A Literate Program for Drawing Dotcharts

Jimmy Oh

Department of Statistics
University of Auckland

Abstract

Dotcharts are used to plot one quantitative variable with labels (Cleveland, 1985) and has many advantages over other ways of displaying labeled data. We present a method for drawing dotcharts in R, one capable of reproducing most dotcharts given in Cleveland, W. S. (1985) **The Elements of Graphing Data.**

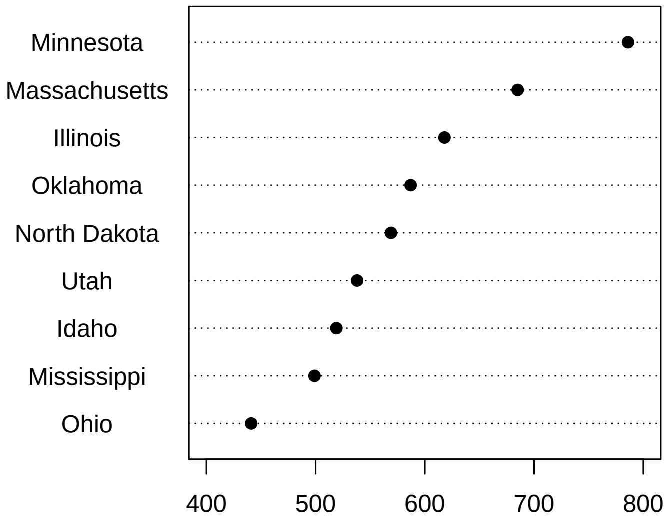

Figure 1: A simple dotchart. The data plots per capita taxes in 1980 of select states in the United States. The position of the points indicate the data values, while the dotted lines help connect the points to their labels.

Contents

1 Introduction **1**
1.1 Code Overview 1

2 The Main Function **3**
2.1 The Parslist Function 6
2.1.1 Plot related arguments 8
2.1.2 Layout related arguments 8
2.1.3 Axis related arguments (including axis labels) 8
2.1.4 Extra Labels related arguments 9
2.1.5 Graphical arguments 10

3 Auxiliary Functions **11**
3.1 The Layout Auxiliary 11
3.1.1 Forming the Plot Region 14
3.1.2 Attaching Axes Regions 16
3.1.3 Attaching Label Regions 18
3.1.4 Finishing off the Layout 20
3.2 The Plot Auxiliary 20
3.2.1 Setting up the Plot Area 21
3.2.2 Drawing the Points and Lines 22
3.2.3 Drawing the Numerical Axes 24
3.2.4 Drawing the Text Labels Axes 26
3.2.5 Drawing the Extra Labels 28
3.3 The Expandpars Auxiliary 31
3.4 Secondary Auxiliaries 32

4 The Back End **33**
4.1 Back End Computation 34
4.1.1 Parslist Wetwork 37
4.2 Call Primary Auxiliaries 43

5 Chunk Index **44**

6 Identifier Index **45**

7 References **46**

On Literate Programs

This software is presented as a *literate program* written in the *noweb* format. It serves as both documentation and as a container for the code. A single `noweb` file can be used to both produce the *literate document* `pdf` file and to extract executable code. The document is separated into *documentation chunks* and named *code chunks*. Each *code chunk* can contain code or references to other *code chunks* which act as placeholders for the contents of the respective *code chunks*. As the name serves as a short description of the code, each *code chunk* can give an overview of what it does via the names it contains, leaving the reader free to delve deeper into the respective *code chunks* for the code if desired.

1 Introduction

Dotchart Plus is a function designed to plot dotcharts in R.

Dotcharts, also known as ‘Cleveland dotplots’ are used to plot one quantitative variable with labels (Cleveland, 1985). They were “invented in response to the standard ways of displaying labeled data - bar charts, divided bar charts, and pie charts - which usually convey quantitative information less well to the viewer than dot charts” (Cleveland, 1985, p144).

A basic `dotchart` function already exists in the default installation of R. This function “was written as a simple placeholder, to be replaced by a better version when the time to create such an improved version became available. Unfortunately the rewrite never occured.”[1]

Discussion on why you should use a dotchart, how `dotchartplus` differs from `dotchart` and `dotplot` (from the ‘lattice’ R package), and some examples on how to use `dotchartplus` are covered in the *Demonstration Document* (`dotchartplus-demos.pdf`).

This document covers the code for the default method for `dotchartplus`. Major functionality includes the ability to juxtapose *groups* of data, to superpose *sets* of data, and to highlight specific points easily.

1.1 Code Overview

The Dotchart Plus function is structured as follows:

1 ⟨*dotchartplus.R* 1⟩≡

```
⟨document header 33⟩
⟨define default Generic 2⟩
setMethod("dotchartplus", signature(object = "list"),
          local({
            ⟨Auxiliary Functions 11a⟩
            ⟨Main Function 3⟩
            })
          )
⟨Parslist Function 6⟩
```

This code is written to file `dotchartplus.R`.

[1]Ross Ihaka, author of the built-in R `dotchart` function

We first define a default Generic function for `dotchartplus` that is essentially an error message. Any unrecognised object types will result in this default function being called.

We additionally define a shorthand function `dcp`, for ease of use.

2 ⟨*define default Generic* 2⟩≡

```
setGeneric("dotchartplus", useAsDefault =
  function(object, ...){
    cat('"', class(object),
        '" is not a recognised object type.',
        '\nThe following are the currently defined methods:',
        '\n(Note that object="ANY" corresponds to this ',
        'error message function)\n', sep = "")
    showMethods("dotchartplus", inherited = FALSE)
  })
dcp = function(...) dotchartplus(...)
```

With the default error message defined, we can now define the true `dotchartplus` function.

This function takes `list` objects, and all other object methods essentially convert those formats into the require `list` object. The Auxiliary and Main functions are defined within a `local` block whose value is the Main function. This provides a way of hiding the utility functions in a scope which is only visible within the body of the function `dotchartplus`.

Auxiliary Functions provides supporting functions. This is further subsectioned into *Primary* and *Secondary*.

Main Function handles certain back end work to ensure all the required variables are in the right format, then calls *Primary* Auxiliary Functions to do the actual plotting.

Parslist Function contains the `parslist` specification function, which defines the default parameters for many of the optional arguments for `dotchartplus`.

2 The Main Function

The *Main Function* works more or less as it appears. It first carries out certain back end work to ensure all the required variables are in the right format then calls first the *Layout Auxiliary* then the *Plot Auxiliary*.

For advanced users, the function also returns several objects invisibly.

3 ⟨*Main Function* 3⟩≡

```
function(object, textlist = NULL, xlab = NULL, col = NULL,
         at = NULL, atsmall = NULL, atlabels = NULL,
         parslist = DefaultParslist, ...){
  datlist = object
  rm(object)
  ⟨back end work 34⟩
  ⟨call layout aux 43a⟩
  ⟨call plot aux 43b⟩
  ⟨return various as invisible 43c⟩
  }
```

Uses `DefaultParslist` 6, `parslist` 34, and `textlist` 37a.

Main Arguments

`datlist` - the data to plot in list form. The only strictly non-optional argument. This should be a list containing matrices (vectors are taken to be a matrix with one column). Each element of the list specifies a *group* of data. Each column of each matrix specifies *sets* of data. Each row of each matrix specifies data points corresponding to the same text label.

`textlist` - the text label for each point of data. Same form as `datlist`, but with each column of each matrix specifying a new column of text to plot.

```
> datlist
[[1]]

   [,1] [,2]
A1    1    4
A2    2    5
A3    3    6
> dotchartplus(datlist)
```

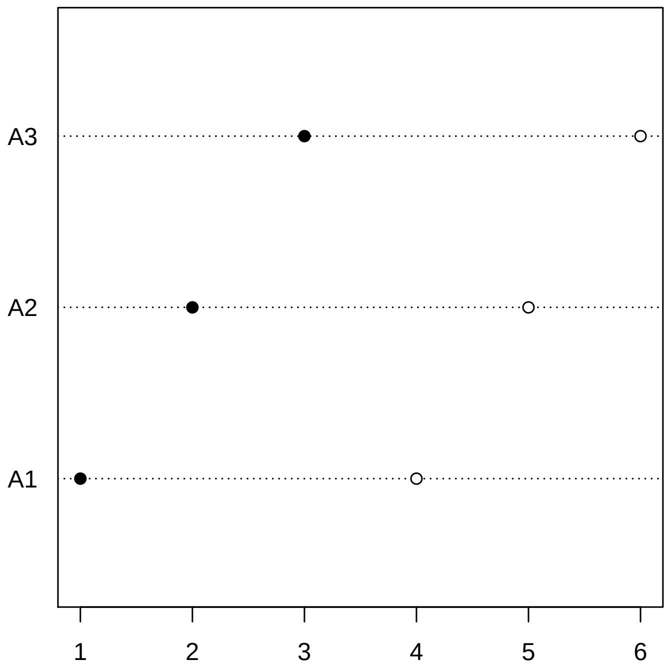

Figure 2: A simple example to demonstrate what `datlist` looks like. Our `datlist` contains 1 group and 2 sets of data. Each set has 3 data points.

```
> datlist
$`Group A`
     black white
[1,]     1     4
[2,]     2     5
[3,]     3     6

> textlist
[[1]]

     [,1]      [,2]
[1,] "Some"    "A1"
[2,] "Made-up" "A2"
[3,] "Data"    "A3"
> dotchartplus(datlist, textlist, adj = c(1, 0))
```

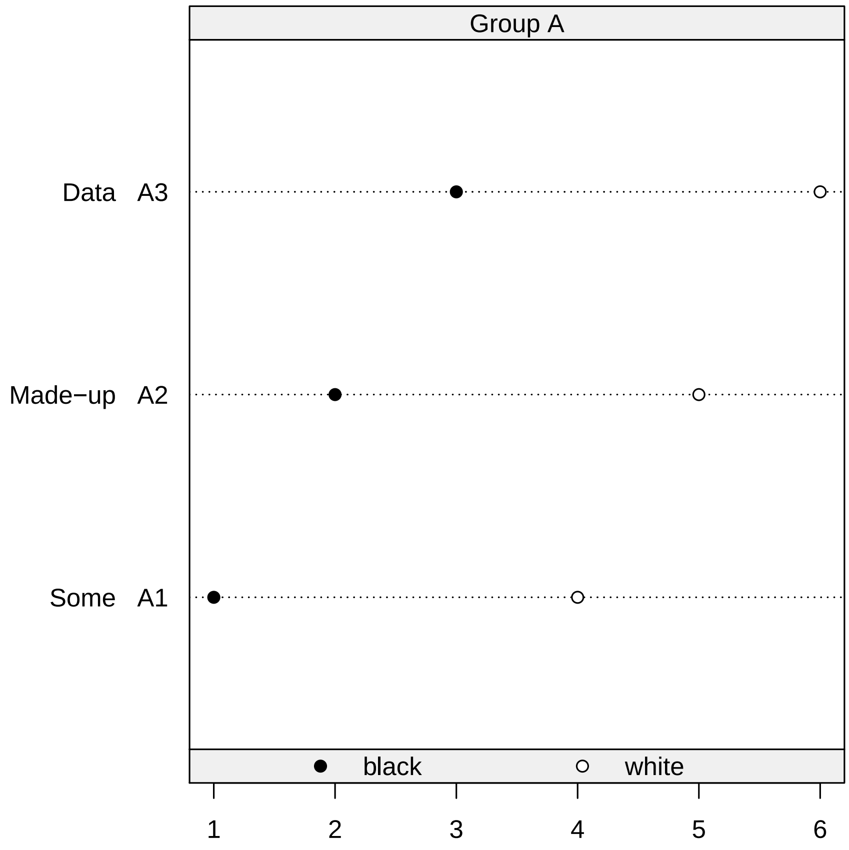

Figure 3: Another example to demonstrate what `datlist` looks like. Here we use a separate `textlist` to specify more than 1 column of text labels.

