot as a grid object is also returned.

5.2. Tools for writing 1d and 2d plot functions

We now present selected tools which help writing user-specific 1d and 2d plots. We also go into some more detail concerning the main underlying paradigm behind zenplots.

Bursting the data and extracting the current plot variables

As mentioned before, the data argument `x` of `zenplot()` is typically a `vector`, `matrix`, `data.frame` or a `list` of such. The auxiliary function `burst()` obtains the arguments `x` and `labs` (a list of length four as described in Section 4.4) and returns a list with the following components:

`xcols`: A list containing the columns of `x` (so `x` "burst" into its columns). These columns are named and the names contain, by default, the combined group and variable labels.

`groups`: The (group) number the respective column belongs to; all equal to 1 if `x` does not consist of different groups.

`vars`: The (variable) number the respective column belongs to within its group.

`glabs`: The group labels (`NULL` if `x` does not consist of different groups).

`vlabs`: The variable labels.

See `?burst` for various examples showing how `burst()` works.

The functions `extract_1d()` and `extract_2d()` obtain the single argument `zargs` and extract (hence the name) useful information for most 1d and 2d plots, respectively. These are widely used in the exported `*_1d_*()` and `*_2d_*()` functions, respectively, and make use of `burst()`. In particular, `extract_1d()` checks whether `zargs` contains `x`, `orientations`, `vars`, `num`, `lim` and `labs` (all discussed before) and returns a list with components:

`x`: The (named) data column to be plotted in the current 1d plot.

`xcols`, `groups`, `vars`, `glabs`, `vlabs`: These are the components as returned by `burst()`.

`horizontal`: A logical indicating whether the current 1d plot is horizontal or not (so vertical).

`xlim`: The axis limits for the current 1d plot.

Similarly, `extract_2d()` checks whether `zargs` contains `x`, `vars`, `num`, `lim` and `labs` and returns:

`x`: The (named) data column to be plotted as x variable in the current 2d plot.

`y`: The (named) data column to be plotted as y variable in the current 2d plot.

`xcols`, `groups`, `vars`, `glabs`, `vlabs`: These are the components as returned by `burst()`.

`xlim`: The x axis limits for the current 2d plot.

`ylim`: The y axis limits for the current 2d plot.

`same.group`: A logical indicating whether the two data columns `x` and `y` belong to the same group or not.

Simply passing `zargs` to the 1d and 2d plot functions `*_1d_*()` and `*_2d_*()` is very convenient. It also allows one to easily write new 1d and 2d plot functions without the need of having to deal with a lot of different arguments; we have already seen the short function `function(zargs) label_1d_graphics(zargs, cex = 0.8)` as `plot1d` in the code for producing the right-hand side of Figure 6. It is thus natural to have the extractor functions `extract_1d()` and `extract_2d()` available to extract the actual information to be used for plotting by most 1d and 2d plot functions. For example, here is the definition of `points_2d_graphics()`:

```
R> points_2d_graphics
```

```
function (zargs, cex = 0.4, box = FALSE, add = FALSE, group... = NULL,
    plot... = NULL, ...)
{
    r <- extract_2d(zargs)
    xlim <- r$xlim
    ylim <- r$ylim
    x <- as.matrix(r$x)
    y <- as.matrix(r$y)
    same.group <- r$same.group
```

```
    if (same.group) {
        if (!add)
            plot_region(xlim = xlim, ylim = ylim, plot... = plot...)
        points(x = x, y = y, cex = cex, ...)
        if (box)
            box(...)
    }
    else {
        args <- c(list(zargs = zargs, add = add), group...)
        do.call(group_2d_graphics, args)
    }
}
<bytecode: 0x7f93230634c8>
<environment: namespace:zenplots>
```

As we can see, `extract_2d()` is used to extract all the plotting information used for the current 2d plot. The component `same.group` is then used to determine whether a normal 2d scatter plot is produced via `points()` or whether a group-specific 2d plot is produced (via `group_2d_graphics()`) to indicate the change of groups.

Bursting `x` in every 1d and 2d plot involved in a zenplot is quite time-consuming and becomes prohibitive in larger zenplots. The extractor functions `extract_1d()` and `extract_2d()` nicely address this problem by setting up an environment containing the burst `x` the first time this needs to be done (which is by the first call of `extract_1d()` if `first1d = TRUE` and by the first call of `extract_2d()` if `first1d = FALSE`). Subsequent calls of `extract_1d()` or `extract_2d()` can simply perform a look-up.

