Recent Changes in grid Graphics

by Paul Murrell

Introduction

The **grid** graphics package provides an alternative graphics system to the "traditional" S graphics system that is provided by the **graphics** package.

The majority of high-level plotting functions (functions that produce whole plots) that are currently available in the base packages and in add-on packages are built on the **graphics** package, but the **lattice** package (Sarkar, 2002), which provides high-level Trellis plots, is built on **grid**.

The **grid** graphics package can be useful for customising **lattice** plots, creating complex arrangements of several **lattice** plots, or for producing graphical scenes from scratch.

The basic features of **grid** were described in an R News article in June 2002 (Murrell, 2002). This article provides an update on the main features that have been added to **grid** since then. This article assumes a familiarity with the basic **grid** ideas of units and viewports.

Changes to grid

The most important organisational change is that **grid** is now a "base" package, so it is installed as part of the standard R installation. In other words, package writers can assume that **grid** is available.

The two main areas where the largest changes have been made to **grid** are viewports and graphical objects. The changes to viewports are described in the next section and summarised in Table 1 at the end of the article. A separate section describes the changes to graphical objects with a summary in Table 2 at the end of the article.

Changes to viewports

grid provides great flexibility for creating regions on the page to draw in (viewports). This is good for being able to locate and size graphical output on the page in very sophisticated ways (e.g., **lattice** plots), but it is bad because it creates a complex environment when it comes to adding further output (e.g., annotating a **lattice** plot).

The first change to viewports is that they are now much more persistent; it is possible to have any number of viewports defined at the same time.

There are also several new functions that provide a consistent and straightforward mechanism for navigating to a particular viewport when several viewports are currently defined.

A viewport is just a rectangular region that provides a geometric and graphical context for drawing. The viewport provides several coordinate systems for locating and sizing output, and it can have graphical parameters associated with it to affect the appearance of any output produced within the viewport. The following code describes a viewport that occupies the right half of the graphics device and within which, unless otherwise specified, all output will be drawn using thick green lines. A new feature is that a viewport can be given a name. In this case the viewport is called `"rightvp"`. The name will be important later when we want to navigate back to this viewport.

```
> vp1 <- viewport(x=0.5, width=0.5,
                  just="left",
                  gp=gpar(col="green", lwd=3),
                  name="rightvp")
```

The above code only creates a description of a viewport; a corresponding region is created on a graphics device by *pushing* the viewport on the device, as shown below.[1]

```
> pushViewport(vp1)
```

Now, all drawing occurs within the context defined by this viewport until we change to another viewport. For example, the following code draws some basic shapes, all of which appear in the right half of the device, with thick green lines (see the right half of Figure 1). Another new feature is that a name can be associated with a piece of graphical output. In this case, the rectangle is called `"rect1"`, the line is called `"line1"`, and the set of circles are called `"circles1"`. These names will be used later in the section on "Changes to graphical objects".

```
> grid.rect(name="rect1")
> r <- seq(0, 2*pi, length=5)[-5]
> x <- 0.5 + 0.4*cos(r + pi/4)
> y <- 0.5 + 0.4*sin(r + pi/4)
> grid.circle(x, y, r=0.05,
```

[1]The function used to be called `push.viewport()`.

```
                name="circles1")
> grid.lines(x[c(2, 1, 3, 4)],
             y[c(2, 1, 3, 4)],
             name="line1")
```

There are two ways to change the viewport. You can *pop* the viewport, in which case the region is permanently removed from the device, or you can navigate *up* from the viewport and leave the region on the device. The following code demonstrates the second option, using the upViewport() function to revert to using the entire graphics device for output, but leaving a viewport called "rightvp" still defined on the device.

```
> upViewport()
```

