A Literate Program for The Layout Helper

Jimmy Oh

Department of Statistics
University of Auckland

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

The Layout Helper library is designed to aid in the creation and manipulation of *layout objects*. The layout object is a *S4 Object Class* that contains the three parameters required for a call to `layout`, the R function. That is, a `matrix` and two `character` vectors defining the `widths` and `heights`. Refer to `help(layout)` for information regarding the function.

The library's basic capabilities can be used as a substitute for `mfrow` and `mfcol` calls to `par`, but with the added flexibility of assigning specified `widths` and `heights`. The library's other functions allows for construction of complex but flexible layouts in a modular fashion (see layout construction in `dotchartplus`).

Contents

2 Code Layout

The code is collected in a file called `layouthelper.R`, which is structured as follows.

2a ⟨*layouthelper.R* 2a⟩≡

⟨*document header* 2b⟩
⟨*define layout Class* 3⟩
⟨*create standard layout object* 5c⟩
⟨*create labels layout object* 16⟩
⟨*combine layout objects* 18a⟩
⟨*replicate layout object* 22b⟩
⟨*add a border around a layout object* 26⟩
⟨*call layout using layout object* 24b⟩

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

We define a short document header to encourage readers to read the literate description rather than studying the R code directly.

2b ⟨*document header* 2b⟩≡

```
## The code in this .R file is machine generated.
## To understand the program, read the literate description
##  (pdf file) rather than studying just the R code.
## No separate Manual exists.
```

2.1 The layout Class

This section describes the layout S4 object class and its associated constructor and accessor functions. The two vectors, `widths` and `heights`, are defined as `character` and not `numeric` to allow for the use of absolute scale (`lcm`) which is stored as `character`.

3 ⟨*define layout Class* 3⟩≡

```
setClass("layout",
         representation(matrix = "matrix",
                        widths = "character",
                        heights = "character"))
```

⟨*constructor function* 5a⟩
⟨*accessor functions* 5b⟩

Here is what a layout object looks like:

```
> LHdefault(udim = c(2, 1), widths = 1, heights = c(lcm(2), 1))
An object of class "layout"
Slot "matrix":
     [,1]
[1,]    1
[2,]    2

Slot "widths":
[1] "1"

Slot "heights":
[1] "2 cm" "1"
```

While this is what the layout object contains, it is more informative to look at layout objects graphically. Figure 1 is a graphical representation of the layout object above.

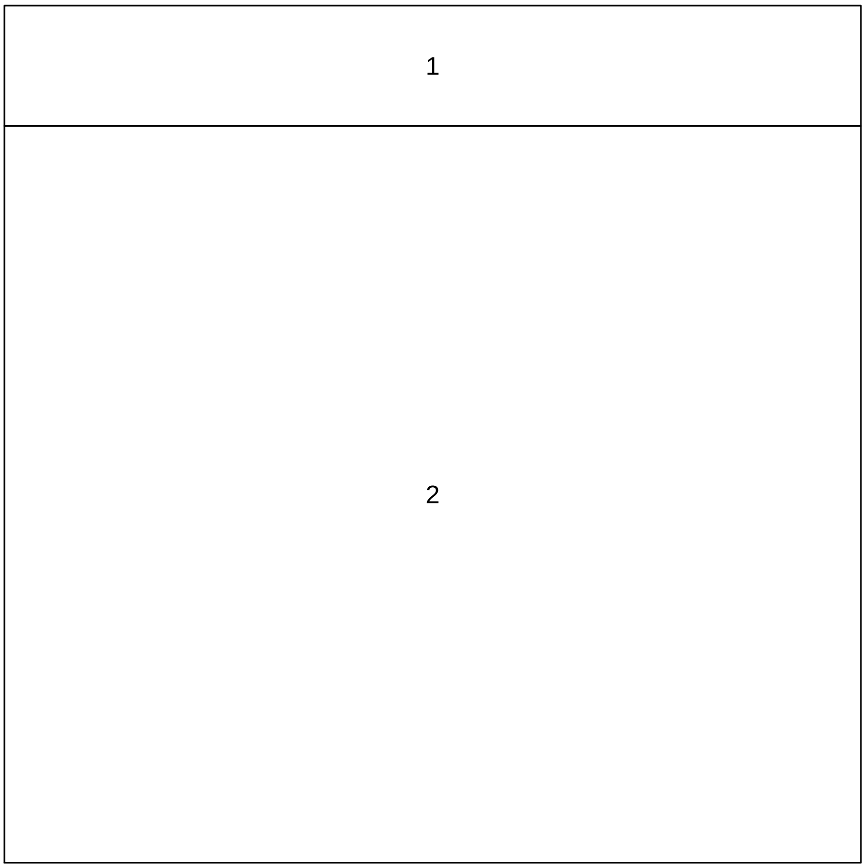

Figure 1: The layout formed by the following call:
`LHdefault(udim = c(2, 1), widths = 1, heights = c(lcm(2), 1))`

2.1.1 Constructor and Accessor Functions

These functions define how layout objects are created and how their component parts accessed. Using accessor functions to always grab components allows for changes in object structure to easily be reflected in all code that uses the objects by changing just the accessors.

5a ⟨*constructor function* 5a⟩≡

```
newlayout =
  function(mat, widths = NA, heights = NA){
    if(!is.matrix(mat))
      stop("mat must be a matrix")
    widths = rep(widths, length = dim(mat)[2])
    heights = rep(heights, length = dim(mat)[1])
    new("layout",
        matrix = mat,
        widths = as.character(widths),
        heights = as.character(heights))
  }
```

Defines:
`newlayout`, used in chunks 5c, 16, 18b, 21b, 23a, and 26.

The provided `widths` and `heights` are always scaled to match the dimensions of the provided `mat`, and have a default value of `NA` (which causes it to be ignored when combined with other layout objects. See Subsection 2.4).

The `getfino` function is used to compute the `fino` for the next layout object, where this new layout needs to be attached to the existing layout.

5b ⟨*accessor functions* 5b⟩≡

```
getmat = function(obj) obj@matrix
getwid = function(obj) obj@widths
gethei = function(obj) obj@heights
getfino = function(obj) max(getmat(obj)) + 1
```

Defines:
`getmat`, used in chunks 19, 21b, 23, 24b, and 26.
`getwid`, used in chunks 19, 21b, 23a, 24b, and 26.
`gethei`, used in chunks 19, 21b, 23a, 24b, and 26.
`getfino`, never used.

2.2 Create Standard layout Object

The function is defined as follows:

5c ⟨*create standard layout object* 5c⟩≡

```
LHdefault =
  function(udim = c(1, 1), byrow = FALSE, fino = 1,
           pad = c(0, 0), padmar = lcm(c(0.5, 0.5)),
           widths = NA, heights = NA, reverse = FALSE){
    ⟨form umat 14a⟩
    ⟨adjust for padding 14b⟩
    newlayout(laymat, widths, heights)
  }
```

Defines:
`LHdefault`, used in chunk 23b.
Uses `newlayout` 5a.

The basic use of this function is simple, `udim` specifies the dimensions of the layout, and `byrow` works as in `matrix`. This usage is similar to `mfrow` and `mfcol` calls to `par`, with the caveat that `widths` and `heights` must be specified. The argument `fino` specifies the first number of the layout. This is important when multiple layouts are being created and combined, but when creating only a single layout the default of 1 should be used.