Setting up the plot region and determining the plot indices

When writing 1d or 2d plot functions, one frequently needs to set up the plot region (of the 1d or 2d plots). To this end, **zenplots** provides the function `plot_region()` in case the graphical system is **graphics**; see the exported functions `*_*_graphics()` such as the above `points_2d_graphics()` where `plot_region()` is used.

```
R> plot_region


function (xlim, ylim, plot... = NULL)
{
    if (is.null(plot...)) {
        plot(NA, xlim = xlim, ylim = ylim, type = "n", ann = FALSE,
            axes = FALSE, log = "")
    }
    else {
        fun <- function(...) plot(NA, xlim = xlim, ylim = ylim,
            ...)
        do.call(fun, plot...)
    }
}
<bytecode: 0x7f93231292c0>
<environment: namespace:zenplots>
```

Similarly for **grid**, where one often needs an auxiliary function for constructing the correct viewport, **zenplots** provides `vport()`; see the exported functions `*_*_grid()` for how to use it.

A short but useful helper function is `plot_indices()`, which allows one to determine, based on `zargs`, the indices of the two variables to be plotted in the current 1d or 2d plot (as mentioned before, for 1d plots, the two are the same); see `?zenplot()` for how to use `plot_indices()` in constructing 1d and 2d plots.

```
R> plot_indices

function (zargs)
zargs$vars[zargs$num, ]
<bytecode: 0x7f93263d4cb0>
<environment: namespace:zenplots>
```

Unfolding the cube

The major helper function which `zenplot()` calls is `unfold()`. Although `unfold()` is probably rather rarely used directly by a user, it provides insight into how zenplots are constructed.

Suppose, for example, we have only $d = 3$ variates, and hence dimensions. Then each 2d plot can be thought of as defining a face of a three dimensional cube as shown in Figure 9. To

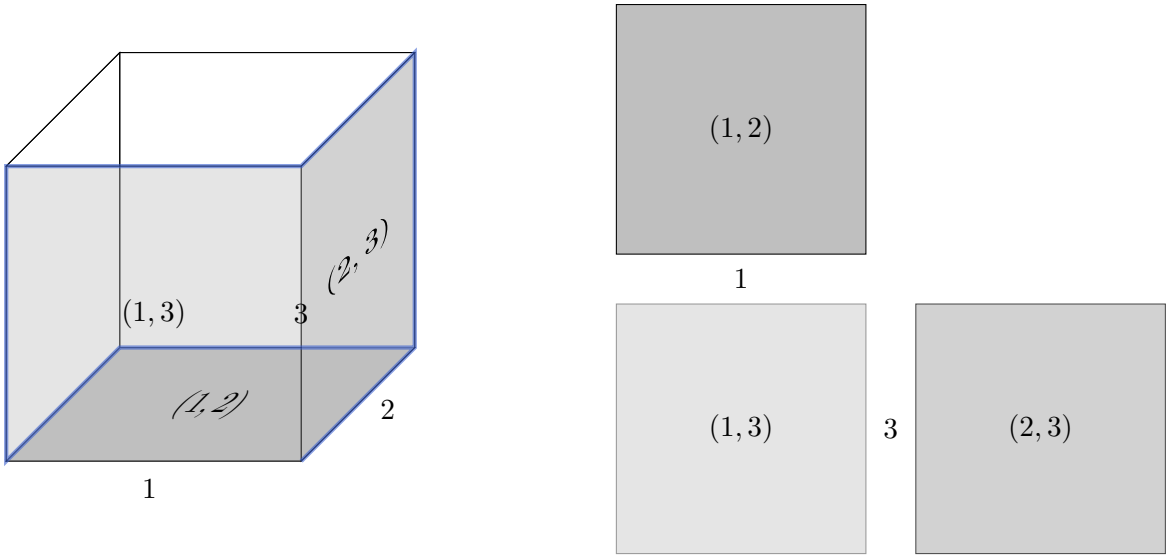

Figure 9: Three-dimensional (folded) hypercube with faces $(1, 2)$, $(1, 3)$ and $(2, 3)$ (left) and unfolded faces after cutting along the blue line (right).

arrive at all two-dimensional plots, we imagine beginning with the bottom most face of the cube, '(1,2)', and placing it in the display at the right of Figure 9. Now roll the cube forward about the edge marked '1', so that its '(1,3)' face is down; this face now appears in the right hand display below '(1, 2)' with the connecting edge '1' between them. With the cube face '(1,3)' now at the bottom, the cube is finally rolled left to right over the '3' edge to land on the '(2, 3)' face; the '3' edge and the '(2, 3)' face now appear in the display at right, reflecting

that movement. It is in this sense that we imagine we are "unfolding the cube". In principle this could continue until all six faces appeared (though three would be redundant). Instead we have unfolded only `nfaces = 3` of the faces from the cube.

With some abuse of language, when the number of dimensions is $d > 3$ we still call the process unfolding (of the "cube") though really the rolling is now from one two-dimensional space to another about a shared axis (variate). It is this common axis that suggests continuing to use the "unfolding cube" metaphor. With d dimensions there are $\binom{d}{2}$ distinct pairs of d plot variables and hence distinct two-dimensional spaces that might be visited, as ordered by the unfolding about shared axes.