Next, a second viewport is defined to occupy the left half of the device, this viewport is pushed, and some output is drawn within it. This viewport is called "leftvp" and the graphical output is associated with the names "rect2", "lines2", and "circles2". The output from the code examples so far is shown in Figure 1.

```
> vp2 <- viewport(x=0, width=0.5,
                  just="left",
                  gp=gpar(col="blue", lwd=3),
                  name="leftvp")
> pushViewport(vp2)
> grid.rect(name="rect2")
> grid.circle(x, y, r=0.05,
              name="circles2")
> grid.lines(x[c(3, 2, 4, 1)],
             y[c(3, 2, 4, 1)],
             name="line2")
```

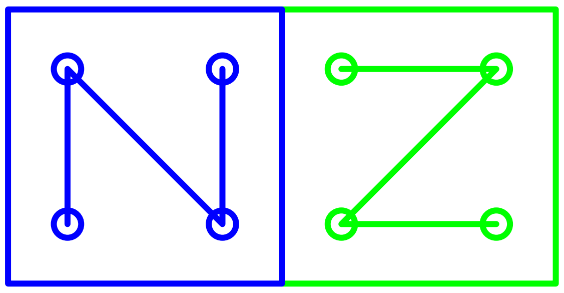
Figure 1: Some simple shapes drawn in some simple viewports.

There are now three viewports defined on the device; the function current.vpTree() shows all viewports that are currently defined.[2]

```
> current.vpTree()
```

```
viewport[ROOT]->(
  viewport[leftvp],
  viewport[rightvp])
```

There is always a ROOT viewport representing the entire device and the two viewports we have created are both direct children of the ROOT viewport. We now have a tree structure of viewports that we can navigate around. As before, we can navigate up from the viewport we just pushed and, in addition, we can navigate *down* to the previous viewport. The function downViewport() performs the navigation to a viewport lower down the viewport tree. It requires the name of the viewport to navigate to. In this case, we navigate down to the viewport called "rightvp".

```
> upViewport()
> downViewport("rightvp")
```

It is now possible to add further output within the context of the viewport called "rightvp". The following code draws some more circles, this time explicitly grey (but still with thick lines; see Figure 2). The name "circles3" is associated with these extra circles.

```
> x2 <- 0.5 + 0.4*cos(c(r, r+pi/8, r-pi/8))
> y2 <- 0.5 + 0.4*sin(c(r, r+pi/8, r-pi/8))
> grid.circle(x2, y2, r=0.05,
              gp=gpar(col="grey"),
              name="circles3")
> upViewport()
```

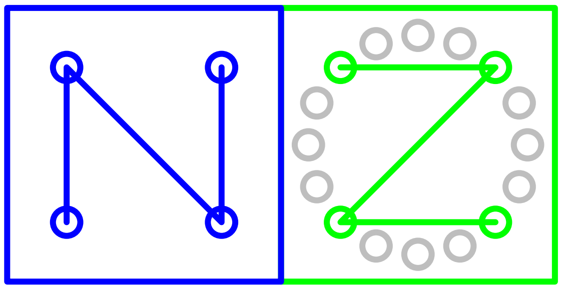
Figure 2: Navigating between viewports and annotating.

What this simple demonstration shows is that it is possible to define a number of viewports during drawing and leave them in place for others to add

[2]The output of current.vpTree() has been reformatted to make the structure more obvious; the natural output is all just on a single line.

further output. A more sophisticated example is now presented using a **lattice** plot.[3]

The following code creates a simple **lattice** plot. The `"trellis"` object created by the `histogram()` function is stored in a variable, `hist1`, so that we can use it again later; printing `hist1` produces the plot shown in Figure 3.[4]

```
> hist1 <- histogram(
    par.settings=list(fontsize=list(text=8)),
    rnorm(500), type = "density",
    panel=
      function(x, ...) {
        panel.histogram(x, ...)
        panel.densityplot(x,
                          col="brown",
                          plot.points=FALSE)
      })
> trellis.par.set(canonical.theme("pdf"))
> print(hist1)
```

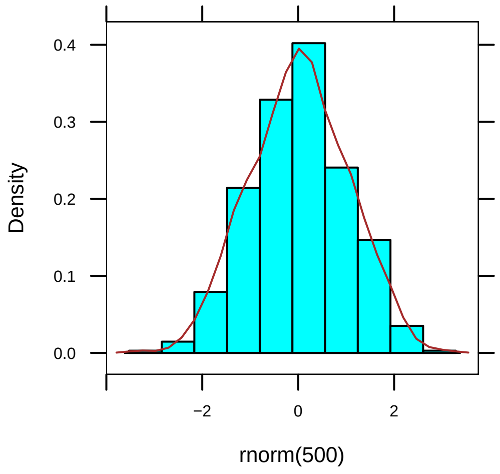

Figure 3: A simple **lattice** plot.