Optional Arguments

`at` - An axis specification argument, similar in working to `axis`. Specifying `at` will result in this being passed to both `at1` and `at3` in `parslist` (while specifying `at1` will change the `at` attribute for axis 1 only). The same thing occurs for `atsmall` and `atlabels`. This makes it easier to specify both axes at the same time, while also retaining the capability for individual specification.

`col` - Specifying `col` will result in this colour being passed to `pbg` (point background colour), which is the most likely candidate to change if the user chooses to specify some custom colour.

`xlab` - `xlab` is passed on to `lab1` or `lab3` based on whether that axis has been specified in `axes`. This is different from manually specifying `lab1` or `lab3`, which will result in that label always being plotted, even if the corresponding axis is not specified in `axes`.

Further optional arguments can be specified either by using a different `parslist` or by passing it through via '`...`'.

2.1 The Parslist Function

This subsection details the `parslist` function, which defines the default parameters for many of the optional arguments for `dotchartplus`. It is also the natural place to discuss these optional arguments.

In most cases, users will want to use the default values, but the option is given for those who want it. Optional arguments can be specified either by using a different `parslist` or by passing these through via '`...`'. A different `parslist` can be constructed either by using this function (specifying which arguments you wish to change), or by using the `parslist` returned from a call to `dotchartplus` (which will incorporate all the changes specified, including those made via '`...`'). In general, it is easiest to simply pass any optional arguments you require through '`...`'.

It is possible to replace the provided `DefaultParslist` with one you create, which will serve to change the default values of all subsequent dotcharts created via `dotchartplus`.

6 ⟨*Parslist Function* 6⟩≡

```
dcpParslist =
  function(cex = c(0.6, 1.1), highlight = NULL,

           widths = NULL, heights = NULL, labwidths = NULL,
           fpad = NULL, pad = c(0, 1), padmar = NULL,
           border = 0.5, newlayout = TRUE,

           axes = c(1, 2),
           lab1 = NULL, lab2 = NULL, lab3 = NULL, lab4 = NULL,
           labcexmult = 1, percentile = FALSE,
           xlim = NULL, xaxs = "r",
           at1 = NULL, atsmall1 = NULL, atlabels1 = NULL,
           at3 = NULL, atsmall3 = NULL, atlabels3 = NULL,
```

```
            main = NULL, maincexmult = 1.5,
            grouplabel = NULL, grouplabcexmult = 1,
            grouplabadj = 0.5, grouplabbg = "#F0F0F0",
            grouplabcol = 1, grouplabfont = 1,
            setslabel = NULL, setslabcexmult = 1,

            pfunc = points, pbg = c("white", "black"),
            pch = c(21, 21, 24, 24), pcol = 1,
            adj = 0.5, fcol = 1, font = 1:4,
            full.lines = NULL, lfunc = segments,
            lcol = 1, lty = 3, lwd = 1)
    as.list(environment())
  DefaultParslist = dcpParslist()
```
Defines:
  `DefaultParslist`, used in chunk 3.
Uses `fpad` 41, `heights` 41, `labwidths` 41, `padmar` 41, `widths` 41, and `xlim` 37b.

2.1.1 Plot related arguments

`cex` specifies character expansion. Similar in usage to other R plot functions. However, `dotchartplus` also allows specification of a range (min, max) within which the function will choose the 'best' cex to make most use of the (vertical) space. Currently, this calculation only occurs at initial plot call, and recalculations do not occur on resizing (although the user can always re-plot after resizing).

`highlight` specifies which points to highlight. A vector specifying the indices of the values to highlight. Where `datlist` contains more than one group, `highlight` can also be a `list` to individually specify highlighting indices for each group. Where more than one highlight method is desired, one can specify a matrix rather than a vector of indices. Each column of the matrix is considered a vector of indices, with the column determining the highlight method. For interaction with superpositioned points and the graphical arguments, refer to Subsection 3.3.

2.1.2 Layout related arguments

`widths` specify `widths` of the plot region. This excludes regions set aside for axes, labels or padding. Similar in usage to `layout`.

`heights` specify `heights` of the plot region. This excludes regions set aside for axes, labels or padding. Similar in usage to `layout`.

`labwidths` *For advances users only.* Specifies the `widths` of the text label region. By default this will automatically be calculated to fit the text labels exactly with slight padding, so specifying one manually is not recommended.

`fpad` specifies the 'slight padding' used in `labwidths`. The default value is the width of the character 'm', which will vary as `cex` varies.

`pad` specifies padding rows and columns between each panel of the plot region. Same usage as in `LHdefault` (see `layouthelper.pdf`).

`padmar` specifies the margins of the padding rows and columns of `pad`. The default value is the width of the character 'm'.

`border` specifies the thickness in cm of the border to be placed around the dotchart. Can be a vector of up to length 4, to individually specify the thickness on the 4 sides individually. This should be `numeric` and not the `character` result of `lcm`.

`newlayout` *For advances users only.* Specifies whether a new layout should be formed. If `FALSE`, layout creation will be skipped and the function will begin plotting on whatever layout is currently in place.

2.1.3 Axis related arguments (including axis labels)

`axes` specifies which axis should be drawn. Axis 1 and 3 are the numeric axes, while axis 2 and 4 are the text labels axes. The function creates a layout appropriate for the current plot, hence altering which `axes` to draw will alter the layout and appearance (mainly aspect ratio) of the plot.

`lab`n specifies the label to plot for the axis specified (i.e. `lab1` is the label for axis 1, `lab2` is for axis 2, etc). Unlike specifying `xlab`, specifying one of the `lab`n arguments will always cause the label to be plotted.

`labcexmult - cex` multiplier for the `lab`n.

`percentile - logical` vector of length 1 specifying if a `percentile` axis should be plotted. This is only appropriate if the data is sorted in ascending order. The `percentile` axis is always plotted on axis 4.

`xlim - numeric` vector of length 2 giving the x coordinate range. By default, the function will grab the range of the data. However `dotchartplus` allows for a `list` to be specified as `xlim`, which results in multiple panels being drawn, each with the `xlim` specified in the matching element of the `list`. This is usually used to have a 'jump' in the x axis.

`xaxs -` the style of axis interval calculation to be used for the x axis. See `par`.

`at`n specifies the position of the tick marks for the axis specified. Only appropriate for axis 1 and 3. See `axis`. As `dotchartplus` allows for a `list` to be specified for `xlim`, it also allows `at`n (along with `atsmall`n and `atlabels`n) to be specified as a `list` to allow axis specification for each `xlim`.

`atsmall`n specifies the position of the small ('minor') tick marks. Small ticks do not have labels.

`atlabels`n specifies the labels for the tick marks corresponding to `at`n. See `axis`.

2.1.4 Extra Labels related arguments

`main` specifies a main title. Works as you would expect.

`maincexmult - cex` multiplier for the main title.

`grouplabel -` (Default `NULL`). If `datlist` contains names for its groups (list elements), these are automatically assigned to `grouplabels`. Otherwise, no group label is plotted. If `TRUE` will always cause a group label to be generated, by assigning a letter of the alphabet to each group. If `FALSE` will always cause the group label to NOT be plotted. Alternatively, can be a `character` vector specifying the group labels to plot. This must be the same length as the number of groups in the `datlist`.

`grouplabcexmult - cex` multiplier for the group labels.

`grouplabadj -` the `adj` to be used for the group labels. Takes a single number between 0 and 1, with 0 meaning draw flush with the left edge of the panel left justified, and 1 meaning the same thing but to the right.

`grouplabbg -` the `bg` colour for the group label panel. Used mainly to clearly distinguish the group label panel from the plotting regions.

`grouplabcol` - the colour of the group labels.

`grouplabfont` - the `font` of the group labels. See `par`.

`setslabel` - (Default `NULL`). If there is more than one set of data and the elements of `datlist` contain `colnames`, these are automatically assigned to `setslabel`. Alternatives are the same as in `grouplabel`.

`setslabcexmult` - `cex` multiplier for the sets labels.

2.1.5 Graphical arguments

Points

`pfunc` *For advances users only.* The function to use for drawing the points. By default, this is `points`, but custom functions can give greater choice.

`pbg` - the 'background' colour of the points. This usually means the interior of the points. Only valid for certain `pch`.

`pch` - the point type. See `par`.

`pcol` - the outline colour of the points.

Text Label

`adj` - text label justification. Takes a single number between 0 and 1, with 0 meaning draw flush with the left edge of the panel left justified, and 1 meaning the same thing but to the right. Where there are multiple columns of text labels, `adj` can be a `vector` to individually specify `adj` for each column.

`fcol` - the colour of the text.

`font` - the format of the text. 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic.

Lines

`full.lines` specifies whether the dotted lines joining the text labels to the points should end at the points (`FALSE`) or go from edge to edge (`TRUE`). By default (`NULL`) the function checks if `xlim` contains 0. If it does, then `full.lines` is set to `FALSE`, otherwise it is set to `TRUE`.

`lfunc` *For advances users only.* The function to use for drawing the lines. By default, this is `segments`, but custom functions can give greater choice.

`lcol` - the colour of the lines.

`lty` - the type of line. See `par`.

`lwd` - the line width. See `par`.

3 Auxiliary Functions

There are three *Primary* Auxiliary Functions: Layout, Plot and Expandpars. Layout is responsible for setting up an appropriate layout and Plot is responsible for plotting everything. In comparison Expandpars is a short and simple function, but it handles all graphical parameter related work, including how sets of data are distinguished and how points are highlighted, making it important. *Secondary* Auxiliaries collect all remaining supporting functions.

Note that in addition to these Auxiliary Functions, `dotchartplus` also requires the Layout Helper Library to work.

11a ⟨*Auxiliary Functions* 11a⟩≡

```
⟨Layout Aux 11b⟩
⟨Plot Aux 20d⟩
⟨Expandpars Aux 31⟩
⟨Secondary Auxiliaries 32⟩
```

3.1 The Layout Auxiliary

The entire function is contained within a `with` call, to allow the objects in `parslist` to be visible.

11b ⟨*Layout Aux* 11b⟩≡

```
dcpLayout =
  function(parslist)
  with(parslist, {
    axes = layoutaxes
    ⟨form plot region 14⟩
    ⟨attach axes 16⟩
    ⟨attach labels 18⟩
    ⟨attach main title area 20a⟩
    ⟨add border 20b⟩
    ⟨call layout 20c⟩
    laymat
  })
```

Defines:
`dcpLayout`, used in chunk 43a.
Uses `laymat` 14 and `parslist` 34.