The `unfold()` function takes, as first argument, `nfaces`, the number of 2d plots/spaces to be "unfolded" and produces the zenpath and zenplot layout required for the function `zenplot()`. Laying out these pairs with a zenplot means "unfolding" (at least a part of) the d-dimensional space. This is what `unfold()` does. It first constructs the zenpath (either based on given turns or based on the given method for constructing turns) with the (non-exported) helper function `get_path()` and then, given the zenpath, determines the layout with the (non-exported) helper function `get_layout()`. Both of these functions are described in more detail in the appendix.

Note that the (invisible) return value of `zenplot()` actually equals the return value of `unfold()`, which we can easily check:

```
R> n2dcols <- ncol(olive) - 1
R> stopifnot(identical(res, unfold(nfaces = n2dcols)))
```

6. Conclusion

We introduced the R package **zenplots** for visualization of high- (but also low-)dimensional data with the notion of zenplots and zenpaths. A zenpath is an order of "interestingness" according to which the data can be sorted and then laid out and plotted with a zenplot. The package's main functions `zenpath()` and `zenplot()` achieve this task as presented in Sections 2 and 3, respectively.

Zenplots can be thought of as a variation on scatterplot matrices, one which sacrifices the (row, column) array look up and the row-wise or column-wise comparisons of a scatterplot matrix in exchange for having an arrangement of more and/or larger two-dimensional displays. With zenplots, comparisons along a common aligned scale are now restricted to a single pair of plots at a time. More importantly, however, unlike scatterplot matrices, zenplots directly address the problem of presenting possibly hundreds of plots, pre-selected for their importance or interesting structure, in a single layout. By accomodating up to a thousand or two plots in a single display, zenplots enable the analyst to consider the most interesting two-dimensional plots even from data having hundreds of dimensions (see Hofert and Oldford (2017) for an example). This means only those two-dimensional plots of greatest interest need be considered by the analyst for further study, and these can be served up automatically.

With zenpaths, measures of interest, and groupings of variables, the R package **zenplots** adds a powerful new tool to the data analyst's arsenal when data dimensionality becomes high. As future work, we identify the need for developing and exporting more 1d and 2d plots specialized for particular analytic purposes such as dependence analysis for mixed categorical

and continuous variates, or model selection and analysis in regression. More generally, an adaptation of zenplots to handle conditioning (analogous to but substantively different from coplots Cleveland (1994)) could provide new tools for multivariate data including multivariate time series.

A. Algorithms

In a "top-down" (top layer to bottom layer) manner, we now describe in terms of pseudo-algorithms how zenplots and zenpaths are constructed "under the hood". The presented algorithms only capture the main purpose, essential inputs (important for the algorithm overall, not necessarily required to be provided), optional inputs, the return value and the major steps; for more details (including more or more general features than captured here), consider the actual source code of **zenplots**. The reason these pieces of information are presented is to convey the intellectual challenge and contribution in constructing zenplots and zenpaths. The algorithms refer to functions which are used by **zenplots** (not all exported and thus visible to the user) but assume a more compact, mathematical notation.

The following algorithm describes the main structure of `zenplot()` as already discussed before. It mainly calls `unfold()` to determine the zenpath in the form of a list containing the vector of turns (denoted by t), a 2-column matrix of positions in the occupancy matrix (denoted by p) and the occupancy matrix itself (denoted by O). The function `unfold()` also determines the corresponding layout of the zenplot.

Algorithm A.1 (Main plot function; `zenplot()`)

Purpose: Compute the zenplot and, invisibly, return the path and layout.

Essential: An (n, d)-matrix or data frame (or list of such) X containing the n d-dimensional data vectors, the turns t (if not provided, they will be computed from the given number $n_{2d} \geq 2$ of columns of 2d plots and the method ("tidy" (the default), "double zigzag", "single zigzag" or "rectangular") used for constructing the path), functions for creating the 1d and 2d plots, and an argument indicating the graphical system used for plotting.

Optional: Logicals indicating whether the first or last 1d plot are to be plotted, the number of 2d plots along the zenpath, a vector of logicals indicating which pieces of information are passed to the 1d and 2d plot functions, how group and variable labels are determined, parameters for adjusting spaces in a zenplot and further parameters such as an ellipsis argument passed to the underlying 1d and 2d plot functions.

Return: Nothing (R's `invisible()`) unless assigned to a variable in which case a list containing the path and layout is returned.

Major steps:

1) Call `unfold()` to compute the path and corresponding layout; see Algorithm A.2.
2) Determine the argument lists passed to the 1d and 2d plot functions, respectively.
3) Setting up the overall layout.
4) Loop over all plots and call the 1d and 2d plot functions with the respective arguments.