The **lattice** package uses **grid** to produce output and it defines lots of viewports to draw in. In this case, there are six viewports created, as shown below.

```
> current.vpTree()
```

```
viewport[ROOT]->(
  viewport[plot1.toplevel.vp]->(
    viewport[plot1.],
    viewport[plot1.panel.1.1.off.vp],
    viewport[plot1.panel.1.1.vp],
    viewport[plot1.strip.1.1.off.vp],
    viewport[plot1.xlab.vp],
    viewport[plot1.ylab.vp]))
```

What we are going to do is add output to the **lattice** plot by navigating to a couple of the viewports that **lattice** set up and draw a border around the viewport to show where it is. We will use the `frame()` function defined below to draw a thick grey rectangle around the viewport, then a filled grey rectangle at the top-left corner, and the name of the viewport in white within that.

```
> frame <- function(name) {
    grid.rect(gp=gpar(lwd=3, col="grey"))
    grid.rect(x=0, y=1,
              height=unit(1, "lines"),
              width=1.2*stringWidth(name),
              just=c("left", "top"),
              gp=gpar(col=NA, fill="grey"))
    grid.text(name,
              x=unit(2, "mm"),
              y=unit(1, "npc") -
                unit(0.5, "lines"),
              just="left",
              gp=gpar(col="white"))
  }
```

The viewports used to draw the **lattice** plot consist of a single main viewport called (in this case) `"plot1.toplevel.vp"`, and a number of other viewports within that for drawing the various components of the plot. This means that the viewport tree has three levels (including the top-most `ROOT` viewport). With a multipanel **lattice** plot, there can be many more viewports created, but the naming scheme for the viewports uses a simple pattern and

[3]This low-level **grid** mechanism is general and available for any graphics output produced using **grid** including **lattice** output. An additional higher-level interface has also been provided specifically for **lattice** plots (e.g., the `trellis.focus()` function).

[4]The call to `trellis.par.set()` sets **lattice** graphical parameters so that the colours used in drawing the histogram are those used on a PDF device. This explicit control of the **lattice** "theme" makes it easier to exactly reproduce the histogram output in a later example.

there are **lattice** functions provided to generate the appropriate names (e.g., trellis.vpname()).

When there are more than two levels, there are two ways to specify a particular low-level viewport. By default, downViewport() will search within the children of a viewport, so a single viewport name will work as before. For example, the following code annotates the **lattice** plot by navigating to the viewport "plot1.panel.1.1.off.vp" and drawing a frame to show where it is. This example also demonstrates the fact that downViewport() returns a value indicating how many viewports were descended. This "'depth" can be passed to upViewport() to ensure that the correct number of viewports are ascended after the annotation.

```
> depth <-
    downViewport("plot1.panel.1.1.off.vp")
> frame("plot1.panel.1.1.off.vp")
> upViewport(depth)
```

Just using the final destination viewport name can be ambiguous if more than one viewport in the viewport tree has the same name, so it is also possible to specify a *viewport path*. A viewport path is a list of viewport names that must be matched in order from parent to child. For example, this next code uses an explicit viewport path to frame the viewport "plot1.xlab.vp" that is a child of the viewport "plot1.toplevel.vp".

```
> depth <-
    downViewport(vpPath("plot1.toplevel.vp",
                        "plot1.xlab.vp"))
> frame("plot1.xlab.vp")
> upViewport(depth)
```

It is also possible to use a viewport name or viewport path in the vp argument to drawing functions, in which case the output will occur in the named viewport. In this case, the viewport path is strict, which means that the full path must be matched starting from the context in which the grob was drawn. The following code adds a dashed white line to the borders of the frames. This example also demonstrates that viewport paths can be specified as explicit strings, with a "::" path separator.

The final output after all of these annotations is shown in Figure 4.

```
> grid.rect(
    gp=gpar(col="white", lty="dashed"),
    vp="plot1.toplevel.vp::plot1.panel.1.1.off.vp")
> grid.rect(
    gp=gpar(col="white", lty="dashed"),
    vp="plot1.toplevel.vp::plot1.xlab.vp")
```

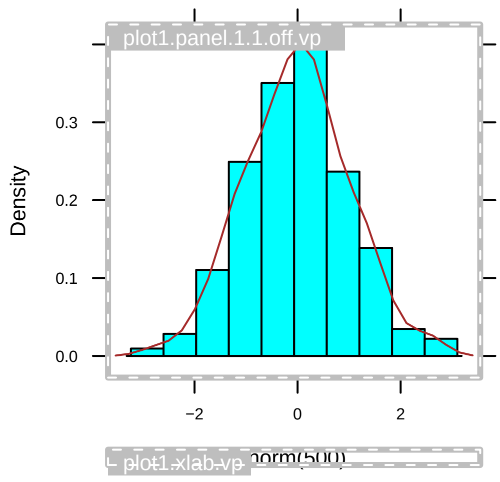

Figure 4: Annotating the simple **lattice** plot.

Changes to graphical objects

All **grid** drawing functions create graphical objects representing the graphical output. This is good because it makes it possible to manipulate these objects to modify the current scene, but there needs to be a coherent mechanism for working with these objects.

There are several new functions, and some existing functions have been made more flexible, to provide a consistent and straightforward mechanism for accessing, modifying, and removing graphical objects.

All **grid** functions that produce graphical output, also create a graphical object, or grob, that represents the output, and there are functions to access, modify, and remove these graphical objects. This can be used as an alternative to modifying and rerunning the R code to modify a plot. In some situations, it will be the only way to modify a low-level feature of a plot.

Consider the output from the simple viewport example (shown in Figure 2). There are seven pieces of output, each with a corresponding grob: two rectangles, "rect1" and "rect2"; two lines, "line1" and "line2"; and three sets of circles, "circles1", "circles2", and "circles3". A particular piece of output can be modified by modifying the corresponding grob; a particular grob is identified by its name.

The names of all (top-level) grobs in the current scene can be obtained using the getNames() function.[5]

[5]Only introduced in R version 2.1.0; a less efficient equivalent for version 2.0.0 is to use grid.get().

```
> getNames()
```

```
[1] "rect1"    "circles1" "line1"    "rect2"
[5] "circles2" "line2"    "circles3"
```

The following code uses the `grid.get()` function to obtain a copy of all of the `grobs`. The first argument specifies the names(s) of the `grob`(s) that should be selected; the `grep` argument indicates whether the name should be interpreted as a regular expression; the `global` argument indicates whether to select just the first match or all possible matches.

```
> grid.get(".*", grep=TRUE, global=TRUE)
```

```
(rect[rect1], circle[circles1], lines[line1],
 rect[rect2], circle[circles2], lines[line2],
 circle[circles3])
```

The value returned is a `gList`; a list of one or more `grobs`. This is a useful way to obtain copies of one or two objects representing some portion of a scene, but it does not return any information about the context in which the `grobs` were drawn, so, for example, just drawing the `gList` is unlikely to reproduce the original output (for that, see the `grid.grab()` function below).

The following code makes use of the `grid.edit()` function to change the colour of the grey circles to black (see Figure 5). In general, most arguments provided in the creation of the output are available for editing (see the documentation for individual functions). It should be possible to modify the gp argument for all `grobs`.

```
> grid.edit("circles3", gp=gpar(col="black"))
```

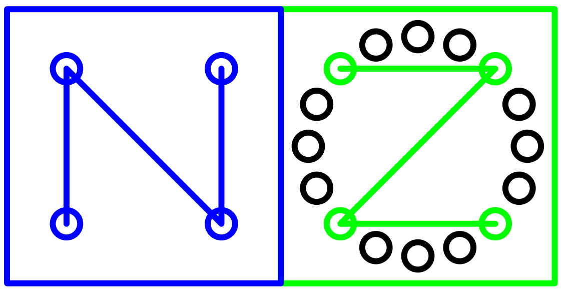

Figure 5: Editing **grid** output.

The following code uses `grid.remove()` to delete all of the `grobs` whose names end with a `"2"` – all of the blue output (see Figure 6).

```
> grid.remove(".+2$", grep=TRUE, global=TRUE)
```

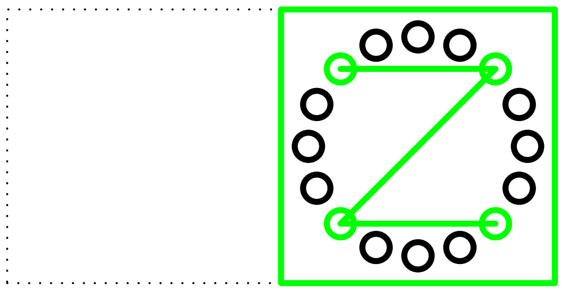

Figure 6: Deleting **grid** output.

These manipulations are possible with any **grid** output. For example, in addition to lots of viewports, a **lattice** plot creates a large number of `grobs` and all of these can be accessed, modified, and deleted using the **grid** functions (an example is given at the end of the article).

Hierarchical graphical objects

Basic `grobs` simply consist of a description of something to draw. As shown above, it is possible to get a copy of a `grob`, modify the description in a `grob`, and/or remove a `grob`.

It is also possible to create and work with more complicated graphical objects that have a tree structure, where one graphical object is the parent of one or more others. Such a graphical object is called a `gTree`. The functions described so far all work with `gTrees` as well, but there are some additional functions just for creating and working with `gTrees`.

In simple usage, a `gTree` just groups several `grobs` together. The following code uses the `grid.grab()` function to create a `gTree` from the output in Figure 6.

```
> nzgrab <- grid.grab(name="nz")
```

If we draw this `gTree` on a new page, the output is exactly the same as Figure 6, but there is now only one graphical object in the scene: the `gTree` called `"nz"`.

```
> grid.newpage()
```

```
> grid.draw(nzgrab)
```

```
> getNames()
```

```
[1] "nz"
```

This gTree has four children; the four original grobs that were "grabbed". The function childNames() prints out the names of all of the child grobs of a gTree.

```
> childNames(grid.get("nz"))
```

```
[1] "rect1"    "circles1" "line1"
[4] "circles3"
```

A gTree contains viewports used to draw its child grobs as well as the grobs themselves, so the original viewports are also available.

```
> current.vpTree()
```

```
viewport[ROOT]->(
  viewport[leftvp],
  viewport[rightvp])
```

The grid.grab() function works with any output produced using **grid**, including **lattice** plots, and there is a similar function grid.grabExpr(), which will capture the output from an expression.[6] The following code creates a gTree by capturing an expression to draw the histogram in Figure 3.

```
> histgrab <- grid.grabExpr(
    { trellis.par.set(canonical.theme("pdf"));
      print(hist1) },
    name="hist", vp="leftvp")
```

The grid.add() function can be used to add further child grobs to a gTree. The following code adds the histogram gTree as a child of the gTree called "nz". An important detail is that the histogram gTree was created with vp="leftvp"; this means that the histogram gets drawn in the viewport called "leftvp" (i.e., in the left half of the scene). The output now looks like Figure 7.

```
> grid.add("nz", histgrab)
```

There is still only one graphical object in the scene, the gTree called "nz", but this gTree now has five children: two lots of circles, one line, one rectangle, and a **lattice** histogram (as a gTree).

```
> childNames(grid.get("nz"))
```

```
[1] "rect1"    "circles1" "line1"
[4] "circles3" "hist"
```

The functions for accessing, modifying, and removing graphical objects all work with hierarchical graphical objects like this. For example, it is possible to remove a specific child from a gTree using grid.remove().

When dealing with a scene that includes gTrees, a simple grob name can be ambiguous because, by default, grid.get() and the like will search within the children of a gTree to find a match.

Just as you can provide a viewport path in downViewport() to unambiguously specify a particular viewport, it is possible to provide a grob path in grid.get(), grid.edit(), or grid.remove() to unambiguously specify a particular grob.

A grob path is a list of grob names that must be matched in order from parent to child. The following code demonstrates the use of the gPath() function to create a grob path. In this example, we are going to modify one of the grobs that were created by **lattice** when drawing the histogram. In particular, we are going to modify the grob called "plot1.xlab" which represents the x-axis label of the histogram.

In order to specify the x-axis label unambiguously, we construct a grob path that identifies the grob "plot1.xlab", which is a child of the grob called "hist", which itself is a child of the grob called "nz". That path is used to modify the grob which represents the xaxis label on the histogram. The xaxis label is moved to the far right end of its viewport and is drawn using a bold-italic font face (see Figure 8).

```
> grid.edit(gPath("nz", "hist", "plot1.xlab"),
            x=unit(1, "npc"), just="right",
            gp=gpar(fontface="bold.italic"))
```

Summary

When a scene is created using the **grid** graphics system, a tree of viewport objects is created to represent the drawing regions used in the construction of the scene and a list of graphical objects is created to represent the actual graphical output.

Functions are provided to view the viewport tree and navigate within it in order to add further output to a scene.

Other functions are provided to view the graphical objects in a scene, to modify those objects, and/or remove graphical objects from the scene. If graphical objects are modified or removed, the scene is redrawn to reflect the changes. Graphical objects can also be grouped together in a tree structure and dealt

[6]The function grid.grabExpr() is only available in R version 2.1.0; the grid.grab() function could be used instead by explicitly opening a new device, drawing the histogram, and then capturing it.

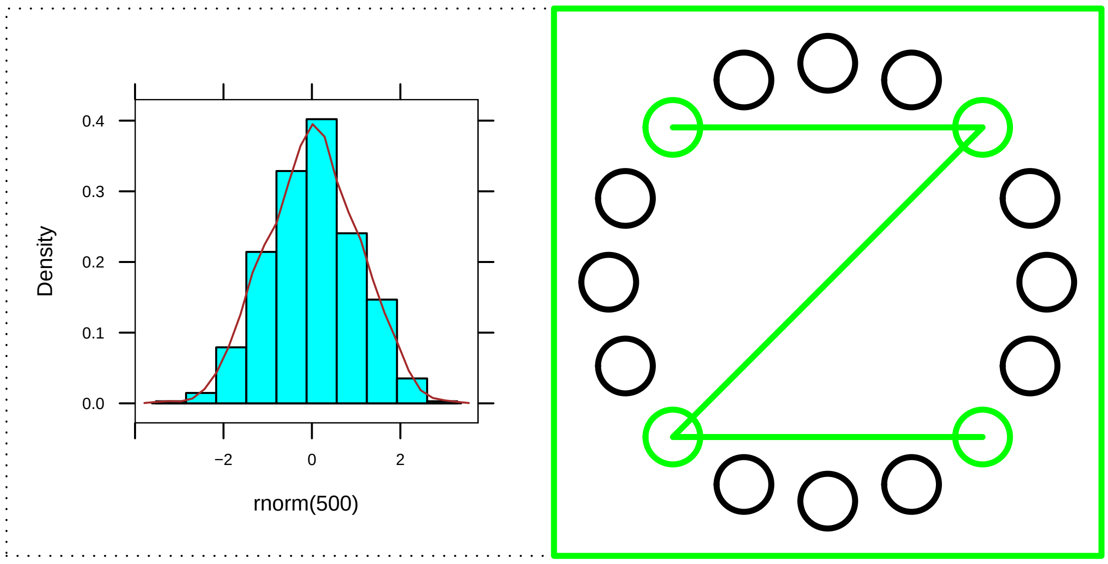

Figure 7: Grabbing **grid** output.

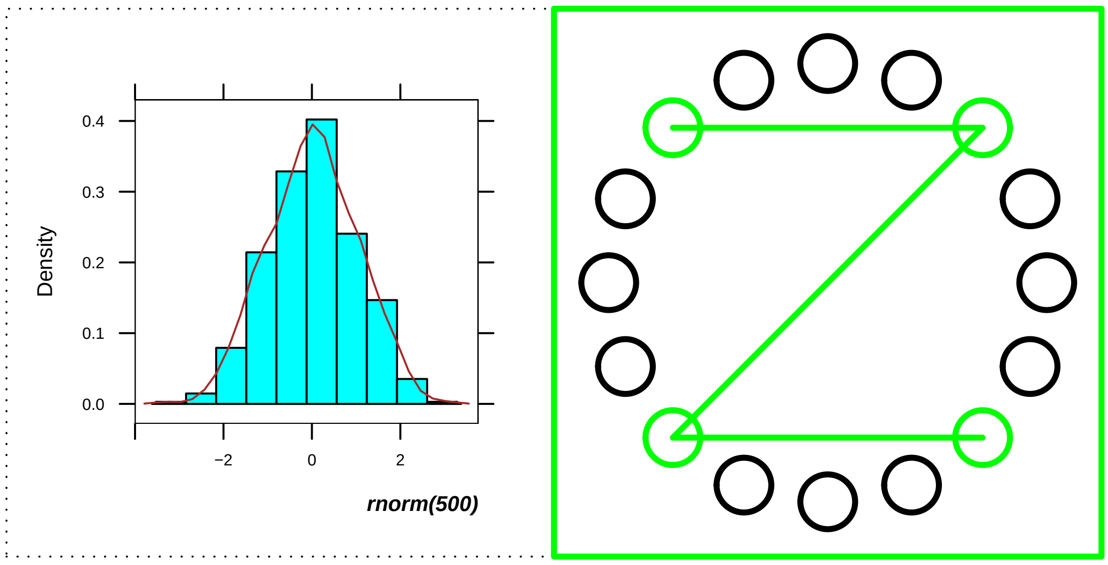

Figure 8: Editing a `gTree`.

Table 1: Summary of the new and changed functions in R 2.0.0 relating to viewports.

Function	Description
`pushViewport()`	Create a region for drawing on the current graphics device.
`popViewport()`	Remove the current drawing region (does not delete any output).
`upViewport()`	Navigate to parent drawing region (but do not remove current region).
`downViewport()`	Navigate down to named drawing region.
`current.viewport()`	Return the current viewport.
`current.vpTree()`	Return the current tree of viewports that have been created on the current device.
`vpPath()`	Create a viewport path; a concatenated list of viewport names.
`viewport(..., name=)`	A viewport can have a name associated with it.

Table 2: Summary of the new and changed functions since R 2.0.0 relating to graphical objects (some functions only available since R 2.1.0).

Function	Description
`grid.get()`	Return a single `grob` or a `gList` of `grobs`.
`grid.edit()`	Modify one or more `grobs` and (optionally) redraw.
`grid.remove()`	Remove one or more `grobs` and (optionally) redraw.
`grid.add()`	Add a `grob` to a `gTree`.
`grid.grab()`	Create a `gTree` from the current scene.
`grid.grabExpr()`	Create a `gTree` from an R expression (only available since R 2.1.0).
`gPath()`	Create a `grob` path; a concatenated list of `grob` names.
`getNames()`	List the names of the top-level `grobs` in the current scene (only available since R 2.1.0).
`childNames()`	List the names of the children of a `gTree`.
`grid.rect(..., name=)`	Grid drawing functions can associate a name with their output.

with as a single unit.

Bibliography

P. Murrell. The grid graphics package. *R News*, 2(2): 14–19, June 2002. URL http://CRAN.R-project.org/doc/Rnews/. 12

D. Sarkar. Lattice. *R News*, 2(2):19–23, June 2002. URL http://CRAN.R-project.org/doc/Rnews/. 12

Paul Murrell
University of Auckland, New Zealand
pmurrell@auckland.ac.nz

hoa: An R package bundle for higher order likelihood inference

by Alessandra R. Brazzale

Introduction

The likelihood function represents the basic ingredient of many commonly used statistical methods for estimation, testing and the calculation of confidence intervals. In practice, much application of likelihood inference relies on first order asymptotic results such as the central limit theorem. The approximations can, however, be rather poor if the sample size is small or, generally, when the average information available per parameter is limited. Thanks to the great progress made over the past twenty-five years or so in the theory of likelihood inference, very accurate approximations to the distribution of statistics such as the likelihood ratio have been developed. These not only provide modifications to well-established approaches, which result in more accurate inferences, but also give insight on when to rely upon first order methods. We refer to these developments as *higher order asymptotics*.

One intriguing feature of the theory of higher order likelihood asymptotics is that relatively simple and familiar quantities play an essential role. The higher order approximations discussed in this paper are for the significance function, which we will use to set confidence limits or to calculate the p-value associated with a particular hypothesis of interest. We start with a concise overview of the approximations used in the remainder of the paper. Our first example is an elementary one-parameter model where one can perform the calculations easily, chosen to illustrate the potential accuracy of the procedures. Two more elaborate examples, an application of binary logistic regression and a nonlinear growth curve model, follow. All examples are carried out using the R code of the `hoa` package bundle.

Basic ideas

Assume we observed n realizations $y_1, \ldots, y_n$ of independently distributed random variables $Y_1, \ldots, Y_n$ whose density function $f(y_i; \theta)$ depends on an unknown parameter θ. Let $\ell(\theta) = \sum_{i=1}^n \log f(y_i; \theta)$ denote the corresponding log likelihood and $\hat{\theta} = \text{argmax}_\theta \ell(\theta)$ the maximum likelihood estimator. In almost all applications the parameter θ is not scalar but a vector of length d. Furthermore, we may re-express it as $\theta = (\psi, \lambda)$, where ψ is the d_0-dimensional *parameter of interest*, about which we wish to make inference, and λ is a so-called *nuisance parameter*, which is only included to make the model more realistic.

Confidence intervals and p-values can be computed using the *significance function* $p(\psi; \hat{\psi}) = \Pr(\hat{\Psi} \leq \hat{\psi}; \psi)$ which records the probability left of the observed "data point" $\hat{\psi}$ for varying values of the unknown parameter ψ (Fraser, 1991). Exact elimination of λ, however, is possible only in few special cases (Severini, 2000, Sections 8.2 and 8.3). A commonly used approach is to base inference about ψ on the *profile log likelihood* $\ell_p(\psi) = \ell(\psi, \hat{\lambda}_\psi)$, which we obtain from the log likelihood function by replacing the nuisance parameter with its constrained estimate $\hat{\lambda}_\psi$ obtained by maximising $\ell(\theta) = \ell(\psi, \lambda)$ with respect to λ for fixed ψ. Let $j_p(\psi) = -\partial^2 \ell_p(\psi) / \partial\psi\partial\psi^\top$ denote the observed information from the profile log likelihood. Likelihood inference for scalar ψ is typically based on the

- Wald statistic, $w(\psi) = j_p(\hat{\psi})^{1/2}(\hat{\psi} - \psi)$;
- likelihood root,
$$r(\psi) = \text{sign}(\hat{\psi} - \psi) \left[2\{\ell_p(\hat{\psi}) - \ell_p(\psi)\}\right]^{1/2};$$
or
- score statistic, $s(\psi) = j_p(\hat{\psi})^{-1/2} d\ell_p(\psi)/d\psi$.

Under suitable regularity conditions on $f(y; \theta)$, all of these have asymptotic standard normal distribution