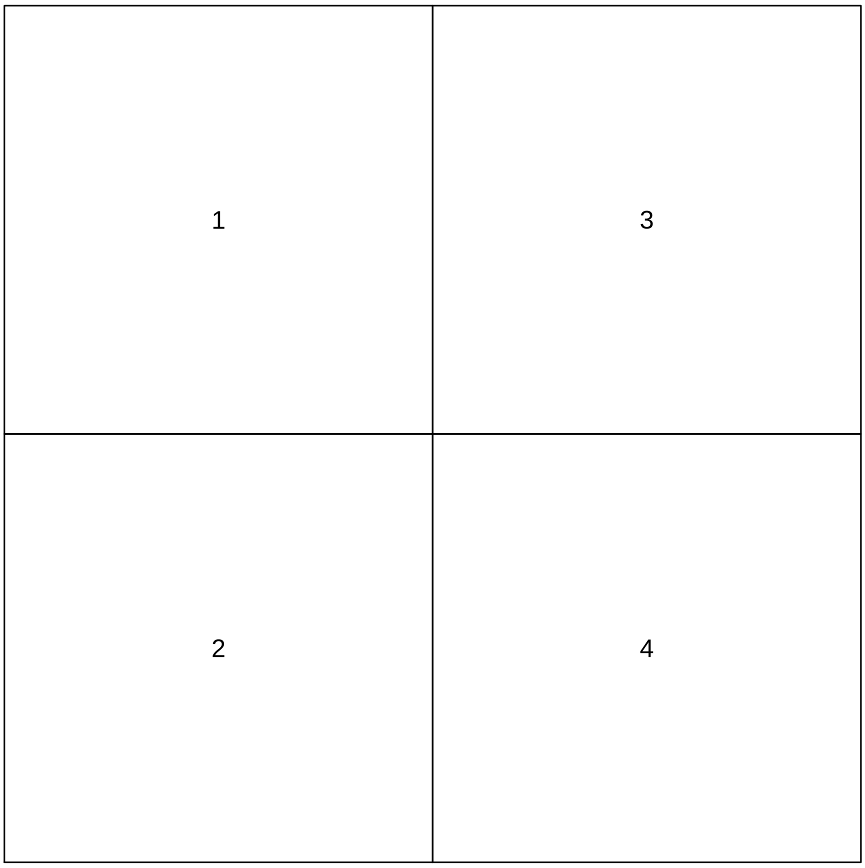

Figure 2: The layout formed by the following call:
`LHdefault(udim = c(2, 2), widths = 1, heights = 1)`

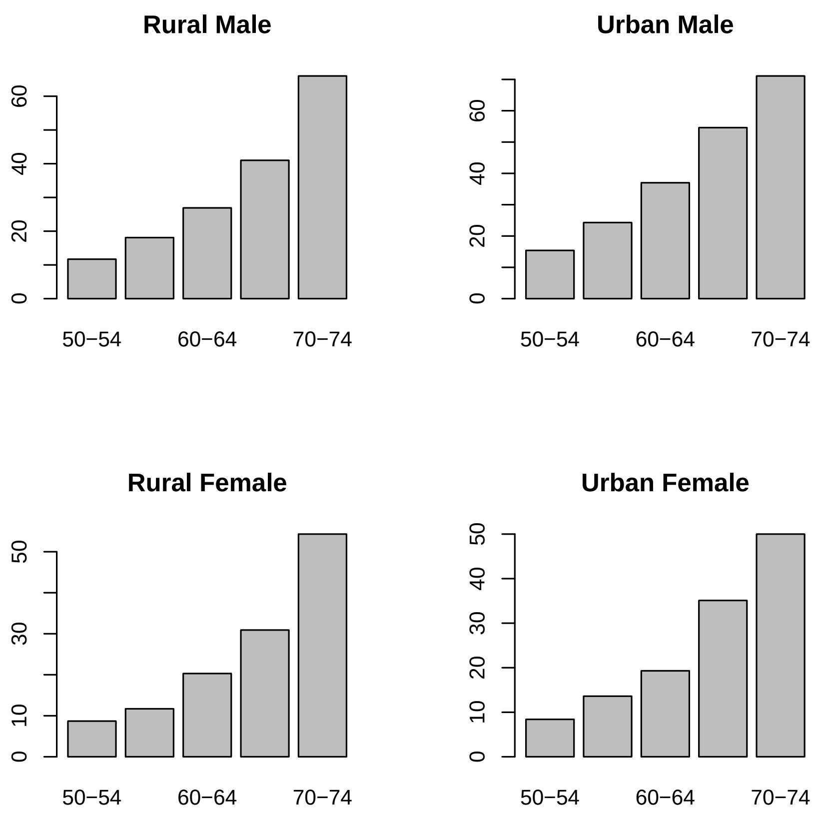

Figure 3: Using the layout shown in Figure 2 with default `mar`, the layout becomes a substitute for the call `par(mfcol = c(2, 2))`. The barplots are of the data `VADeaths` which is one of the included datasets for R.

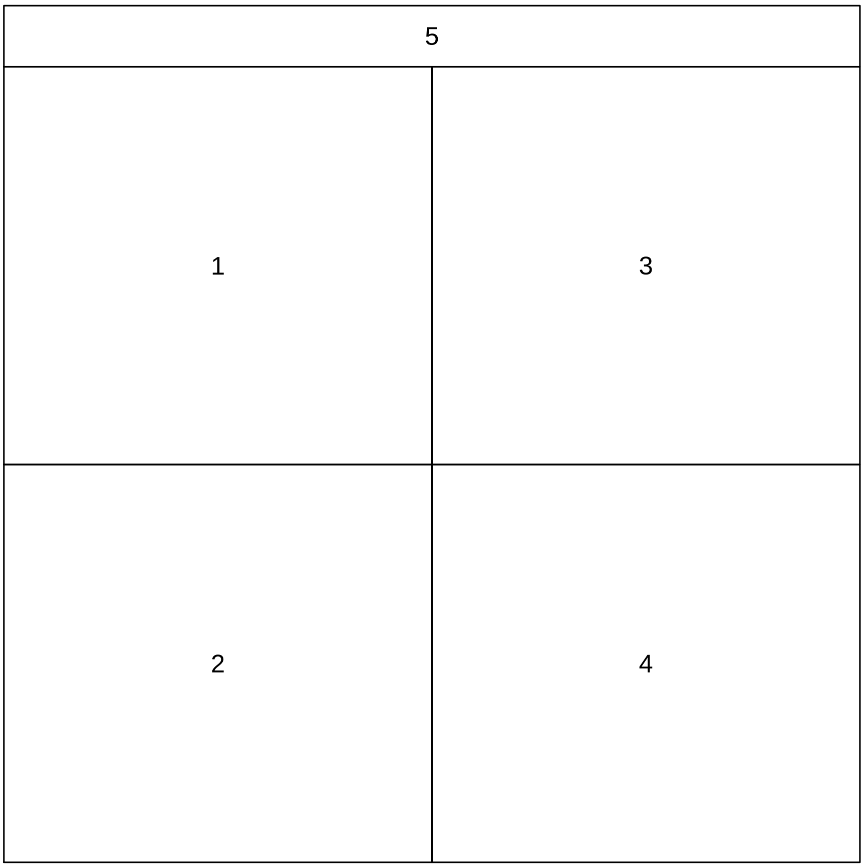

Figure 4: The four barplots on their own as in Figure 3 is not very informative. Adding a main title for the four plots would be useful, and we can do exactly that by attaching a panel at the top using the bind methods discussed later on in the document.

