Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Importing Vector Graphics: the grImport package for R

Paul Murrell
The University of Auckland

Abstract

This article describes an approach to importing vector-based graphical images into statistical software. The approach is implemented in a package for the R statistical computing environment called **grImport**. This package consists of three components: a function for converting PostScript files to an XML format; a function for reading the XML format into `Picture` objects in R; and functions for manipulating and drawing `Picture` objects. Several examples and applications are also presented.

Keywords: PostScript, R, statistical graphics, XML.

1. Introduction

One of the important features of statistical software is the ability to create sophisticated statistical plots of data. Software systems such as the **lattice** (Sarkar 2008) package in R (R Development Core Team 2008) can produce complex images from a very compact expression. For example, the following code is all that is needed to produce the image in Figure 1, which shows density estimates for the number of moves in a set of chess games, broken down by the result of the games.

```
> xyplot(Freq ~ nmoves | result, data=chess.df, type="h",
         layout=c(1, 3), xlim=c(0, 100))
```

On the other hand, statistical graphics software does *not* typically provide support for producing more free-form or artistic graphical images. As a very simple example, it would be difficult to produce an image of a chess piece like the pawn shown below, using statistical software.

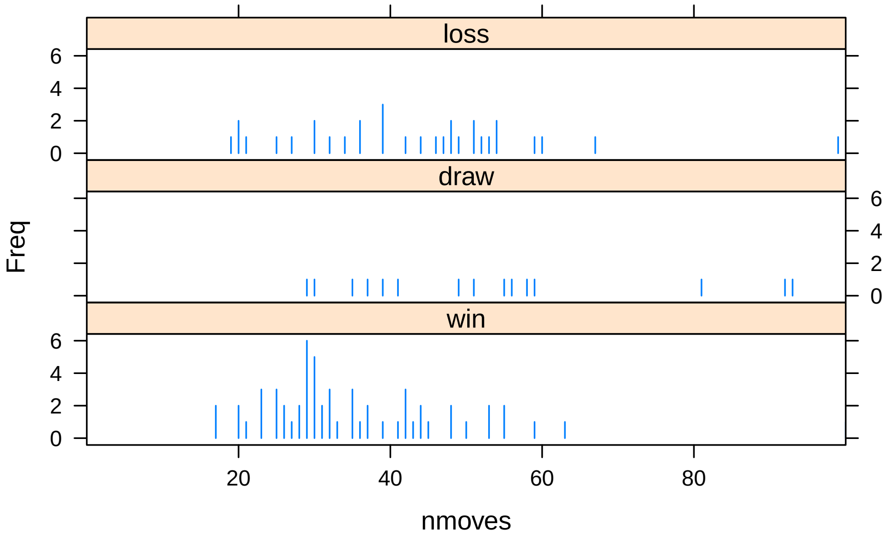
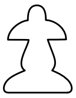


Figure 1: A statistical plot produced in R using the **lattice** package. The data are from chess games involving Louis Charles Mahe De La Bourdonnais between 1821 and 1838 (original source: http://www.chessgames.com/).



In the case of R, there is a general polygon-drawing function, but determining the vertices for the boundary of this pawn image would be non-trivial; these sorts of artistic images are produced much more easily using the tools that are provided by drawing software such as the Gimp (Kylander and Kylander 1999, http://www.gimp.org/) or Inkscape (Bah 2007, http://www.inkscape.org/), not to mention that producing an aesthetically pleasing result for this sort of image also requires a healthy dose of artistic skill.

However, there are situations where it is useful to be able to include artistic images as part of a statistical plot. Ignoring for the moment the principles of effective plot design, a simple example of this sort of situation is the addition of free-form annotations to a plot. Figure 2 demonstrates this sort of annotation by adding a pawn to each panel of the plot from Figure 1, to provide an additional visual cue as to whether the games in the panel were won (white pawn), drawn (grey pawn) or lost (black pawn).

This is one example of the problem that is addressed in this article. Stating the issue more generally, this article is concerned with the ability to import graphical images that have been generated using third party software into a statistical software system so that the images can be manipulated within the statistical software, including incorporating the images within statistical plots.

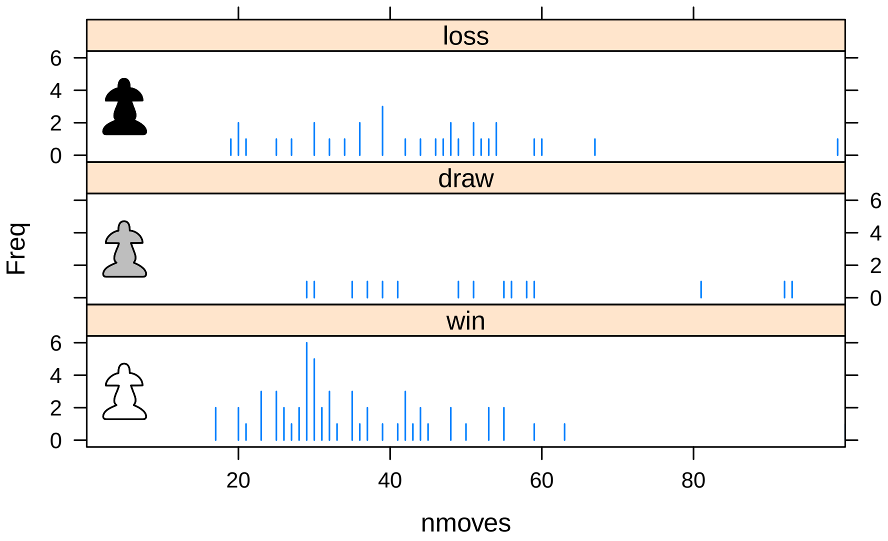


Figure 2: The statistical plot from Figure 1 with a pawn added to each panel.

1.1. Raster images versus vector images

There are two basic types of image formats: *raster* images and *vector* images. A raster image consists of a matrix of *pixels* (picture elements) and the image is represented by recording a different color for each pixel. A vector image consists of a set of mathematical shapes, such as lines and polygons, and and the image is represented by recording the locations and colors of the shapes.

The images below demonstrate the difference between raster and vector images. On the left is a raster image of a circle both at normal size and zoomed to show that the image is made up of a set of square pixels. At an appropriate size, this image looks fine, but it does not scale well. On the right is a vector version of the same image. At the smaller scale, this image appears very similar to the raster image, but the zoomed portion shows that the vector image is made up of a curve and this can be rendered effectively at any size.

Importing raster images is a different problem from importing vector images. For the R system, several packages, including **pixmap** (Bivand, Leisch, and Maechler 2008), **rimage** (Nikon Systems Inc. 2005), and **EBimage** (Sklyar, Pau, Smith, and Huber 2008), provide functions for reading various raster image formats into R.

This article is concerned with reading *vector* image formats into R.

1.2. Vector image formats

The problem addressed in this article is essentially a *translation* problem. The original image, in its original vector format, needs to be translated into a format that R can understand (and draw).

There are many different vector image formats, with some major examples being PostScript (Adobe Systems 1999), PDF (Adobe Systems 2005), and SVG (Ferraiolo, Jun, and Jackson 2003). These are more accurately described as *metaformats*, because they allow an image to consist of both raster and vector components. However, the important point for the current context is that these are very popular formats for storing an image as a description of a set of mathematical shapes.

Rather than attempt to translate all possible vector image formats, the approach taken in this article is to provide tools to translate a single format, PostScript, and rely on other software to translate images in other formats to PostScript. For example, the convert utility from the ImageMagick graphics suite (Still 2005, `http://www.imagemagick.org/`) can be used to convert between a huge variety of graphics formats. For the particular case of PDF to PostScript, the ghostscript (Merz 1997, `http://pages.cs.wisc.edu/~ghost/`) utility pdf2ps is quite effective, and Inkscape has produced good results for converting from SVG to PostScript. There are some limitations to this dependence on PostScript images, but a proper discussion of these will be deferred until Section 5.

The reasons *for* choosing PostScript as the single format to focus on are explained in the next section.

2. The grImport package

The solution that is described in this article for importing vector graphics into statistical software is implemented in an R package called **grImport**.

The solution provided by the **grImport** package consists of three separate steps: transforming from a PostScript image format to a specialized RGML format; importing the RGML format into R data structures; and drawing the R data structures. Each of these steps is described in a separate section below.

2.1. PostScript to XML

The starting point for any import using this system is a PostScript file and the first step in the import process is a translation of this PostScript file to a new file in a special XML (Bray, Paoli, Sperberg-McQueen, Maler, and Yergeau 2006) format called RGML (R Graphics Markup Language).

As a simple example to follow through in detail, Figure 3 shows a file, `petal.ps`, that consists of PostScript code for drawing a simple "petal" shape, which is shown below.

This simple example demonstrates some basic PostScript commands. A shape, called a *path*,

```
%!PS
newpath                    % start a new shape
0 0 moveto                 % move to a start location
-5 10 lineto               % line to a new location
-10 20 10 20 5 10 curveto  % curve to a third location
5 10 lineto                % line to a fourth location
closepath                  % connect back to the start location
0 setgray                  % set the drawing colour to black
fill                       % fill the current shape
```

Figure 3: The file `petal.ps`, which contains PostScript code to draw a simple "petal" shape.

is defined by specifying lines and curves between a set of points and this shape is then filled with a color.

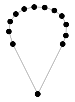
Another common PostScript operation involves drawing just the boundary outline of the shape. This could be achieved in the example in Figure 3 by replacing the command `fill` with the command `stroke` (PostScript calls this *stroking* a path).

The user interface provided by **grImport** for the translation from PostScript to RGML format is very simple, consisting of a single function, `PostScriptTrace()`. In the most basic usage, the only required argument is the name of the PostScript file to translate, as in the code below. The resulting RGML file, `petal.ps.xml` is shown in Figure 4.

```
> PostScriptTrace("petal.ps")
```

The RGML format in this example is roughly a one-to-one translation of the PostScript code. The shape is recorded as a `<path>` element that has a `type` attribute with the value `fill` indicating that the shape should be filled. The `<context>` element for this shape specifies the colour to be used to fill the shape and then a series of `<move>` and `<line>` elements describe the outline of the shape itself. A `<summary>` element provides information on how many paths there are in the image, plus bounding box information.

One detail to notice is that the `curveto` in the PostScript file has become a series of `<line>` elements in the RGML file. We will discuss this issue further in Section 3. The main point to focus on for now is that the image has become a set of (x, y) locations that describe the outline of the shape in the image, as illustrated below.



One reason for choosing PostScript as the original format to focus on is that it is a sophisticated graphics language. PostScript has commands to draw a wide variety of shapes and PostScript provides advanced facilities to control the placement of shapes and to control such things as the colors and line styles for filling and stroking the shapes. This means that PostScript is capable of describing very complex images; by focusing on PostScript we should be able to import virtually any vector image no matter how complicated it is.

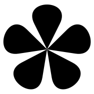
```
<?xml version=’1.0’?>

<picture version=’2’ xmlns:rgml=’http://r-project.org/RGML’>

<path type=’fill’ id=’1’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>

    <move y=’0.0’ x=’0.0’/>
    <line y=’100.0’ x=’-50.0’/>
    <line y=’109.082’ x=’-53.9705’/>
    <line y=’117.578’ x=’-56.5674’/>
    <line y=’125.488’ x=’-57.8828’/>
    <line y=’132.812’ x=’-58.0078’/>
    <line y=’139.551’ x=’-57.0344’/>
    <line y=’145.703’ x=’-55.0537’/>
    <line y=’151.27’ x=’-52.1577’/>
    <line y=’156.25’ x=’-48.4375’/>
    <line y=’160.645’ x=’-43.9851’/>
    <line y=’164.453’ x=’-38.8916’/>
    <line y=’167.676’ x=’-33.249’/>
    <line y=’170.312’ x=’-27.1484’/>
    <line y=’172.363’ x=’-20.6819’/>
    <line y=’173.828’ x=’-13.9404’/>
    <line y=’174.707’ x=’-7.01611’/>
    <line y=’175.0’ x=’0.0’/>
    <line y=’174.707’ x=’7.01587’/>
    <line y=’173.828’ x=’13.9404’/>
    <line y=’172.363’ x=’20.6816’/>
    <line y=’170.312’ x=’27.1484’/>
    <line y=’167.676’ x=’33.2488’/>
    <line y=’164.453’ x=’38.8916’/>
    <line y=’160.645’ x=’43.9849’/>
    <line y=’156.25’ x=’48.4375’/>
    <line y=’151.27’ x=’52.1575’/>
    <line y=’145.703’ x=’55.0537’/>
    <line y=’139.551’ x=’57.0342’/>
    <line y=’132.812’ x=’58.0078’/>
    <line y=’125.488’ x=’57.8826’/>
    <line y=’117.578’ x=’56.5674’/>
    <line y=’109.082’ x=’53.9702’/>
    <line y=’100.0’ x=’50.0’/>
    <line y=’100.0’ x=’50.0’/>
    <line y=’0.0’ x=’0.0’/>
</path>

<summary count=’1’ ymax=’175.0’ ymin=’0.0’ xmax=’58.0078’ xmin=’-58.0078’/>

</picture>
```

Figure 4: The file `petal.ps.xml`, which contains RGML code created by calling `PostScriptTrace()` on the PostScript code in Figure 3.

```
%!PS
/petal {
    newpath
    0 0 moveto
    -5 10 lineto
    -10 20 10 20 5 10 curveto
    5 10 lineto
    closepath
    0 setgray
    fill
} def
20 20 translate
5 {
    petal 72 rotate
} repeat
showpage
```

Figure 5: The file `flower.ps`, which contains PostScript code to draw a simple "flower" shape.

PostScript is also a complete programming language. As a simple demonstration of this, Figure 5 shows a file, `flower.ps`, that contains PostScript code for drawing a simple "flower" shape, which is shown below.



The important feature of this PostScript code is that it defines a "macro" that describes how to draw a petal, then it runs this macro five times (at five different angles) to produce the overall flower.

This complexity presents an imposing challenge for us. How can we translate PostScript code when the code can be extremely complicated? The approach taken by the **grImport** package is to use the power of PostScript against itself.

The first part of the solution is based on the fact that it is possible to write PostScript code that runs other PostScript code. The basis for the translation from an original PostScript file to an RGML file is a set of additional PostScript code that processes the original PostScript file.

The other part of the solution is based on the fact that it is possible to *redefine* some of the core PostScript commands. For example, the **grImport** PostScript code redefines the meaning of the PostScript commands `stroke` and `fill` so that, instead of drawing shapes, these commands print out information about the shapes that would have been drawn.

Figure 6 shows a very simplified version of how the **grImport** PostScript translation code works. This code first defines a macro, `printtwo` that prints out two values. It also defines a macro, `donothing`, which does nothing. The next macro, `fill`, is the important one. This is *redefining* the standard PostScript `fill` command. This macro, instead of filling a shape, breaks any curves in the current path into short line segments (`flattenpath`), then it

```
%!PS
/printtwo {
    /printnum {
        100 mul round 100 div 20 string cvs print
    } def
    printnum
    ( ) print
    printnum
    (\n) print
} def
/donothing { } def
/fill {
    flattenpath
    {printtwo} {printtwo} {donothing} {donothing} pathforall
} def
(petal.ps) run
```

Figure 6: The file `translate.ps`, which contains PostScript code to process the PostScript file `petal.ps`.

calls the `pathforall` command. This command converts the current path into four possible operations: a move, a line, a curve, or a closing of the path. The four values in front of `pathforall` specify what to do for each of these operations. Overall, the code says that, if there is a move or a line in the path, then we should print out two values (the position moved to or the position lined to). For curves and closes, we do nothing. The final line of code in Figure 6 says to run the PostScript code in the file `petal.ps`.

The PostScript code in the **grImport** package is a lot more complicated than the code in Figure 6, but this demonstrates the main idea of the code.

At this point, we have new PostScript code that can process the original PostScript code, but we still need software to *run* the new PostScript code. The **grImport** package uses ghostscript for this purpose. For example, the code below shows how to run the simplified translation code in Figure 6, with the resulting output shown below that. Several of the values printed out should be recognisable from the PostScript code in the file `petal.ps` (see Figure 3).

```
$ gs -dBATCH -dQUIET -dNOPAUSE translate.ps
0.0 0.0
10.0 -5.0
10.41 -4.78
11.41 -4.14
12.64 -3.15
13.75 -1.87
14.39 -0.36
14.22 1.33
12.87 3.13
10.0 5.0
10.0 5.0
```

The beauty of this solution is that, no matter how complicated the PostScript code gets, it ultimately calls `stroke` or `fill` to do the actual drawing. For example, the code in Figure 5 performs a loop to draw five petals, but we do not have to write code that understands PostScript loops; all we have to do is to ensure that *whenever* the PostScript code ultimately tries to fill one of the petals, we intervene and simply print out the information about the petal instead.

Figure 7 shows the RGML file that results from running `PostScriptTrace()` on the PostScript code in the file `flower.ps`. Many of the `<line>` elements have been left out in order to show the overall structure of the file.

```
> PostScriptTrace("flower.ps")
```

The overall effect is that the PostScript *program* in the file `flower.ps` has become a much longer, but much simpler RGML file consisting simply of descriptions of the five shapes that would have been drawn if the PostScript file had been viewed normally.

The PostScript code that is used to perform the translation from the original PostScript file to an RGML file can be found within the file `PostScript2RGML.R` of the **grImport** package.

At this point, there might appear to be little cause for celebration. All we have achieved is to convert the PostScript file into an RGML file. It is important to highlight how much closer that has taken us to working with the image in R.

The main point is that the RGML format is *simple*. An RGML file *only* contains shape descriptions, so all R has to do is read the information about each shape and draw it. With the **XML** package (Temple Lang 2008), reading an XML file into R is straightforward and R has facilities for drawing each of the shapes in the RGML file.

A secondary point is that the RGML format is XML code. This is useful because XML can be produced and consumed by many different software systems. For example, it would be quite straightforward to write XSL (Clark 1999) code that would translate an RGML file to SVG with the help of the xsltproc utility from the libxml library (Veillard 2008).

Because the RGML format would also be easy for other statistical software systems to read, this format could be the basis for importing vector images into other statistical systems.

Another important class of software that can work with XML documents is text editor software. One of the nice features of XML code is that it can be viewed and modified with very simple tools. In this context, basic image editing can be performed with a text editor.

Taking this idea a bit further, with the image in a very simple plain text format, more complex and large-scale edits become simple text processing tasks. For example, the following R code uses a search-and-replace to modify the `<rgb>` elements in the RGML file `flower.ps.xml` so that the flower is filled with a blue color instead of being black. The modified flower image is shown below the R code.

```
> flowerRGML <- readLines("flower.ps.xml")
> blueFlower <- gsub("<rgb.+$",
                     "<rgb r='0.3' g='0.6' b='0.8'/>",
                     flowerRGML)
> writeLines(blueFlower, "blueflower.ps.xml")
```

```
<?xml version=’1.0’?>
<picture version=’2’ xmlns:rgml=’http://r-project.org/RGML’>
<path type=’fill’ id=’1’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>
    <move y=’200.0’ x=’200.0’/>
    ...
    <line y=’200.0’ x=’200.0’/>
</path>
<path type=’fill’ id=’2’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>
    <move y=’200.0’ x=’200.0’/>
    ...
    <line y=’200.0’ x=’200.0’/>
</path>
<path type=’fill’ id=’3’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>
    <move y=’200.0’ x=’200.0’/>
    ...
    <line y=’200.0’ x=’200.0’/>
</path>
<path type=’fill’ id=’4’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>
    <move y=’200.0’ x=’200.0’/>
    ...
    <line y=’200.0’ x=’200.0’/>
</path>
<path type=’fill’ id=’5’>
    <context>
        <rgb r=’0.0’ g=’0.0’ b=’0.0’/>
        <style lwd=’1.33’/>
    </context>
    <move y=’200.0’ x=’200.0’/>
    ...

    <line y=’200.0’ x=’200.0’/>
</path>
<summary count=’5’ ymax=’375.0’ ymin=’44.0947’ xmax=’370.366’ xmin=’29.634’/>
</picture>
```

Figure 7: The file `flower.ps.xml`, which contains RGML code created by calling `PostScriptTrace()` on the PostScript code in Figure 5. Most of the `<line>` elements have been removed and replaced with `...` so that the overall structure of the complete file can be displayed.

A final point is that once the image has been converted into the RGML format, there is no further need for ghostscript. The image can be freely shared between users, with the only requirement being the availability of R (and the **XML** package).

In summary, the **grImport** package provides a function called `PostScriptTrace()`, which uses ghostscript to process an original PostScript file and convert it into an RGML file.

2.2. XML to R

The next step in importing a PostScript image into R involves reading the RGML format into R.

As mentioned previously, reading XML files is straightforward with the **XML** package. However, the R objects that are generated by the functions in the **XML** package are very general-purpose, so the **grImport** package provides a function that produces an R object that is specifically designed for representing a graphical image.

The function used to read RGML files is called `readPicture()`. This function has only one argument, which is the name of the RGML file. The following code uses this function to read the petal image from the file `petal.ps.xml`.

```
> petal <- readPicture("petal.ps.xml")
```

The resulting object, `petal`, is a `Picture` object, with two slots: one slot contains all of the paths from the image and the other slot contains the summary information about the image. In this case, there is only one path and it is a `PictureFill` object (i.e., a shape that should be filled with a color).

```
> str(petal)

Formal class 'Picture' [package "grImport"] with 2 slots
  ..@ paths  :List of 1
  .. ..$ path:Formal class 'PictureFill' [package "grImport"] with 4 slots
  .. .. .. ..@ x  : Named num [1:36] 0 -50 -54 -56.6 -57.9 ...
  .. .. .. .. ..- attr(*, "names")= chr [1:36] "move" "line" "line" "line" ...
  .. .. .. ..@ y  : Named num [1:36] 0 100 109 118 125 ...
  .. .. .. .. ..- attr(*, "names")= chr [1:36] "move" "line" "line" "line" ...
  .. .. .. ..@ rgb: chr "#000000"
  .. .. .. ..@ lwd: num 1.33
  ..@ summary:Formal class 'PictureSummary' [package "grImport"] with 3 slots
  .. .. ..@ numPaths: Named num 1
  .. .. .. ..- attr(*, "names")= chr "count"
  .. .. ..@ xscale  : Named num [1:2] -58 58
  .. .. .. ..- attr(*, "names")= chr [1:2] "xmin" "xmax"
  .. .. ..@ yscale  : Named num [1:2] 0 175
  .. .. .. ..- attr(*, "names")= chr [1:2] "ymin" "ymax"
```

The Picture object has a clear one-to-one coprrespondence with the information in the XML file and, again, we might question what we have gained by generating this object. Why not just draw the information from the RGML file directly?

The main reason for having the special S4 class of Picture objects in R is that we can work with the image using all of the powerful data processing tools that are available in R. One specific example that is explicitly supported by the **grImport** package is the ability to subset paths from an image.

As a simple example of subsetting, consider the Picture object that is generated by reading in the RGML file that was generated from the PostScript file flower.ps (see Figure 7). Only the summary information for this Picture object is shown.

```
> flower <- readPicture("flower.ps.xml")
```

```
> str(flower@summary)
```

```
Formal class 'PictureSummary' [package "grImport"] with 3 slots
  ..@ numPaths: Named num 5
  .. ..- attr(*, "names")= chr "count"
  ..@ xscale  : Named num [1:2] 29.6 370.4
  .. ..- attr(*, "names")= chr [1:2] "xmin" "xmax"
  ..@ yscale  : Named num [1:2] 44.1 375
  .. ..- attr(*, "names")= chr [1:2] "ymin" "ymax"
```

This Picture object has five paths, corresponding to the five petals. A subsetting method for Picture objects is defined by the **grImport** package so that we can extract just some of the petals from the image as shown in the code below.

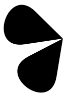
```
> petals <- flower[2:3]
```

The result is a new Picture object consisting of just the second and third paths from the original Picture object. As the code below shows, the summary information has been updated as well.

```
> str(petals@summary)
```

```
Formal class 'PictureSummary' [package "grImport"] with 3 slots
  ..@ numPaths: int 2
  ..@ xscale  : num [1:2] 29.6 200
  ..@ yscale  : num [1:2] 44.1 297.4
```

Visually, the new picture is just the second and third petals from the original image, as shown below.



In summary, the **grImport** package provides a function `readPicture()` that reads an RGML file and creates a `Picture` object. `Picture` objects are used to draw the image, but they can also be manipulated to modify the image. For example, a `Picture` object can be subsetted to extract individual paths from the overall image.

2.3. R to grid

Having read an RGML file into R as a `Picture` object, the final step is to draw the `Picture` object.

Conceptually, this step is very straightforward. A path is just a set of (x, y) pairs and R graphics functions such as `lines()` and `polygon()`, in the **graphics** package, and `grid.lines()` and `grid.polygon()`, in the **grid** package, can be used to stroke or fill these paths (Murrell 2005).

The main inconvenience in this step lies in dealing with *coordinate systems*. As the code below demonstrates for the `petal Picture`, the (x, y) locations for an image can be on an arbitrary scale.

```
> petal@summary@xscale

    xmin     xmax
-58.0078  58.0078

> petal@summary@yscale

ymin ymax
   0  175
```

In order to position and size the image in useful ways, the (x, y) locations for the paths need to be scaled. Viewports in the **grid** package provide a convenient way to establish appropriate coordinate systems for drawing, so the **grImport** package provides several functions based on **grid** for drawing `Picture` objects.

The first of these, the `grid.symbols()` function, can be used to draw several copies of a `Picture` object at a set of (x, y) locations and at a specified size. The following code makes use of this function to draw the `flower` image as data symbols on a **lattice** scatterplot (see Figure 8). The important arguments are the `Picture` object to draw, the (x, y) locations (and the coordinate system that those locations refer to), and the size of the individual images (in this case, each flower image is 5mm high).

```
> library(cluster)

> xyplot(V8 ~ V7, data=get("flower", "package:cluster"),
         xlab="Height", ylab="Distance Apart",
         panel=function(x, y, ...) {
             grid.symbols(flower, x, y, units="native",
                          size=unit(5, "mm"))
         })
```

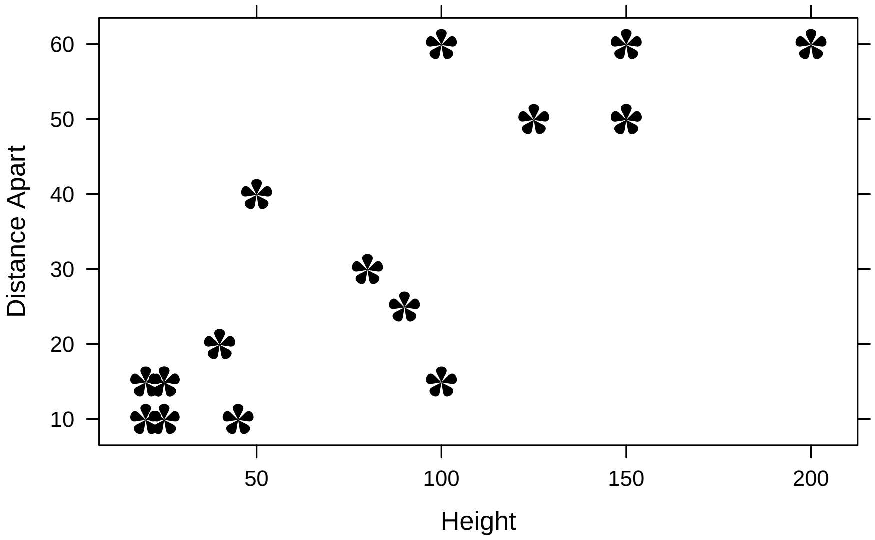


Figure 8: A statistical plot produced in R using the **lattice** package, with an imported "flower" image used as the plotting symbol. The data are the heights of 18 popular flower varieties and the distance that should be left between plants when sowing seeds. These data are in a data frame called `flower` in the **cluster** package.

This example also demonstrates one of the major reasons for going to all of the effort to *import* an image into R in order to combine it with an R plot. An alternative approach to adding an image to a plot is to only create the plot using R and then combine that plot with other images using tools such as ImageMagick's compose utility. However, the problem with that approach is that it is impractical, if not impossible, to position the images relative to the coordinate systems within the plot.

By importing an image to R, the image can be drawn within the same set of coordinate systems that are used to produce the plot, so the positioning of images is straightforward and accurate.

In addition to the `grid.symbols()` function, the **grImport** package also provides a `grid.picture()` function. This is used to add a single copy of an image to a page.

The `grid.picture()` function also provides a little more flexibility in how the image is drawn, compared to the `grid.symbols()` function; an example of this flexibility will be described in Section 3.7.

As a simple demonstration of the `grid.picture()` function, the following code translates, reads, and draws the "tiger" example PostScript file that is distributed with ghostscript (minus its grey background rectangle). The tiger image to the right of the code is produced by R.

```
> PostScriptTrace("tiger.ps")
> tiger <- readPicture("tiger.ps.xml")
> grid.picture(tiger[-1])
```

In summary, the **grImport** package provides two functions, `grid.picture()` and `grid.symbols()`, for drawing `Picture` objects. The `grid.picture()` function draws a single copy of the `Picture` at a particular location and size and the `grid.symbols()` function draws several copies of the `Picture` at a set of (x, y) locations.

The overall steps involved in importing an original image into R are as follows: generate a PostScript version of the original image; use `PostScriptTrace()` to convert the image to an RGML format; use `readPicture()` to read the RGML file into a `Picture` object; and use `grid.picture()` or `grid.symbols()` to draw the `Picture` object.

3. Further details

The previous section provided an overview of the structure of the **grImport** solution to importing vector graphics into statistical software. In order to make that overview as straightforward as possible, some important details were ignored; this section fills in some additional details about how the **grImport** package works.

3.1. Flattening PostScript paths

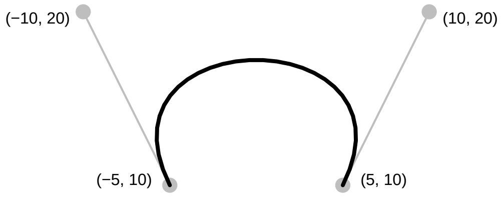
The PostScript language provides four basic operations for constructing a path: *move* to a location, draw a (straight) *line* to a location, draw a *curve* to a location, and show *text* at a location. The discussion in Section 2 only properly addressed moving and drawing lines. The simple petal image and flower image examples did actually include paths with curves, but that was not properly dealt with. We will now look more closely at how curves in PostScript files are handled by **grImport**. Section 3.2 will deal with text.

Looking again at the PostScript code in Figure 3, the path that describes the petal image consists of a move to the location $(0, 0)$, followed by a line to the location $(-5, 10)$, followed by a curve. The PostScript code describing the curve is reproduced below.

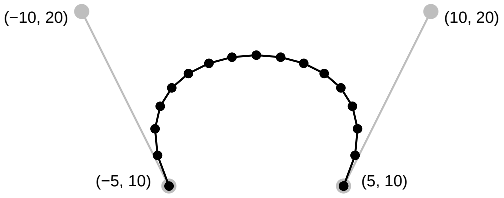
```
-10 20 10 20 5 10 curveto
```

This curve creates the nice round "end" for the petal shape.

In PostScript, these curves are cubic Bézier curves; a smooth curve is drawn from the previous location, in this case $(-5, 10)$, to the last location mentioned in the `curveto` command, $(5, 10)$, with the other two locations, $(-10, 20)$ and $(10, 20)$, specifying *control points* that control the shape of the curve. Specifically, the start of the curve is tangent to a line joining the first two locations and the end of the curve is tangent to a line joining the last two locations, as shown below.



Unfortunately, the R graphics system cannot natively draw Bézier curves, and it does not have the notion of a general path consisting of both straight lines and curves; it can only draw a series of straight lines. Consequently, the translation performed by `PostScriptTrace()` breaks, or *flattens*, any curves into many short straight lines, as shown below.



In this way, the paths in an RGML file only consist of movements and lines, as can be seen by looking at the RGML code in Figure 4.

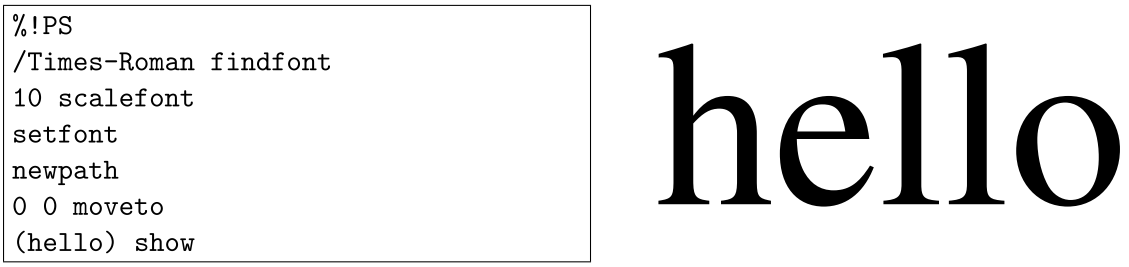
This flattening of curves is not ideal because, although the resulting straight lines appear to the eye as a smooth curve, under certain conditions, for example at large magnification or when lines are very thick, the corners where the straight lines meet can become noticeable. Because of this, `PostScriptTrace()` has an argument called `setflat`, which controls how many straight lines the curve is broken into. Larger values (up to a maximum of 100) result in fewer straight lines and smaller values (down to a minimum of 0.2) result in more straight lines. The downside of a small value of `setflat` is that the RGML file will be much larger because there will be many more `<line>` elements produced.

3.2. Text

The previous section explained how PostScript curves are handled by **grImport**, but the ability to display *text* in a PostScript file has been completely ignored up to this point. That omission is rectified in this section.

One reason for ignoring text in PostScript files is because the main focus of this article is on importing images that are made up of shapes rather than text

Another good reason for ignoring text in PostScript files is the fact that importing text is *hard*. In particular it is very difficult to replicate the exact *font* that is used in the original PostScript file because that information can be extremely complex.



```
%!PS
/Times-Roman findfont
10 scalefont
setfont
newpath
0 0 moveto
(hello) show
```

Figure 9: The file `hello.ps`, which contains PostScript code to draw the word "hello" in a Times Roman font. The resulting image is shown to the right of the PostScript code.

Despite these objections, the **grImport** package provides two simple approaches to importing text from a PostScript image. Neither of these approaches is ideal, but they may be better than nothing for certain images.

As a simple example to demonstrate these approaches, we will work with the file shown in Figure 9, which displays the word "hello" in a Times Roman font.

The first approach to importing this text into R is to convert each character in the text into (flattened) paths. The advantage of this approach is that the resulting text will look quite a lot like the original text because it will be based on the actual outlines of the characters in the original text.

There are two major drawbacks to this approach. The first is that translating each individual letter of text into its own path can result in a very large RGML file. The second problem is that the paths for characters that have a "hole" in them, such as an 'e' or an 'o', will be hard to fill correctly.

Because of the second problem, the paths that are recorded to represent the characters in the text are *stroked* rather than filled, by default. The following code and the resulting image demonstrate this.

```
> PostScriptTrace("hello.ps")
```

```
> hello <- readPicture("hello.ps.xml")
> grid.picture(hello)
```

An argument, `fillText` is provided to instruct `grid.picture()` to *attempt* to fill the text. This uses a simple heuristic to fill all holes in a letter with a background color. The background color is controlled by another argument, `bgText`, which has the value `"white"` by default.

For the current simple example, the heuristic works, as shown below, but the result may be different for other text or for other fonts.

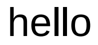
```
> grid.picture(hello, fillText=TRUE)
```

The other approach to importing text from a PostScript file is to completely ignore the font that is being used and just import the actual character values from the file.

The `charpath` argument to the `PostScriptTrace()` function is used to trigger this option. When drawing the resulting text, **grImport** attempts to get the size of the text roughly the same as the original, but differences in fonts will mean that the location and size of text will not be identical.

The following code imports just the text from the file `hello.ps` and the resulting image (shown below the code) is approximately the right size, but uses a completely different font.

```
> PostScriptTrace("hello.ps",
                  "hellotext.xml",
                  charpath=FALSE)

> hellotext <- readPicture("hellotext.xml")
> grid.picture(hellotext)
```



One problem that can completely stymie attempts to import text from a PostScript file is that some font outlines are "protected" and they will resist **grImport**'s attempts to extract them.

3.3. Bitmaps

As mentioned back in Section 1.2, PostScript is really a *metaformat* rather than just a vector graphics format, which means that a PostScript file can contain raster elements as well as shapes and text.

Currently, **grImport** will completely ignore any raster elements in a PostScript file.

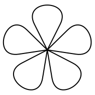
3.4. Graphical parameters

The description of an image in a PostScript file consists of a description of shapes, or paths, plus a description of whether to stroke or fill each path, *plus* a description of what colors and line styles to use when filling or stroking each path. This section addresses the last part: how does **grImport** handle importing *graphical parameters* such as colors and line styles?

Whenever a path is translated from PostScript to RGML, in addition to recording the set of locations that describe the path, `PostScriptTrace()` records the color, as an RGB triplet, and the line width that are used to stroke or fill the path. A minor detail is that the line width is scaled up by a factor of 4/3 because a line width of 1 corresponds to 1/72 inches in PostScript, but a line width of 1 corresponds to roughly 1/96 inches on R graphics devices.

By default, the colors and line widths that are recorded in the RGML file are used when drawing the image in R. This was vividly demonstrated on page 15 with the tiger image. However, both the `grid.picture()` and `grid.symbols()` functions provide a `use.gc` argument that allows the default graphical parameters to be overridden. As a simple example, the following code draws just the outline of the flower image by turning off the default graphical parameter settings and specifying a transparent fill and a black border instead.

```
> grid.picture(flower,
               use.gc=FALSE,
               gp=gpar(fill=NA, col="black"))
```



The following code demonstrates a similar usage of `grid.symbols()`, except in this case the black fill has been retained and a white border has been *added*. This makes it is easier to see where flower images overlap within the plot. Figure 10 shows the resulting plot.

```
> xyplot(V8 ~ V7, data=get("flower", "package:cluster"),
         xlab="Height", ylab="Distance Apart",
         panel=function(x, y, ...) {
             grid.symbols(flower, x, y, units="native",
                          size=unit(5, "mm"),
                          use.gc=FALSE,
                          gp=gpar(col="white", fill="black", lwd=.5))
         })
```

3.5. The **RGML** format

This section provides a more complete description of the structure of RGML files, which may be helpful for working directly with RGML files, for example, using R functions other than those provided by the **grImport** package or when using other software altogether.

The root element for an RGML file is a `<picture>` element. This element will have a `version` attribute that distinguishes between different versions of the RGML format.

The content of the `<picture>` element will typically consist mostly of `<path>` elements. Each `<path>` element is made up of `<move>` elements and `<line>` elements that describe a shape and the `type` attribute of the `<path>` element is either `"fill"` or `"stroke"` to indicate whether that shape should be filled or stroked. Each `<move>` and `<line>` element has two attributes, `x` and `y`, which provide the location of a vertex on the boundary of the shape that is being described.

Each `<path>` element also contains a `<context>` element, which in turn contains an `<rgb>` element and a `<style>` element, with information about the color and line width that should be used to fill or stroke the path.

The final element in an RGML file is a `<summary>` element with attributes recording the total number of paths and bounding box information for the image.

The basic structure of an RGML file consisting of just a single `<path>` element is shown below (this is the petal image from Section 2.1).

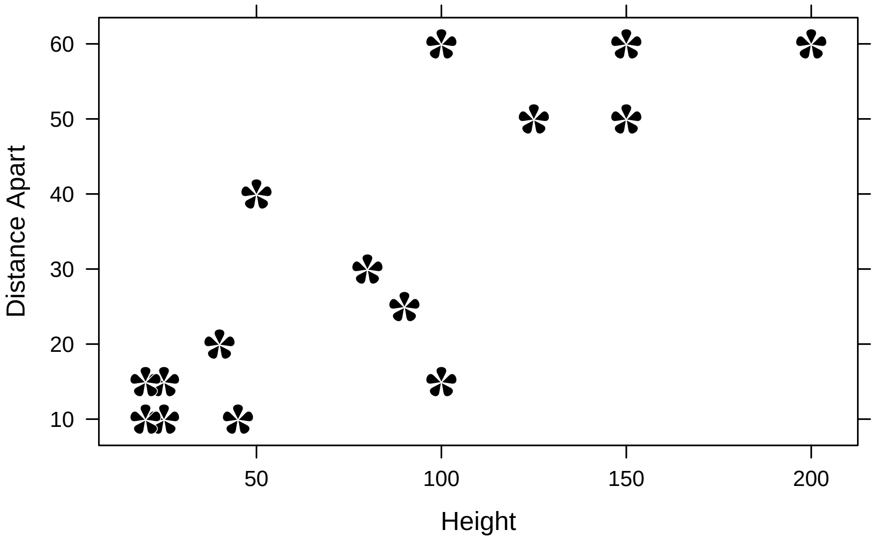


Figure 10: A statistical plot produced in R using the **lattice** package, with an imported "flower" image used as the plotting symbol. This is very similar to Figure 8, but with a white border added to each petal within each flower symbol.

```
<?xml version='1.0'?>
<picture version='2' xmlns:rgml='http://r-project.org/RGML'>
<path type='fill' id='1'>
    <context>
        <rgb r='0.0' g='0.0' b='0.0'/>
        <style lwd='1.33'/>
    </context>
    <move y='0.0' x='0.0'/>
    <line y='100.0' x='-50.0'/>
    ...
    <line y='0.0' x='0.0'/>
</path>
<summary count='1' ymax='175.0' ymin='0.0' xmax='58.0078' xmin='-58.0078'/>
</picture>
```

In addition to `<path>` elements with `type="fill"` and `type="stroke"`, there may also be `<path>` elements with `type="char"`, in which case, the `<move>` elements and `<line>` elements describe the outline of a single letter from a piece of text. This is represented as a different sort of `<path>` element because the default drawing algorithm for text can be different from normal paths.

As well as `<path>` elements, a `<picture>` element may also contain `<text>` elements, which represent complete pieces of text. These contain only a `<context>` element; the text itself, plus its location and size are recorded as attributes of the `<text>` element.

The basic structure of an RGML file consisting of just a single `<text>` element is shown below (this is the hello image from Section 3.2).

```
<?xml version='1.0'?>
<picture version='2' xmlns:rgml='http://r-project.org/RGML'>
<text  id='1' string='hello' y='0.0' x='0.0' width='196.101' height='69.3005'>
    <context>
        <rgb r='0.0' g='0.0' b='0.0'/>
        <style lwd='1.33'/>
    </context>
</text>
<summary count='1' ymax='69.3005' ymin='0.0' xmax='200.0' xmin='0.0'/>
</picture>
```

The **grImport** package provides a DTD file, `rgml.dtd`, that formalizes the RGML document structure.

3.6. The `Picture` class

This section provides a more complete description of the `Picture` class, and the other associated classes, that are used to represent imported images in R. This information is useful for dealing directly with `Picture` objects.

As mentioned previously, the components of a `Picture` data structure have a one-to-one correspondence with the elements of an RGML file, so most elements and attributes from the previous section are represented as slots in an R object within this section. For example, where an RGML file has one or more `<path>` elements, the `Picture` class has a `paths` slot containing a list of paths.

A `Picture` object has two slots: the `paths` slot contains a list of shapes that describe the image and the `summary` slot contains the summary information for the image.

```
> slotNames(petal)
```

```
[1] "paths"   "summary"
```

The components of the list in the `paths` slot are all S4 objects, each with a class corresponding to one type of `<path>` or `<text>` element in an RGML file:

RGML element	S4 class
`<path type="stroke">`	`PictureStroke`
`<path type="fill">`	`PictureFill`
`<path type="char">`	`PictureChar`
`<text>`	`PictureText`

`PictureStroke`, `PictureFill`, and `PictureChar` objects all have four slots: the `x` and `y` slots contain numeric vectors that specify the locations of the vertices of the path, an `rgb` slot contains the color for the path, and an `lwd` slot contains the line width.

The following code and output shows the first (and only) path in the imported petal image. This path is a `PictureFill` object.

```
> str(petal@paths[[1]])
```

```
Formal class 'PictureFill' [package "grImport"] with 4 slots
  ..@ x  : Named num [1:36] 0 -50 -54 -56.6 -57.9 ...
  .. ..- attr(*, "names")= chr [1:36] "move" "line" "line" "line" ...
  ..@ y  : Named num [1:36] 0 100 109 118 125 ...
  .. ..- attr(*, "names")= chr [1:36] "move" "line" "line" "line" ...
  ..@ rgb: chr "#000000"
  ..@ lwd: num 1.33
```

A `PictureText` object has three additional slots: the `string` slot contains the text to draw, and the `w` and `h` slots contain the width and height of the text, respectively.

The following code and output shows the first (and only) path in the imported text image. This path is a `PictureText` object.

```
> str(hellotext@paths[[1]])
```

```
Formal class 'PictureText' [package "grImport"] with 7 slots
  ..@ string: Named chr "hello"
  .. ..- attr(*, "names")= chr "string"
  ..@ w     : num 196
  ..@ h     : num 69.3
  ..@ x     : num 0
  ..@ y     : num 0
  ..@ rgb   : chr "#000000"
  ..@ lwd   : num 1.33
```

The `summary` slot of a `Picture` object is a `PictureSummary` object with slots for the number of shapes in the image, plus bounding box information. The following code and output shows the summary information from the imported petal image.

```
> str(petal@summary)
```

```
Formal class 'PictureSummary' [package "grImport"] with 3 slots
  ..@ numPaths: Named num 1
  .. ..- attr(*, "names")= chr "count"
  ..@ xscale  : Named num [1:2] -58 58
  .. ..- attr(*, "names")= chr [1:2] "xmin" "xmax"
  ..@ yscale  : Named num [1:2] 0 175
  .. ..- attr(*, "names")= chr [1:2] "ymin" "ymax"
```

3.7. `Picture` objects to grid grobs

The `grid.picture()` function uses the **grid** package to draw a `Picture` object.

```
> library(colorspace)
> blueShade <- function(inrgb) {
      rgb <- col2rgb(inrgb)
      RGB <- RGB(t(rgb)/255)
      # Special case "black"
      if (all(coords(RGB) == 0))
          RGB <- RGB(0, 0, .1)
      LCH <- as(RGB, "polarLUV")
      lch <- coords(LCH)
      # Scale the chroma so greys become blues
      hcl(240, 20 + .8*lch[2], lch[1])
  }
```

Figure 11: R code that defines a function to convert an RGB color into a corresponding shade of blue.

This involves two steps: first the `Picture` object is converted into several **grid** *grobs* (graphical objects) and then those grobs are drawn within a **grid** viewport that takes care of all of the necessary coordinate system transformations.

The `grid.picture()` function converts a `Picture` object to grobs by calling the function `grobify()` on each component of the `paths` slot in the `Picture` object. The `grobify()` function is an S4 generic function with methods for `PictureFill`, `PictureStroke`, `PictureChar`, and `PictureText` objects. For example, the `grobify()` method for `PictureFill` objects creates a polygon grob, whereas the method for `PictureText` objects creates a text grob.

The `grid.picture()` function provides an argument called `FUN` that allows the `grobify()` function to be replaced with a custom function. This makes it possible to fully control the conversion of the `Picture` paths into **grid** grobs.

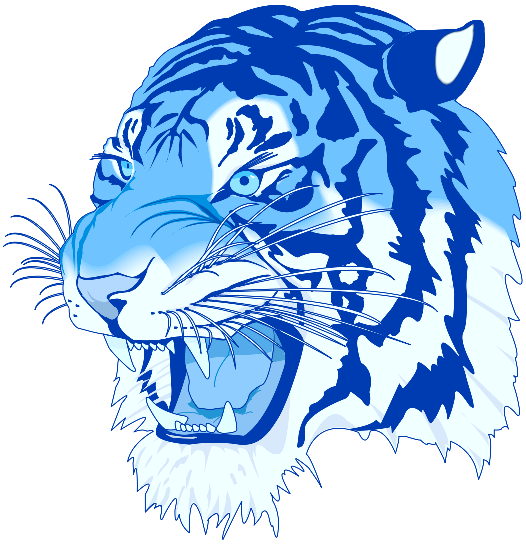
Figure 12 shows an example of this sort of customization. An S4 generic function called `blueify()` is defined, with methods for `PictureFill` and `PictureStroke` objects. The `PictureFill` method produces a polygon grob and the `PictureStroke` method produces a polyline grob, just like the standard `grobify()` function would do. The difference is that the `blueify()` methods set the fill and border colors for these grobs by converting the original RGB color from the the original image to a corresponding shade of blue (using the `blueShade()` function that is defined in Figure 11).

With this `blueify()` generic function defined, we can draw the `tiger` image that we saw on page 15, but this time using different shades of blue. The following code does this drawing and the resulting image is shown to the right of the code.

```
> setGeneric("blueify",
             function(object, ...) {
                 standardGeneric("blueify")
             })
> setMethod("blueify", signature(object="PictureFill"),
            function (object, ...) {
                polygonGrob(object@x, object@y,
                            default.units="native",
                            gp=gpar(col=NA,
                                    fill=blueShade(object@rgb)),
                            ...)
            })
> setMethod("blueify", signature(object="PictureStroke"),
            function (object, ...) {
                polylineGrob(object@x, object@y,
                             default.units="native",
                             gp=gpar(col=blueShade(object@rgb)),
                             ...)
            })
```

Figure 12: R code that defines a custom transformation from `Picture` paths to **grid** grobs. The function `blueify()` is an S4 generic function with methods for `PictureFill` and `PictureStroke` objects. This function generates grobs that have a fill color or border color based on the original colors from the image, but converted into a corresponding shade of blue, using the `blueShade()` function from Figure 11

```
> grid.picture(tiger[-1], FUN=blueify)
```



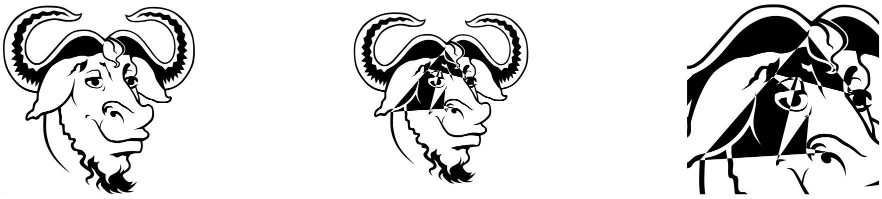
3.8. Complex paths

One of the more sophisticated features of PostScript is that it allows paths to be quite complex. For example, a path may intersect itself and a path may be disjoint, being composed of more than one shape, with the shapes able to overlap and create holes within one another. This can create a problem for importing such an image to R because R graphics cannot draw such complex paths.

In order to demonstrate these ideas, the next example introduces a new image, which is a logo for the GNU project (designed by Aurélio A. Heckert). The original file is in an SVG format and this was converted to a PostScript format using Inkscape. The PostScript file, `GNU.ps`, can be imported using the tools described previously, as shown by the following code. However, the result is not quite right.

```
> PostScriptTrace("GNU.ps")
> GNU <- readPicture("GNU.ps.xml")
> grid.picture(GNU)
```

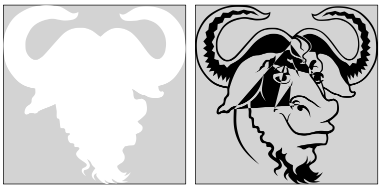
Shown below are the original logo, on the left, the image that is produced by **grImport**, in the middle, and a zoomed view of the problem area in the **grImport** effort, on the right.



The problem is that the GNU logo contains some of these complex paths that consist of multiple disjoint shapes that possibly overlap with each other. The **grImport** package provides two tools that can be used to diagnose and possibly solve this sort of problem.

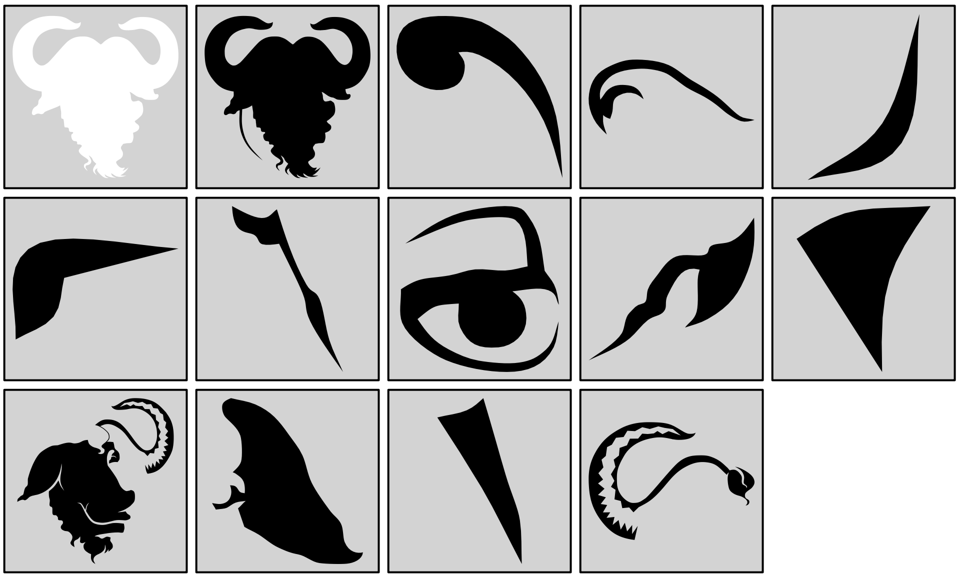
The `picturePaths()` function draws each separate path within an imported image. The following code uses this function to show that there are only two paths in the GNU logo, and the second one is the complex one.

```
> picturePaths(GNU, nr=1, nc=2, label=FALSE)
```



The other tool that is useful in this situation is the `explodePaths()` function. This function takes any path that consists of more than one disjoint shape and breaks it into several distinct paths, each consisting of just a single shape. The following code demonstrates this function being used to explode the paths in the GNU logo and the subsequent broken paths are shown again using `picturePaths()`, this time with `freeScales=TRUE`, which means that each individual path is drawn on its own scale.

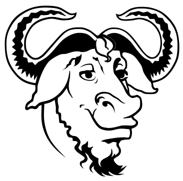
```
> brokenGNU <- explodePaths(GNU)
> picturePaths(brokenGNU, nr=3, nc=5,
               label=FALSE, freeScales=TRUE)
```



The output shows that the complex second path in the original image was composed of 13 disjoint shapes. The usefulness of exploding the original image like this is that it is sometimes possible to resurrect the original complex path by drawing the individual paths in the right order, with the right selection of colours.

The following code demonstrates this idea with the broken-apart GNU logo, by drawing the second path first, followed by paths 11 to 14, each filled with white rather than black, and finally paths 3 to 10. This order and the colors were obtained after a little trial and error. The drawing code is wrapped within a simple function, `GNUlogo()`, so that we can use it again later.

```
> GNUlogo <- function() {
      xscale <- GNU@summary@xscale
      yscale <- GNU@summary@yscale
      grid.picture(brokenGNU[2],
                   xscale=xscale, yscale=yscale)
      grid.picture(brokenGNU[11:14], use.gc=FALSE, gp=gpar(fill="white"),
                   xscale=xscale, yscale=yscale)
      grid.picture(brokenGNU[3:10],
                   xscale=xscale, yscale=yscale)
  }

> GNUlogo()
```



The result is now faithful to the original logo.

4. Applications and examples

This section describes and demonstrates some possible uses of the **grImport** package.

A straightforward use of the `grid.symbols()` function is to import an external image as a custom plotting symbol for a scatterplot, as was previously demonstrated in Figure 8.

A straightforward use of the `grid.picture()` function is to add a company logo to a plot. This has been done “for real” within a large pharmaceutical research and development company, but unfortunately legal constraints prevent the publishing of that example. Instead, the code below demonstrates the basic idea by adding the GNU logo from Section 3.8 as a “watermark” background to a lattice barchart.

```
> barchart(~ cit, main="Number of Citations per Year", xlab="",
           panel=function(...) {
               GNUlogo()
               grid.rect(gp=gpar(fill=rgb(1, 1, 1, .9)))
               panel.barchart(...)
           })
```

This code washes out the logo by simply drawing a semitransparent rectangle over the top. The resulting plot is shown in Figure 13.

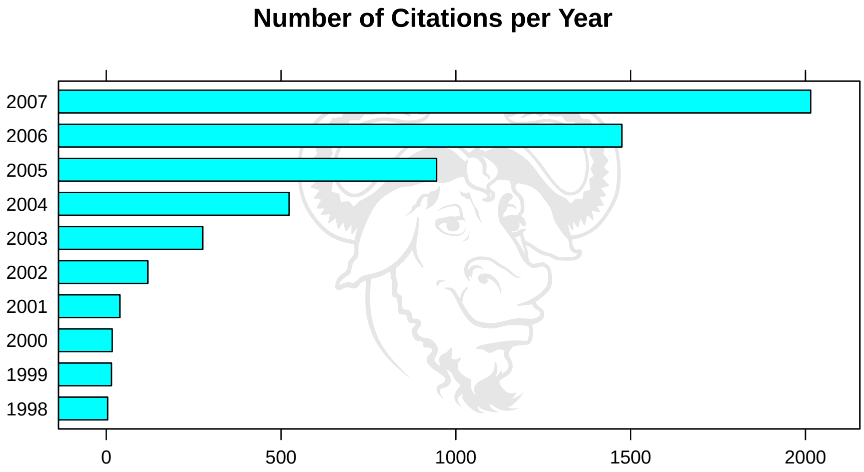


Figure 13: A barplot of the exponential growth in the number of citations of R, with a GNU logo watermark.

4.1. Scraping data from images

A less obvious application of the **grImport** package, and one that demonstrates working directly with the `Picture` objects in R, was suggested by Daniel Jackson of the MRC Biostatistics Unit in Cambridge (private communication). The context for this application is the practice of meta-analysis, specifically in the area of survival data.

A problem that researchers face in this area is how to obtain data from published articles when the original data are not provided. An article may include summary tables and plots, but raw data values may not be available. In practice, researchers sometimes resort to measuring plots, such as survival curves, with a pencil and ruler in order to retrieve at least some raw data points. If an article of interest is published in an electronic format, it may be possible to use the **grImport** package to radically improve both the accuracy and efficiency of such a task.

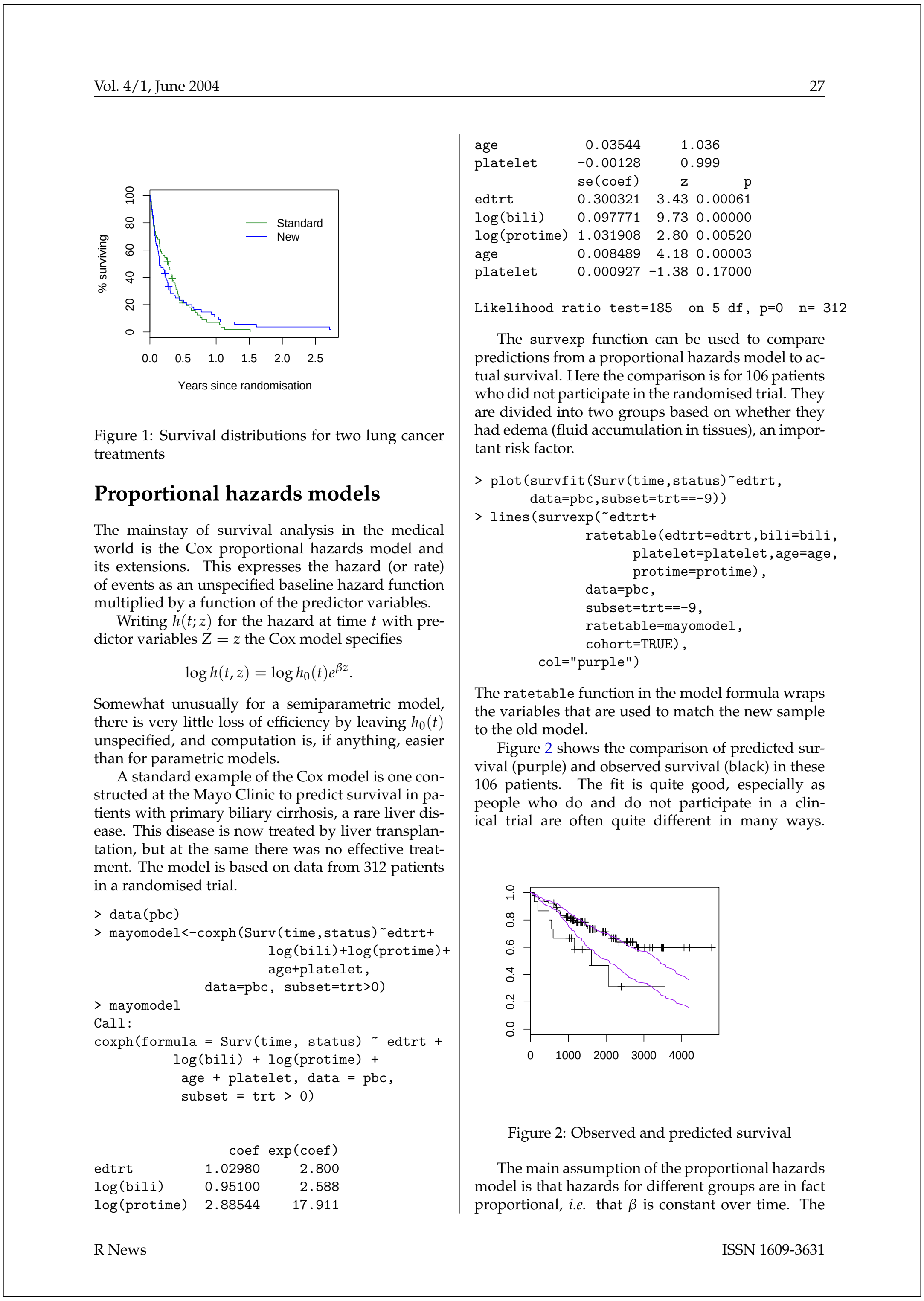
In order to demonstrate this idea, the following example will process a survival curve that was published in the newsletter of the R project for statistical computing (Lumley 2004).

A survival plot appears on page 27 of this R News issue. Figure 14 shows the original context of the plot.

A number of tools can be used to extract just a single page from a multiple-page PDF document and convert that page to PostScript format. In this case, the resulting file is called `page27.ps` and this is converted to RGML format by the following code.

```
> PostScriptTrace("page27.ps")
```

We do not need the entire page, but using the `picturePaths()` function and a little trial and error, it is possible to determine which paths constitute the survival plot in the top-left corner of the page. The following code reads the RGML file into a `Picture` object and extracts just



Vol. 4/1, June 2004 27

Figure 1: Survival distributions for two lung cancer treatments

Proportional hazards models

The mainstay of survival analysis in the medical world is the Cox proportional hazards model and its extensions. This expresses the hazard (or rate) of events as an unspecified baseline hazard function multiplied by a function of the predictor variables.

Writing $h(t; z)$ for the hazard at time t with predictor variables $Z = z$ the Cox model specifies

$$\log h(t, z) = \log h_0(t) e^{\beta z}.$$

Somewhat unusually for a semiparametric model, there is very little loss of efficiency by leaving $h_0(t)$ unspecified, and computation is, if anything, easier than for parametric models.

A standard example of the Cox model is one constructed at the Mayo Clinic to predict survival in patients with primary biliary cirrhosis, a rare liver disease. This disease is now treated by liver transplantation, but at the same there was no effective treatment. The model is based on data from 312 patients in a randomised trial.

```
> data(pbc)
> mayomodel<-coxph(Surv(time,status)~edtrt+
                      log(bili)+log(protime)+
                      age+platelet,
              data=pbc, subset=trt>0)
> mayomodel
Call:
coxph(formula = Surv(time, status) ~ edtrt +
          log(bili) + log(protime) +
           age + platelet, data = pbc,
           subset = trt > 0)


                 coef exp(coef)
edtrt         1.02980     2.800
log(bili)     0.95100     2.588
log(protime)  2.88544    17.911
age           0.03544     1.036
platelet     -0.00128     0.999
             se(coef)      z       p
edtrt        0.300321   3.43 0.00061
log(bili)    0.097771   9.73 0.00000
log(protime) 1.031908   2.80 0.00520
age          0.008489   4.18 0.00003
platelet     0.000927  -1.38 0.17000

Likelihood ratio test=185  on 5 df, p=0  n= 312
```

The `survexp` function can be used to compare predictions from a proportional hazards model to actual survival. Here the comparison is for 106 patients who did not participate in the randomised trial. They are divided into two groups based on whether they had edema (fluid accumulation in tissues), an important risk factor.

```
> plot(survfit(Surv(time,status)~edtrt,
       data=pbc,subset=trt==-9))
> lines(survexp(~edtrt+
              ratetable(edtrt=edtrt,bili=bili,
                    platelet=platelet,age=age,
                    protime=protime),
              data=pbc,
              subset=trt==-9,
              ratetable=mayomodel,
              cohort=TRUE),
        col="purple")
```

The `ratetable` function in the model formula wraps the variables that are used to match the new sample to the old model.

Figure 2 shows the comparison of predicted survival (purple) and observed survival (black) in these 106 patients. The fit is quite good, especially as people who do and do not participate in a clinical trial are often quite different in many ways.

Figure 2: Observed and predicted survival

The main assumption of the proportional hazards model is that hazards for different groups are in fact proportional, *i.e.* that β is constant over time. The

R News ISSN 1609-3631

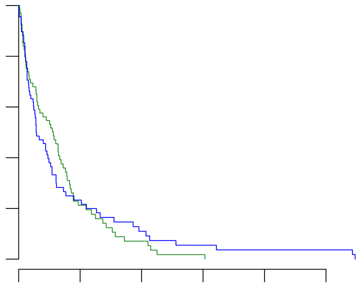
Figure 14: A page from the newsletter of the R project for statistical computing that includes a survival curve at top left.

the paths that make up the crucial part of the plot: paths 3 to 16 draw the axes, path 18 is the green curve, and path 27 is the blue curve.

```
> page27 <- readPicture("page27.ps.xml")
> survivalPlot <- page27[c(3:16, 18, 27)]
```

The following code draws the extracted paths and the result is shown to the right of the code.

```
> pushViewport(viewport(gp=gpar(lex=.2)))
> grid.picture(survivalPlot)
> popViewport()
```



The original image shows that the outer tick marks are at locations 0 and 100 on the y-axis and 0 and 2.5 on the x-axis. These locations can be matched to the locations of the paths that make up those tick marks in the `Picture` object in order to establish a scale for the paths that make up the green and blue curves. For example, the `picturePaths()` function can be used to determine that the lowest tick mark on the y-axis is drawn by the ninth path in `survivalPlot` and zero on the vertical scale is at the y-location of this path.

```
> zeroY <- survivalPlot@paths[[9]]@y[1]
> zeroY

   move
6308.73
```

Similarly, the uppermost tick mark is path 14, so a unit step on the vertical scale is $\frac{1}{100}$th of the difference between the y-location of this path and that of path 9.

```
> unitY <- (survivalPlot@paths[[14]]@y[1] - zeroY)/100
> unitY

move
   9
```

The survival percentages from the green curve (path 15 of `survivalPlot`) can now be determined using this scale information. Each y-value repeats twice because the green curve is drawn as a step function.

```
> greenY <- (survivalPlot@paths[[15]]@y - zeroY)/unitY
> head(round(unname(greenY), 1), n=20)
```

```
 [1] 100.0 100.0  98.6  98.6  97.1  97.1  95.7  95.7  92.8
[10]  92.8  89.9  89.9  88.4  88.4  85.5  85.5  84.1  84.1
[19]  82.6  82.6
```

Happily, these numbers match quite well with the values that were used to produce the original plot.

```
> library(survival)
> sfit <- survfit(Surv(time, status) ~ trt, data=veteran)
> originalGreenY <- sfit$surv[1:sfit$strata[1]]
> head(round(originalGreenY*100, 1), n=9)
```

```
[1] 98.6 97.1 95.7 92.8 89.9 88.4 85.5 84.1 82.6
```

The x-values for the green curve and the data from the blue curve could be extracted in an analogous fashion.

This idea of importing images just to extract the locations from the boundaries of shapes in the image might also be usefully applied to map data that is only available in pictorial form.

4.2. Importing WebLogo images

The next example demonstrates a more sophisticated use of `grid.picture()` based on work by Toby Dylan Hocking (http://www.ocf.berkeley.edu/~tdhock).

The images to be imported are *sequence logos* (Schneider and Stephens 1990) as generated by the WebLogo software (Crooks, Hon, Chandonia, and Brenner 2004, http://weblogo.berkeley.edu/).

The idea of sequence logos is to display patterns in aligned genetic sequences. An example of a sequence logo that was created to visualize the importance of different amino acids in a phage display experiment is shown in Figure 13.

The logo displays the relative freqeuncy of amino acids at each binding position in the experiment; large letters represent amino acids that are "strong signals" for each position. For example, D and E are strong signals for position 1 and G and S are strong signals for position 3.

A question of interest is whether D at position 1 correlates strongly with G at position 3, or with S at position 3. To answer this, a more detailed plot was produced that combined the overall sequence logo with a dendrogram of the experimental binding results, *plus* further sequence logos based on important subsets of the overall result. This plot is shown in Figure 16 and it suggests that D at position 1 does correspond to G at position 2, while E at position 1 corresponds to S at position 2.

The more detailed plot was generated by combining sequence logos that were produced by WebLogo with a dendrogram that was produced by R. WebLogo can produce sequence logos in PostScript format, so these were imported to R using **grImport** and subsetted to remove the axes that WebLogo produces. The R graphics system was then used to produce a dendrogram, with space left on either side for the logos; the logos were positioned relative to the appropriate portions of the dendrogram using **grid** viewports and the **gridBase** package (Murrell 2003).

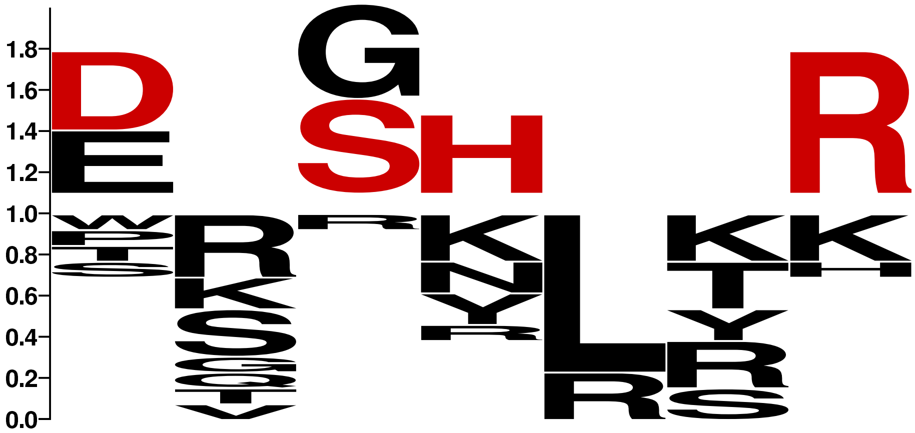


Figure 15: An example of a sequence logo produced by the WebLogo software.

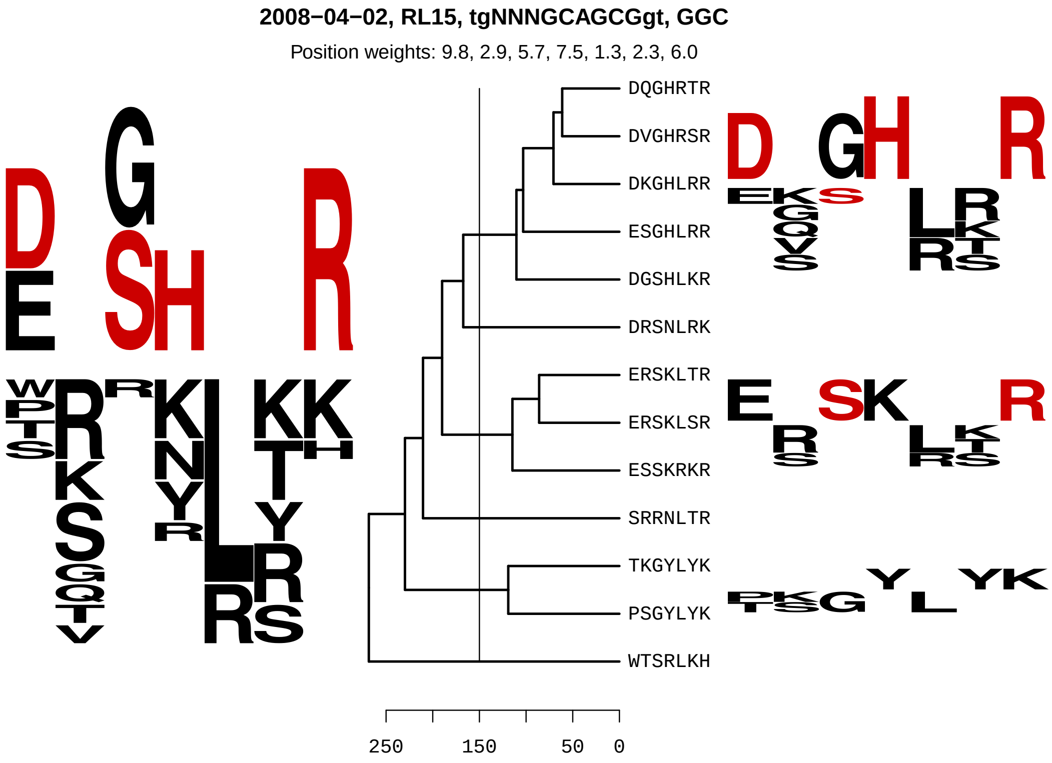


Figure 16: A dendrogram drawn using R's `plot.dendrogram()` function, with four imported WebLogo sequence logos drawn alongside: an overall sequence logo on the left and three subsequence logos on the right, next to the relevant leaves of the dendrogram.

5. Limitations

The **grImport** package is based on PostScript as the original image format and XML as an intermediate image format. These technologies were chosen for a number of reasons, as outlined in previous sections, but they are not without their disadvantages. This section discusses some of the limitations of the **grImport** approach to importing vector graphics.

One of the reasons for choosing PostScript as the original image format is that PostScript is a sophisticated graphics language. However, there are some things that PostScript cannot represent, one major example being semitransparent colors. This means that if an original image is in a format that allows semitransparent colors, such as PDF or SVG, the transformation to PostScript may lose important features of the original image.

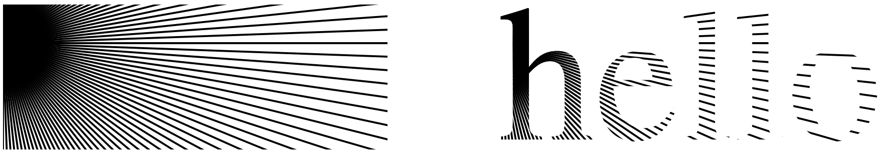
It is possible to convert a PDF image that contains semitransparency to a PostScript format using ghostscript, but the result is a *raster* image in the PostScript format, so for the purposes of importing the image using **grImport**, the image is destroyed.

So one problem with PostScript is that it is lacks some useful graphical features that are available in other vector formats. A different problem with using PostScript is that it *possesses* some useful graphical features that R graphics lacks.

An example of this problem was discussed in Section 3.8; a PostScript path can be more complex than R is capable of drawing. Previous sections have also mentioned that a PostScript file can contain raster elements, which **grImport** just ignores, and Bézier curves, which **grImport** flattens into a series of straight lines.

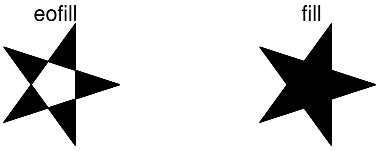
Some other PostScript features that are not supported by either R graphics in general or **grImport** specifically are:

Clipping to arbitrary paths: PostScript allows clipping to the current path, which can be a very complex shape. As a simple example, the image below on the left shows a pattern of radiating lines; the image on the right shows this pattern being clipped to the outline of a piece of text.



R graphics only allows clipping to a rectangle and **grImport** currently completely ignores any clipping in a PostScript image, so an original image that makes uses of clipping can be imported but it will not reproduce correctly in R.

The path fill rule: When a self-intersecting path is filled with a color, there are two main ways to decide what constitutes the "inside" of the resulting shape. The two algorithms are called the *non-zero winding rule* and the *even-odd rule*. The image below demonstrates the difference by showing exactly the same (self-intersecting) path being filled with the two algorithms (the even-odd rule is on the left).



> PostScript can perform either type of fill, but R graphics only uses the non-zero winding rule, so an original PostScript image that uses the even-odd rule can be imported, but it will not reproduce correctly in R.

Moving on to the choice of XML as an intermediate format, the main problem is that XML is a verbose language that can result in large files. A related problem is that reading such files into R can be slow. However, the **grImport** package, and R graphics in general, are designed for high flexibility in the production of static, presentation-quality graphics, so these problems have so far proven to be an inconvenience rather than an insurmountable obstacle.

6. Availability

The **grImport** package is available from the Comprehensive R Archive Network (R Foundation for Statistical Computing 2008). This article describes version 0.4 of the package.

7. Conclusion

The **grImport** package implements a three-stage approach to importing vector-based graphical images into the R software environment for statistical computing and graphics.

It is assumed that the original image can be converted to a PostScript format, then functions in the **grImport** package are provided to convert the PostScript file to an intermediate RGML format, to read the RGML file into S4 objects in R, and to manipulate and draw those objects.

There are limitations to the system, due to the limitations of the PostScript language and due to the limitations of the R graphics system, although some of these issues can be worked around. Furthermore, despite these limitations, the **grImport** package has been employed in a variety of ways in several real-world applications.

One of the important ideas to take away is that the **grImport** package is not just about drawing pictures. The package starts with a vector image and transforms the image into data. One of the things that can be done with data in R is to draw it, but there are many other potential applications for the image data.

Acknowledgements

Thanks to the authors of the freely-available images that are used in this article. The chess pawn is from a public domain chess board image in SVG format by José Hevia, which was originally sourced from the Open Clip Art Library `http://openclipart.org/clipart//recreation/games/chess/chess_game_01.svg`, but is now available from `http://www.public-domain-photos.com/free-cliparts/other/chess/chess_game_01-3657.htm`. The GNU logo is by Aurélio A. Heckert and is available from `http://commons.wikimedia.org/wiki/Image:Heckert_GNU_white.svg` under a Free Art Licence. The tiger image is distributed as part of the ghostscript software system.

Thanks also for ideas and data to Toby Dylan Hocking and Daniel Jackson. The R citation data used in Figure 13 were taken from an email to the R-help mailing list by John Maindonald.

Aspects of the **grImport** package were contributed to by Richard Walton during a Faculty of Science summer scholarship at the University of Auckland (2005/2006).

References

Adobe Systems (1999). *Postscript Language Reference*. Addison-Wesley Professional, 3rd edition. ISBN 9780201379228.

Adobe Systems (2005). *PDF Reference Version 1.6*. Adobe Press, 5th edition. ISBN 9780321304742.

Bah T (2007). *Inkscape: Guide to a vector drawing program*. Prentice Hall Press, Upper Saddle River, NJ, USA. ISBN 9780131357945.

Bivand R, Leisch F, Maechler M (2008). *pixmap: Bitmap Images ("Pixel Maps")*. R package version 0.4-9.

Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F (2006). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C). URL http://www.w3.org/TR/2006/REC-xml-20060816/.

Clark J (1999). *XSL Transformations (XSLT)*. World Wide Web Consortium (W3C). URL http://www.w3.org/TR/xslt/.

Crooks GE, Hon G, Chandonia JM, Brenner SE (2004). "WebLogo: A sequence logo generator." *Genome Research*, **14**, 1188–1190.

Ferraiolo J, Jun F, Jackson D (2003). *Scalable Vector Graphics (SVG) 1.1 Specification*. World Wide Web Consortium (W3C). URL http://www.w3.org/TR/SVG11/.

Kylander OS, Kylander K (1999). *Gimp the Official Handbook*. Coriolis Value. ISBN 1576105202.

Lumley T (2004). "The survival Package." *R News*, **4**(1), 26–28. URL http://CRAN.R-project.org/doc/Rnews/.

Merz T (1997). *Ghostscript User Manual*. URL ftp://mirror.cs.wisc.edu/pub/mirrors/ghost/gs5man_e.pdf.

Murrell P (2003). "Integrating grid Graphics Output with Base Graphics Output." *R News*, **3**(2), 7–12. URL http://CRAN.R-project.org/doc/Rnews/.

Murrell P (2005). *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL. ISBN 1-584-88486-X, URL http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html.

Nikon Systems Inc (2005). *rimage: Image Processing Module for R*. R package version 0.5-7.

R Development Core Team (2008). *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.

R Foundation for Statistical Computing (2008). *Comprehensive R Archive Network*. URL http://cran.r-project.org/.

Sarkar D (2008). *Lattice: Multivariate Data Visualization with R*. Springer. ISBN 9780387759685.

Schneider TD, Stephens RM (1990). "Sequence Logos: A New Way to Display Consensus Sequences." *Nucleic Acids Res*, **18**, 6097–6100.

Sklyar O, Pau G, Smith M, Huber W (2008). *EBimage: Image processing and image analysis toolkit for R*. R package version 2.4.0, URL http://www.ebi.ac.uk/~osklyar/EBImage/.

Still M (2005). *The Definitive Guide to ImageMagick*. Apress. ISBN 9781590595909.

Temple Lang D (2008). *XML: Tools for parsing and generating XML within R and S-Plus*. R package version 1.95-3.

Veillard D (2008). "libxml home page." http://xmlsoft.org/.

Affiliation:

Paul Murrell
Department of Statistics
The University of Auckland
Private Bag 92019
Auckland
New Zealand
64 9 3737599 x85392
E-mail: paul@stat.auckland.ac.nz
URL: http://www.stat.auckland.ac.nz/~paul/