DEPRECATED: Commented source code for regular and annotated heatmaps in package `Heatplus`

Alexander Ploner
Medical Epidemiology & Biostatistics
Karolinska Institutet, Stockholm
email: `alexander.ploner@ki.se`

October 26, 2021

Abstract

This is an older, commented version of the R source for providing annotated and regular heatmaps. The package source code used to be a tangled version of this file, but this is no longer the case.

This vignette is DEPRECATED.

Contents

1 Helper functions

1.1 Defining the plot layout

We start with an empty maximal layout (i.e. space for the central image, both row and column dendrograms and annotations, and legends on all four sides). We then fill in the numbers of the plots in a given order, as well as setting the corresponding widths and heights. At the end, we remove all empty plotting slots, compressing the maximal layout to the actual layout.

```
> heatmapLayout = function(dendrogram, annotation, leg.side=NULL, show=FALSE)
+ {
+     ## Start: maximum matrix, 5 x 5, all zero
+     ## Names for nice display post ante
+     ll = matrix(0, nrow=5, ncol=5)
+     ll.width = ll.height = rep(0, 5)
+     cnt = 1
+     rownames(ll) = c("leg3", "colDendro","image", "colAnn", "leg1")
+     colnames(ll) = c("leg2", "rowDendro","image", "rowAnn", "leg4")
+
+     ## The main plot
+     ll[3,3] = cnt
+     ll.width[3] = ll.height[3] = 5
+     cnt = cnt+1
+     ## The column dendrogram
+     if (dendrogram$Col$status=="yes") {
+         ll[2, 3] = 2
+         ll.width[3]  = 5
+         ll.height[2] = 2
+         cnt = cnt+1
+     }
+     ## The row dendrogram
+     if (dendrogram$Row$status=="yes") {
+         ll[3, 2] = cnt
+         ll.width[2]  = 2
+         ll.height[3] = 5
+         cnt = cnt+1
+     }
```

```
+     # Column annotation
+     if (!is.null(annotation$Col$data)) {
+         ll[4, 3] = cnt
+         ll.width[3] = 5
+         ll.height[4] = 2
+         cnt = cnt+1
+     }
+     ## Row annotation
+     if (!is.null(annotation$Row$data)) {
+         ll[3, 4] = cnt
+         ll.width[4]  = 2
+         ll.height[3] = 5
+         cnt = cnt+1
+     }
+     ## Legend: if no pref specified, go for empty, if possible
+     if (is.null(leg.side)) {
+         if (dendrogram$Row$status != "yes") {
+             leg.side = 2
+         } else if (is.null(annotation$Row$data)) {
+             leg.side = 4
+         } else if (is.null(annotation$Col$data)) {
+             leg.side = 1
+         } else if (dendrogram$Col$status != "yes") {
+             leg.side = 3
+         } else {
+             leg.side = 4
+         }
+     }
+     ## Add the legend space
+     if (leg.side==1) {
+         ll[5,3] = cnt
+         ll.width[3] = 5
+         ll.height[5] = 1
+     } else if (leg.side==2) {
+         ll[3,1] = cnt
+         ll.width[1] = 1
+         ll.height[3] = 5
+     } else if (leg.side==3) {
```

```
+         ll[1,3] = cnt
+         ll.width[3]  = 5
+         ll.height[1] = 1
+     } else if (leg.side==4) {
+         ll[3,5] = cnt
+         ll.width[5]  = 1
+         ll.height[3] = 5
+     }
+
+     ## Compress
+     ndx = rowSums(ll)!=0
+     ll  = ll[ndx, , drop=FALSE]
+     ll.height = ll.height[ndx]
+     ndx = colSums(ll)!=0
+     ll  = ll[, ndx, drop=FALSE]
+     ll.width = ll.width[ndx]
+     ## Do it - show it
+     if (show) {
+         layout(ll, width=ll.width, height=ll.height, respect=TRUE)
+         layout.show(max(ll))
+     }
+     return(list(plot=ll, width=ll.width, height=ll.height, legend.side=leg.sid
+ }
```

1.2 Processing the argument lists

Seeing as we have very much the same possible arguments for row and column dendrograms, annotations and clustering, we follow the example of the lattice argument scales, which uses sub-lists named x and y to set axis-specific elements, and otherwise assumes that all other arguments are set for both axes. The underlying code can be seen in `xyplot.formula` and `construct.scales` in package `lattice` and `modifyList` (which has migrated from `lattice` to `utils`).

`modifyList` fills in recursively all entries from a source list into a target list, including entries that do not exist in the target list. I'd like to be a bit more strict here, and only copy entries from the source list that already exist under the same name in the target list (only overwrite, no new write):

```
> modifyExistingList = function(x, val)
+ {
+     if (is.null(x)) x = list()
+     if (is.null(val)) val = list()
+     stopifnot(is.list(x), is.list(val))
+     xnames <- names(x)
+     vnames <- names(val)
+     for (v in intersect(xnames, vnames)) {
+         x[[v]] <- if (is.list(x[[v]]) && is.list(val[[v]]))
+             modifyExistingList(x[[v]], val[[v]])
+         else val[[v]]
+     }
+     x
+ }
```

Note that this code accepts NULL as valid shorthand for the empty list list(). We can use this generic function within a specific function for extracting heatmap-related arguments:

```
> extractArg = function(arglist, deflist)
+ {
+     if (missing(arglist)) arglist = NULL
+     al2 = modifyExistingList(deflist, arglist)
+     row = col = al2
+     row = modifyExistingList(row, arglist[["Row"]])
+     col = modifyExistingList(col, arglist[["Col"]])
+     list(Row=row, Col=col)
+ }
```

`arglist` is the list of arguments as passed into the calling function; `deflist` is a complete list of all possible entries (arguments) for this list, but without the recursive sub-lists `Row` and `Col`. The function first copies the globally set options from the argument list into the default template, using this as the base for the row- and column specific argument lists, and only then copies in the entries set in the sub-lists. It returns a list with two entries, `Row` and `Col`, where all possible options are set (possibly to their default), making it easy to process.