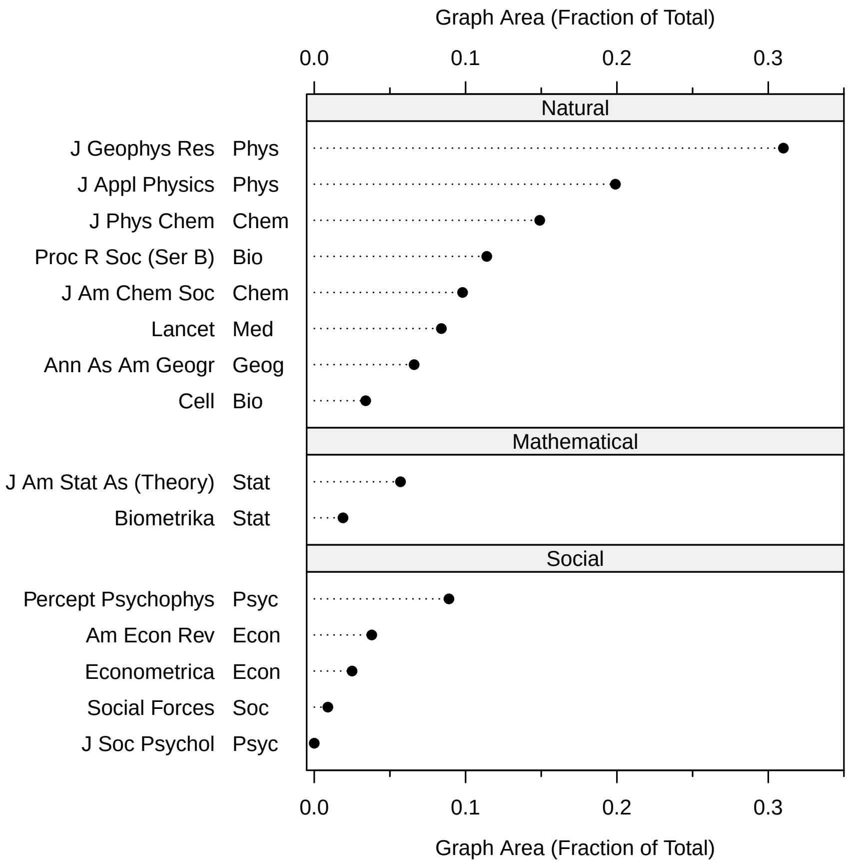

Figure 4: A reproduction of Figure 3.26 in The Elements of Graphing Data (Cleveland, 1985, p145). There are some minor differences, most notably in the positioning of the group labels. This dotchart showcases many features of `dotchartplus` and will be used alongside the code to graphically explain the purpose of the code.

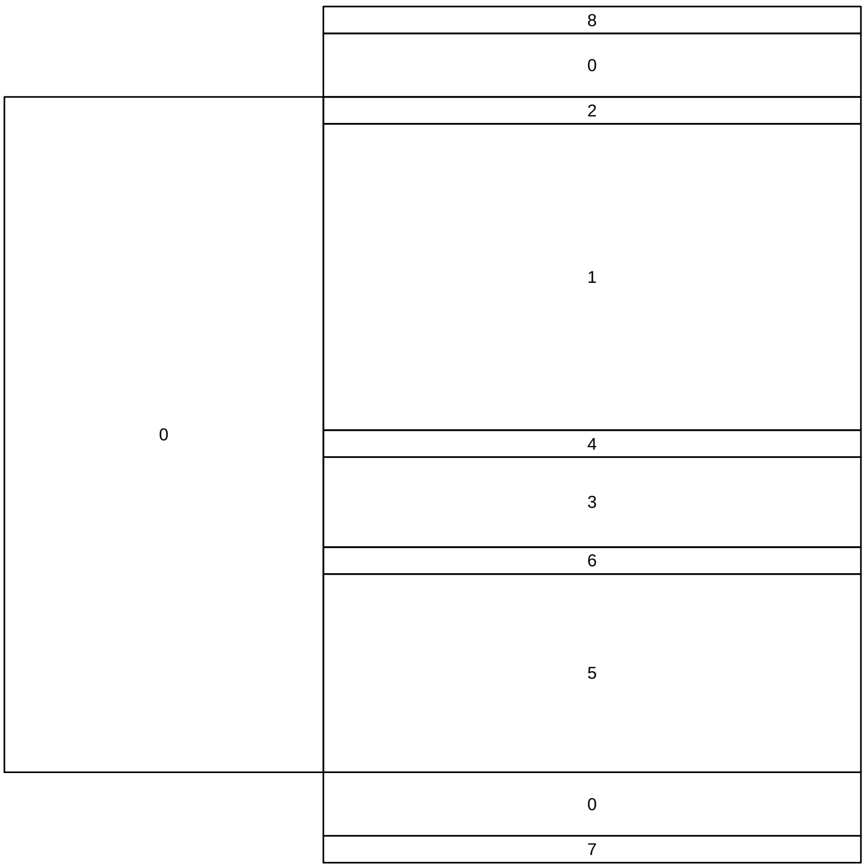

Figure 5: A representation of the layout used to create Figure 4. Note that a modified form of `layout.show` was used to show certain 0 regions, as these spaces will be used to plot the axes using `axis` and `mtext`.

3.1.1 Forming the Plot Region

If there is no `grouplabel` specified, this is a straight-forward call to `LHdefault`. If there is a `grouplabel`, we must first create a single 'panel', which we call `vlay`. This panel sets up the main plotting region and the group label region, and while this is of a fixed form in the code (plot region with a single label region above), because of the modular fashion in which layouts are built with the *Layout Helper*, it can be more complex if desired.

We set the `heights` to be an identifying `character` vector, which we use to replace the plot region `heights` and group label region `heights` separately after replication.

Once we have our completed layout, we need to check if we need a `setslabel` and attach such a region.

Finally, we update the `metamat` (meta matrix) which is a 2×3 matrix which contains the following information regarding the `laymat`:

$$\begin{bmatrix} \text{Rows above plot region} & \text{Plot region rows} & \text{Rows below plot region} \\ \text{Cols above plot region} & \text{Plot region cols} & \text{Cols below plot region} \end{bmatrix}$$

This information is useful for when we start attaching labels.

14 ⟨*form plot region* 14⟩≡

```
if(is.null(grouplabel)){
  laymat = LHdefault(udim, pad = pad, padmar = padmar,
    widths = widths, heights = heights)
  } else{
    vlay = LHdefault(c(2, 1), widths = 1,
      heights = c("plot", "grouplab"), reverse = TRUE)
    ulay = rep(vlay, udim[1], udim[2], pad = pad,
      padmar = padmar)
    uhei = gethei(ulay)
    uhei[uhei == "plot"] = heights
    uhei[uhei == "grouplab"] =
      llines(nlines(grouplabel) + 0.1, cex = grouplabcexmult)
    laymat = newlayout(getmat(ulay), widths, uhei)
  }
if(!is.null(setslabel))
  laymat = rbind(laymat, LHdefault(fino = getfino(laymat),
    heights = llines(nlines(setslabel) + 0.1,
      cex = setslabcexmult)))
metamat = matrix(c(0, 0, dim(getmat(laymat)), 0, 0), nrow = 2)
```

Defines:
`laymat`, used in chunks 11b, 16, 18, 20, and 43.
`metamat`, used in chunks 16 and 18.

Uses `heights` 41, `llines` 32, `nlines` 32, `padmar` 41, `udim` 39b, and `widths` 41.

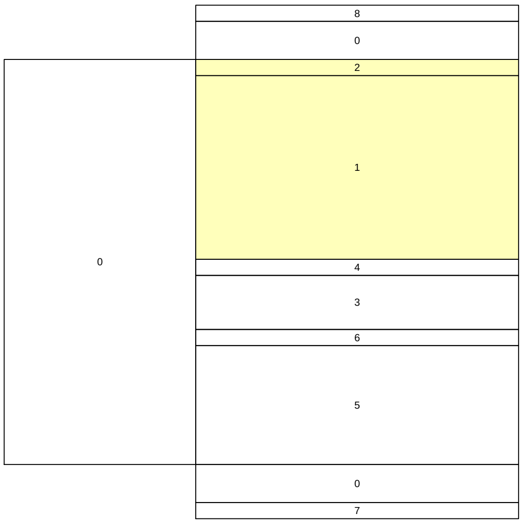

Figure 6: To create the main plot area with group labels, we first create a single 'panel' (called `vlay` in the code), which consists of the actual plot area and a group label area above it. This is shown as the highlighted area in our figure.

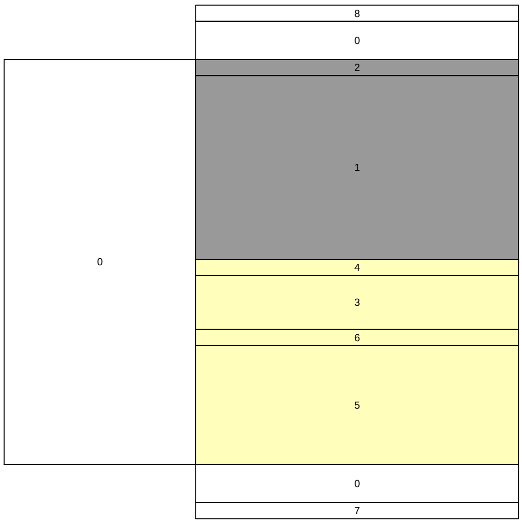

Figure 7: Once a single 'panel' is created, it's easy to replicate this to the required dimensions (3 in our case) using the Layout Helper library's `rep` method for layout objects. The darker highlighted area represents what we have (`vlay`), while the lighter highlighted areas represent what `rep` will add (together, the highlighted areas represent our current layout). Once we have created the desired `ulay`, we have one more step, which is to add the correct `heights`. This completes our `laymat` for the plot region.

3.1.2 Attaching Axes Regions

For the attaching of regions for the axes, we simply bind a 1×1 layout matrix numbered 0 with either `widths` or `heights` set to the computed `axiswidhei`. These attached regions are essentially white space of the appropriate size, within which we can draw our axes. For each axis attached, we also update `metamat` for later use.

16 ⟨*attach axes* 16⟩≡

```
if(any(axes == 1)){
  laymat = rbind(laymat,
    LHdefault(fino = 0, heights = axiswidhei[1]))
  metamat[1, 3] = metamat[1, 3] + 1
}
if(any(axes == 2)){
  laymat = cbind(LHdefault(fino = 0,
    widths = axiswidhei[2]), laymat)
  metamat[2, 1] = metamat[2, 1] + 1
}
if(any(axes == 3)){
  laymat = rbind(LHdefault(fino = 0,
    heights = axiswidhei[3]), laymat)
  metamat[1, 1] = metamat[1, 1] + 1
}
if(any(axes == 4)){
  laymat = cbind(laymat,
    LHdefault(fino = 0, widths = axiswidhei[4]))
  metamat[2, 3] = metamat[2, 3] + 1
}
```

Uses `axiswidhei` 42b, `heights` 41, `laymat` 14, `metamat` 14, and `widths` 41.

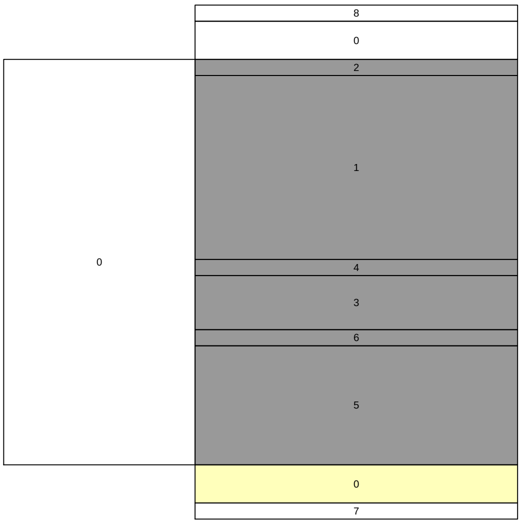

Figure 8: Attaching an axis area is easy, the bind methods for layout objects automatically stretch as necessary, so we can simply bind a 1×1 layout object with the correct `widths` or `height`, without having to factor in the dimensions of our current layout.

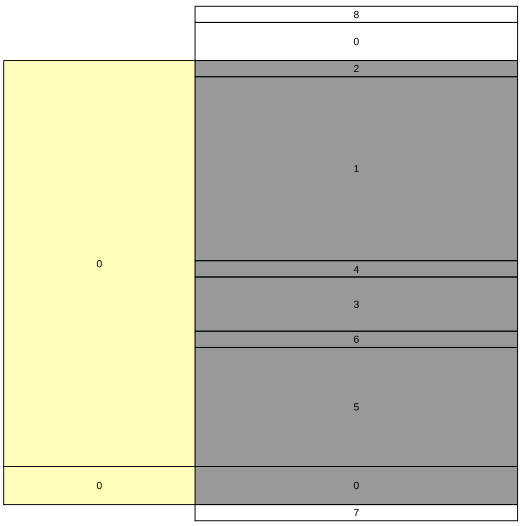

Figure 9: The same process as before works for attaching further axis regions, again because the bind methods automatically stretch to the required dimensions.

3.1.3 Attaching Label Regions

As the attachment of label regions require long-ish calls but are quite similar, we define a function that does it. The function takes three arguments, the label (`lab1`, `lab2`, `lab3` or `lab4`), whether this label is a column vector (2 and 4 are, 1 and 3 are not) and whether to reverse the argument list. The default order has `laymat` first, meaning the label will be attached to either the bottom or to the right, so we need to reverse the argument list for labels 2 and 3.

The label region itself is created using `LHlabels`. For precise mechanics, refer to the Layout Helper document. Essentially, we want our labels to be centred on the plot region only, and not over the axis regions as well. Thus, we use the information in the `metamat` to create an appropriate vector that is numbered for the part that attaches to the plot region, and 0 for the part that attaches to the axis regions. We also assign an appropriate `widths` or `heights` based on the the lines of text in our label.

18 ⟨*attach labels* 18⟩≡

```
attachlabelsf = function(labn, colvec, reverse){
  arglist = list(laymat,
    LHlabels(c(0, getfino(laymat), 0),
             metamat[2 - as.numeric(colvec),],
             colvec = colvec,
             widhei = llines(nlines(labn) + 0.1,
               cex = labcexmult)))
  if(reverse) arglist = rev(arglist)
  do.call(if(colvec) "cbind" else "rbind", arglist)
}
if(!is.null(lab1)){
  laymat = attachlabelsf(lab1, FALSE, FALSE)
  metamat[1, 3] = metamat[1, 3] + 1
}
if(!is.null(lab2)){
  laymat = attachlabelsf(lab2, TRUE, TRUE)
  metamat[2, 1] = metamat[2, 1] + 1
}
if(!is.null(lab3)){
  laymat = attachlabelsf(lab3, FALSE, TRUE)
  metamat[1, 1] = metamat[1, 1] + 1
}
if(!is.null(lab4)){
  laymat = attachlabelsf(lab4, TRUE, FALSE)
  metamat[2, 3] = metamat[2, 3] + 1
}
```

Uses `laymat` 14, `llines` 32, `metamat` 14, and `nlines` 32.

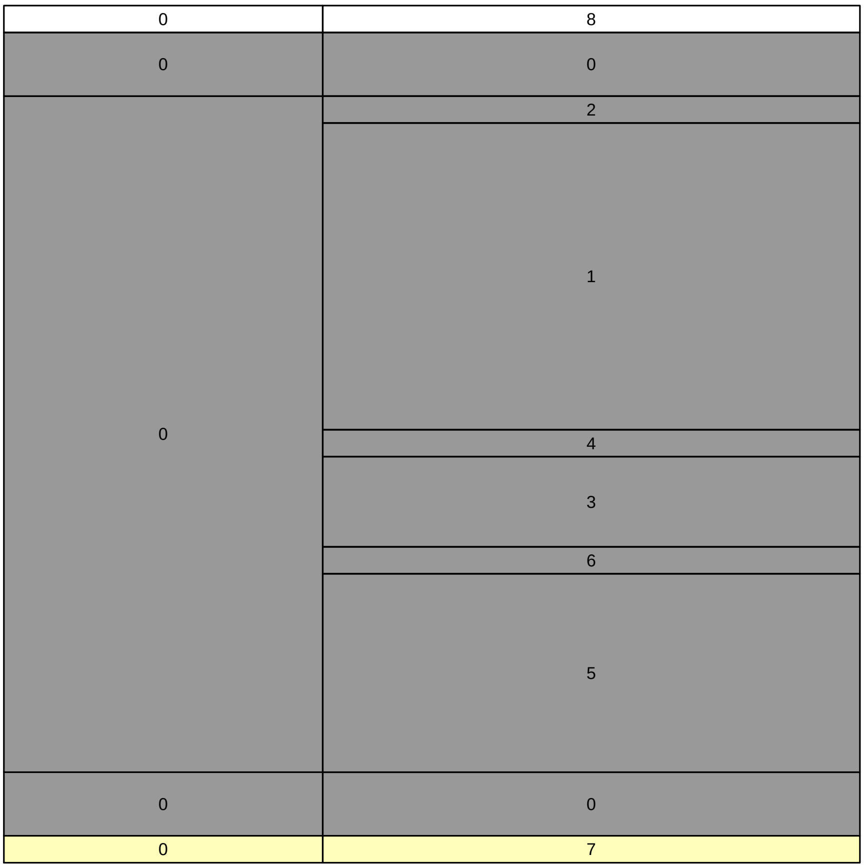

Figure 10: Attaching the label regions is slightly more complex than the axis regions because we can't simply stretch to fit. We require our labels to be centred on the plot region, and a simple stretch would in fact centre it over both the plot and the axis region.

3.1.4 Finishing off the Layout

If some `main` is supplied, attach an appropriate region to the top of the layout.

20a ⟨*attach main title area* 20a⟩≡

```
if(!is.null(main))
  laymat = rbind(LHdefault(fino = getfino(laymat),
    heights = llines(nlines(main) + 0.1, cex = maincexmult)),
    laymat)
```

Uses `heights` 41, `laymat` 14, `llines` 32, and `nlines` 32.

If some positive `border` is supplied, add this around our layout.

20b ⟨*add border* 20b⟩≡

```
if(any(border > 0))
  laymat = LHborder(laymat, border)
```

Uses `laymat` 14.

Finally we make a call to `layout` via the Layout Helper function for convenience. We supply the `cex` to revert the automatic change `layout` will make (see `help(layout)` for more details).

20c ⟨*call layout* 20c⟩≡

```
LHcall(laymat, cex)
```

Uses `laymat` 14.

3.2 The Plot Auxiliary

The entire function is contained within a `with` call, to allow the objects in `parslist` to be visible.

The function cycles through each 'panel' and calls `plotloop`. Once all the panels have been plotted, the remaining extra labels are drawn to finish off the plotting.

20d ⟨*Plot Aux* 20d⟩≡

```
dcpPlot =
  function(datlist, textlist, parslist)
  with(parslist, {
    ⟨plotloop 21a⟩
    for(colj in 1:udim[2])
      for(rowi in 1:udim[1])
        plotloop(rowi, colj)
    ⟨plot setslabel 29⟩
    ⟨plot xlab and main 30⟩
  })
```

Defines:
`dcpPlot`, used in chunk 43b.
Uses `parslist` 34, `plotloop` 21a, `textlist` 37a, and `udim` 39b.

The `plotloop` sub-function does most of the heavy-lifting. It is defined within the function and within the `with` block, giving access to all the required variables without having to pass it through as arguments. The only required arguments are `rowi` and `colj` which specify which row and column of the plotting region we are currently plotting in, and hence which data to plot.

21a ⟨*plotloop* 21a⟩≡

```
plotloop =
  function(rowi, colj){
    ⟨setup plot area 21b⟩
    ⟨form graphpars 21c⟩
    ⟨plot points and lines 22⟩
    ⟨plot numerical axis 24a⟩
    ⟨plot percentile axis 24b⟩
    ⟨plot text axis 26⟩
    ⟨plot grouplabel 28b⟩
  }
```

Defines:
`plotloop`, used in chunk 20d.

3.2.1 Setting up the Plot Area

As `xlim` and `ylim` are `list` objects, we grab the appropriate one for this plot.

We also compute `x` and `y` here, as they will be used for many of the subsequent processes.

21b ⟨*setup plot area* 21b⟩≡

```
plot.new()
plot.window(xlim = xlim[[colj]], ylim = ylim[[rowi]],
            xaxs = xaxs, yaxs = "i")
box()
x = datlist[[rowi]]
datlen = nrow(x)
datcol = ncol(x)
y = 1:datlen
```

Defines:
`y`, used in chunks 22, 24b, and 26.
`datlen`, used in chunks 21c, 24b, and 26.
`datcol`, used in chunks 21c and 22.
Uses `xlim` 37b and `ylim` 38c.

We call `expandpars` to expand the required graphical parameters to the vectorised form that we can use.

21c ⟨*form graphpars* 21c⟩≡

```
graphpars = expandpars(parslist[c("fcol", "font",
  "pbg", "pch", "pcol", "lcol", "lty", "lwd")], datlen,
  datcol, highlight[[rowi]])
```

Uses `datcol` 21b, `datlen` 21b, `expandpars` 31, and `parslist` 34.

3.2.2 Drawing the Points and Lines

Our data, `x`, is a matrix, with the columns indicating *sets*. We desire to draw all *sets* in a single vectorised call, so we create a vector version of `x`, `xv`, and likewise we create a `yv` that matches the length of `xv`.

We also need to create two new vectors to be used as the start and end points of the lines. These values depend on whether full lines are specified or not.

If `full.lines = TRUE`, the the lines will all stretch from the left edge of the plot region to the right edge. We can obtain these points by calling `par("usr")`.

If `full.lines = FALSE`, we set the lines to start from 0, and to end at `xv`. We fix the start a 0 so that the length of the line has meaning (length corresponds to actual value).

We now have our vectors in the required form, so it is simple to make a single call to `lfunc` to draw our lines, and `pfunc` to draw our points.

22 ⟨*plot points and lines* 22⟩≡

```
xv = as.vector(x)
yv = rep(y, datcol)
if(full.lines){
  curusr = par("usr")
  xstart = curusr[1]
  xend = curusr[2]
} else{
  xstart = 0
  xend = xv
}

lfunc(xstart, yv, xend, yv,
         col = graphpars$lcol,
         lty = graphpars$lty,
         lwd = graphpars$lwd)
pfunc(xv, yv,
       bg = graphpars$pbg,
       pch = graphpars$pch,
       col = graphpars$pcol)
```

Uses `datcol` 21b and `y` 21b.

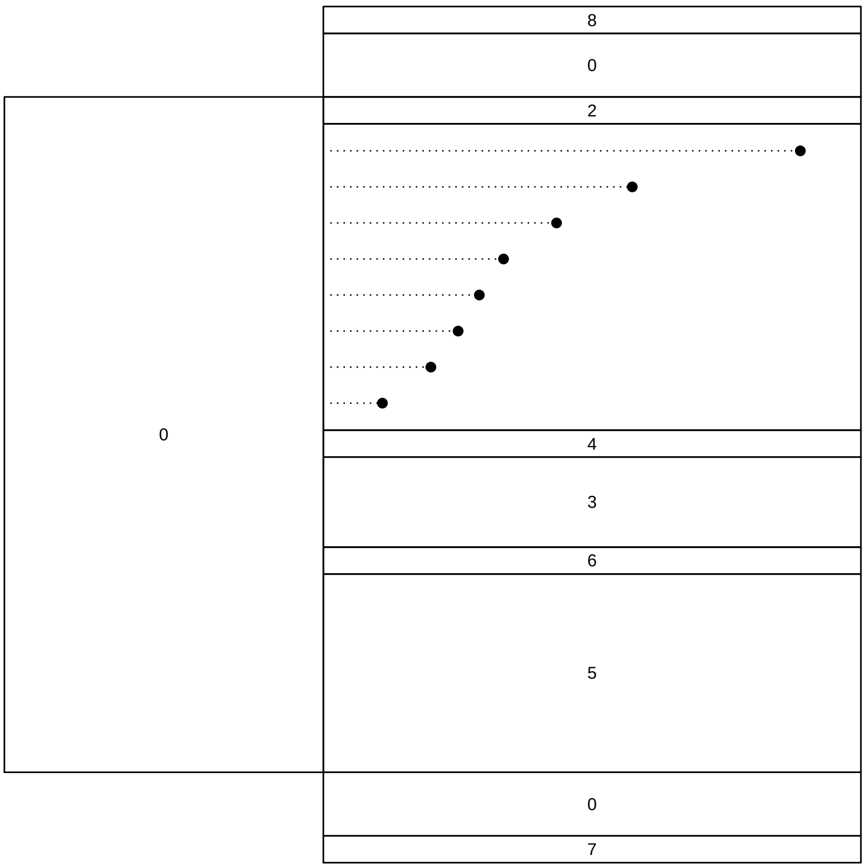

Figure 11: The plot so far. We have only drawn the points and dotted lines, but the layout has also been drawn to some extent for reference. In truth, this page should be blank except for the points and dotted lines.

3.2.3 Drawing the Numerical Axes

The general idea is simple. We check if the particular axis is specified in `axes`, and whether where we're currently plotting is at the very top (`rowi = 1`) or at the very bottom (`rowi = udim[1]`). We use `axis.cus` to allow us to plot small ticks as well. Complications can arise if we have a region set aside for either the group or sets labels. In this case, we must compute the height of the group label region (`labjump`) and specify the `line` argument so the axis can jump over the label region and plot in the right place.

The axis labels are plotted separately under 'plot xlab and main', as those labels go in their own panel and are not done using `mtext` (mainly for centering reasons).

24a ⟨*plot numerical axis* 24a⟩≡

```
grouplabjump =
  if(is.null(grouplabel)) NA
  else (nlines(grouplabel) + 0.1) * grouplabcexmult
setslabjump =
  if(is.null(setslabel)) NA
  else (nlines(setslabel) + 0.1) * setslabcexmult
if(any(axes == 1) && rowi == udim[1])
  axis.cus(1, at1[[colj]], atsmall1[[colj]], atlabels1[[colj]],
           line = setslabjump)
if(any(axes == 3) && rowi == 1)
  axis.cus(3, at3[[colj]], atsmall3[[colj]], atlabels3[[colj]],
           line = grouplabjump)
```

Uses `axis.cus` 32, `nlines` 32, and `udim` 39b.

The 'percentile' axis is special kind of numerical axis. The number of percentile tick marks is based on the size of the data. We can use `quantile` to determine the actual values to place the tick marks.

24b ⟨*plot percentile axis* 24b⟩≡

```
if(percentile == TRUE && colj == udim[2]){
  pvaln = c(1, 2, 4, 5)[datlen < c(10, 20, 30, Inf)][1]
  pvallabels = (0:pvaln)/pvaln
  pvalat = quantile(y, pvallabels)
  axis.cus(4, pvalat, NULL, pvallabels * 100)
}
```

Uses `axis.cus` 32, `datlen` 21b, `udim` 39b, and `y` 21b.

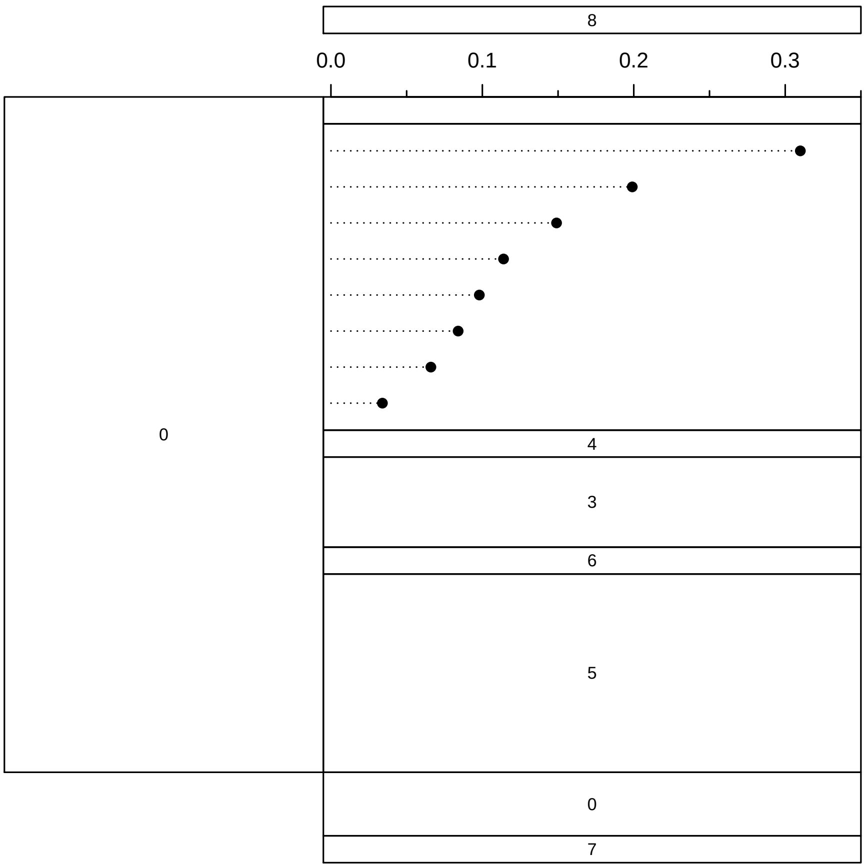

Figure 12: The plot so far. We have drawn the points, dotted lines and the numerical axis. Because there are group labels, we had to 'jump' over it when plotting the numerical axis.

3.2.4 Drawing the Text Labels Axes

We draw these labels using `mtext`, which is intended for text parallel with the plot edge, however in our case the orientation will be perpendicular, which will make things a little complicated. Further complicating matters, we can have columns of text that can be given different justification via `adj`. These all lead to an involved `txtjump` calculation which is covered in the next code chunk.

Once we know `txtjump`, the actual call is simple. As with the numerical axes above, we check if the axis is requested and whether we're currently plotting in an appropriate region. Then we loop through each column of the text labels to plot.

For axis 2 (the left axis), our `txtjump` requires a slight adjustment. As the natural reading direction is left-to-right, the `txtjump` value is also from left-to-right. However, for axis 2, we require a jump from right-to-left. Thus, we first jump to the left edge of the text axis region (`sum(labwidths) + fpad`), then take away our `txtjump` value.

That done, the remaining arguments to `mtext` should be self explanatory.

26 ⟨*plot text axis* 26⟩≡

```
⟨txtjump computation 28a⟩
txt = textlist[[rowi]]
if(any(axes == 2) && colj == 1)
  for(txtcol in 1:ncol(txt)){
    txtjump = sum(lcmTOlines(sumlcm(labwidths, fpad))) -
      txtjumpf(txtcol)
    mtext(txt[,txtcol], 2, txtjump, at = y,
          adj = adj[txtcol], cex = cex, las = 2,
          col = graphpars$fcol[1:datlen],
          font = graphpars$font[1:datlen])
  }
if(any(axes == 4) && colj == udim[2])
  for(txtcol in 1:ncol(txt)){
    mtext(txt[,txtcol], 4, txtjumpf(txtcol), at = y,
          adj = adj[txtcol], cex = cex, las = 2,
          col = graphpars$fcol[1:datlen],
          font = graphpars$font[1:datlen])
  }
```

Uses `datlen` 21b, `fpad` 41, `labwidths` 41, `lcmTOlines` 32, `sumlcm` 32, `textlist` 37a, `txtjumpf` 28a, `udim` 39b, and `y` 21b.

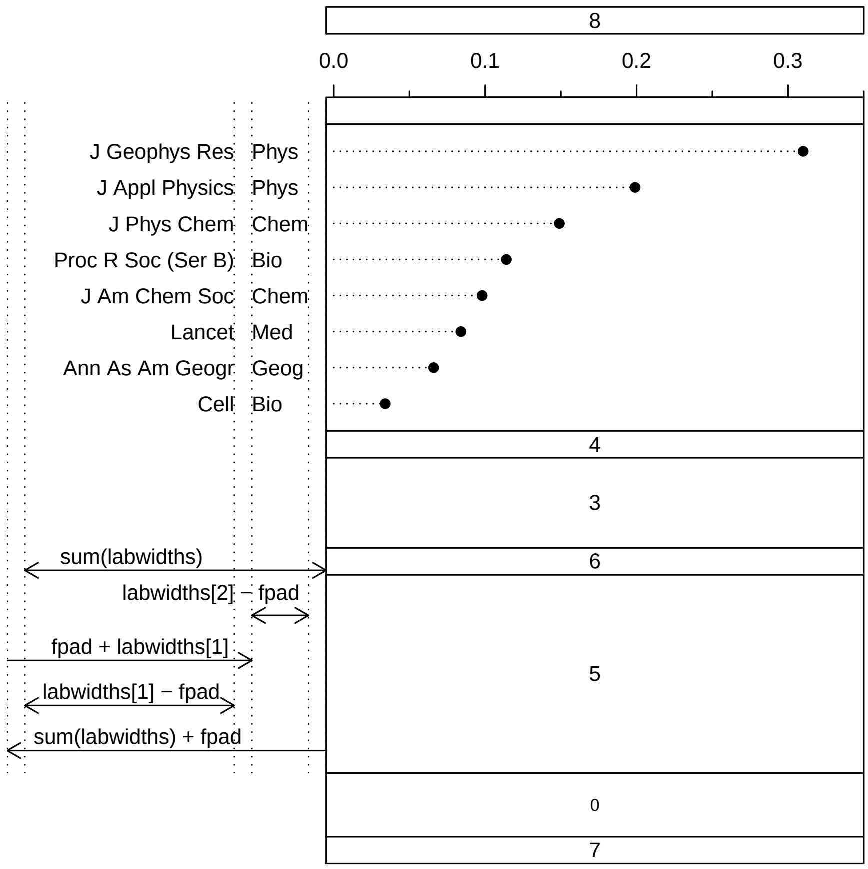

Figure 13: The plot so far. We have drawn the points, dotted lines, numerical axis and text labels. This figure also includes some visual aids to better understand how the text label area is divided into columns for the text.

It should be noted that most of our computed units are stored in cm, while `mtext` shifting works in terms of lines of text. Thus, we make use of `lcmTOlines` to convert cm values to lines of text.

A reminder, `labwidths` is a vector containing the largest width of each column of text, plus an `fpad` for padding. The area set aside for the text labels is the sum of all the `labwidths` plus an additional `fpad`. Natural reading direction is left-to-right, so `txtjumpf` is natural for axis 4. The computation is broken down thus:

- The first line is to jump `fpad`.
- The second line is to jump over all previous column widths.
- The third line is the current column width, less the padding, which is multiplied by...
- The fourth line, which is the `adj` for the current column.

These combined give us the lines of text to jump from left-to-right. As mentioned above, this can be used directly for axis 4, but requires a little bit of extra work for axis 2.

28a ⟨*txtjump computation* 28a⟩≡

```
txtjumpf = function(txtcol){
  lcmTOlines(fpad) +
    sum(lcmTOlines(labwidths[0:(txtcol - 1)])) +
      lcmTOlines(sumlcm(labwidths[txtcol], -fpad)) *
        adj[txtcol]
}
```

Defines:
`txtjumpf`, used in chunk 26.
Uses `fpad` 41, `labwidths` 41, `lcmTOlines` 32, and `sumlcm` 32.

3.2.5 Drawing the Extra Labels

As we have an area set aside for drawing group labels, it's simple to call `text.cus` to draw the thing.

28b ⟨*plot grouplabel* 28b⟩≡

```
if(!is.null(grouplabel)){
    text.cus(grouplabel[rowi],
             pos = c(grouplabadj, 0.5),
             col = grouplabcol,
             font = grouplabfont,
             cex = grouplabcexmult,
             bg = grouplabbg)
    box()
  }
```

Uses `text.cus` 32.

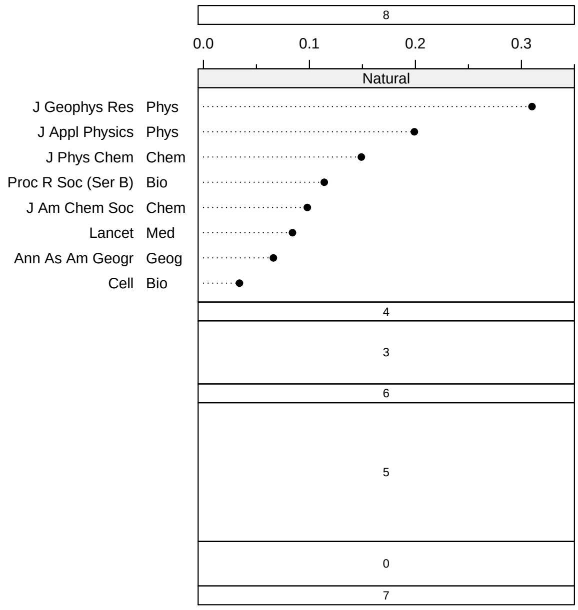

Figure 14: The plot so far. We have drawn the points, dotted lines, numerical axis, text labels and group label. We now need to loop through the remaining panels.

Drawing the sets labels is slightly harder. We need to grab what the points look like using `expandpars`, then draw the labels in an appropriately spaced manner across the plot.

29 ⟨*plot setslabel* 29⟩≡

```
if(!is.null(setslabel)){
  nsets = length(setslabel)
  graphpars = expandpars(parslist[c("pbg", "pch",
    "pcol")], 1, nsets)
  plot.new()
  plot.window(xlim = c(0.5, nsets + 1), ylim = c(0, 2),
              xaxs = "i", yaxs = "i")
  rect(0.5, 0, nsets + 1, 2, col = grouplabbg,
       border = NA)
  box()
  pchjump = strwidth("MM")
  for(i in 1:nsets){
    points(i, 1, bg = graphpars$pbg[i],
           col = graphpars$pcol[i], pch = graphpars$pch[i])
    text(i + pchjump, 1, setslabel[i], adj = 0,
         cex = setslabcexmult)
  }
}
```

Uses `expandpars` 31, `parslist` 34, `xlim` 37b, and `ylim` 38c.

Drawing of the ‘xlab’ and main title are also simple as we will have areas set aside for them. The vertical labels (2 and 4) are rotated 90 degrees in conventional R style.

30 ⟨*plot xlab and main* 30⟩≡
```
if(!is.null(lab1))
  text.cus(lab1, labcexmult)
if(!is.null(lab2))
  text.cus(lab2, labcexmult, srt = 90)
if(!is.null(lab3))
  text.cus(lab3, labcexmult)
if(!is.null(lab4))
  text.cus(lab4, labcexmult, srt = 90)
if(!is.null(main))
  text.cus(main, maincexmult)
```
Uses `text.cus` 32.

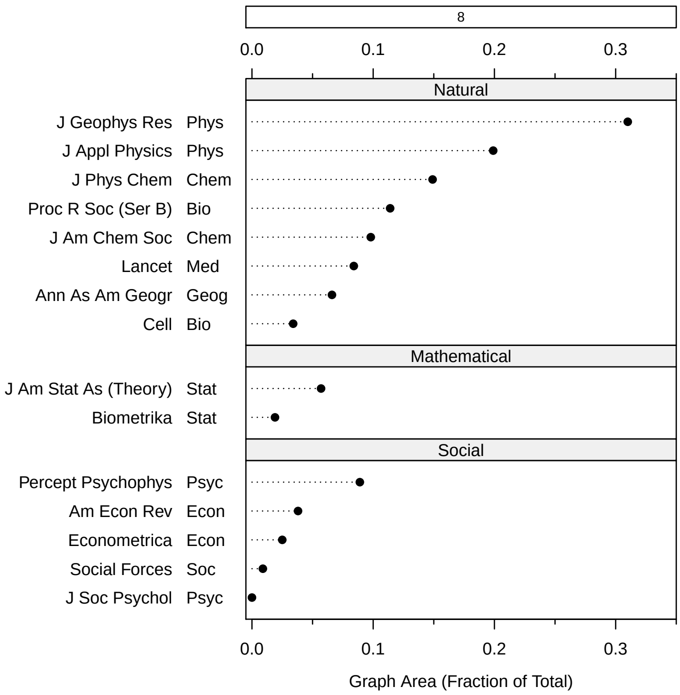

Figure 15: The plot so far. Having looped through all the panels, it’s now time to finish off the plot by drawing in the axis labels.

3.3 The Expandpars Auxiliary

The idea here is to consolidate the user arguments, specifically the graphical parameters, *sets* and highlighting, into something that can be used directly in a vectorised call to the points and lines functions.

The function will first assign the graphical parameters to the *sets* of data, then assign the remainder for use in highlighting. For instance, if there are *n sets* of data, the first *n* elements of the graphical parameters will be used to distinguish between the *sets* of data. These graphical parameters are grabbed using `lrow`, replicated to match the length of the data, then stored in a `data.frame`, which allows us to extract specific columns (which will correspond to a specific graphical parameter, e.g. `pch`), or specific rows (which will correspond to all the graphical parameters for a specific data point). Row operations using data frames in R are inefficient, but in our case the size of the data frame is small, so the convenience outweighs the inefficiency.

If we have no highlighting, our job ends here, as we now have the required vectorised form. Highlighting occurs by replacing the appropriate subset of the data frame with the graphical parameters assigned to the particular highlight method. As we have used the first *n* elements to distinguish between the *sets* of data, the *n + 1* element will be used for the first highlighting method (column 1 of the `highlight` matrix), the *n + 2* element will be used for the second highlighting method, and so on. This replacement uses the data frame row operations mentioned in the previous paragraph.

The function takes four arguments. The `parslist` argument is usually a subset of the full `parslist` containing only the relevant graphical parameters (e.g. `parslist[c("pbg", "pch", "pcol")]`). The next two arguments, `nrows` and `ncols`, are the rows and columns of the current *group* of data (`datlist[[i]]`), which will be a matrix. `nrows` can be considered to be the length of the data, while `ncols` is the number of *sets* of data in the current group. The last argument, `highsub` (which can be `NULL`), is the `highlight` specified for the current *group* (`highlight[[i]]`), and will also be a matrix.

31 ⟨*Expandpars Aux* 31⟩≡
```
expandpars =
  function(parslist, nrows = 1, ncols = 1, highsub = NULL){
    parsdf = data.frame(lapply(lrow(parslist, 1:ncols),
      function(x) rep(x, each = nrows)))
    if(!is.null(highsub))
      for(j in 1:ncol(highsub))
        for(i in 1:ncols){
          rowtochange = highsub[,j] +
            sign(highsub[,j]) * nrows * (i - 1)
          parsdf[rowtochange,] =
            data.frame(lrow(parslist, j * ncols + i))
          }
    parsdf
  }
```
Defines:
`expandpars`, used in chunks 21c, 29, and 42a.
Uses `lrow` 32 and `parslist` 34.

3.4 Secondary Auxiliaries

The *Secondary* Auxiliary Functions can also be called Trivial Auxiliaries. All the short and simple supporting functions are collected here.

`llines` - Calculates the height of `x` lines of text, expanded by `cex`, in cm.

`nlines` - Calculates the number of lines in the given character vector `x` (i.e. returns count of `"\n"` + 1).

`lcmTOlines` - A pseudo-inverse of `llines`. Converts x cm into a value relative to the height of lines of text.

`strwidth.cm` - Calculates the width of a character string `x`, with format `font` and expanded by `cex`, in cm.

`sumlcm` - Sums cm values (i.e. character strings created by `lcm`, of the form "x cm"), and outputs the sum in cm.

`axis.cus` - A custom `axis` function that can support the plotting of 'small' ticks (also known as 'minor' ticks) in addition to the regular ticks (also known as 'major' ticks). The small ticks always have `tcl` (tick size) half the size of the regular ticks, and never have labels. An example call is: `axis.cus(side = 1, at = 2 * (0:5), atsmall = 1:9)`, which will plot a regular ticks with default labels at 0, 2, ..., 10, and small ticks at 1, 3, ..., 9 (technically, small ticks are plotted at 1, 2, ..., 9, however, some of these will be hidden by the regular ticks).

`text.cus` - A custom `text` function, meant to be a 'high-level' text plotting function. The function will form a new plotting area, set the `x` and `y` limits to be exactly 0 and 1, then plot the text given at the position given, with character expansion `cex`. Thus, `pos` should be specify the `x` and `y` coordinates on a scale from 0 to 1, where (0, 0) specifies the lower-left corner, and (1, 1) specifies the upper-right.

`lrow` - A function to obtain the 'rows' of a list. In effect, it's like taking the row subset of a data.frame, but `lrow` features automatic recycling, and its output remains a list object. Without the recycling feature, `lrow(lobj, i)` is similar conceptually to `lobj[i,]`.

32 ⟨*Secondary Auxiliaries* 32⟩≡

```
llines = function(x, cex = 1)
  lcm(2.54 * cex * x * par("csi"))
nlines = function(x)
  nchar(x) - nchar(gsub("\n", "", x)) + 1
lcmTOlines = function(x, cex = 1)
  as.numeric(sub(" cm", "", x))/(2.54 * cex * par("csi"))
strwidth.cm =
  Vectorize(function(x, font = par("font"), cex = 1)
            2.54 * strwidth(x, "inches", font = font, cex = cex))
sumlcm = function(...)
  lcm(do.call(sum, lapply(list(...), function(x)
                          as.numeric(sub(" cm", "", x)))))
```

```
axis.cus =
  function(side, at = NULL, atsmall = NULL, labels = TRUE,
           tcl = par("tcl"), ...){
    if(!is.null(atsmall))
      axis(side, at = atsmall, labels = FALSE, tcl = tcl/2, ...)
    axis(side, at = at, labels = labels, tcl = tcl, ...)
  }
text.cus =
  function(txt, cex = 1, pos = c(0.5, 0.5), bg = NULL, ...){
    plot.new()
    plot.window(xlim = 0:1, ylim = 0:1, xaxs = "i", yaxs = "i")
    if(!is.null(bg))
      rect(0, 0, 1, 1, col = bg, border = NA)
    text(pos[1], pos[2], txt, adj = pos, cex = cex, ...)
  }
lrow =
  function(lobj, row)
  lapply(lobj, function(x) x[(row - 1) %% length(x) + 1])
```

Defines:
- `llines`, used in chunks 14, 18, 20a, and 42b.
- `nlines`, used in chunks 14, 18, 20a, and 24a.
- `lcmTOlines`, used in chunks 26 and 28a.
- `strwidth.cm`, used in chunks 41 and 42a.
- `sumlcm`, used in chunks 26, 28a, and 42b.
- `axis.cus`, used in chunk 24.
- `text.cus`, used in chunks 28b and 30.
- `lrow`, used in chunk 31.