The following algorithm unfolds the hypercube along a (provided or determined) set of edges and returns the corresponding zenpath and a layout of 1d and 2d plots along it.

Algorithm A.2 (Unfolding the hypercube; `unfold()`)

Purpose: Computes the path and corresponding layout by unfolding a cube (the sides of which are interpreted as the pairs of variables to be plotted in the 2d plots of a zenplot).

Essential: The number of faces of the hypercube (which equals the number $n_{2\text{dp}}$ of 2d plots in a zenplot, so typically the number of columns d of X minus one), the number $n_{2\text{dc}}$ of columns containing 2d plots and the method (see Algorithm A.1).

Optional: The turns t, logicals indicating whether the first or last 1d plot are to be plotted and widths of 1d and 2d plots in the layout.

Return: Returns the path (t, p, O) and corresponding layout (a list with orientations, plot dimensions, plot variables (2-column matrix), layout width and height and bounding boxes); see the output for Figure 1 we discussed earlier in the paper.

Major steps:

1) Construct the path via `get_path()`; see Algorithm A.3.

2) Determine the layout (corresponding to the turns) via `get_layout()`; see Algorithm A.6.

We now turn to determining the path. This is done via `get_path()` (Algorithm A.3) which is an auxiliary function of `unfold()` (Algorithm A.2) and which itself has two more auxiliary functions discussed below; these are `get_zigzag_turns()` (Algorithm A.4) and `next_move_tidy()` (Algorithm A.5).

Algorithm A.3 (Determining the path; `get_path()`)

Purpose: Determine the path (turns t, positions p and occupancy matrix O).

Essential: The turns t, the number $n_{2\text{dc}}$ of columns containing 2d plots, the number $n_{2\text{dp}}$ of 2d plots and the method (see Algorithm A.1).

Optional: Logicals indicating whether the first or last 1d plot are to be plotted.

Returns: The path in the form of a list.

Major steps:

1) If the turns t are provided, say, n_{t}-many, proceed as follows.

   1.1) Initialize the $(n_{\text{t}}, 2)$-matrix p of positions with zeros, set the current location p_{cur} to $(0, 0)$ and the horizontal and vertical limits of the number of plots to $(0, 0)$ each; the latter determine the extend of the occupancy matrix in horizontal and vertical direction.

   1.2) If $n_{\text{t}} > 1$, loop over $i \in \{2, \ldots, n_{\text{t}}\}$, set the current position p_{cur} to the position obtained by moving in the direction of turn $i - 1$ (the turn into the current position) and set the ith row of p to p_{cur}. Then check whether the horizontal or vertical limits have to be extended and extend them if necessary.

   1.3) To obtain the final matrix of positions, "shift" all coordinates appropriately (by subtracting the minimum of each column of p from the respective column and adding 1 to each value).

1.4) Initialize the occupancy matrix O with zeros (no position occupied); note that the maximum of the first and second column of p determine the dimensions of O. Then loop over all rows of p and set the respective entry in O to 1, 2, 3 or 4, depending on whether the turn out of the current position p_{cur} is "l", "r", "d" or "u", respectively.

2) If the turns t are not provided, proceed as follows. Let n_{plots} denote the number of all plots (1d or 2d).

2.1) For method "rectangular", proceed as follows.

- Set $n_{\text{plots}} = 2n_{\text{2dp}} + 1 - (1 - \mathbb{1}_{\text{first1d}}) - (1 - \mathbb{1}_{\text{last1d}})$, where $\mathbb{1}_{\text{first1d}}$ ($\mathbb{1}_{\text{last1d}}$) is the indicator of the event that the first (last) 1d plot is to be displayed.
- Set the number n_{2dr} of rows with 2d plots to $n_{\text{2dr}} = \lceil n_{\text{2dp}}/n_{\text{2dc}} \rceil$.
- Determine the turns to fill the rectangular layout (determined by n_{2dr} and n_{2dc}), remove the first turn or append the last (depending on whether the first and last 1d are to be displayed) and extract the n_{plots}-many turns required for the zenplot.
- Loop over the turns to determine the positions p and from there the occupancy matrix O.
- Return the path (list of turns t, positions p and the occupancy matrix O).

2.2) For all other methods, set $n_{\text{plots}} = 2n_{\text{2dp}} + 1$ and proceed as follows.