1.3 Adding annotation

This is a completely reworked version of the original picket plot. The most important differences from a user point of view are that a) we can have more than one covariate (non-factor), and b) we do not have to convert factors to binaries by hand. On the technical side, this is done by drawing everything in one plot window (instead of having a separate plot window for the covariate, as previously).

We use the pre-processor function `convAnnData` to convert the data to a numerical matrix of binary indicators and numerical variables.

In terms of mostly hidden functionality, picket plots can now be plotted vertically as well as horizontally: in principle, a small change, but a clean implementation requires some re-arrangement of the code. What I have done here is to separate the set-up of the plot and the actual drawing as far as possible, so that essentially, I only have to swap coordinates at one place, if required. A really clean implementation would split the function into a generating function and a plotting function. Note that internally, we work with matrices with two columns for x- and y-coordinates, even for single points.

The current implementation could be improved – the function has been simplified by moving all of the pre-processing to `convAnnData`, but some of the internal looping does not take this into account and could be simplified.

```
> picketPlot = function (x, grp=NULL, grpcol, grplabel=NULL, horizontal=TRUE, as
+ #
+ # Name: picketPlot (looks like a picket fence with holes, and sounds like the
+ #                   pocketplot in geostatistics)
+ # Desc: visualizes a pattern of 0/1/NAs by using bars, great for annotating a
+ #       heatmap
+ # Auth: Alexander.Ploner@meb.ki.se  181203
+ #
+ # Chng: 221203 AP loess() with degree < 2
+ #       260104 AP
+ #       - made loess() optional
+ #       - better use of space if no covariate
+ #       030304 AP
+ #       - added RainbowPastel() as default colors
```

```
+ #       - check grplabel before passing it to axis
+ #       2010-07-08 AP
+ #       - complete re-write
+ #       2010-08-28
+ #       - re-arranged code for vertical/horizontal drawing
+ {
+     # deal with the setup
+     cc = picketPlotControl()
+     cc[names(control)] = control
+
+     ## Convert/check the data
+     x = convAnnData(x, asIs=asIs)
+
+     # Count variables, panels, types
+     nsamp  = nrow(x)
+     npanel = ncol(x)
+     bpanel = apply(x, 2, function(y) all(y[is.finite(y)] %in% c(0,1)) )
+
+     # Compute panel heights, widths
+     panelw = nsamp*(cc$boxw+2*cc$hbuff)
+     panelh = cc$boxh+2*cc$vbuff
+     totalh = sum(panelh * ifelse(bpanel, 1, cc$numfac))
+     LL = cbind(0, 0)
+     UR = cbind(panelw, totalh)
+
+     # Set up the x-values for a single panel
+     xbase = seq(cc$hbuff, by=cc$boxw+2*cc$hbuff, length=nsamp)
+     xcent = xbase + cc$boxw/2
+
+     # if we get a cluster variable, we have to set differently colored
+     # backgrounds; this assumes that the  grp variable is sorted in the
+     # way it appears on the plot
+     if (!is.null(grp)) {
+         grp = as.integer(factor(grp, levels=unique(grp)))
+         tt = table(grp)
+         gg = length(tt)
+         grpcoord = c(0,cumsum(tt/sum(tt))*panelw)
+         grp0 = cbind(grpcoord[1:gg], rep(0, gg))
```

```
+         grp1 = cbind(grpcoord[2:(gg+1)], rep(totalh, gg))
+         if (missing(grpcol)) {
+             grpcol=BrewerClusterCol
+         }
+         if (is.function(grpcol)) grpcol = grpcol(gg)
+         ## In case of manually specified group colors, we only check/use the
+         ## relevant colors and ignore the rest
+         grpcol = grpcol[1:gg]
+         if (gg > 1) {
+             if ( any(grpcol[-1] == grpcol[-gg]) ) warning("neighboring cluster
+         }
+     }
+
+     # Loop over vars and fill in the panels
+     panels = list()
+     voff = 0
+     for (i in 1:npanel) {
+         if (bpanel[i]) {
+             ## Coordinates
+             x0 = xbase
+             x1 = x0+cc$boxw
+             y0 = voff + cc$vbuff
+             y1 = y0   + cc$boxh
+             ## Set fill
+             fill = ifelse(x[, i, drop=FALSE]==1, "black", "transparent")
+             fill[is.na(fill)] = cc$nacol
+             label = colnames(x)[i]
+             labcc = if (!is.null(label)) (y0+y1)/2 else NULL
+             panels[[i]] = list(ll=cbind(x0, y0), ur=cbind(x1, y1), fill=fill,
+             voff = voff + panelh
+         } else {
+             xv = x[,i]
+             rr = range(xv, na.rm=TRUE)
+             yval = voff + cc$vbuff*cc$numfac + ((xv - rr[1])/(rr[2] - rr[1]))*
+             if ((cc$degree>0) & (cc$span>0)){
+                 yy = predict(loess(yval~xcent, span=cc$span, degree=cc$degree)
+             } else {
+                 yy = rep(NA, length(xcent))
```

```
+             }
+             label = colnames(x)[i]
+             labcc = if (!is.null(label)) mean(range(yval, na.rm=TRUE)) else NU
+             axlab = pretty(range(xv, na.rm=TRUE))
+             axcc  = voff + cc$vbuff*cc$numfac + ((axlab - rr[1])/(rr[2] - rr[1
+             panels[[i]] = list(raw=cbind(xcent, yval), smo=cbind(xcent, yy), l
+             voff = voff + panelh*cc$numfac
+         }
+     }
+
+     # if grplabels are given, we add another horizontal axis to the
+     # last plot (independent of whether it is binvar or contvar)
+     if (!is.null(grp) & !is.null(grplabel)) {
+         mids = (grpcoord[1:gg] + grpcoord[2:(gg+1)])/2
+         # Is the grplabel ok?
+         labelnum = length(grplabel)
+         if (labelnum < gg) {
+             warning("more groups than labels (filling up with blanks)")
+             grplabel = c(grplabel, rep(" ", gg-labelnum))
+         } else if (gg < labelnum) {
+             warning("more labels than groups (ignoring the extras)")
+             grplabel = grplabel[1:gg]
+         }
+     }
+
+     ## Switch coordinates, if you have to
+     h2v = function(cc) cbind(cc[,2]-totalh, cc[,1])
+     if (horizontal) {
+         grpaxis = 1
+         labaxis = 2
+         covaxis = 4
+         las = 1
+     } else {
+         grpaxis = 4
+         labaxis = 3
+         covaxis = 1
+         las = 3
+         ## Rotate
```

```
+         LL = h2v(LL)
+         UR = h2v(UR)
+         if (!is.null(grp)) {
+             grp0 = h2v(grp0)
+             grp1 = h2v(grp1)
+         }
+         for (i in 1:npanel) {
+             panels[[i]][[1]] = h2v(panels[[i]][[1]])
+             panels[[i]][[2]] = h2v(panels[[i]][[2]])
+             panels[[i]]$labcc = panels[[i]]$labcc - totalh
+             panels[[i]]$axcc  = panels[[i]]$axcc - totalh
+         }
+     }
+
+     # Set up the plot
+     plot(rbind(LL, UR), type="n", xaxt="n", yaxt="n", xlab="", ylab="")
+     # Add the colored rectangles, if required
+     if (!is.null(grp)) {
+         rect(grp0[,1], grp0[,2], grp1[,1], grp1[,2], col=grpcol, border="trans
+     }
+     # Loop over vars and fill in the panels
+     for (i in 1:npanel) {
+         if (bpanel[i]) {
+             ## Do the rectangles
+             with(panels[[i]], rect(ll[,1], ll[,2], ur[,1], ur[,2], col=fill, b
+         } else {
+             with(panels[[i]], points(raw[,1], raw[,2], pch=cc$pch, cex=cc$cex.
+             if ((cc$degree>0) & (cc$span>0)){
+                 with(panels[[i]], lines(smo[,1], smo[,2]))
+             }
+             with(panels[[i]], axis(covaxis, at=axcc, label=axlab))
+         }
+         ## Name panel (regardless of type)
+         if (!is.null(panels[[i]]$label)) {
+             axis(labaxis, at=panels[[i]]$labcc, label=panels[[i]]$label, las=l
+         }
+     }
+     # if grplabels are given, we add another horizontal axis to the
```

```
+     # last plot (independent of whether it is binvar or contvar)
+     if (!is.null(grp) & !is.null(grplabel)) {
+         axis(grpaxis, grpcoord, label=FALSE, tcl=-1.5)
+         axis(grpaxis, mids, label=grplabel, font=2, cex.axis=cc$cex.label, tic
+     }
+     invisible(panels)
+ }
```