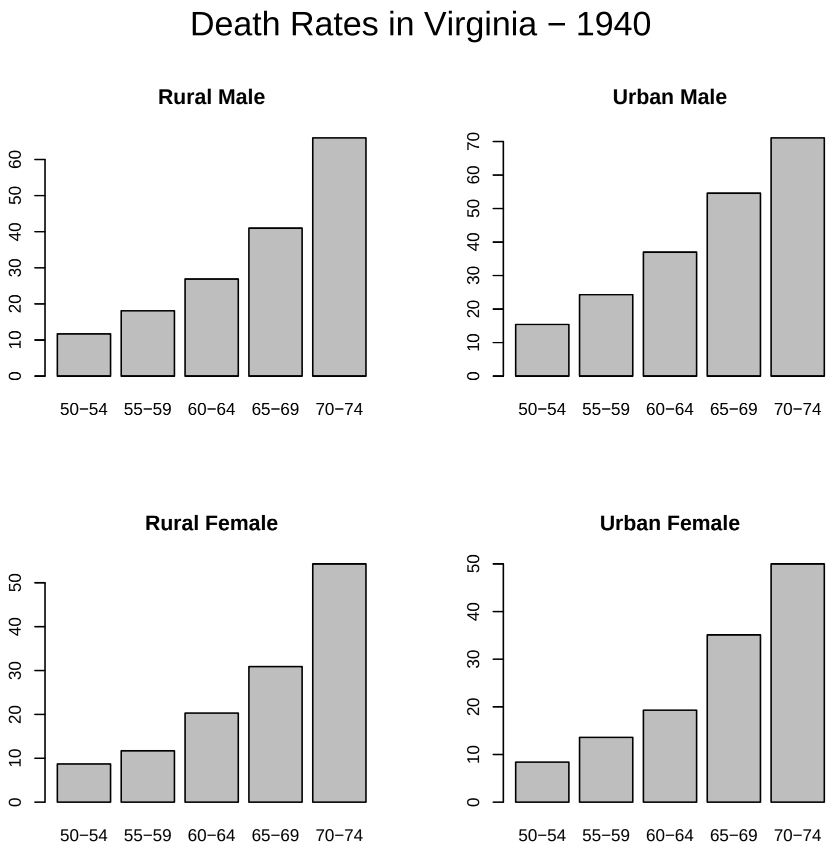

Figure 5: Figure 3 replotted using the new layout shown in Figure 4, with an added main title. This kind of plot cannot be replicated using `par` alone, and is one of the many advantages of `layout`. That said, this is still not a very good graph. The y axis are different, making direct comparisons difficult, and there is a lot of wasted white space by using the default `mar` settings.

Specification of `widths` and `heights` are intuitive, being the same as with a direct call to `layout`, except that they are always replicated to match the dimensions specified.

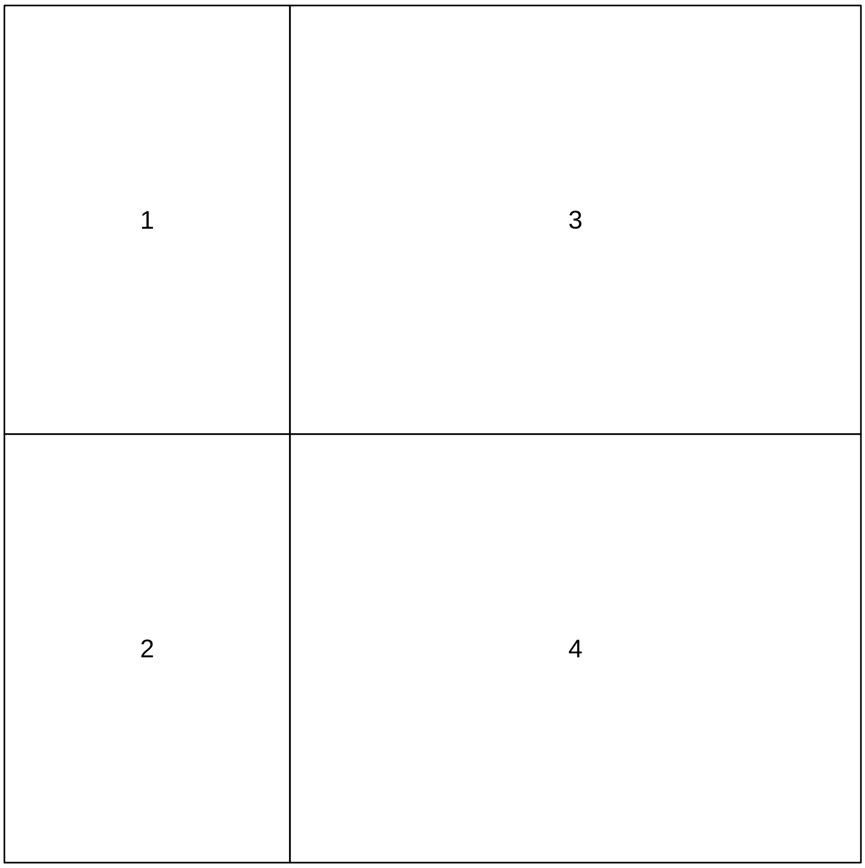

Figure 6: The layout formed by the following call:
`LHdefault(udim = c(2, 2), widths = c(1, 2), heights = 1)`

Use of `mar` calls to `par` can be restrictive, and for greater control it is best to use rows and columns of 0's within layout. For this reason, the arguments `pad` and `padmar` exist.