- If $n_{\text{plots}} = 1$, set turns $t = (\text{"d"})$, positions $p = (1, 1)$, occupancy matrix $O = (3)$.
- If $n_{\text{plots}} = 2$, set turns $t = (\text{"d"}, \text{"r"})$, positions $p = \begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix}$, occupancy matrix $O = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$.
- If $n_{\text{plots}} = 3$, set turns $t = (\text{"d"}, \text{"r"}, \text{"r"})$,

$$\text{positions } p = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 2 & 2 \end{pmatrix}, \quad \text{occupancy matrix } O = \begin{pmatrix} 3 & 0 \\ 2 & 2 \end{pmatrix}.$$

- If $n_{\text{plots}} \geq 4$, proceed as follows.

2.2.1) For method "double zigzag" or "single zigzag", the idea is to build all turns right away (the positions and the occupancy matrix can the be determined from the turns):

- Compute the turns via `get_zigzag_turns()`; see Algorithm A.4.
- Initialize the occupancy matrix O as an $(1, 2n_{2d} - 1)$-matrix containing zeros and the positions p as an $(n_{\text{plots}}, 2)$-matrix containing zeros.
- Encode the turns "l", "r", "d", "u" as the numeric values 1–4, respectively.
- Set $p_{\text{cur}} = (1, 1)$ and the first row of p to p_{cur}. Furthermore, set O at p_{cur} to the numeric value of the first turn.
- Loop over $i \in \{2, \ldots, n_{\text{plots}}\}$, set the next position p_{next} to the value obtained by moving from p_{cur} in the direction of turn $i-1$ and set the ith row of p to p_{cur}. Then update p_{cur} to p_{next}. If p_{cur}'s first entry exceeds the maximal row number of O append another row of zeros to O. Then set O at p_{next} to the numeric value of the ith turn.
- Trim off the last columns of zeros (if any) from O.

2.2.2) For method "tidy":

- Initialize the n_{plots}-vector turns t, and the $(1, 2n_{2d}+1)$ occupancy matrix O with zeros and the positions p as an $(n_{\text{plots}}, 2)$-matrix containing zeros. Furthermore, set the first turn to d, the first row of p to $(1, 2)$ and the occupancy matrix O at $(1, 2)$ to 3 (numeric value corresponding to a turn "d").
- Loop over $i \in \{2, \ldots, n_{\text{plots}}\}$, determine the next position p_{next} and the turn t_{next} out of p_{next} via `next_move_tidy()`, see Algorithm A.5, set the ith turn to t_{next} and the ith row of p to p_{next}. If the first entry of p_{next} exceeds the maximal row number of O append another row to O. Then set O at p_{next} to the numeric value of the ith turn.
- If it exists, trim off the last column of zeros from O.
- If it exists, trim off the first column of zeros from O and subtract 1 from the second column of p (as all positions relative the occupancy matrix changed).

- Depending on whether the first and last 1d plot are to be displayed, trim the path accordingly.
- Return the path (list of turns t, positions p and the occupancy matrix O).

Algorithm A.4 (Compute the turns for zenplots with method "double zigzag" or "single zigzag"; `get_zigzag_turns()`)

Purpose: Compute the vector of turns for zenplots with method "double zigzag" or "single zigzag".

Essential: The number of plots n_{plots}, the number $n_{2d} \geq 2$ of columns of 2d plots, the method used for constructing the path ("double zigzag" or "single zigzag").

Returns: The turns t.

Major steps:

1) Define the horizontal 2d pattern ("r", ..., "r", "l" ..., "l") (each of "r", "l" appears $2(n_{2d}-1)$ times). So far this only contains horizontal movements, vertical movements will be added in Step 4) below.

2) Define the vertical 2d subpattern corresponding to the first part ("r", ..., "r") of the horizontal 2d pattern in the following way:

   - For method "single zigzag", the vertical 2d subpattern is ("d", ..., "d") (of length $n_{2d} - 1$).
   - For method "double zigzag", the vertical 2d subpattern can be constructed as follows: If $n_{2d} = 2$, it is ("d"). If $n_{2d} > 2$, take $n_{2d} - 3$ elements from the alternating sequence "d", "u", "d", "u", ... and append two "d"'s (so, ("d", "d") for $n_{2d} = 3$, ("d", "d", "d") for $n_{2d} = 4$, ("d", "u", "d", "d") for $n_{2d} = 5$, etc.).

3) Define the vertical 2d pattern corresponding to the horizontal 2d pattern from Step 1) by repeating the vertical 2d subpattern from Step 2) twice, i.e., (vertical 2d subpattern, vertical 2d subpattern).

4) Set components of the horizontal 2d pattern from Step 1) with even indices to the values of the vertical 2d pattern from Step 3). With this merge, we obtain the 2d pattern which repeats itself until we run out of plots.

5) Repeat each element of this 2d pattern twice to obtain the overall pattern; this is to account for 1d plots, so the pattern now contains turns for both 1d plots and 2d plots.

