Pictor Reference Manual

Allan R. Wilks

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

created: 18 August 1995

ABSTRACT

Pictor* is an environment for statistical graphics that promotes simple commands for common uses, while also offering the ability to experiment with whole new paradigms. It is implemented in Version 4 of S (Chambers, 1994 and Becker, et. al., 1988) and this paper is a guide for its use.

1. Introduction

Pictor is an environment for statistical graphics that promotes simple commands for common uses, while also offering the ability to experiment with whole new paradigms. Its design comprises several levels, from its high-level graphics model to its low-level device model. The latter is described in Wilks (1994b) and the material there is useful background material for this paper. Pictor describes statistical graphs as a collection of pieces whose coordinate systems are tied together via various types of constraints; see Wilks (1994a) for a detailed account of how constraints are established and resolved.

Pictor is implemented in Version 4 of S (Chambers, 1994 and Becker, et. al., 1988). This paper is a guide for its basic use. In the next section we begin with an example of S code that uses Pictor to draw simple scatterplots. Many of the features of the S implementation of Pictor appear briefly in the example. The remainder of the paper deals in greater depth with the collection of classes and functions that comprise the implementation. In section 3 device drivers, device descriptors and device control are introduced. In section 4 the rendering environment of Pictor is dealt with. With this in hand, Pictor's graphical objects, or *grobs*, are then described, both in their simple, primitive form in section 5, and their most general form in section 6. Section 6 also describes the basic tools used for constructing grobs.

---

*Lacaille, one of the first astronomers systematically to chart the southern skies, gave the name *Equuleus Pictoris*, meaning *Painter's Easel*, to the grouping of stars between declinations -43° and -64° and right ascensions $4^h 30^m$ and $6^h 50^m$. Today the name of this constellation has been shortened to simply *Pictor*, meaning *Painter* and can be found just north of the Large Magellanic Cloud.

2. An example

We begin with an example. Suppose we wish to write a function that will make a simple scatterplot of some (x,y) data. What we actually write in Pictor is a function to create a grob that describes the scatterplot. Simply invoking the function will return the grob and since the default print method (see Chambers 1994) for objects of class `"grob"` is to cause them to be rendered on the current display, the scatterplot will appear as desired. Of course, as with ordinary S computations, the return value of the function can be saved via an assignment for later manipulation or display.

Here is the code for a simple version of `scat()`, which takes two numeric vectors, possibly of different lengths and produces a grob describing the scatterplot:

```
 1 scat <- function(x, y)
 2 {
 3         rx <- range(x)
 4         dx <- diff(rx)
 5         Rx <- (rx + c(-1,1)*dx*0.07)
 6
 7         ry <- range(y)
 8         dy <- diff(ry)
 9         Ry <- (ry + c(-1,1)*dy*0.07)
10
11         gpaste(
12                 bottom = baxis(Rx),
13                 left = laxis(Ry),
14                 scatter = marks(x, y),
15                 box = polylines(Rx[c(1,2,2,1,1)], Ry[c(1,1,2,2,1)]),
16                 constraints = list(
17                         constraint("xexact", "/bottom", "/scatter"),
18                         constraint("ymatch", "/bottom", "/scatter", Ry[1]),
19                         constraint("yexact", "/left", "/scatter"),
20                         constraint("xmatch", "/left", "/scatter", Rx[1]),
21                         constraint("exact", "/scatter", "/box")
22                 )
23         )
24 }
```

The code begins in lines 3–5 with a simple calculation that leaves in `Rx` a pair of numbers that span the range of the x values with some extra room left symmetrically on both ends; the total extra room is equal to 7% of the range of the x data. In lines 7–9, a similar calculation computes `Ry` for the y data. These four numbers in `Rx` and `Ry` define the data box inside of which the (x,y) data will be plotted. The 7% ‘‘white space’’ around the data follows conventional wisdom (see Cleveland 1994).

The remainder of the code consists of a single call to the function `gpaste()`, beginning at line 11, which computes the grob that `scat()` returns. The call has five named arguments. The first four of these are themselves grobs; it is the task of `gpaste` to combine these four into a single grob. That grob will have a *root*, named `"/"`, and the root will have four *children*, named `"bottom"` (this describes the bottom or x-axis), `"left"` (describing the left or y-axis), `"scatter"` (the actual scatter of points) and `"box"` (the box around the plot). Each *node* of the grob has a *full name*. For example, the full name of the root is `"/"` and the full name of the first child of the root is

"/bottom".

The fifth argument to `gpaste()` is a list of *constraints*. Since each node of the grob potentially has its own coordinate system, those coordinate systems must be related to one another in some manner before it is possible to render the grob. The constraints do precisely this. Each constraint has a `type` (the first argument), the nodes `from` which and `to` which the constraint applies and an optional additional vector of numeric parameters that further specify the constraint. Thus, for example, the first constraint says that in the horizontal direction, the coordinate systems of the x-axis and the scatter exactly match.

The second and third arguments to `gpaste()` (lines 14 and 15) are *drawing primitives*. Each corresponds exactly to one of Pictor's device drawing primitives. The first requests a mark at each of the (x,y) pairs (with the usual S convention about repeating the shorter vector until its length matches the longer). The second requests a single polyline that forms the box determined by `Rx` and `Ry`.

The first two arguments to `gpaste()` (lines 12 and 13) are not drawing primitives, but rather are grobs with their own children. The `baxis()` routine looks like this:

```
1  baxis <- function(Rx)
2  {
3          xpos <- nice(Rx, out=F)
4          nx <- length(xpos)
5          gpaste(
6                  axisline = segments(xpos[1], 0, xpos[nx], 0),
7                  ticks = segments(xpos, rep(0,nx), xpos, rep(1,nx)),
8                  ticklabels = text(xpos, 1, format(xpos), textadjust="top+"),
9                  constraints = list(constraint("yscale", "/", "", -0.2))
10         )
11 }
```

The call to `nice()` in line 3 returns a set of positions that roughly speaking span the range in `Rx` and that are ''nice'' numbers (powers of 10 times 1, 2 or 5). The positions are supposed to be appropriate locations for tick marks on an axis. This is very similar to the S function `pretty()`. As with `scat()`, the main work of the function is in its last expression, another call to `gpaste()`. This time the computed grob has three leaf nodes, two calling for sets of segments (the first is one segment—the axis line, and the other is a set of short segments for the tick marks), and one for a set of text strings (the call to `format()` produces the text strings and the `textadjust` argument requests that the strings be centered horizontally and just below the tick marks).

The fact that the single constraint generated for the `baxis()` grob applies to the whole grob (it goes *from* `"/"`—the root of the grob) means that all three parts have the same coordinate system. This coordinate system measures horizontally in data units and vertically in units of -0.2 inches, with the negative sign implying that an increase in the y-coordinate goes *down* the page. This absolute condition on the y-coordinate is captured in the constraint, which says that one y unit in any part of the grob is equal to -0.2 inches in the display coordinate system (indicated by the null name, `""`).

The `laxis()` function is very similar to `baxis()`. These two functions, together with `scat()`, are all that is needed to produce simple scatterplots in Pictor. Of course, more is needed: labels for the two axes; an overall title for the plot; ability to change colors and other rendering details; and so on. However, all of this functionality can be added easily to the code given above, and does not even double it in size. In the remainder of this paper we will take a close look at the various mechanisms, both the data structures and the functions, underlying the `scat()` example.

3. Getting started (and finished)

In order to use Pictor, there typically needs to be one or more devices active. A device is made active by invoking a function, such as `ps()` for the PostScript (Adobe 1985) driver or `gl()` for the GL driver (for graphics in a window on an SGI machine) or `motif()` for a generic Motif window. If the call to the device function is successful, the return value is a small integer, the *device descriptor*, which can be the final argument to many Pictor functions; when this is possible, the argument name is always `device`. In general, however, it does not need to be given as an explicit argument as there is always a *current device*, which is the default value of the device descriptor for those functions. The current device is always available as the return value of the function `getDevice()`, and may be set as the single argument to `setDevice()`. When the device driver is to be retired, its descriptor can be given to `killDevice()`, whose default argument is the current device.

One of the tasks that the device function performs is to initialize the rendering state; we turn now to a discussion of rendering parameters and rendering maps.

4. Rendering parameters and maps

Pictor's device model defines a set of *rendering parameters* that control the appearance of everything that gets displayed (see Wilks 1994b). In the process of using Pictor there is always a current *rendering state* kept for each active device. The state is captured in a single S object of class `"rpar"`, which has slots for each of the rendering parameters. Here is the class definition:

```
Rpar <- setClass("rpar", representation(
     color = "numeric",
     fill = "numeric",
     linestyle = "numeric",
     linewidth = "numeric",
     font = "numeric",
     pointsize = "numeric",
     textrotation = "numeric",
     textadjust = "character",
     marktype = "numeric",
     marksize = "numeric",
     markrotation = "numeric"
))
```

Each slot is a numeric vector, except for the `textadjust` slot, which is a character vector. Again, the reader is referred to Wilks (1994b) for a detailed explanation of the meaning of each of these parameters.

The main function for interfacing with the rendering state is `rpar()`. It's optional first argument is an `"rpar"` object and its optional second argument is the device descriptor, which defaults to the current device. If an `"rpar"` object is given, the rendering state is set to that value and if no first argument is given, the current rendering state is returned. To set individual slots of the current rendering state, the `rpar.set()` function may be used, as in these examples:

```
rpar.set(color=1:10)
rpar.set(marktype=rep(1:3,12))
rpar.set(textrotation=seq(0,350,by=10), font=2)
```

Some of the rendering parameters are indexes into rendering maps: `color` and `fill` index the `colormap`, `linestyle` indexes the `linemap`, `font` indexes the `fontmap` and `marktype` indexes the `markmap`. A map is a collection of (index,definition) pairs, where the index is a possible value for the rendering parameter and the definition is an appropriate object. For each type of map there is a class that represents a definition for that map and a function that initializes an object of that class.

A color definition is given in one of several *color models*, such as RGB, CMY, HLS, HSV, HSB, YIQ and a special one called NAME. All but the last follow the classical color models of those names, each requiring a triple of numbers. The last is a character string giving a color name, like `"red"`, `"white"` and so on. To create a color definition object the `colordef()` function takes arguments giving the definition and the model, and returns a `"colordef"` object. Here are some examples:

```
colordef(1,0,0,"rgb")    # a way to define red
colordef("red")          # another way; default model is "name"
colordef(0,0.5,1,"hls")  # yet another way
```

A linestyle definition is created by a call to the `linedef()` function whose two arguments are a vector of dash and gap lengths and an offset; the offset is zero by default (see Wilks 1994b for details on linestyles). Some examples:

```
linedef(0)               # a solid line
linedef(c(1,1)/20)       # dashes and gaps both 1/20 inch
linedef(c(1,1)/20,1/20)  # same, but starting on a gap
```

A font definition is created by `fontdef()` and simply takes a single character string as an argument:

```
fontdef("Helvetica")
fontdef("Symbol")
fontdef("ZapfChanceryItalic")
```

Mark definitions are more complex. They encapsulate the idea, in some sense, of a little graphical subroutine to be executed at each mark position. The `markdef()` function takes as its arguments expressions that evaluate to grobs. We haven't yet shown any such expressions, so the following examples may not yet be completely understandable, but the basic idea should be clear.

```
markdef(arcs(0,0,1,0,360))
                         # a filled circle (bullet)
markdef(arcs(0,0,1,0,360,fill=NA))
                         # an un-filled circle
markdef(polygons(c(-1,1,1,-1),c(-1,-1,1,1)))
                         # a filled square
markdef(polylines(c(-1,1,1,-1,-1),c(-1,-1,1,1,-1)),
     segments(c(-1,0),c(0,-1),c(1,0),c(0,1)))
                         # a target: unfilled square with cross-hairs
```