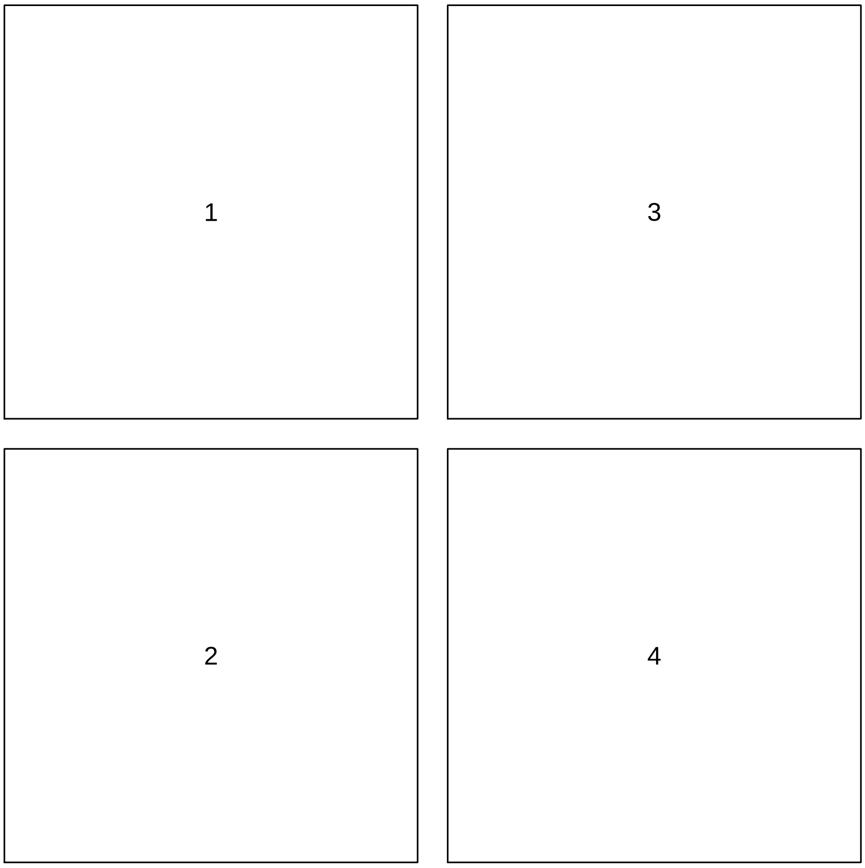

Figure 7: The layout formed by the following call:
`LHdefault(udim = c(2, 2), widths = 1, heights = 1, pad = 1)`.
The argument supplied to `pad` will be replicated to length 2, hence simply supplying 1 will specify 1 padding row and column.

A single value can be specified for `pad`, or a vector of length 2 to specify the number of padding rows and columns separately.

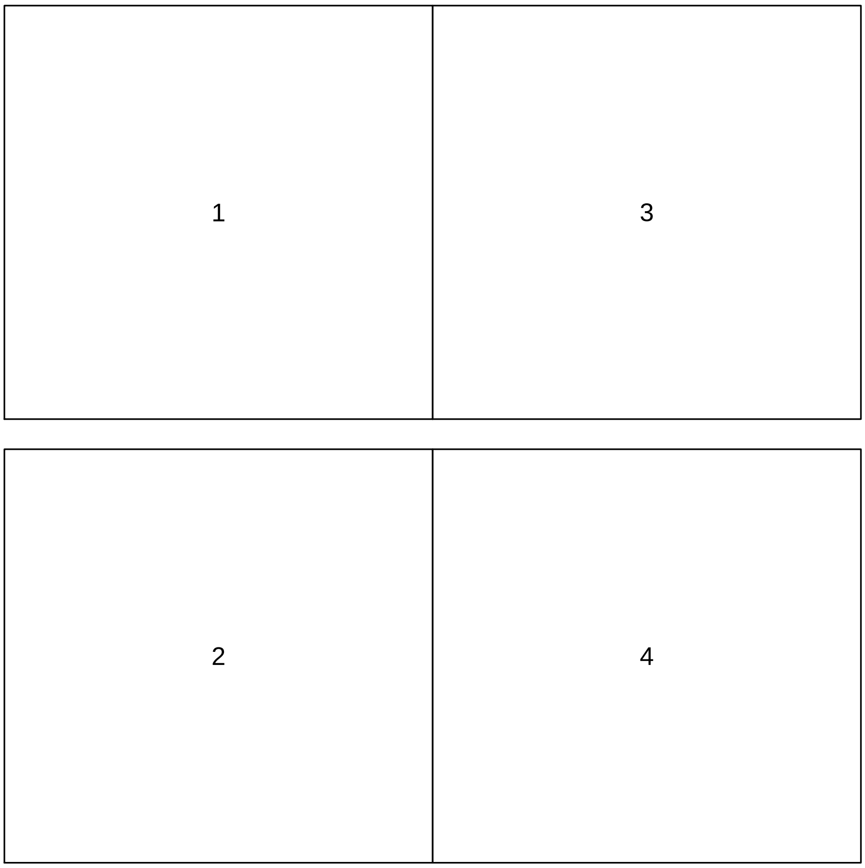

Figure 8: The layout formed by the following call:
`LHdefault(udim = c(2, 2), widths = 1, heights = 1, pad = c(1, 0))`.
This time, we are individually specifying the padding rows (1) and columns (0), unlike in Figure 7.

Notice that with padding, `udim` specifies the dimensions of the `umat` ('useful' matrix), which is the matrix that actually defines plotting regions. For example, in Figure 7, we specify a `udim = c(2, 2)`, but because of the padding, the actual matrix ix 3×3. Of course, looking at the graphical representation of the layout, the advantages of being able to specify `udim` and `pad` separately are clear, allowing separation of actual graphing elements with presentation elements (padding margins).

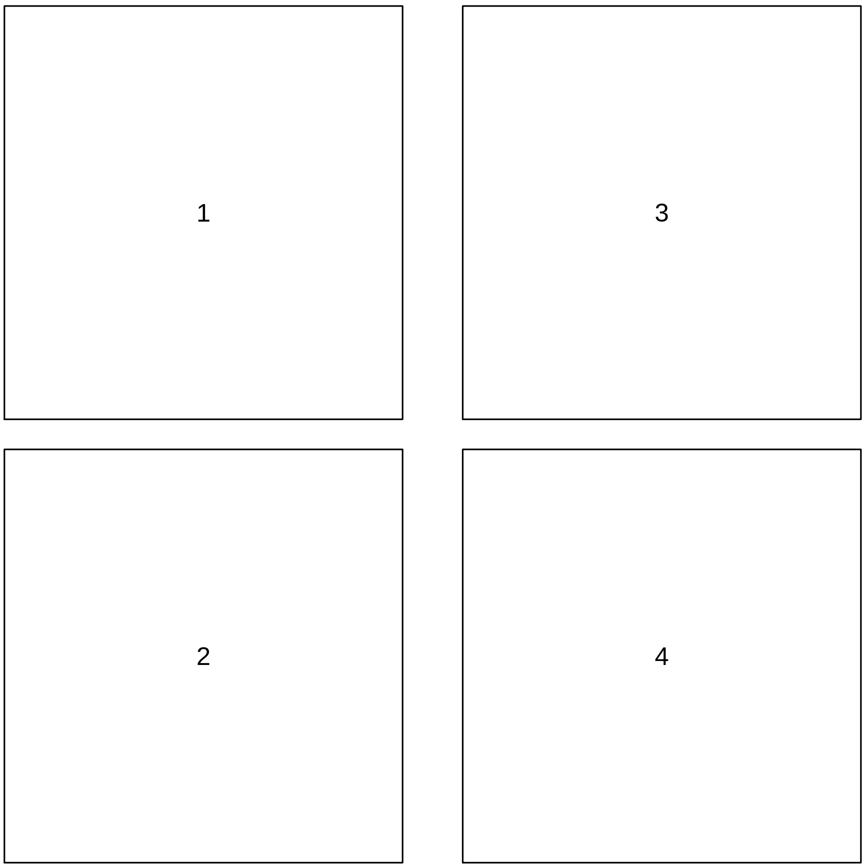

Figure 9: The layout formed by the following call:

```
LHdefault(udim = c(2, 2), widths = 1, heights = 1,
          pad = c(1, 1), padmar = lcm(c(0.5, 1))).
```

Another example of having both padding rows and columns, but this time we are individually assigning different padding margins. The margins are specified in absolute units (cm) through the use of the function `lcm`.

The code to form the padded matrix also utilises the idea of a `umat`, first forming this 'useful' matrix, then using this to fill in the larger padded matrix.

Forming the umat

We first form the `umat`, generating the required sequence of numbers based on `fino` and `udim`, with the sequence being reversed if `reverse = TRUE`. In addition, we ensure the supplied `widths` and `heights` are of the correct length by recycling. The actual matrix is declared as `laymat` as this may be further adjusted for padding before being returned.

14a ⟨*form umat* 14a⟩≡
```
useq = seq(fino, length = prod(udim))
if(reverse){
  useq = rev(useq)
  heights = rev(heights)
  widths = rev(widths)
}
laymat = matrix(useq, udim[1], udim[2], byrow)
heights = rep(heights, length = udim[1])
widths = rep(widths, length = udim[2])
```

Adjusting for Padding

We then adjust the `umat` for any padding specified. The argument `pad` is a vector of length 2, with the first element specifying the number of padding rows between each element of the `umat`, and the second element the padding columns. Generally, only 0 or 1 is used, with `padmar` being used to specify the `widths` or `heights` of the padding rows/columns.

The actual adjustment is accomplished by first forming a larger matrix (`Lmat`) containing only 0's, then replacing the correct elements with the elements from the `umat`. Similarly, we also adjust the `widths` and `heights` by first forming a larger vector containing only `padmar`, then replacing the correct elements with the `umat` `widths` and `heights`. For computational purposes, the row and col indices (the locations to insert the `umat` elements) are calculated first.

Finally, `laymat`, `widths` and `heights` are updated to their padded variants.

14b ⟨*adjust for padding* 14b⟩≡
```
pad = rep(pad, length = 2)
padmar = rep(padmar, length = 2)
if(any(pad > 0)){
  ⟨compute rowcol indices 15a⟩
  ⟨form larger matrix 15b⟩
  ⟨replace correct elements 15c⟩
  laymat = Lmat
  heights = Lheights
  widths = Lwidths
}
```

The row and col indices computation can be understood in this manner: For illustration purposes, let's only look at row positions.

1. Each element of the `umat` has row positions 1, 2, ... , `udim[1]` (`1:udim[1]`) to begin with.
2. These must then be adjusted for padding in between each element. Suppose we have padding of 1 row. Then the new row positions are 1, 3, 5, ...
3. We note that these positions can be generated by the following equation: `1:udim[1] + 1 * (1:udim[1] - 1)`.
4. Now suppose we have padding of 2 rows. Then the new row positions are 1, 4, 7, ...
5. We note that these positions can be generated by the following equation: `1:udim[1] + 2 * (1:udim[1] - 1)`.
6. More generally, for any row padding, `pad[1]`, the new row positions can be generated by the following equation: `1:udim[1] + pad[1] * (1:udim[1] - 1)`. An equivalent statement that may be harder to understand, but is marginally more efficient, is: `(1 + pad[1]) * 1:udim[1] - pad[1]`. Which is the form used in the actual code.

15a ⟨*compute rowcol indices* 15a⟩≡

```
rowind = (1 + pad[1]) * 1:udim[1] - pad[1]
colind = (1 + pad[2]) * 1:udim[2] - pad[2]
```

As padding only applies in between elements of the `umat`, the largest indices gives us the dimensions of the `Lmat`. All that remains is to simply call `matrix` and recycle the `padmar` to match the dimensions of the `Lmat`.

15b ⟨*form larger matrix* 15b⟩≡

```
Lmat = matrix(0, max(rowind), max(colind))
Lheights = rep(padmar[1], length = dim(Lmat)[1])
Lwidths = rep(padmar[2], length = dim(Lmat)[2])
```

Finally, we replace the right elements (which we have already computed) with the elements from `umat`, `widths` and `heights`.

15c ⟨*replace correct elements* 15c⟩≡

```
Lmat[rowind, colind] = laymat
Lheights[rowind] = heights
Lwidths[colind] = widths
```

2.3 Creating Label layout Objects

When forming more complex layouts, the plotting of axis labels that are correctly aligned can become challenging. Several cases can present this problem, such as with asymmetric axis (an axis on one side but not the other), where the plotting region consists of an even number of panels (meaning no 'middle' panel exists to use as the centre position), or where the plot panels are not equally sized (again causing problems with finding the centre).

`LHlabels` is designed for cases like these, when you wish to create a suitably sized layout to attach to the main one, such that the label region correctly matches to the plotting regions, with white-space to match the axis.

The use of the function is thus, the argument `x` specifies the numbering, each of length corresponding to `lengths`. That is, the following call:

```
LHlabels(x = c(1, 2), lengths = c(2, 1), colvec = FALSE, widhei = 1)
```

Will produce a layout object containing the row vector `1 1 2`, with height 1 and widths `NA`.

In general, `x` will be `c(0, 1, 0)`, where 1 can be substituted for the appropriate number for the label panel. Then `lengths` can be matched to the size of the axis on either side and to the plotting region.

Finally, the arguments `colvec` specifies whether the output is a column vector or not (row vector) and `widhei` specifies the `widths` or `heights` as appropriate. As the intent is for matching to some existing layout, `widhei` is passed to whichever is necessary for meshing, while the other remains `NA` so that it is ignored.

Thus, if the label is a column vector, the `widths` of this column must be specified, while the `heights` of the existing layout can be used. Hence `widhei` will be specifying the `widths` of this column vector.

16 ⟨*create labels layout object* 16⟩≡

```
LHlabels =
  function(x, lengths = c(0, 1, 0), colvec = TRUE, widhei = NA){
    if(colvec){
      ncol = 1
      widths = widhei
      heights = NA
    } else{
      ncol = sum(lengths)
      widths = NA
      heights = widhei
    }

    maxlen = max(length(x), length(lengths))
    x = rep(x, maxlen)
    lengths = rep(lengths, maxlen)

    labvec = NA
    for(i in 1:maxlen)
      labvec = c(labvec, rep(x[i], lengths[i]))
    labvec = labvec[-1]
```

```
    newlayout(matrix(labvec, ncol = ncol), widths, heights)
  }
```

Defines:

  `LHlabels`, never used.

Uses `newlayout` 5a.

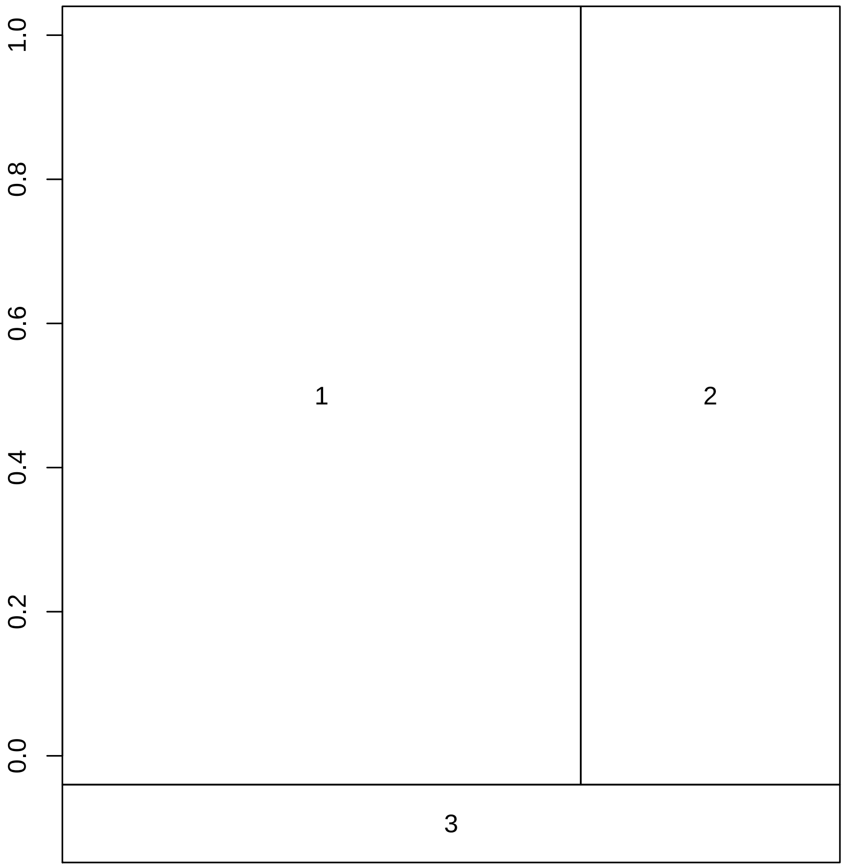

Figure 10: Our 'main' layout has unequal widths, and also has an axis on the left. This makes centering a xlab problematic, but `LHlabels` provides a way to do this. The call made is
`LHlabels(c(0, 3, 0), c(1, 2, 0), colvec = FALSE, widhei = 0.1)`
Thus we have one 0 to match the left axis, two 3's to match the two plotting panels, and zero 0's to the right, as we have no right axis in this case. We also specify a suitable `widhei` which provides the height, while the widths will automatically match the 'main' layout once we bind these together using the methods discussed later on in the document.

2.4 Combining layout Objects

A single standard layout object generally doesn't give us the complete layout matrix. Instead, we must combine several layout objects.

18a ⟨*combine layout objects* 18a⟩≡

```
⟨define a layout cbind 18b⟩
⟨define a layout transpose 21b⟩
⟨define a layout rbind 22a⟩
```

2.4.1 Binding layouts by columns

Often we will desire a combination of layout objects that will have the same matrix dimensions. However, occasionally we may wish to combine layout objects with different matrix dimensions, usually where one layout object is a 1×1 matrix, which we wish to simply stretch to fit the larger matrix. Thus, unlike the matrix method for `cbind`, the layout method accepts matrices of different dimensions by scaling smaller matrices. The scaling method is a 'stretch' method, with 'over-stretched' excess removed without warning. This is perhaps best understood by way of example. Consider the following 1×2 matrix:

$$\begin{bmatrix}1 & 2\end{bmatrix}$$

Now suppose we wish to scale this to match a 1×4 matrix, then it will be stretched to the following:

$$\begin{bmatrix}1 & 1 & 2 & 2\end{bmatrix}$$

Now consider if we wished to scale the original 1×2 matrix to match a 1×3 matrix, then it will be stretched to the following:

$$\begin{bmatrix}1 & 1 & 2\end{bmatrix}$$

Here, the excess column containing 2 has been removed. This is how all scaling in the `bind` method is handled.

18b ⟨*define a layout cbind* 18b⟩≡

```
cbind.layout =
  function(..., reverse = FALSE){
    ⟨form list and remove nulls 18c⟩
    ⟨separate components 19⟩
    ⟨scale matrices 20⟩
    ⟨combine matrices widths and heights 21a⟩
    newlayout(new.mat, new.wid, new.hei)
  }
```

Uses `newlayout` 5a.

We pass the arguments into a list, removing any `NULL` objects. In addition, if `reverse = TRUE`, then reverse the order of the supplied layout objects.

18c ⟨*form list and remove nulls* 18c⟩≡

```
parlist = list(...)
parlist = parlist[!sapply(parlist, is.null)]
if(reverse) rev(parlist)
```

We separate each component as they require separate handling. Each component will be stored in a list.

19 ⟨*separate components* 19⟩≡

```
parmat = lapply(parlist, function(x) getmat(x))
parwid = lapply(parlist, function(x) getwid(x))
parhei = lapply(parlist, function(x) gethei(x))
```

Uses `gethei` 5b, `getmat` 5b, and `getwid` 5b.

For scaling the matrices to fit, we first find the largest row dimension, then scale all matrices to have this number of rows. The scaling can be understood in this manner:

1. We note that R stores matrices column-wise as a vector.

2. If the matrix is taken to be a vector and replicated by a call to `rep` supplying an argument for `each`, each column of the matrix can be 'stretched' downward. We then need to know how much to stretch by.

3. Suppose we wish to stretch a matrix `x`. `rowmax/dim(x)[1]` gives us the exact proportion of the largest row (`rowmax`) to the rows of the current matrix (`x`).

4. `ceiling(rowmax/dim(x)[1]))` then gives us the integer multiple (rounded-up) to stretch by. Note this will 'over-stretch' the matrix beyond what we want (`rowmax`).

5. Specifying `ncol = dim(x)[2]` ensures the resulting matrix has the correct column dimension.

6. Taking the subset `[1:rowmax, ,drop = FALSE]` ensures the resulting matrix has the correct row dimension (by removing the excess created by the 'over-stretch').

Let us go over the above with a concrete example. Let us consider the following 2×2 matrix:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

In R, this matrix is stored as a vector of length 4 (1 2 3 4). If we were to make a call to `rep` supplying `each = 2`, the resulting vector would be: 1 1 2 2 3 3 4 4. If we then converted this back to a matrix, supplying `ncol` as the original number of columns (`dim(x)[2]` $= 2$), we get the following 'stretched' matrix:

$$\begin{bmatrix} 1 & 3 \\ 1 & 3 \\ 2 & 4 \\ 2 & 4 \end{bmatrix}$$

We have, in effect, 'stretched' the matrix downward. If we wished a fewer number of rows, we can simply take a subset. Specifying the argument `drop = FALSE` ensures that the subset we take remains a matrix.

20 $\langle$*scale matrices* 20$\rangle\equiv$

```
rowmax = max(sapply(parmat, nrow))
matscaled = lapply(parmat, function(x)
      matrix(rep(x, each = ceiling(rowmax/dim(x)[1])),
             ncol = dim(x)[2])[1:rowmax, ,drop = FALSE]
      )
```

As we're binding by columns, we simply merge the individual `widths` vectors to form the new `widths`. The handling of `heights` is somewhat more complicated. Firstly, we ignore any `heights` vector which contains NA. Of the remaining vectors, we grab the first longest vector. This way of handling is useful when you wish to `cbind` a new layout object to an existing one, and you wish for the new layout object to share the same `heights` as the existing. Rather than having to grab the `heights` from the existing, one can simply form the new layout object with `heights = NA`. Upon combination, this NA `heights` vector will be ignored, and the new combined layout object will carry the correct `heights` vector.

21a ⟨*combine matrices widths and heights* 21a⟩≡

```
new.mat = do.call(cbind, matscaled)
new.wid = unlist(parwid)
notna.hei = parhei[sapply(parhei, function(x) all(!is.na(x)))]
lens.hei = sapply(notna.hei, length)
firstlongest.hei = which(lens.hei == max(lens.hei))[1]
new.hei = notna.hei[[firstlongest.hei]]
```

2.4.2 Transpose of a layout Object

The matrix scaling involved in `cbind.layout` causes some problems in writing a similar `rbind.layout`. For the early iterations of the Layout Helper, there were two separate methods, but often they behaved in different ways with respect to the scaling. It was deemed wiser to simply transpose the layout object, apply `cbind.layout`, then transpose back, rather than having a separate `rbind.layout` method. Thus, we must define what a transpose of a layout object is. A transpose of a layout object is simply a transpose of the matrix, and an exchange of the `widths` and `heights`.

21b ⟨*define a layout transpose* 21b⟩≡

```
setMethod("t", signature(x = "layout"),
  function(x) newlayout(t(getmat(x)), gethei(x), getwid(x)))
```

Uses `gethei` 5b, `getmat` 5b, `getwid` 5b, and `newlayout` 5a.

2.4.3 Binding layouts by rows

Now that we have defined the transpose of a layout object, it is easy to define a method for binding by rows. We take the provided arguments, place into a list, transpose every element, call cbind, then back transpose the combined layout object.

As with `cbind.layout`, we remove any `NULL` objects, as a transpose of a `NULL` is not defined.

22a ⟨*define a layout rbind* 22a⟩≡

```
rbind.layout =
  function(..., reverse = TRUE){
    parlist = list(...)
    parlist = parlist[!sapply(parlist, is.null)]
    tparlist = lapply(parlist, t)
    combined = do.call(cbind, tparlist)
    t(combined)
  }
```

2.5 Replicating a layout Object

Rather than combining distinct layout objects, we may sometimes be interested in replicating the same layout object to create multiple panels. Replicating layout objects works slightly differently from the default `rep` for the crucial reason that the numbering must be adjusted for the replicated layout object to be useful. This is most easily done by utilising the afore defined bind functions, rather than utilising `rep`.

22b ⟨*replicate layout object* 22b⟩≡

```
⟨shift layout numbering 23a⟩
setMethod("rep", signature(x = "layout"),
          local({
            ⟨rep function 23b⟩
            ⟨rep ordering function 24a⟩
            }))
```

We define two support functions, one within a local block whose value is the ordering function, and another outside the local block, as it has potential use beyond just the rep function. The use of the local block 'hides' the support function, preventing it from being called separately.

The first support function is to shift the numbering. As mentioned, for a replicated layout object to be useful, the numbering must be shifted (thereby defining new plotting regions when `layout` is called). This is done by simply adjusting all non-zero elements in the matrix by adding the previous maximum number (`prevmax`). This way of shifting relies on the matrix having every integer number from 1 to its maximum value at least once (i.e. being a valid `layout` matrix). The `prevmax` can either be supplied as an argument, or is otherwise calculated by taking the `max` of the supplied matrix.

23a ⟨*shift layout numbering* 23a⟩≡

```
LHshift = function(x, prevmax = NULL){
  mat = getmat(x)
  if(is.null(prevmax)) prevmax = max(mat)
  mat[mat > 0] = mat[mat > 0] + prevmax
  newlayout(mat, getwid(x), gethei(x))
}
```

Defines:
`LHshift`, used in chunks 23b and 26.
Uses `gethei` 5b, `getmat` 5b, `getwid` 5b, and `newlayout` 5a.

The support function in the local block does the actual replication. The 'replication' is conducted by creating a replicate layout object with shifted numbers, then binding this to the existing layout object. This is then looped to however many replications are desired. The actual `repfunc` can be difficult to use, so we also create `repright` and `repdown` for a more inuitive way to call.

23b ⟨*rep function* 23b⟩≡

```
repfunc = function(x, times, what = "cbind",
  pad = 0, padmar = c(NA, NA)){
  padobj = if(pad > 0)
    rep(list(LHdefault(fino = 0, widths = padmar[2],
                       heights = padmar[1])), length = pad)
    else NULL
  newx = x
  if(times > 1)
    for(i in 2:times)
      newx = do.call(what, c(list(newx), padobj,
        list(LHshift(x, max(getmat(newx))))))
  newx
}
repright = function(x, times, pad = 0, padmar = lcm(0.5))
  repfunc(x, times, "cbind", pad = pad, padmar = c(NA, padmar))
repdown = function(x, times, pad = 0, padmar = lcm(0.5))
  repfunc(x, times, "rbind", pad = pad, padmar = c(padmar, NA))
```

Defines:
`repfunc`, never used.
`repright`, used in chunk 24a.
`repdown`, used in chunk 24a.
Uses `getmat` 5b, `LHdefault` 5c, and `LHshift` 23a.

Finally, the ordering function, which can be considered the 'main' function. The argument `x` is the layout object to be replicated, `byrow` specifies whether the replication should occur by rows first (i.e. to the right) or by columns first (i.e. downwards). The arguments `coltimes` and `rowtimes` works like `times` in the default `rep` function.

24a ⟨*rep ordering function* 24a⟩≡

```
function(x, rowtimes = 1, coltimes = 1, byrow = FALSE,
         pad = c(0, 0), padmar = lcm(c(0.5, 0.5))){
  if(byrow){
    newx = repright(x, coltimes, pad[2], padmar[2])
    repdown(newx, rowtimes, pad[1], padmar[1])
  } else{
    newx = repdown(x, rowtimes, pad[1], padmar[1])
    repright(newx, coltimes, pad[2], padmar[2])
  }
}
```

Uses `repdown` 23b and `repright` 23b.

2.6 Calling layout

This is a trivial function that calls `layout` with the correct components of the provided layout object. If provided with a custom argument `cex`, the function will also set the `par` value for cex to the given value (as `layout` will automatically adjust `cex` based on the size of the matrix).

24b ⟨*call layout using layout object* 24b⟩≡

```
LHcall =
  function(obj, cex = NULL){
   layout(getmat(obj), getwid(obj), gethei(obj))
   if(!is.null(cex)) par(cex = cex)
  }
```

Defines:
`LHcall`, never used.
Uses `gethei` 5b, `getmat` 5b, and `getwid` 5b.

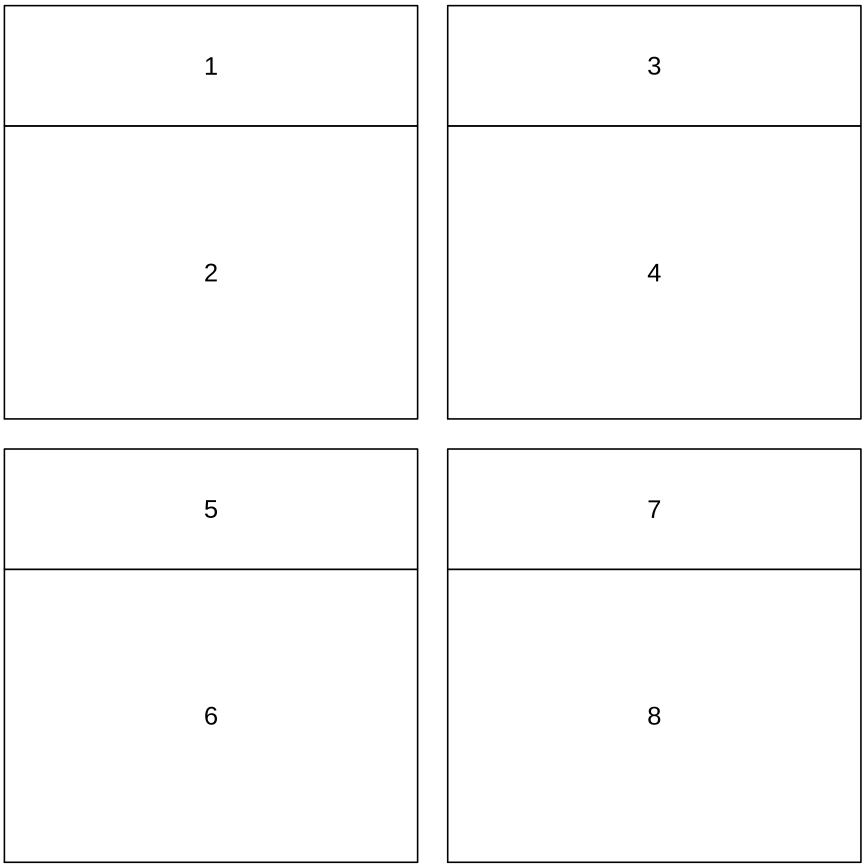

Figure 11: This gives an example of the layout featured in Figure 1 replicated with parameters:
`rowtimes = 2, coltimes = 2, byrow = TRUE, pad = c(1, 1)`
Notice that as the top panel has a fixed height of 2 cm, with the replication, the lower panel has a comparatively smaller height.

2.7 Adding Borders

This is a trivial function for adding a border (usually of 0's) around a layout object. We first add top and bottom, then add sides, which includes the corners.

This function doesn't use the bind methods defined above: firstly because this was written earlier, and secondly because this is more efficient.

The argument `border` is specified in much the same way as `oma` in `par`, with the difference that the specified number is interpreted as cm, not lines of text. Thus the default of 0.5 specifies a border on all sides of 0.5 cm. One could supply `border = c(1, 0.5, 1, 0.5)` for a 1 cm border on the top and bottom, and a 0.5 cm border on the sides.

If `numbered = FALSE`, the border consists of 0's.

If `numbered = TRUE`, the border is numbered, meaning it can be used as a plotting region. By bordering some layout object with a numbered border, one can treat the entire bordered layout as a single panel, as well as treating it as individual panels.

The actual procedure of adding the border is similar under both cases, except if the border is numbered, the numbering is taken to be the smallest number in the layout object, and the layout object's numbers are shifted to account for the numbered border.

As a layout object may not have `widths` or `heights` of value 0, the function will automatically replace any 0's given with 1 angstrom, which is 10^{-8} cm. This is most useful when adding a numbered border, as it is often desirable for the numbered border to have (effectively) no separate `widths` or `heights`.

26 ⟨*add a border around a layout object* 26⟩≡

```
LHborder =
  function(obj, border = 0.5, numbered = FALSE){
    border[border == 0] = 10^-8
    border = rep(border, length = 4)
    if(numbered) obj = LHshift(obj, 1)
    mat = getmat(obj)
    wid = getwid(obj)
    hei = gethei(obj)

    dims = dim(mat)
    bordnum = if(numbered) min(mat[mat > 0]) - 1 else 0

    bordtop = rep(bordnum, dims[2])
    bordsides = rep(bordnum, dims[1] + 2)

    new.mat = rbind(bordtop, mat, bordtop, deparse.level = 0)
    new.mat = cbind(bordsides, new.mat, bordsides, deparse.level = 0)

    new.wid = c(lcm(border[2]), wid, lcm(border[4]))
    new.hei = c(lcm(border[3]), hei, lcm(border[1]))

    newlayout(new.mat, new.wid, new.hei)
  }
```

Defines:
`LHborder`, never used.

Uses `gethei` 5b, `getmat` 5b, `getwid` 5b, `LHshift` 23a, and `newlayout` 5a.

Note that the calls to `rep`, `rbind` and `cbind` are calls to the standard methods that comes with R, not the ones defined for layout objects. We specify `deparse.level = 0` as otherwise the combined matrix will gain col and row names.

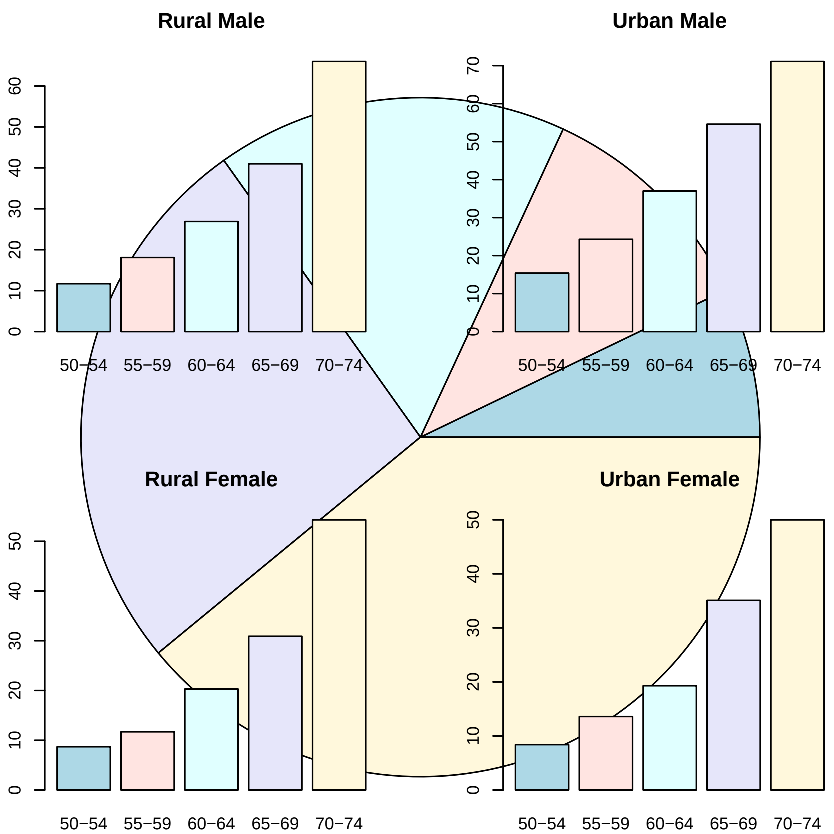

Figure 12: Figure 3 redrawn with a numbered border. Using the numbered border, we can draw over the entire plot region (in this case, we've drawn a piechart of the age groups summed over the population groups), then go through each individual panel and still do the barplots as before. Note that piecharts are almost always a bad idea (refer to R help for `pie`, under Note), and drawing a graph behind several other graphs is also usually a bad idea. This is done purely for demonstration purposes. Usually, a numbered border is best used to do something simple, like drawing a border around the entire plotting region.

3 Chunk Index

4 Identifier Index

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