6) The vector of turns can now be obtained by taking "d" as first component (since the first turn is always "d") and then taking the first n_{plots} elements of an imaginary sequence consisting of the overall pattern repeated after each other; recall that n_{plots} denotes the total number of 1d and 2d plots.

Algorithm A.5 (Determine the next position and turn out of there; `next_move_tidy()`)

Purpose: Determine, for the method "tidy", the next position p_{next} to move to and the turn t_{next} out of this next position.

Essential: The current plot number n_{p}, the total number of plots n_{plots} and the current path as has been determined up to and including n_{p}. Note that this function assumes that the zenplot starts with a 1d plot; `get_path()` trims off the first 1d plot if necessary.

Returns: A list containing the next position p_{next} (i.e., the position of plot numbered $n_{\text{p}}+1$) and the turn t_{next} out of this next position.

Major steps:

1) Let $n_{\text{left}} = n_{\text{plots}} - n_{\text{p}}$ denote the number of remaining plots.
   - If $n_{\text{p}} = 1$, return $p_{\text{next}} = (2, 2)$ and $t_{\text{next}} =$ "r".
   - If $n_{\text{p}} = 2$, return $p_{\text{next}} = (2, 3)$ and $t_{\text{next}} =$ "r".
   - If $n_{\text{p}} = 3$, set $t_{\text{next}} = u$ if $n_{\text{left}} \leq 2$ and set $t_{\text{next}} =$ "d" otherwise. Return $p_{\text{next}} = (2, 4)$ and t_{next}.

2) Determine the current position p_{cur} (corresponding to row n_{p} in the positions matrix p), the turn t_{in} into p_{cur} (element $(n_{\text{p}} - 1)$ in the vector of turns t), the turn t_{out} out of p_{cur} (element n_{p} of t) and the next position p_{next} when moving from p_{cur} in the direction given by t_{out}.

3) If n_{p} is even (i.e., the current plot is a 2d plot according to the above assumption), return p_{next} and t_{out}.

4) Now the current plot numbered n_{p} is a 1d plot. We proceed by determining the horizontal moving direction out of this plot. To this end, consider the turns at $n_{\text{p}} - 2$ and $n_{\text{p}} - 1$. If "r" is among any of these two turns, the horizontal moving direction is "r", otherwise it is "l". Furthermore, determine the distance to the margin of the occupancy matrix O in the horizontal moving direction; if the latter is "r", the distance is the number of columns in O minus the second component of p_{cur} (distance to the right end of O); otherwise it is the second component of p_{cur} minus 1 (distance to the left end of O).

5) We now determine the turn t_{next} out of p_{next}. To this end, we distinguish two cases:

   5.1) If t_{out} is "d" or "u" (i.e., the current 1d plot is horizontal), proceed as follows.

      5.1.1) If $n_{\text{left}} \leq 2$, check the location of the 2d plot which comes after the next 2d plot in opposite horizontal moving direction.

         - If this location does not exist in O (which can only happen if $t_{\text{out}} =$ "d" in which case O is missing a new row), proceed as follows. Put the last 1d plot in opposite horizontal moving direction if we are near the margin

(i.e., set t_{next} = "l" if the horizontal moving direction is "r" and t_{next} = "r" otherwise; we follow this strategy here since we occupy an additional column otherwise). Otherwise, put it in horizontal moving direction (i.e., set t_{next} to the horizontal moving direction; this can be done since we are away from the margin, i.e., "inside" O).

- If this location exists, check the occupancy matrix at this location to see whether it is occupied. If it is not occupied, set t_{next} to the opposite horizontal moving direction ("l" if the horizontal moving direction is "r" and vice versa). Otherwise, set t_{next} to the current horizontal moving direction.

5.1.2) If $n_{\text{left}} \geq 3$ (so there are at least two more 2d plots left), change the horizontal moving direction if and only if the distance to the margin is less than or equal to two.

5.2) If t_{out} is "l" or "r" (i.e., the current 1d plot is vertical and the distance to the margin is at least two in this case), proceed as follows. Ideally, we would like to go up (in order to save space in the plotting region), but if there are too many plots left, we can only go up if we have the space to go down again. To explain this part of the algorithm, we utilize the concept of a U-turn (going up and back down again).

5.2.1) Given p_{cur}, the horizontal moving direction and the occupancy matrix O so far, determine the number of plots which fit in the U-turn starting from p_{cur}, i.e., the length $l_{\text{U-turn}}$ of the U-turn. This can be done as follows.