The expressions are arbitrary, with the proviso that they must evaluate to a grob that has a single leaf node (necessarily a Pictor drawing primitive). So, for example, an argument to `markdef()` could be a compound expression enclosed in braces, as long as the final sub-expression evaluated as just prescribed. Here is a more complex (perhaps somewhat silly) example of a mark that is an octagonal doughnut: two concentric octagons outlined in the current color, painted color number 2 between them and painted color number 0 (the background color) inside the inner one:

```
markdef({
        theta <- seq(0,2*pi,length=9)[-1]
        polygons(cos(theta),sin(theta),fill=2)
},{
        theta <- seq(0,2*pi,length=9)[-1]
        polygons(0.5*cos(theta),0.5*sin(theta),fill=0)
})
```

Corresponding to each of these four definition functions is a map function used for establishing the relationship between indexes and definitions. All four map functions have exactly the same syntax, so we will only look at one of them, `colormap()`:

```
colormap(index=NULL,definition=NULL,map=NULL,device=getDevice())
```

In its most common usage `colormap()` adds a single color definition to the color table for the current device, as in:

```
colormap(2,colordef("red"))
```

which establishes color number 2 as red. Vectors (of equal length) may be given as the first two arguments to add more than one definition at a time:

```
colormap(2:3,list(colordef("red"),colordef("green")))
```

If the second argument is missing the color table is read instead of written. So to retrieve color number 2:

```
colormap(2)
```

When just one color is requested in this way, the return value is a `"colordef"` object; otherwise a list of `"colordef"` objects is returned. If both the first and second arguments are missing then the entire color table is returned. In this case not just a list of definitions is returned but rather an object of class `"colormap"`, which contains not only the definitions but also the indexes associated with them.

In any of these calls to `colormap()` an optional third argument may be specified, which should be a `"colormap"` object. Then, instead of modifying the color table of the current device, `colormap()` will operate on the supplied color table. If the color table is to be written, the modified table is returned and if it is to be read, a single `"colordef"` or a list of them is returned. When the `"map"` argument is supplied, the `"device"` argument is usually just ignored. The only exception to this is if the first two arguments are missing; this is taken as an indication that the color table for the specified device should be unconditionally set to that given in the `"map"` argument. As an example, suppose that you wanted to set the color table of the current device to that of device number 2:

```
colormap(map=colormap(device=2))
```

The inner call to `colormap()` retrieves the entire color table for device number 2 and returns it as a `"colormap"` object. This is then given to a further call to `colormap()` as the value of the `"map"` argument and since the first two arguments are missing, the color table of the current device (the default value of the `"device"` argument) is set to the given map. As another example, the following will initialize the color table of the current device to be empty:

```
colormap(map=new("colormap"))
```

Alternatively, you may want to establish a color map for later use that defines colors 1 through 10 according to the 10 definitions in the list `my.defs`. After the expression

```
my.map <- colormap(1:10, my.defs, map=new("colormap"))
```

the `"colormap"` object `my.map` would be available for use as the `"map"` argument to further calls to `colormap`.

Each of the four tables works in the manner just outlined for the color table. There are functions `linemap()`, `fontmap()` and `markmap()`, each with the same arguments as `colormap()`, and there are classes `"linemap"`, `"fontmap"`, and `"markmap"` that these functions manipulate; everything we have said about color tables applies to all four table types.

There is one subtle point, however, to mention about the mark table. Recall that a mark definition is an expression. Any time one of the active device tables is modified, the device is notified of it. In the case of marks, this means actually evaluating the expression and sending the resulting primitive grobs to the device. If the expression refers to global data, the current value of this data will be used in the evaluation, so that if the data changes later, the mark will not follow suit. This is not an issue if the expression does not refer to global data.

5. Primitive grobs

With rendering parameters and rendering maps in hand we now proceed to the simplest sort of grobs, namely, the *primitive* grobs. A primitive grob describes exactly one of the primitive device operations, as defined in Wilks (1994b). It refers to a particular geometric concept, such as a set of line segments or a set of circular arcs; it gives the geometric data that precisely describe the object and it optionally gives rendering information that says how the object should be drawn.

Currently there are eight kinds of primitive grobs and each has its own class. These classes, however, have some commonality, which is collected together in another class, the `"groblet"` class. The slots for this class allow the specification of a complete set of rendering parameters, a complete set of rendering tables, a coordinate transformation matrix that maps the intrinsic coordinates of the groblet into the absolute coordinate system of the device, and bounding box data, both geometric and rendering. (Wilks 1994a has a detailed description of coordinate systems and bounding boxes.) Though this sounds like a great deal of information to carry around for a simple primitive grob, in fact it usually turns out not to be very much, due to the Pictor's method of propagating constraints, to which we shall return later.

Each of the primitive grob classes, then, is a `"groblet"` object with additional information. For example, here is the definition for the `"polylines"` class:

```
SetClass("polylines",representation(Groblet,x="numeric",y="numeric"))
```

Hence, a `"polylines"` object is a `"groblet"` object with two additional slots, `x` and `y` giving the (x,y) pairs that define the polyline.

The user should never have to deal with the slots of a groblet or with those of one its derivitave primitive grobs directly; this is left to the rest of the Pictor infrastructure. For example, corresponding to each kind of primitive grob there is a function that will create an instance of it, and the name of the function is the same as the name of the class. Each function takes first of all a set of arguments in one-one correspondence with the slots its class adds to the generic groblet. As an example, the `polylines()` function begins with two arguments, `x` and `y`.

Next, the function allows the argument `rpar`, which should be an object of class `"rpar"`, and which specifies some or all of the rendering parameters that are to apply to the primitive grob. Finally, each of the rendering parameters that apply to the primitive may be specified individually (and will override the same parameters if they were specified in the `rpar` argument). Here is the full argument list, then, of `polylines()`:

```
polylines(x,y,rpar=new(Rpar),color=NULL,linestyle=NULL,linewidth=NULL)
```

Only three rendering parameters appear explicitly because it is only these three that affect the appearance of polylines. More parameters could be specified, if desired, in the `rpar` argument, but this would probably be silly, at best extraneous, as the specifications would have no effect.

A groblet is not a grob, though as we shall see in the next section, grobs are composed of a set of groblets arranged in a tree-like structure. The return value of functions like `polylines()` is, in fact, a grob, albeit a very simple one with just one groblet in it. We now turn to a description of full-fledged grobs.

6. Grobs

A picture is generally composed of a number of primitive grobs. We now describe how these primitive grobs are organized in a single object whose class is just `"grob"`. Recall from section 2 that a grob is conceptually a tree. Each node of the tree has a *local name* and the local names of the children of a particular node must be unique. Each node also has a *full name*, composed by concatenating the local names of its ancestors, separated by "/" characters. This is analogous to the naming of files in the UNIX file system, and the analogy carries over to the root of the tree, which has the full name `"/"`.

The leaves of a grob are always primitive grobs. Conversely, interior nodes of a grob are never primitive grobs. Rather, they are intended to be place holders that help to organize the leaves conceptually into groups, groups of groups and so on.

When first defined, a primitive grob has geometric data that is given in an intrinsic coordinate system, not yet related to the absolute coordinate system of the display. Though this relationship is not known precisely at the time of definition, usually there is partial information available that relates the primitive grobs to one another or to the display itself. This partial information may be specified in the form of *geometric constraints*, as shown in the example in section 2 and as described in full detail in Wilks (1994a).

The `constraint()` function creates an object of class `"constraint"`, given the `type` of the constraint, the full names of the `from` and `to` nodes and possible additional numeric data. If either node of a constraint is not a leaf node then there is an implicit assumption that all nodes below that node, right down to the leaves, all have the same coordinate system and that therefore, the constraint is replicated for each of the descendent leaves.

A grob is a class with three slots. Here is the class definition:

```
setClass("grob", representation(
     parts="list",
     names="character",
     constraints="list"
))
```

The `parts` slot is a list of all the groblets in the grob in a certain order: beginning with the root node, the nodes are listed in depth first order and the children of a given node appear (not necessarily contiguously) in left-to-right order. The left-to-right order of the children is established at the time of the creation of the grob and has a bearing on how the grob gets rendered, as we shall see.

The `names` slot is a character vector of the same length as the `parts` list and gives the full name of each of the parts. Finally, the `constraints` slot is a list of `"constraint"` objects, each of whose `from` and `to` slots uses names from the `names` slot of the grob.

There are several fundamental tools used for grob creation in Pictor. The first of these has already been mentioned in the last section, namely, the collection of functions that create primitive grobs, like `polylines()`, `segments()`, and so on. The primitive grob returned by one of these functions has just one part—the requested specialized groblet—with name `"/"` and with no constraints.

The second main tool for grob creation is the `gpaste()` function whose arguments are arbitrary grobs:

```
gpaste(..., constraints=)
```

These grobs are made into children of a new, common root node in the order given (left-to-right). It is expected that each of the arguments will be named; these names become the local names of the children of the new root (boring default names, like `"1"` and `"2"`, are chosen if not supplied by the user). In addition to the grob arguments, `gpaste()` may take one additional argument, named `constraints`, that gives a list of constraints between the other grob arguments.

The `gpaste()` function just bundles up its grob arguments and passes them to `gpaste.list()`:

```
gpaste.list(glist, constraints=)
```

It may sometimes be more convenient to use `gpaste.list()` directly, after creating a collection of grobs in a list and then passing in the whole list.

A third tool for grob creation is the `ginsert()` function:

```
ginsert(g, h, childof="/", after=)
```

Its first two arguments are grobs, with the understanding that the first grob is to be inserted somewhere in the second grob. The location of the insertion can be specified with the two optional arguments. If the full name of one of the parts of `h` is given as the `childof` argument then `g` is inserted as the last (rightmost) child of that part of `h`. Alternatively, if the value of the `after` argument is the full name of a part of `h` then `g` is inserted as a sibling, and just after, that part. Both arguments may be supplied; `childof` will be ignored unless `after` is a local name, in which case it should be the name of a child of the `childof` argument; if so, `g` will be inserted just after that child. If neither argument is supplied, `g` becomes the rightmost child of the root of `h`. This gives an alternative method of incrementally adding children to the root node of a grob, but it is probably more efficient to add a whole list of grobs at once with `gpaste.list()`.

7. Conclusion

With the classes and tools we have described, it is possible to produce complex graphical displays in a natural and straightforward way. Pictor is currently accessible as a library in Version 4 of S and my be attached simply by invoking the `pictor()` function.

References

Adobe Systems Incorporated (1985), *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts.

Becker, Richard A., John M. Chambers, Allan R. Wilks (1988), *The New S Language*, Wadsworth, Pacific Grove, California.

Chambers, John M. (1994), *Overview of Version 4 of S*, AT&T Bell Labs Research Report, Murray Hill, New Jersey.

Cleveland, William S. (1994), *The Elements of Graphing Data*, Second edition, Hobart Press, Summit, New Jersey.

Wilks, Allan R. (1994a), *Constraint-Based Representations of Statistical Graphs*, AT&T Bell Labs Research Report, Murray Hill, New Jersey.

Wilks, Allan R. (1994b), *Controlling Graphics Devices for Statistical Graphics*, AT&T Bell Labs Research Report, Murray Hill, New Jersey.