Note that `picketPlot` makes use of a small helper function that returns the list of default settings for the `control` list. This has been externailzed so that `annHeatmap2` has access to it for setting up a correct parsing template (see below): not pretty, but simple.

```
> picketPlotControl = function()
+ {
+     list(boxw=1, boxh=4, hbuff=0.1, vbuff=0.1, span=1/3, nacol=gray(0.85),
+          degree=1, cex.label=1.5, numfac=2, pch=par("pch"), cex.pch=par("cex")
+          col.pch=par("col") )
+ }
```

1.4 Defining nice breaks with nice colors

Breaks I find that the specification of the classes is entirely insufficient in the old version: essentially using the equidistant classes from `image`, with the `trim` and `equalize` modifiers. This, of course, will not do: we want nice breaks (round numbers), we want our equidistant classes centered around zero, and we want to be able to specifiy a set of breaks of our own choosing.

This is achieved via the argument `breaks` that works a little bit like the argument of the same name for function `cut`: a vector specifies the actual breaks, and a single number is interpreted as the desired number of equidistant classes. However, we do not just chop up the range of the values into a suitable number of classes, we use the function `pretty` to get nice round breaks; the number of resulting intervals can very slightly from the suggested number.

For the special case that we have both negative and positive numbers, we make sure that zero is one of the breaks, so that the intervals extend somewhat symmetrically around zero. For the default behaviour of standardizing the rows of the data matrix, this is eminently reasonable.

All of this is packed into a helper function, of course:

```
> niceBreaks = function(xr, breaks)
+ {
+     ## If you want it, you get it
+     if (length(breaks) > 1) {
+         return(sort(breaks))
+     }
+     ## Ok, so you proposed a number
+     ## Neg and pos?
+     if ( (xr[1] < 0) & (xr[2] > 0) ) {
+         xminAbs = abs(xr[1])
+         xmax    = xr[2]
+         nneg = max(round(breaks * xminAbs/(xmax+xminAbs)), 1)
+         npos = max(round(breaks * xmax/(xmax+xminAbs)), 1)
+         nbr  = pretty(c(xr[1], 0), nneg)
+         pbr  = pretty(c(0, xr[2]), npos)
+         ## Average of the proposed interval lengths,
+         ##  nice enough for us
+         diff = ( (nbr[2]-nbr[1]) + (pbr[2] - pbr[1]) ) / 2
+         nbr  = diff * ( (xr[1] %/% diff)  : 0 )
+         pbr  = diff * ( 1 : (xr[2] %/% diff + 1) )
+         breaks = c(nbr, pbr)
+     } else { ## only pos or negs
+         breaks = pretty(xr, breaks)
+     }
+     breaks
+ }
```

Note that the user is free to specify breaks that do not cover the whole range of the data, in which case we could

- replace values outside the breaks with the smallest or largest break (for display only, of course),
- cleverly indicate in the legend that there are more values beyond the indicated limits, by adding a $<$ or $>$ sign in the annotation.

This would neatly implement functionality comparable to the original `trim`-argument, avoiding the problem of outliers compressing the display scale.

Potentially, we may want use quantile-based breaks (i.e. equal counts instead of equal lengths), but this remains to be seen.

Colors Once we have defined a nice set of breaks, we want to be able to specify the colors for the corresponding intervals. Again, we aim for convenience in case of the default situation of standardized data – i.o.w. when we have breaks that are symmetric around zero, we want to have colors that are symmetric around zero, or more precisely, a color range that is symmetric around zero, whether or not the actual values are. E.g. in case of the classical green to black to red scale for (log) gene expression and data ranging from -2 to +3, we want to have black (with red or green tinge) in the center, full green at -3 and full red at +3; this means that there will be no full red in central plot, but we get a visual impression of the asymmetry in the data. (Note that this can be handled in different ways in the legend).

On the other hand, we want to allow the user to specify *exactly* the colors they feel are necessary. Therefore, colors can be specified via the argument `col` in two ways:

1. as a vector of colors (generally through a call to a palette function as in e.g. `heat.colors(12)`),
2. as the name of a palette function; in this case, the result will depend on whether or not the breaks are symmetric around zero: if not, we get the standard palette for the implied number of classes; if symmetric, we get the extended color space as outlined above.

```
> breakColors = function(breaks, colors, center=0, tol=0.001)
+ {
+     ## In case of explicit color definitions
+     nbreaks = length(breaks)
+     nclass  = nbreaks - 1
+     if (!is.function(colors)) {
+         ncolors = length(colors)
+         if (ncolors > nclass) {
+             warning("more colors than classes: ignoring ", ncolors-nclass, " l
+             colors = colors[1:nclass]
+         } else if (nclass > ncolors) {
+             stop(nclass-ncolors, " more classes than colors defined")
```

```
+         }
+     } else {
+         ## Are the classes symmetric and of same lengths?
+         clens = diff(breaks)
+         aclen = mean(clens)
+         if (aclen==0) stop("Dude, your breaks are seriously fucked up!")
+         relerr = max((clens-aclen)/aclen)
+         if ( (center %in% breaks) & (relerr < tol) ) { ## yes, symmetric
+             ndxcen = which(breaks==center)
+             kneg = ndxcen -1
+             kpos = nbreaks - ndxcen
+             kmax = max(kneg, kpos)
+             colors = colors(2*kmax)
+             if (kneg < kpos) {
+                 colors = colors[ (kpos-kneg+1) : (2*kmax) ]
+             } else if (kneg > kpos) {
+                 colors = colors[ 1 : (2*kmax - (kneg-kpos)) ]
+             }
+         } else {                                        ## no, not symmetric
+             colors = colors(nclass)
+         }
+     }
+     colors
+ }
```

We also offer a re-implementation of the green to black to red scale that a) actually goes from green to red (instead of the wrong way round) and b) keeps a red or green tinge for the "black" colors left and right of zero (equal number of classes only). This is of course just an updated version of the old `RGBColVec`:

```
> g2r.colors = function(n=12, min.tinge = 0.33)
+ {
+     k <- trunc(n/2)
+     if (2 * k == n) {
+         g <- c(rev(seq(min.tinge, 1, length = k)), rep(0, k))
+         r <- c(rep(0, k), seq(min.tinge, 1, length = k))
+         colvec <- rgb(
```

```
+         r, g, rep(0, 2 * k))
+     }
+     else {
+         g <- c(rev(seq(min.tinge, 1, length = k)), rep(0,
+             k + 1))
+         r <- c(rep(0, k + 1), seq(min.tinge, 1, length = k))
+         colvec <- rgb(r, g, rep(0, 2 * k + 1))
+     }
+     colvec
+ }
```

1.5 Adding a legend

Here we have several options that could be considered (and may even be implemented one day):

- a simple strip with an axis, indicating the range of values and colors, as in the old heatmaps;
- a stack of colored rectangles, indicating the different colors, and giving the class limits (i.e. the size of the color rectangle is not proportional to the class width);
- some variant of the cool histogram legend they have in `heatmap2`.

```
> doLegend = function(breaks, col, side)
+ {
+     zval = ( breaks[-1] + breaks[-length(breaks)] ) / 2
+     z  = matrix(zval, ncol=1)
+     if (side %in% c(1,3)) {
+         image(x=zval, y=1, z=z, xaxt="n", yaxt="n", col=col, breaks=breaks , x
+     } else {
+         image(x=1, y=zval, z=t(z), xaxt="n", yaxt="n", col=col, breaks=breaks,
+     }
+     axis(side, las=1)
+ }
```

1.6 Pre-process annonation data frames

An annotation data frame should have either factors or numerical variables. This requires e.g. conversion of logical values, but also numerically coded factors (say 0/1). We take a leave from Frank Harrel's book and convert numerical variables with fewer than some c variables into factors.

The binary indicators are provided courtesy of `model.matrix`. The argument `inclRef` determines whether the reference level is included or not. I have also included a pass-through option `asIs` that only checks the argument whether it is a proper numerical matrix and returns it unchanged if ok.

```
> convAnnData = function(x, nval.fac=3, inclRef=TRUE, asIs=FALSE)
+ {
+     if (is.null(x)) return(NULL)
+     if (asIs) {
+         if (is.matrix(x) & is.numeric(x)) return(x)
+         else stop("argument x not a numerical matrix, asIs=TRUE does not work"
+     }
+
+     x = as.data.frame(x)
+     if (!is.null(nval.fac) & nval.fac>0) doConv = TRUE
+     vv = colnames(x)
+     for (v in vv) {
+         if (is.logical(x[,v])) {
+             x[,v] = factor(as.numeric(x[,v]))
+         }
+         if (doConv & length(unique(x[is.finite(x[,v]),v])) <= nval.fac) {
+             x[,v] = factor(x[,v])
+         }
+     }
+     ret  = NULL
+     ivar = 0
+     for (v in vv) {
+          xx = x[, v]
+          if (is.factor(xx)) {
+             nandx = is.na(xx)
+             if (length(unique(xx[!nandx])) > 1) {
+                 naAction = attr(na.exclude(x[, v, drop=FALSE]), "na.action")
```

```
+                 modMat   = model.matrix(~xx-1)
+                 if (!inclRef) modMat = modMat[ , -1, drop=FALSE]
+                 binvar   = naresid(naAction, modMat)
+                 colnames(binvar) = paste(v, "=", levels(xx)[if (!inclRef) -1 e
+             } else {
+                 nlev = length(levels(xx))
+                 ilev = unique(as.numeric(xx[!nandx]))
+                 if (length(ilev)==0) {
+                     binvar = matrix(NA, nrow=length(xx), ncol=nlev)
+                 } else {
+                     binvar = matrix(0, nrow=length(xx), ncol=nlev)
+                     binvar[, ilev] = 1
+                     binvar[nandx, ] = NA
+                 }
+                 colnames(binvar) = paste(v, "=", levels(xx), sep="")
+             }
+             ret = cbind(ret, binvar)
+             ivar = ivar + ncol(binvar)
+          } else {
+             ret = cbind(ret, x[,v])
+             ivar = ivar + 1
+             colnames(ret)[ivar] = v
+          }
+     }
+     ret
+ }
```

1.7 Cut a dendrogram

An annoying feature of the old version was that we were schlepping around the hclust and the dendrogram version of the same object, because the cut function for dendrograms does not return a nice clustering variable.

This is fairly straightforward, with an interesting complication: for an organically grown dendrogram, `order.dendrogram` returns the index of the leaves wrt to the original order of the underlying data, and we can use it to re-order in an intuitive manner. For dendrograms that result from cutting up such an organically grown dendrogram, this index remains unchanged: in other

words, just looking at the sub-dendrogram, we only get the clustering for the leaves in the sub-tree, with the ordering pointing to a potentially much larger data set of unknown size. We could return a padded vector with lots of NAs, but here I just check and fail informatively when we encounter this situation.

```
> cutree.dendrogram = function(x, h)
+ {
+     # Cut the tree, get the labels
+     cutx = cut(x, h)
+     cutl = lapply(cutx$lower, getLeaves)
+     # Set up the cluster vector as seen in the plot
+     nclus = sapply(cutl, length)
+     ret   = rep(1:length(nclus), nclus)
+     # Return cluster membership in the order of the original data, if possible
+     ord = order.dendrogram(x)
+     # Is the order a valid permutation of the data?
+     if (!all(sort(ord)==(1:length(ret)))) {
+         stop("dendrogram order does not match number of leaves - is this a sub
+     }
+     # Ok, proceed
+     ret[ord] = ret
+     ret = as.integer(factor(ret, levels=unique(ret))) # recode for order of cl
+     names(ret)[ord] = unlist(cutl)
+     ret
+ }
```

Note that since R version 2.13.0, there is a function `as.hclust.dendrogram`, so that we could just convert our dendrogram internally and use `cutree`. First of all, I don't like to force people to upgrade to the latest version just for one silly command, secondly, this combo does currently not deal correctly with the sub-tree situation either, but returns crap instead of failing. So maybe somwhere down the road...

A nice little helper-helper that traverses the dendrogram and returns the labels of the leaves. Note that a simple recursion would work as well for binary trees, but it fails for dendrograms with more than two branches (see examples in `?dendrogram` for such a beast). So we sacrifice a bit of performance for (strictly speaking unnecessary) generality:

```
> getLeaves = function(x)
+ {
+     unlist(dendrapply(x, function(x) attr(x, "label")))
+ }
```

1.8 A reasonable print method

We have decided to stick with S3 classes – if it's good enough for lattice graphics, it's definitely good enough for us, god bless them. Below a prototype for a showing a short description:

```
> print.annHeatmap = function(x, ...)
+ {
+     cat("annotated Heatmap\n\n")
+     cat("Rows: "); show(x$dendrogram$Row$dendro)
+     cat("\t", if (is.null(x$annotation$Row$data)) 0 else ncol(x$annotation$Row
+     cat("Cols: "); show(x$dendrogram$Col$dendro)
+     cat("\t", if (is.null(x$annotation$Col$data)) 0 else ncol(x$annotation$Col
+     invisible(x)
+ }
```

1.9 Nice colors for clusters

This is the new default color scheme for clusters, see remarks below with `RainbowPastel`. This function wraps the qualitative palettes from `RColorBrewer` and will return a color vector of any specified length: if the number of required colors is larger than the number of available colors in the specifed palette, the colors will be recycled – visually, this is not a problem, because adjacent clusters will have different colors, and clusters of the same color will be well separated.

In the spirit of the previous coloring, I have chosen `Pastel1` as the default palette.

```
> BrewerClusterCol = function(n, name="Pastel1")
+ {
+     ## Check the name of the palette
+     qualpal = subset(RColorBrewer::brewer.pal.info, category=="qual")
```

```
+     name = match.arg(name, rownames(qualpal))
+     nmax = qualpal[name, "maxcolors"]
+
+     ## Get the full color vector of the palette
+     cols = RColorBrewer::brewer.pal(nmax, name)
+
+     ## Build the (shortened or recycled) index vector
+     ndx  = rep(1:nmax, length=n)
+
+     cols[ndx]
+ }
```

1.10 From the previous version

These functions are copied from the old Heatplus, with little or no changes. Nice colors for the different clusters:

```
> RainbowPastel =  function (n, blanche=200, ...)
+ #
+ # Name: RainbowPastel
+ # Desc: constructs a rainbow clolr vector, but more pastelly
+ # Auth: Alexander.Ploner@mep.ki.se      030304
+ #
+ # Chng:
+ #
+
+ {
+     cv = rainbow(n, ...)
+     rgbcv = col2rgb(cv)
+     rgbcv = pmin(rgbcv+blanche, 255)
+     rgb(rgbcv[1,], rgbcv[2,], rgbcv[3, ], maxColorValue=255)
+ }
```

Actually, these colors are NOT nice, in that the extra white component will lead to color vectors with duplicate colors next to each other, already for `n=5`. Consequently, this is no longer the default color scheme for clusters, and only kept around for traditionalists who like the classic color scheme.

Compared to the previous version, this one checks whether the propose cutting height is greater than the tree height (in which case it just plots the tree and moves on), and deals correctly with singleton clusters. Note that the treatment of singletons is not great, as it colors the corresponding branch all the way up to the specified cutting height, while non-singletons are only colored up to the fork; this will have to do for now, but is clearly a FIXME (technically, coloring already the trunk above the cut sounds feasible).

```
> cutplot.dendrogram = function(x, h, cluscol, leaflab= "none", horiz=FALSE, lwd
+ #
+ # Name: cutplot.dendrogram
+ # Desc: takes a dendrogram as described in library(mva), cuts it at level h,
+ #       and plots the dendrogram with the resulting subtrees in different
+ #       colors
+ # Auth: obviously based on plot.dendrogram in library(mva)
+ #       modifications by Alexander.Ploner@meb.ki.se  211203
+ #
+ # Chng: 050204 AP
+ #       changed environment(plot.hclust) to environment(as.dendrogram) to
+ #       make it work with R 1.8.1
+ #       250304 AP added RainbowPastel() to make it consistent with picketplot
+ #       030306 AP slightly more elegant access of plotNode
+ #       220710 AP also for horizontal plots
+ #       120811 AP use edgePar instead of par() for col and lwd
+ #
+ {
+     ## If there is no cutting, we plot and leave
+     if (missing(h) | is.null(h)) {
+         return(plot(x, leaflab=leaflab, horiz=horiz, edgePar=list(lwd=lwd), ..
+     }
+     ## If cut height greater than tree, don't cut, complain and leave
+     treeheight = attr(x, "height")
+     if (h >= treeheight) {
+         warning("cutting height greater than tree height ", treeheight, ": tre
+         return(plot(x, leaflab=leaflab, horiz=horiz, edgePar=list(lwd=lwd), ..
+     }
+
+     ## Some param processing
```

```
+     if (missing(cluscol) | is.null(cluscol)) cluscol = BrewerClusterCol
+
+     # Not nice, but necessary
+     pn  = stats:::plotNode
+
+     x = cut(x, h)
+     plot(x[[1]], leaflab="none", horiz=horiz, edgePar=list(lwd=lwd), ...)
+
+     x = x[[2]]
+     K = length(x)
+     if (is.function(cluscol)) {
+        cluscol = cluscol(K)
+     }
+     left = 1
+     for (k in 1:K) {
+         right = left + attr(x[[k]],"members")-1
+         if (left < right) {         ## not a singleton cluster
+             pn(left, right, x[[k]], type="rectangular", center=FALSE,
+                  leaflab=leaflab, nodePar=NULL, edgePar=list(lwd=lwd, col=clus
+         } else if (left == right) { ## singleton cluster
+             if (!horiz) {
+                 segments(left, 0, left, h, lwd=lwd, col=cluscol[k])
+             } else {
+                 segments(0, left, h, left, lwd=lwd, col=cluscol[k])
+             }
+         } else stop("this totally should not have happened")
+         left = right + 1
+    }
+
+ }
```

2 Working functions

2.1 Generating function

This is the actual workhorse underlying the different functions for generating standard plots. It allows any combination of row and column dendrograms and annotations, and has a consistent, if not very friendly interface, based on lists within lists, as described above. The point of this function is to offer an expert access to the bells and whistles in a sane manner; the casual user will prefer one of the wrapping functions creating e.g. a standard annotated heatmap with column annotation and without row dendrogram.

```
> annHeatmap2 = function(x, dendrogram, annotation, cluster, labels, scale=c("ro
+ #
+ # Name: annHeatmap2
+ # Desc: a (possibly doubly) annotated heatmap
+ # Auth: Alexander.Ploner@ki.se 2010-07-12
+ #
+ # Chng:
+ #
+ {
+     ## Process arguments
+     if (!is.matrix(x) | !is.numeric(x)) stop("x must be a numeric matrix")
+     nc = ncol(x); nr = nrow(x)
+     if (nc < 2 | nr < 2) stop("x must have at least two rows/columns")
+
+     ## Process the different lists: dendrogram, cluster, annotation
+     ## See lattice:::xyplot.formula, modifyLists, lattice:::construct.scales
+     def = list(clustfun=hclust, distfun=dist, status="yes", lwd=3, dendro=NULL
+     dendrogram = extractArg(dendrogram, def)
+     def = list(data=NULL, control=picketPlotControl(), asIs=FALSE, inclRef=TRU
+     annotation = extractArg(annotation, def)
+     def = list(cuth=NULL, grp=NULL, label=NULL, col=BrewerClusterCol)
+     cluster = extractArg(cluster, def)
+     def = list(cex=NULL, nrow=3, side=NULL, labels=NULL)
+     labels = extractArg(labels, def)
+
+     ## Check values for the different lists
```

```
+
+
+     ## Generate the layout: TRUE means default, FALSE means none
+     ## Otherwise, integer 1-4 indicates side
+     if (is.logical(legend)) {
+         if (legend) leg = NULL else leg = 0
+     } else {
+         if (!(legend %in% 1:4)) stop("invalid value for legend: ", legend)
+         else leg=legend
+     }
+     layout = heatmapLayout(dendrogram, annotation, leg.side=leg)
+
+     ## Copy the data for display, scale as required
+     x2 = x
+     scale = match.arg(scale)
+     if (scale == "row") {
+         x2 = sweep(x2, 1, rowMeans(x, na.rm = TRUE))
+         sd = apply(x2, 1, sd, na.rm = TRUE)
+         x2 = sweep(x2, 1, sd, "/")
+     }
+     else if (scale == "col") {
+         x2 = sweep(x2, 2, colMeans(x, na.rm = TRUE))
+         sd = apply(x2, 2, sd, na.rm = TRUE)
+         x2 = sweep(x2, 2, sd, "/")
+     }
+
+     ## Construct the breaks and colors for display
+     breaks = niceBreaks(range(x2, na.rm=TRUE), breaks)
+     col    = breakColors(breaks, col)
+
+     ## Generate the dendrograms, if required; re-indexes in any cases
+     ## We could put some sanity checks on the dendrograms in the else-branches
+     ## FIXME: store the names of the functions, not the functions in the objec
+     dendrogram$Row = within(dendrogram$Row,
+         if (!inherits(dendro, "dendrogram")) {
+             dendro = clustfun(distfun(x))
+             dendro = reorder(as.dendrogram(dendro), rowMeans(x, na.rm=TRUE))
+         }
```

```
+     )
+     dendrogram$Col = within(dendrogram$Col,
+         if (!inherits(dendro, "dendrogram")) {
+             dendro = clustfun(distfun(t(x)))
+             dendro = reorder(as.dendrogram(dendro), colMeans(x, na.rm=TRUE))
+         }
+     )
+     ## Reorder the display data to agree with the dendrograms, if required
+     rowInd = with(dendrogram$Row, if (status!="no") order.dendrogram(dendro) e
+     colInd = with(dendrogram$Col, if (status!="no") order.dendrogram(dendro) e
+     x2 = x2[rowInd, colInd]
+
+     ## Set the defaults for the sample/variable labels
+     labels$Row = within(labels$Row, {
+         if (is.null(cex)) cex = 0.2 + 1/log10(nr)
+         if (is.null(side)) side = if (is.null(annotation$Row$data)) 4 else 2
+         if (is.null(labels)) labels = rownames(x2)
+     })
+     labels$Col = within(labels$Col, {
+         if (is.null(cex)) cex = 0.2 + 1/log10(nc)
+         if (is.null(side)) side = if (is.null(annotation$Col$data)) 1 else 3
+         if (is.null(labels)) labels = colnames(x2)
+     })
+
+     ## Generate the clustering, if required (cut, or resort the cluster var)
+     ## FIXME: does not deal with pre-defined grp form outside
+     cluster$Row = within(cluster$Row,
+         if (!is.null(cuth) && (cuth > 0)) {
+             grp = cutree.dendrogram(dendrogram$Row$dendro, cuth)[rowInd]
+         })
+     cluster$Col = within(cluster$Col,
+         if (!is.null(cuth) && (cuth > 0)) {
+             grp = cutree.dendrogram(dendrogram$Col$dendro, cuth)[colInd]
+         })
+
+     ## Process the annotation data frames (factor/numeric, re-sort?)
+     annotation$Row = within(annotation$Row, {
+         data = convAnnData(data, asIs=asIs, inclRef=inclRef)
```

```
+     })
+     annotation$Col = within(annotation$Col, {
+         data = convAnnData(data, asIs=asIs, inclRef=inclRef)
+     })
+
+
+     ## Generate the new object
+
+     ## print, return invisibly
+     ret = list(data=list(x=x, x2=x2, rowInd=rowInd, colInd=colInd, breaks=brea
+     class(ret) = "annHeatmap"
+     ret
+
+ }
```

2.2 Plotting function

This function takes the `annHeatmap` object generated from the user input and plots as required.

```
> plot.annHeatmap = function(x, widths, heights, ...)
+ {
+         ## Preserve parameters that are set explicitly below
+         ## Not doing this has lead to Issue 8: inconsistent distance
+         ## between dendrogram and heatmap after repeated calls on same device
+         opar = par("oma", "mar", "xaxs", "yaxs")
+         on.exit(par(opar))
+     ## If there are cluster labels on either axis, we reserve space for them
+     doRClusLab = !is.null(x$cluster$Row$label)
+     doCClusLab = !is.null(x$cluster$Col$label)
+     omar = rep(0, 4)
+     if (doRClusLab) omar[4] = 2
+     if (doCClusLab) omar[1] = 2
+     par(oma=omar)
+     ## Set up the layout
+     if (!missing(widths)) x$layout$width = widths
+     if (!missing(heights)) x$layout$height = heights
```

```
+     with(x$layout, layout(plot, width, height, respect=TRUE))
+
+     ## Plot the central image, making space for labels, if required
+     nc = ncol(x$data$x2); nr = nrow(x$data$x2)
+     doRlab = !is.null(x$labels$Row$labels)
+     doClab = !is.null(x$labels$Col$labels)
+     mmar = c(1, 0, 0, 2)
+     if (doRlab) mmar[x$labels$Row$side] = x$labels$Row$nrow
+     if (doClab) mmar[x$labels$Col$side] = x$labels$Col$nrow
+     with(x$data, {
+         par(mar=mmar)
+         image(1:nc, 1:nr, t(x2), axes = FALSE, xlim = c(0.5, nc + 0.5), ylim =
+     })
+     with (x$labels, {
+         if (doRlab) axis(Row$side, 1:nr, las = 2, line = -0.5, tick = 0, label
+         if (doClab) axis(Col$side, 1:nc, las = 2, line = -0.5, tick = 0, label
+     })
+
+     ## Plot the column/row dendrograms, as required
+     with(x$dendrogram$Col,
+         if (status=="yes") {
+             par(mar=c(0, mmar[2], 3, mmar[4]))
+             cutplot.dendrogram(dendro, h=x$cluster$Col$cuth, cluscol=x$cluster
+         })
+     with(x$dendrogram$Row,
+         if (status=="yes") {
+             par(mar=c(mmar[1], 3, mmar[3], 0))
+             cutplot.dendrogram(dendro, h=x$cluster$Row$cuth, cluscol=x$cluster
+         })
+
+     ## Plot the column/row annotation data, as required
+     if (!is.null(x$annotation$Col$data)) {
+         par(mar=c(1, mmar[2], 0, mmar[4]), xaxs="i", yaxs="i")
+         picketPlot(x$annotation$Col$data[x$data$colInd, ,drop=FALSE],
+           grp=x$cluster$Col$grp, grplabel=x$cluster$Col$label, grpcol=x$cluste
+           control=x$annotation$Col$control, asIs=TRUE)
+     }
+     if (!is.null(x$annotation$Row$data)) {
```

```
+         par(mar=c(mmar[1], 0, mmar[3], 1), xaxs="i", yaxs="i")
+         picketPlot(x$annotation$Row$data[x$data$rowInd, ,drop=FALSE],
+           grp=x$cluster$Row$grp, grplabel=x$cluster$Row$label, grpcol=x$cluste
+           control=x$annotation$Row$control, asIs=TRUE, horizontal=FALSE)
+     }
+
+     ## Plot a legend, as required
+     if (x$legend) {
+         if (x$layout$legend.side %in% c(1,3)) {
+             par(mar=c(2, mmar[2]+2, 2, mmar[4]+2))
+         } else {
+             par(mar=c(mmar[1]+2, 2, mmar[3]+2, 2))
+         }
+         doLegend(x$data$breaks, col=x$data$col, x$layout$legend.side)
+     }
+
+     invisible(x)
+
+ }
```

FIXME: This terminates currently ungracefully if the dimension of the annotation frame does not match the dimension of the data; some kind of check is required, either in the generating function or here where we try to plot.

3 Wrapping functions

3.1 Generic methods

We define a set of standard for invocations for S3 classes.

```
> regHeatmap = function(x, ...) UseMethod("regHeatmap")
> annHeatmap = function(x, ...) UseMethod("annHeatmap")
```

3.2 Default standard heatmap

We plot a standard heatmap without annotation:

```
> regHeatmap.default = function(x, dendrogram=list(clustfun=hclust, distfun=dist
+ {
+     ret = annHeatmap2(x, dendrogram=dendrogram, annotation=NULL, cluster=NULL,
+     ret
+ }
```

3.3 Default standard annotated heatmap

We plot a standard annotated heatmap (annotated columns, no row dendrogram); note that we explicitly check for `annotation` to be a data frame (legitimate for an annotated heatmap) to avoid cryptic error messages downstream.

```
> annHeatmap.default = function(x, annotation, dendrogram=list(clustfun=hclust,
+ {
+         if (!is.data.frame(annotation)) stop("Argument 'annoation' needs to be
+
+     ret = annHeatmap2(x, dendrogram=dendrogram, annotation=list(Col=list(data=
+     ret
+ }
```

FIXME: The glaring disdavantage is that while this makes specification of the annotation slightly simpler, it does not allow to specify e.g. `inclRef`. FIXME: the check for data frame is only a start, there have to be better checks wrt to data / annotation agreement; hopefully this can be done downstream

3.4 Annotated heatmap for ExpressionSet

```
> annHeatmap.ExpressionSet =function(x, ...)
+ {
+     expmat = exprs(x)
+     anndat = pData(x)
+     annHeatmap(expmat, anndat, ...)
+ }
```