- Let p_{check} denote the position obtained from p_{cur} by going up two rows and then one column in the horizontal moving direction. If p_{check} exists in the occupancy matrix O and is occupied, return $l_{\text{U-turn}} = 1$ (as only one additional plot fits in along this U-turn). If it does not exist in O, return $l_{\text{U-turn}} = 2$.
- Update p_{check} by going two columns further in the horizontal moving direction. If p_{check} does not exist in O or if it exists but is occupied, return $l_{\text{U-turn}} = 4$.
- Update p_{check} by going two rows down. If p_{check} does not exist in O or if it exists but is occupied, return $l_{\text{U-turn}} = 6$.
- Update p_{check} by going two columns further in the horizontal moving direction. If p_{check} does not exist in O or if it exists but is occupied, return $l_{\text{U-turn}} = 8$. Otherwise return $l_{\text{U-turn}} = 10$ (which means that at least 10 plots fit in the U-turn and we can do a complete U-turn by going up and back down again).

5.2.2) If the first component of the current position p_{cur} is less than or equal to two (i.e., we are in the second row of O and thus cannot go up) or if $l_{\text{U-turn}} \leq 9$ (i.e., at most 9 plots fit in the U-turn starting from p_{cur}) but $n_{\text{left}} > l_{\text{U-turn}}$ (i.e., there are more plots left than fit in the U-turn), set t_{next} = "d". Otherwise, set t_{next} = "u".

6) Return p_{next} and t_{next}.

Finally, we describe how the layout can be constructed.

Algorithm A.6 (Determining the layout; `get_layout()`)

Purpose: Determine the layout (1d and 2d plot orientations, plot dimensions, plot variables, total width and height of the layout and bounding boxes).

Essential: The turns t and number n_{2dp} of 2d plots.

Optional: Logicals indicating whether the first or last 1d plot are to be plotted and widths of 1d and 2d plots in the layout.

Returns: The layout (a list with orientations of 1d and 2d plots, their dimensions (either 1d or 2d), which variables are plotted in each 1d and 2d plot, layout widths and heights, and bounding boxes; see Section 5.1).

Major steps:

1) Set up an empty 4-column matrix containing the left, right, bottom and top coordinates (in units of the widths of 1d and 2d plots) of the bounding box of each plot. Furthermore, set up a 2-column matrix containing the variables of each 1d or 2d plot.

2) Loop over all 1d and 2d plots and determine, for each, its orientation, plot variables involved and coordinates of the bounding box. In particular, this loop requires as input the number of plots, the current and the last plot variable. Some more details about this loop:

   - Determining the current plot variables: For 1d plots, this is the current plot variable repeated twice. For 2d plots which turn left or right out of the current position in O, this is the last and current plot variable; for 2d plots turning down or up out of the current position in O, this is the current and the last plot variable (notice the difference in order).
   - Determining the current bounding box: This is a function of the last turn (i.e., the turn into the current plot position), the last bounding box and the width and height of the bounding box. The latter two are easy to determine based on the provided widths of the 1d and 2d plots. And the location of the bounding box for the current plot equals the bounding box of the previous plot shifted in the direction of the last turn (so the turn into the current plot position).

References

Asimov D (1985). "The Grand Tour: A Tool for Viewing Multidimensional Data." *SIAM Journal on Scientific and Statistical Computing*, **6**(1), 128–143.

Azzalini A, Torelli N (2007). "Clustering via Nonparametric Density Estimation." *Statistics and Computing*, **17**, 71–80.

Ball GH, Hall DJ (1970). "Some Implications of Interactive Graphic Computer Systems for Data Analysis and Statistics." *Technometrics*, **12**(1), 17–31.

Buja A, Hurley CB, McDonald JA (1986). "A Data Viewer for Multivariate Data." *Computing Science and Statistics*, **18**, 171–174.

Cleveland WS (1994). "Coplots, Nonparametric Regression, and Conditionally Parametric Fits." *Lecture Notes-Monograph Series*, **24**, 21–36. ISSN 07492170. URL http://www.jstor.org/stable/4355791.

Emerson JW, Green WA, Schloerke B, Crowley J, Cook D, Hofmann H, Wickham H (2013). "The Generalized Pairs Plot." *Journal of Computational and Graphical Statistics*, **22**(1), 79–91. doi:10.1080/10618600.2012.694762. URL http://dx.doi.org/10.1080/10618600.2012.694762.

Forina M, Armanino C, Lanteri S, Tiscornia E (1983). "Classification of Olive Oils from Their Fatty Acid Composition." In H Martens, HJ Russwurm (eds.), *Food Research and Data Analysis*, pp. 189–214. Applied Science Publishers.

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

Gentleman R, Whalen E, Huber W, Falcon S (2019). ***graph**: A Package to Handle Graph Data Structures*. URL http://www.bioconductor.org/packages/release/bioc/html/graph.html.

Hartigan JA (1975). "Printer Graphics for Clustering." *Journal of Statistical Computation and Simulation*, **4**(3), 187–213.

Hofert M, Mächler M (2014). "A Graphical Goodness-of-Fit Test for Dependence Models in Higher Dimensions." *Journal of Computational and Graphical Statistics*, **23**(3), 700–716. doi:10.1080/10618600.2013.812518.