Uses `xlim` 37b and `ylim` 38c.

4 The Back End

We place a document header at the top of the extracted code to encourage people to read the literate description.

33 ⟨*document header* 33⟩≡

```
##---------------------------------------
## The code in this .R file is machine generated.
## To understand the program, read the literate description
##  pdf rather than studying just the R code.
##---------------------------------------
```

4.1 Back End Computation

The following is a collection of the various preliminary computations that are required before the actual plotting of the dotchart. Here, cag stands for 'check and generate'.

34 ⟨*back end work* 34⟩≡

```
⟨pass on pars to parslist 35a⟩
⟨check and force datlist to matrix 35b⟩
⟨compute best cex 36a⟩
⟨save original par and opt 36b⟩
⟨cag textlist 37a⟩
parslist = with(parslist, {
  ⟨cag xlim 37b⟩
  ⟨cag fulllines 38a⟩
  ⟨cag axis at 38b⟩
  ⟨generate ylim 38c⟩
  ⟨cag adj 38d⟩
  ⟨check highlight 39a⟩
  ⟨compute udim 39b⟩
  ⟨adjust axes for percentile 39c⟩
  ⟨cag grouplabel 40a⟩
  ⟨cag setslabel 40b⟩
  ⟨cag widths and heights 41⟩
  ⟨layout required computations 42b⟩
  ⟨return updated parslist 42c⟩
})
```

Defines:
`parslist`, used in chunks 3, 11b, 20d, 21c, 29, 31, 35, 36, 42, and 43.

Uses `heights` 41, `textlist` 37a, `udim` 39b, `widths` 41, `xlim` 37b, and `ylim` 38c.

Optional arguments specified in '...' are passed through to `parslist` via name matching, except in the case of the 'special arguments', `at`, `atsmall`, `atlabels`, `col` and `xlab`, which are passed through based on the rules mentioned in Section 2.

35a ⟨*pass on pars to parslist* 35a⟩≡

```
if(!is.null(col))
  parslist$pbg = col
if(!is.null(at))
  for(subs in c("at1", "at3"))
    parslist[[subs]] = at
if(!is.null(atsmall))
  for(subs in c("atsmall1", "atsmall3"))
    parslist[[subs]] = atsmall
if(!is.null(atlabels))
  for(subs in c("atlabels1", "atlabels3"))
    parslist[[subs]] = atlabels
optpars = list(...)
for(i in 1:length(optpars))
  if(any(names(parslist) == names(optpars)[i]))
    parslist[names(optpars)[i]] = optpars[i]
if(!is.null(xlab)){
  if(any(parslist$axes == 1))
    parslist$lab1 = xlab
  if(any(parslist$axes == 3))
    parslist$lab3 = xlab
  }
```

Uses `parslist` 34.

The default `dotchartplus` can accept lists containing both vectors or matrices, but for processing purposes, we desire it to be all the same type. As vectors are easily coerced into matrices, this is what we do.

35b ⟨*check and force datlist to matrix* 35b⟩≡

```
if(!is.list(datlist))
  stop("datlist must be a list.")
datlist = lapply(datlist, as.matrix)
```

This code chunk contains some complicated looking calculations. Essentially, it is taking into account all the text in the dotchart and computing a `cex` value that minimises wasted white (vertical) space.

36a ⟨*compute best cex* 36a⟩≡
```
parslist$cex = with(parslist, {
  cex = rep(cex, length = 2)
  cex.best = function(){
    dat.length = sum(sapply(datlist, function(x) nrow(x) + 2))
    total.lines = 1.05 * dat.length +
      2.6 * (any(axes == 1) + any(axes == 3)) +
        labcexmult * (!is.null(lab1) + !is.null(lab3)) +
          maincexmult * (!is.null(main))
    dcm = 2.54 * par("din")[2] - 2 * border
    cm.per.line = dcm/total.lines
    cm.per.line/(2.54 * par("csi"))
  }
  min(max(cex[1], cex.best()), cex[2])
})
```
Uses `parslist` 34.

The `dotchartplus` function needs to make some changes to `par` and `options`. We set `cex` to the specified `cex`, and set `mar` to 0, as margins will be handled via `layout`. The `stringsAsFactors` option is set to `FALSE` as a `data.frame` is used in some of the processing, and we require `strings` (`character` vectors) to remain as `strings`. The original `par` and `options` are saved and are restored once `dotchartplus` finishes running via a call to `on.exit`.

If `newlayout = TRUE`, we also desire to restore the original `layout`. However there exists no way to query whether the existing `layout` was done by a call to `mfcol`, `mfrow` or `layout`. Having no means to restore the original, instead we will call `layout(1)`, to restore the `layout` to a 'pseudo-default' state.

36b ⟨*save original par and opt* 36b⟩≡
```
opar = par(cex = parslist$cex, mar = rep(0, 4))
oopt = options(stringsAsFactors = FALSE)
on.exit({
  par(opar)
  options(oopt)})
if(parslist$newlayout)
  on.exit(layout(1), add = TRUE)
```
Uses `parslist` 34.

If no `textlist` is provided, we try to make one by taking the rownames of the matrices in the `datlist`. If the lengths of `textlist` don't match with the corresponding element of `datlist`, we generate a new one ourselves using a combination of letters and numbers (following the pattern A1, A2, ... for group 1. B1, B2, ... for group 2, etc). Note that if no rownames were found, the result will be `NULL`. We define a new function, `txtlen` to compute the 'length' of each element of `textlist`. This is required because `textlist` can be `NULL`, a `vector` or a `matrix`. We can't directly tackle the problem using `as.matrix` because such a call on `NULL` will fail. So we must first check if it is `NULL`, in which case we return 0 (`length(NULL)`). Otherwise we call `as.matrix` and compute the rows. As with `datlist` above, we again coerce every entry into a matrix. Note that where a `textlist` is provided, the code does not currently check if it is appropriate (e.g. that the lengths match).

37a ⟨*cag textlist* 37a⟩≡

```
if(is.null(textlist))
  textlist = lapply(datlist, rownames)
txtlen = function(txt)
  if(is.null(txt)) 0 else nrow(as.matrix(txt))
for(i in 1:length(datlist)){
  if(txtlen(textlist[[i]]) != nrow(datlist[[i]]))
    textlist[[i]] =
      paste(LETTERS[i], 1:nrow(datlist[[i]]) , sep = "")
}
textlist = lapply(textlist, as.matrix)
```

Defines:
`textlist`, used in chunks 3, 20d, 26, 34, 38d, 42a, and 43b.

4.1.1 Parslist Wetwork

This subsubsection details operations done inside the `with(parslist)`. This gives access to all the elements of `parslist`. At the end of all the 'wetwork', the updated variables are all collected into a `list` to update our `parslist`.

If no `xlim` is provided, we generate one by taking the range of all values in the `datlist`. Similarly with `datlist` above, we coerce `xlim` into a list for convenience. Finally, we ensure that each `xlim` is of the correct length (2).

37b ⟨*cag xlim* 37b⟩≡

```
if(is.null(xlim))
  xlim = range(unlist(datlist))
if(!is.list(xlim))
  xlim = list(xlim)
xlim = lapply(xlim, function(x) rep(x, length = 2))
```

Defines:
`xlim`, used in chunks 6, 21b, 29, 32, 34, 38, 39b, and 41.

Having full.lines = FALSE only makes sense if the lines will carry meaning. Thus, by default (NULL), check if xlim contains 0 and there is no split (length(xlim) > 1). Then set full.lines as appropriate.

38a ⟨*cag fulllines* 38a⟩≡

```
if(is.null(full.lines))
  if((all(xlim[[1]] != 0) &&
      sum(sign(xlim[[1]] + diff(xlim[[1]]) *
               0.04 * c(-1, 1))) != 0) || length(xlim) > 1)
    full.lines = TRUE
  else
    full.lines = FALSE
```

Uses xlim 37b.

To allow individual assignment of at, atsmall and atlabels for each split of the xlim, we require a list, which should match the length of xlim. For code-length efficiency, we do this using a loop with eval, substitute and get.

What this actually does it quite simple. The essence of it is:

```
if(!is.list(at1)) at1 = list(at1);
at1 = rep(at1, length = length(xlim))
```

That is, if at1 is not a list, we place it in a list. Then we recycle to match the length of xlim. However, we need to do this not just for at1, but also for atsmall1, at3, etc. So we loop through each of these.

38b ⟨*cag axis at* 38b⟩≡

```
for(subs in c("at1", "at3", "atsmall1", "atsmall3",
              "atlabels1", "atlabels3"))
  eval({
    substitute({
      if(!is.list(get(curat))) '='(curat, list(get(curat)));
      '='(curat, rep(get(curat), length = length(xlim)))},
               list(curat = subs))
  })
```

Uses xlim 37b.

We require some kind of ylim to order the data. It makes intuitive sense for each positive integer value of y (1, 2, 3, ...) to represent a data point, and we wish for a slight bit of padding at the top and bottom, so we generate ylim by taking 0.25 as the lower limit and the number of data points + 0.75 (nrow(x) + 0.75) as the upper limit, which gives us 0.75 padding on either end.

38c ⟨*generate ylim* 38c⟩≡

```
ylim = lapply(datlist, function(x) c(0.25, nrow(x) + 0.75))
```

Defines:
ylim, used in chunks 21b, 29, 32, 34, 39b, and 41.

We require adj to match the number of columns in textlist.

38d ⟨*cag adj* 38d⟩≡

```
adj = rep(adj, length = ncol(textlist[[1]]))
```

Uses textlist 37a.

As with `xlim` and `datlist` above, we coerce `highlight` into a list containing matrices. We also check to make sure that the highlight subset specified is a valid one (as per usual R subset rules) by checking the signs. If it is invalid, that particular subset is effectively removed (by setting to 0) and a warning message is given. Finally, we replicate `highlight` to match the length of `datlist`.

39a ⟨*check highlight* 39a⟩≡
```
if(!is.null(highlight)){
  if(!is.list(highlight))
    highlight = list(highlight)
  highlight = lapply(highlight, as.matrix)
  highlight = lapply(highlight, function(x){
    for(j in 1:ncol(x))
      if(!all(sign(x[,j]) >= 0) && !all(sign(x[,j]) <= 0)){
        warning(paste("Only 0's may be mixed with negative",
                      "subscripts.",
                      "\nInvalid highlight columns removed."),
                call. = FALSE)
        x[,j] = 0
      }
    x
  })
  highlight = rep(highlight, length = length(datlist))
}
```

We compute the `udim` ('useful' dimensions, see the Layout Helper for more details) simply by taking the length of `ylim` and `xlim`, which will give us the number of elements in the respective lists, and hence the 'useful' dimensions of our plotting area. Additionally we also replicate `pad` to the correct length here.

39b ⟨*compute udim* 39b⟩≡
```
udim = c(length(ylim), length(xlim))
pad = rep(pad, length = 2)
```
Defines:
`udim`, used in chunks 14, 20d, 24, 26, and 34.
Uses `xlim` 37b and `ylim` 38c.

The `percentile` axis is always plotted on axis 4 (to the right of the plot). Hence if `percentile` is specified but the user has also specified axis 4 in `axes`, this is overriden with a warning.

39c ⟨*adjust axes for percentile* 39c⟩≡
```
if(percentile && any(axes == 4)){
  warning(paste("The percentile axis is always drawn on axis",
                "4 (to the right of the plot).",
                "The specification of axis 4 for a label",
                "axis has been overriden to accomodate the",
                "percentile axis.", sep = ""),
          call. = FALSE)
  axes = axes[axes != 4]
  }
```

The `grouplabel` argument can take four types.

`NULL` - the default. If `datlist` contains names for its groups (list elements), these are automatically assigned to `grouplabels`. Otherwise, no group label is plotted.

`TRUE` - will always cause a group label to be generated, by assigning a letter of the alphabet to each group.

`FALSE` - will always cause the group label to NOT be plotted.

`character` - a `character` vector of the same length as the number of groups in the `datlist`, specifying the labels.

After this code chunk, `grouplabel` can be a `character` vector, or something else that says to not plot the `grouplabel`. Consistency with argument evaluation would mean this 'something else' is `FALSE`. However, a `logical` is still a vector. For a more elegant method of distinguishing between plotting and not plotting, the 'something else' is instead chosen to be `NULL`.

40a ⟨*cag grouplabel* 40a⟩≡
```
if(is.null(grouplabel))
  grouplabel = names(datlist)
if(any(grouplabel == TRUE))
  grouplabel = paste("Group", LETTERS[1:length(datlist)])
if(any(grouplabel == FALSE))
  grouplabel = NULL
if(!is.null(grouplabel) &&
   length(datlist) != length(grouplabel)){
  grouplabel = NULL
  warning(paste("length(datlist) != length(grouplabel).",
                "grouplabel forced to NULL"),
          call. = FALSE)
}
```

The `setslabel` argument is corrected in much the same way as `grouplabel` above, and takes the same types of arguments.

40b ⟨*cag setslabel* 40b⟩≡
```
if(is.null(setslabel) && (ncol(datlist[[1]]) > 1))
  setslabel = dimnames(datlist[[1]])[[2]]
if(any(setslabel == FALSE))
  setslabel = NULL
if(!is.null(setslabel) &&
   ncol(datlist[[1]]) != length(setslabel)){
  setslabel = NULL
  warning(paste("ncol(datlist[[1]]) != length(sets",
                "label)). setslabel forced to NULL",
                sep = ""), call. = FALSE)
}
```

If no `widths` or `heights` are provided, the default is to create one such that resolution is preserved across the panels, i.e. the same distance will represent the same numerical difference. This is done by simply taking the difference of the `xlim` and `ylim` values for each panel, and setting this to be our `widths` or `heights` (respectively), e.g. If we had one group with 10 data points and another with 5 data points, the `heights` assigned will be 10 and 5, so the group with twice as many data points has twice the height.

The `labwidth` computation is somewhat more complex, and covered in its own code chunk.

41 ⟨*cag widths and heights* 41⟩≡

```
if(is.null(widths))
  widths = abs(sapply(xlim, diff))
if(is.null(heights))
  heights = abs(sapply(ylim, diff))
if(is.null(padmar))
  padmar = lcm(rep(strwidth.cm("m"), 2))
if(is.null(fpad))
  fpad = strwidth.cm("m")
if(is.null(labwidths)){
  ⟨labwidths computation 42a⟩
  labwidths = labwidthsf()
  rm(labwidthsf)
}
```

Defines:
`widths`, used in chunks 6, 14, 16, and 34.
`heights`, used in chunks 6, 14, 16, 20a, and 34.
`padmar`, used in chunks 6 and 14.
`fpad`, used in chunks 6, 26, 28a, and 42.
`labwidths`, used in chunks 6, 26, 28a, and 42b.
Uses `labwidthsf` 42a, `strwidth.cm` 32, `xlim` 37b, and `ylim` 38c.

The `textlist` is a list containing matrices. Each column of the matrix represents a different column of text labels, so we wish to obtain the widths of each column. We do this by looping through the list, computing the `highlight` adjusted string widths for each column, and storing this in a matrix (`labwidmat`). Then we use `apply` on the columns of `labwidmat` to extract the maximum label width for each column. We add `fpad` to give a slight bit of white space along the sides, so the labels don't extend to the very edges of their column.

42a ⟨*labwidths computation* 42a⟩≡
```
labwidthsf = function(){
  textlen = length(textlist)
  textcols = ncol(textlist[[1]])
  labwidmat = matrix(NA, nrow = textlen, ncol = textcols)
  for(i in 1:textlen){
    curtext = textlist[[i]]
    curfont = expandpars(parslist["font"], nrow(curtext),
      textcols, highlight[[i]])
    for(j in 1:textcols)
      labwidmat[i, j] = max(strwidth.cm(curtext[,j],
                   font = curfont$font[1:nrow(curtext)]))
  }
  lcm(apply(labwidmat, 2, max) + fpad)
}
```
Defines:
`labwidthsf`, used in chunk 41.
Uses `expandpars` 31, `fpad` 41, `parslist` 34, `strwidth.cm` 32, and `textlist` 37a.

The required layout computations is merely the `widths` and `heights` of the 4 axes. Axes 1 and 3 (bottom and top) will always have tick axis plotted, so 2.6 lines of text is assigned. Axis 2 (left) will always have a text axis plotted, so it is the sum of the individual `labwidths` (using `sumlcm` as `labwidths` is stored as a character string including ' cm') plus an additional `fpad`. Axis 4 (right) can be a tick axis (if `percentile == TRUE`) or it could also be a text axis. This vector of `widths` and `heights` is passed to the *Layout Child*, which then uses these as required depending on which `axes` are asked for.

42b ⟨*layout required computations* 42b⟩≡
```
axiswidhei = numeric(4)
axiswidhei[c(1, 3)] = llines(2.6)
axiswidhei[c(2, 4)] = sumlcm(labwidths, fpad)
if(percentile){
 layoutaxes = c(axes, 4)
 lab4 = "Percentile"
 axiswidhei[4] = llines(2.6)
} else layoutaxes = axes
```
Defines:
`axiswidhei`, used in chunk 16.
Uses `fpad` 41, `labwidths` 41, `llines` 32, and `sumlcm` 32.

All the changes we have made to the elements of `parslist` are now returned.

42c ⟨*return updated parslist* 42c⟩≡
```
as.list(environment())
```

4.2 Call Primary Auxiliaries

If `newlayout` is true, we call the *Layout Auxiliary* to set-up a new layout. We also save this to `laymat` to be returned invisibly later.

43a ⟨*call layout aux* 43a⟩≡

```
laymat = if(parslist$newlayout) dcpLayout(parslist) else NULL
```

Uses `dcpLayout` 11b, `laymat` 14, and `parslist` 34.

Once the layout has been set (either beforehand or by the call to the *Layout Auxiliary*, we call the *Plot Auxiliary* to do the actual plotting.

43b ⟨*call plot aux* 43b⟩≡

```
dcpPlot(datlist, textlist, parslist)
```

Uses `dcpPlot` 20d, `parslist` 34, and `textlist` 37a.

While `dotchartplus` will usually be called for its 'side-effect' of producing a dotchart, it also returns some objects invisibly. The returned `parslist` is the one that has been fully updated with any optional arguments.

43c ⟨*return various as invisible* 43c⟩≡

```
invisible(list(layout = laymat, parslist = parslist))
```

Uses `laymat` 14 and `parslist` 34.

5 Chunk Index

⟨*add border* 20b⟩
⟨*adjust axes for percentile* 39c⟩
⟨*attach axes* 16⟩
⟨*attach labels* 18⟩
⟨*attach main title area* 20a⟩
⟨*Auxiliary Functions* 11a⟩
⟨*back end work* 34⟩
⟨*cag adj* 38d⟩
⟨*cag axis at* 38b⟩
⟨*cag fulllines* 38a⟩
⟨*cag grouplabel* 40a⟩
⟨*cag setslabel* 40b⟩
⟨*cag textlist* 37a⟩
⟨*cag widths and heights* 41⟩
⟨*cag xlim* 37b⟩
⟨*call layout* 20c⟩
⟨*call layout aux* 43a⟩
⟨*call plot aux* 43b⟩
⟨*check and force datlist to matrix* 35b⟩
⟨*check highlight* 39a⟩
⟨*compute best cex* 36a⟩
⟨*compute udim* 39b⟩
⟨*define default Generic* 2⟩
⟨*document header* 33⟩
⟨*dotchartplus.R* 1⟩
⟨*Expandpars Aux* 31⟩
⟨*form graphpars* 21c⟩
⟨*form plot region* 14⟩
⟨*generate ylim* 38c⟩
⟨*labwidths computation* 42a⟩
⟨*Layout Aux* 11b⟩
⟨*layout required computations* 42b⟩
⟨*Main Function* 3⟩
⟨*Parslist Function* 6⟩
⟨*pass on pars to parslist* 35a⟩
⟨*Plot Aux* 20d⟩
⟨*plot grouplabel* 28b⟩
⟨*plot numerical axis* 24a⟩
⟨*plot percentile axis* 24b⟩
⟨*plot points and lines* 22⟩
⟨*plot setslabel* 29⟩
⟨*plot text axis* 26⟩
⟨*plot xlab and main* 30⟩
⟨*plotloop* 21a⟩
⟨*return updated parslist* 42c⟩
⟨*return various as invisible* 43c⟩
⟨*save original par and opt* 36b⟩

⟨*Secondary Auxiliaries* 32⟩
⟨*setup plot area* 21b⟩
⟨*txtjump computation* 28a⟩

6 Identifier Index

Numbers indicate the chunks in which the function appears. Underline indicates the chunk where the function is defined.

`axis.cus`: 24a, 24b, 32
`axiswidhei`: 16, 42b
`datcol`: 21b, 21c, 22
`datlen`: 21b, 21c, 24b, 26
`dcpLayout`: 11b, 43a
`dcpPlot`: 20d, 43b
`DefaultParslist`: 3, 6
`expandpars`: 21c, 29, 31, 42a
`fpad`: 6, 26, 28a, 41, 42a, 42b
`heights`: 6, 14, 16, 20a, 34, 41
`labwidths`: 6, 26, 28a, 41, 42b
`labwidthsf`: 41, 42a
`laymat`: 11b, 14, 16, 18, 20a, 20b, 20c, 43a, 43c
`lcmTOlines`: 26, 28a, 32
`llines`: 14, 18, 20a, 32, 42b
`lrow`: 31, 32
`metamat`: 14, 16, 18
`nlines`: 14, 18, 20a, 24a, 32
`padmar`: 6, 14, 41
`parslist`: 3, 11b, 20d, 21c, 29, 31, 34, 35a, 36a, 36b, 42a, 43a, 43b, 43c
`plotloop`: 20d, 21a
`strwidth.cm`: 32, 41, 42a
`sumlcm`: 26, 28a, 32, 42b
`text.cus`: 28b, 30, 32
`textlist`: 3, 20d, 26, 34, 37a, 38d, 42a, 43b
`txtjumpf`: 26, 28a
`udim`: 14, 20d, 24a, 24b, 26, 34, 39b
`widths`: 6, 14, 16, 34, 41
`xlim`: 6, 21b, 29, 32, 34, 37b, 38a, 38b, 39b, 41
`y`: 21b, 22, 24b, 26
`ylim`: 21b, 29, 32, 34, 38c, 39b, 41

7 References

- Cleveland, W. S. (1985) **The Elements of Graphing Data.** Monterey, CA: Wadsworth.
- Ihaka, R. (2011) **The brain of.** 303.375.
- R Development Core Team (2011). **R: A language and environment for statistical computing.** R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.
- Ramsey, N. (1994) **Literate programming simplified.** IEEE Software. URL http://www.cs.tufts.edu/∼nr/noweb/.