Hofert M, Oldford RW (2017). "Visualizing Dependence in High-dimensional Data: An Application to S&P 500 Constituent Data." *Econometrics and Statistics*. doi:10.1016/j.ecosta.2017.03.007.

Huang B, Cook D, Wickham H (2012). "tourrGui: A gWidgets GUI for the Tour to Explore High-Dimensional Data Using Low-Dimensional Projections." *Journal of Statistical Software*, **49**(6), 1–12.

Hurley CB, Oldford RW (2010). "Pairwise Display of High-dimensional Information via Eulerian Tours and Hamiltonian Decompositions." *Journal of Computational and Graphical Statistics*, **19**(4), 861–886.

Hurley CB, Oldford RW (2011a). "Eulerian Tour Algorithms for Data Visualization and the PairViz Package." *Computational Statistics*, **26**(4), 613–633.

Hurley CB, Oldford RW (2011b). "Graphs as Navigational Infrastructure for High Dimensional Data Spaces." *Computational Statistics*, **26**(4), 585–612.

Im JF, McGuffin MJ, Leung R (2013). "GPLOM: The Generalized Plot Matrix for Visualizing Multidimensional Multivariate Data." *IEEE Transactions on Visualization and Computer Graphics*, **19**(12), 2606–2614.

Murrell P (2016). *R Graphics*. CRC Press.

Oldford RW, Waddell A (2011). "Visual Clustering of High-dimensional Data by Navigating Low-dimensional Spaces." STS057, pp. 3294–3303. International Statistical Institute, The Hague, The Netherlands. URL http://2011.isiproceedings.org/papers/650370.pdf.

R Core Team (2017a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

R Core Team (2017b). *The R* ***graphics*** *Package*. R Foundation for Statistical Computing, Vienna, Austria.

Ripley BD (2017). ***MASS****: Support Functions and Datasets for Venables and Ripley's MASS*. URL https://CRAN.R-project.org/package=MASS.

Temple Lang D, Swayne D, Wickham H, Lawrence M (2018). ***rggobi****: Interface Between R and 'GGobi'*. URL https://cran.r-project.org/web/packages/rggobi/index.html.

Tukey JW, Tukey PA (1983). "Some Graphics for Studying Four-Dimensional Data." In KW Heiner, RS Sacher, JW Wilkinson (eds.), *Computer Science and Statistics: Proceedings of the 14th Symposium on the Interface*, pp. 60–66. Springer-Verlag, New York, NY, USA.

Tukey JW, Tukey PA (1985). "Computer Graphics and Exploratory Data Analysis: An Introduction." In *Proceedings of the Sixth Annual Conference adn Exposition*, volume III. National Computer Graphics Association.

Tukey PA, Tukey JW (1981a). "Graphical Display of Data Sets in 3 or more Dimensions – Data-Driven View Selection; Agglomeration and Sharpening." In V Barnett (ed.), *Interpreting Multivariate Data*, pp. 215–243. John Wiley & Sons, Chichester.

Tukey PA, Tukey JW (1981b). "Graphical Display of Data Sets in 3 or more Dimensions – Preparation; Prechosen Sequences of Views." In V Barnett (ed.), *Interpreting Multivariate Data*, pp. 189–213. John Wiley & Sons, Chichester.

Waddell A, Oldford RW (2011). "**RnavGraph**: A Visualization Tool for Navigating Through High-dimensional Data." IPS117, pp. 1852–1860. International Statistical Institute, The Hague, The Netherlands. URL http://2011.isiproceedings.org/papers/450430.pdf.

Waddell A, Oldford RW (2014). ***RnavGraph****: Using Graphs as a Navigational Infrastructure*. URL http://www.navgraph.com.

Waddell A, Oldford RW (2017). ***loon****: Interactive Statistical Data Visualization*. URL https://CRAN.R-project.org/package=loon.

Wickham H (2016). ***ggplot2****: Elegant Graphics for Data Analysis*. Springer-Verlag.

Wickham H, Chang W (2016). ***ggplot2****: Create Elegant Data Visualisations Using the Grammar of Graphics*. URL https://CRAN.R-project.org/package=ggplot2.

Wilkinson L, Anand A, Grossman RL (2005). "Graph-Theoretic Scagnostics." In *INFOVIS*, volume 5, pp. 157–164.

Affiliation:

Marius Hofert
Department of Statistics and Actuarial Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
E-mail: marius.hofert@uwaterloo.ca
URL: http://www.math.uwaterloo.ca/~mhofert/

Wayne Oldford
Department of Statistics and Actuarial Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
E-mail: rwoldford@uwaterloo.ca
URL: http://www.math.uwaterloo.ca/~rwoldfor/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd