Images in S

John M. Chambers
AT&T Bell Laboratories, Murray Hill, New Jersey

Neil Crellin
Statistics Department, Stanford University

November 23, 1991

Contents

Abstract

We describe techniques for extending S to handle the class of *image* objects. Specifically, software is described to represent images, to carry out numerical and graphical computations, to interface to image-processing packages, and to use S as a prototyping language for studying image algorithms.

1 Introduction.

The software described here provides a class of *image* objects in S, with functions and methods to create and modify image objects, to display them graphically, and to interface to specialized packages of image software. There are three different views of the software:

1. From the view of the S user, our image objects are numeric or logical objects, possibly quite large. The objects are matrix-like with additional properties that let them be plotted and otherwise treated in conventional image style.
2. In the context of image-processing software, we are providing gray-level (i.e., univariate) images in byte or numeric form, without any specific color information. Interfaces to image-processing packages are part of the software design.
3. In terms of statistical computing research, the image project is an application (the first of any size) of some new ideas about *external* objects in S. Images inherit from class `"external"`, methods for which are designed to make general computations available for large objects.

The immediate goal of the software is to make images accessible for data analysis and related tasks. To achieve this we provide:

- basic computational facilities to treat images like ordinary S objects;
- graphical displays for images that are integrated with the usual S graphics;
- a paradigm and some specific examples of interfaces to traditional image-processing packages;
- an implementation using classes and methods that should assist extensions of the computations to new methods and new kinds of data.

An image represents some data values over a regular rectangular grid of specified width and height. Many kinds of data might be organized as images; our basic implementation has only two: byte and numeric. Byte data are thought of as a having unsigned values between 0 and 255. Numeric data are ordinary S numeric (double-precision) data. These are sufficient for most image computations. Ordinary use of the image software does not require the user to know whether an image object is currently in byte or double form. Extensions to handle other kinds of data can be made, either by modifying the methods currently used or by defining new classes of objects.

Image objects are closely related to ordinary S matrices, but several considerations make a special class of image objects useful. First, image objects tend to be large — it is not unusual to have the width and height be several hundred pixels each. We handle this by making images inherit from the class of external objects, a general mechanism for dealing with large objects (see section 5). Second, both image graphics and interfaces to image-processing packages will be more efficient and easier if the images are organized internally in an appropriate way. The provision of image data stored externally in byte form is partly motivated by this consideration, as well as by the desire for more compact storage. Third, images are usually considered as being sorted by raster line (i.e., row) and from top to bottom, which would occasionally cause confusion, since S matrices are stored by columns and graphical co-ordinates in S go from bottom to top.

Section 3 describes basic facilities for creating, displaying and manipulating images in S. Many users will also want to use some of the many special computations developed for images. There are a number of packages for image processing, with overlapping but distinct facilities. We will describe a general approach to constructing interfaces to image packages in section 4, with an interface to the `pbmplus` package as an example. Generally, we expect that most "end users" of image software in S will want to augment our software with facilities for their application. The software described in this paper provides a base to work on and has been designed with extension in mind. Section 5 is recommended reading for those who plan to design and write such extensions. Finally, we hope S will be a useful prototyping environment for new techniques to analyze, display or process image data.

This work is still at a fairly early stage. We are reasonable happy with the design so far; in particular, it has proved simple and unrestrictive. We are anxious to hear from users and would welcome any suggestions for improvements in the design or implementation.

2 Examples of Images

Images are most often thought of as digitized pictures, photographs that have been scanned and converted to a digital representation. Monochrome or single-channel images, which are all that we deal with here, can be represented as a matrix of numeric values (gray levels), recorded to some chosen precision. In practice, this usually means that the levels are coded as some multiple of bytes (thus, 8-bit, 16-bit or 32-bit unsigned integers), or as floating-point numbers. In our software the first and the last of these are the options (8-bit bytes or double-precision numbers). Data coming from scanning or other image observations typically arrives in byte form. Numeric images can arise from computations on byte images or from other forms of numeric data that are convenient to interpret as images. Multi-channel images (for example, with color separation) need some additional software to relate the multiple channels to some color map. The software described here can then function below such software in the same way it does for single-channel images.

Figures omitted in this version of the paper

Figure 1: *In the left panel, an image originating as a digitized photograph showing a portion of a wafer from an experiment on wave soldering. The four larger rectangles represent soldering under different experimental conditions. On the right, a photo of the interior of a section of optical fiber preform, from a study of factors affecting the uniformity of the resulting fiber.*

For purposes of data analysis, images play a number of useful roles. The pictures they represent may be directly related to the data being analyzed. Figure 2, for example, shows two images resulting from scanned photographs; on the left, a portion of a wafer from a designed experiment to study the quality of a wave-soldering process for electronic components (see, for example, [3]). Such pictures are useful identifiers of the numerical results from the experiment, and in addition processing the images may be informative itself. The right panel in Figure 2 is a photomicrograph taken from a different experiment, in this case on the properties of optical fiber preform. As image data becomes more generally accessible and usable, data analysis will benefit from being able to use pictures where appropriate to identify observations.

Not all image data need come from pictures, however. The same experiment shown in the photograph on the right of Figure 2 also produced

Figures omitted in this version of the paper

Figure 2: *On the left, plots of the position of a deflected laser beam, as the laser passes by a section of optical fiber preform. Four successive passes are shown, with the curve shifted on the plot each time to increase visibility. On the right, a numeric image representing all the data from 45 such curves. The vertical dimension represents position along the preform and the horizontal dimension is the position of the laser. Taking minus the absolute value makes the spikes on the curves appear dark on the image.*

voluminous numeric data about the interior surface of the preform. A laser passes horizontally by a piece of optical fiber preform, the fiber held vertically. As the laser passes, its light is deflected, and patterns in the deflection can be interpreted in terms of the internal surface of the preform. Each pass was recorded as 2000 deflections and 45 successive vertical positions were used (see the left panel of Figure 2). On top of this the observations are repeated for different rotations of the fiber and, of course, for fibers produced under different conditions. Because of the volume of data and because it is, at least in part, naturally thought of in two-dimensions, representing the measurements as an image object is useful. The data is stored and manipulated as numeric values; only on plotting, as on the right of Figure 2 must these numeric value be interpreted as gray level (bright for high values, dark for low values).

A variety of computational sources may be involved in obtaining image data. Photographs will be digitized by a scanner and recorded as files of gray levels or as objects in some image-processing package. The numeric data in Figure 2 originated with automatically recorded experimental results, stored and shipped as raw data files. Sections 3 and 4 will show how images in S can be created from various sources.

3 Computations on Images

Because images are kept as nonstandard S objects, arbitrary S expressions involving them may act in different ways. You can try out computations that seem reasonable. The following three groups of functions apply to images:

1. For a number of basic S functions, special methods have been written that handle images. Calls to these functions with image arguments will not break down suddenly, even for large images, since the methods have been written to handle large, external S objects. These methods have

added leverage in that other functions may apply to images as well, if the image arguments only arise in calls to the functions of this group. See the example below on page 11.

2. Some functions may be explicitly written for images; for example, to invoke an interface to a package. These functions are distinguished from the first group in that, since they are not methods for generic S functions, they will not have the leverage described above.

3. A third case arises when some of the functions in the expression do not treat images specially, but do coerce the relevant objects to either vectors or matrices, through the functions `as.vector()` and `as.matrix()`, for which methods convert images to ordinary S objects. Like the first group, these functions can pass on their ability to other functions using them. However, these functions will slow down more rapidly than those of the first group if the image objects are large.

If these possibilities do not include all the functions used in the expression, the computation is likely to fail. Functions that neither have methods for images nor explicitly coerce their arguments to vectors or matrices will find that image objects look like character strings. The symptom of this third situation is therefore often some complaint about non-numeric data or missing values generated from numeric operations applied to character data.

A long-range goal of the work on images, and on external objects in general, is to expand the expressions for which some reasonable computations can be done. An adequate treatment that includes arbitrary computations on arbitrarily large objects will require some new mechanisms at a fundamental level. Only some general notions exist at the moment for these changes.

Here are a few additional notes on the three groups of functions. The first group includes roughly 75 functions using methods for external objects for all the basic mathematical functions, operators, summaries, and subset calculations. Also included are special routines for graphics, and a few special methods. See sections 3.2–3.4.

Computations using `as.matrix()` and `as.vector()` will fail with an error if an image to be converted exceeds a maximum permitted size.

```
> as.matrix(fiber)
Error in as.matrix.image(fiber): object has 554800 elements,
     maximum for coercion is 500000
```

A similar message is generated from `as.vector(fiber)`. The choice of a maximum size for coercion is heuristic; currently, one tenth of the maximum object size is used. To increase or decrease the maximum size allowed, change the option; for example,

```
> options(object.size = 6e06)
```

would allow conversion of images up to length 60,000. Any choice, however, may be satisfactory for some matrix computations but either to lenient or too strict for others. The conversion mechanism is not ideal, but it does allow substantially more computations to be used on images.

The conversion by `as.matrix()` produces a matrix that "looks like" the image; that is, the first row of the matrix corresponds to the first row of the image, and so on. Because images and matrices have different conventions about how the data is stored, elements of the image will be permuted. In contrast, `as.vector()` leaves the data elements in the same order; in fact, the method involved is not an image method at all, but a method for the underlying `external` class.

The remainder of this section discusses the various computations defined for image data.

3.1 Creating Images

Image objects are created or modified by computations on existing image objects or by inserting ordinary S data into an image object. The function `image()` generates image objects; for example

```
x <-  image(width, height)
```

creates a new image object of the specified width and height and assigns it the name `x`. An optional third argument controls the mode (`"logical"` or `"numeric"`). The image created by the above is initialized to 0.

Sometimes, a file will have been created with the appropriate data in it. Most often these will be the raw bytes for the image, maybe preceded by some header information. The `image()` function can be instructed to read in this data by supplying additional arguments `external.mode`, `file`, and optionally, `skip`. It should be in typical scan-line order for images; that is, stored a scan-line at a time, going from the upper-left to the lower-right of the image. For example, suppose a file `"raw.fiber"` contains all the data for the optical fiber picture in Figure 2 in this form, as raw byte data. The image can then be created, with the data read in, by:

```
fiber <- image(760, 730, external.mode = "byte",
     file = "raw.fiber")
```

which creates an image of width 760 and height 730. One further extension allows a number of lines to be skipped in the file before reading the data. It's often not necessary to understand the format of the file in detail to see how to convert it in this form. For example, suppose the data were on a file `"fiber.out"` from a package with unknown format. Seeing the line

```
WINDOW=0 0 760 730
```

and noting that the file had 9 lines of header would be enough to guess that the image could be created by the expression:

```
fiber <- image(760, 730,  external.mode = "byte",
    file = "raw.fiber", skip=9)
```

The general problem of format conversion of images is hard, perhaps intrinsically unsolvable in a fully satisfactory way (see [4]). But for our modest purposes, a few fairly simple ideas will provide for many common cases. If the originating package can produce either the raw data alone or the data with a header as above, we can copy it in directly. When a package is used extensively, a specific conversion method can be written.

After creating an image object, any subset of the values may then be set by the matrix-type assignment,

```
x[i,j] <-  value
```

where `i` and `j` select rows and pixels within rows, respectively. Either subscript can be omitted; in particular, `x[] <- value` sets the entire contents of the image. Vector-style subscripts can be used also:

```
x[i] <-  value
```

When using these, remember that images are stored raster-wise, so that the first `width` values form the first row, etc.

As noted, when image data is imported frequently from a particular package or other outside source, it is convenient to supply a function to get the data and convert it to an S image:

```
get.pgm.image(file)
```

converts an image in the `pbmplus` package, stored on `file`. Writing these `get` method requires some knowledge of the external package (`pbmplus` in this case)—in particular, its format for storing image attributes and data. We will discuss this example and some general considerations in section 4.

3.2 Graphical Display of Images

There are two basic graphical functions for images:

```
plot(image, . . .)
show.image(image, where, . . .)
```

The first, a call to the generic function `plot()`, actually calls the methods `plot.image()`. It generates a new plot, using the full display surface or the current figure region as defined by the graphics parameters or by `split.screen()`. The call to `show.image()` does not generate a new plot, but adds the image to the current plot, using the rectangular portion of the plot defined by the `where` argument. This argument specifies the rectangle in the style of the graphics parameter `usr`; that is, `where` should be of the form

```
c(xleft, xright, ybottom, ytop)
```

with the values being defined in current user co-ordinates.

When plotting images with either function, two additional considerations arise. First, assuming that the `width` and `height` attributes of the image represent its original physical shape, should the plot retain that aspect ratio or should the image be reshaped to fill the region on the display? Second, should the data values in the image be rescaled to span the range of colors available for images in the current device, or should the numerical values be interpreted as indexes into a particular pre-defined range of colors? Arguments `reshape` and `rescale` control to either of the graphics functions control these. By default, the image is reshaped if it will fill less than half of the proposed space on the device, and the values are rescaled to span all the available image colors.

Both graphics functions can take graphical parameters as arguments. In the case of `plot.image()`, this includes the high-level parameters such as the labels and limits for the `x` and `y` axes. The default values for some of these parameters are different from the usual scatterplot. By default there are no axes, tick marks, or axis labels and the plotted image fills the entire plot region. If you do want the usual axes, supply `xaxt="s"` and `yaxt="s"` as arguments.

Not all graphics devices are capable of displaying images. Plots can currently be done with `postscript()`, `iris4d()`, `news()`, and `x11()`. Except for the case of `postscript()`, displaying an image generally implies converting the image data to some range of color numbers. This range needs to have been initialized in a color map to an appropriate set of actual colors (for example, a gray scale). The image software itself knows nothing about the contents of color maps. It is the responsibility of the graphics device to set up a suitable range before images are plotted. This should happen automatically if the device supports image display. The color numbers currently being used for images can be retrieved and set by:

```
get.image.range()
set.image.range(x)
```

The first function returns a vector with two numbers, the first and last color number used for images. The second function sets the first and last such color numbers according to its argument, which should be a numeric vector of length 2. Thus, for example,

```
set.image.range(rev(get.image.range())
```

would invert the image colors, producing a photographic-negative effect.

3.3 Basic Methods: Numeric Computations

All the basic mathematical and summary functions, and unary and binary operators are defined for images—specifically, all the functions listed in table A.2 of [2], page 479. So in particular, arithmetic operations and numeric transformations can be applied to images. The implementation uses the notion of group methods ([2], page 741), so that methods for the `Math`, `Ops` and `Summary` groups support many of the functions. The methods use the implementation of images on top of external objects (section 5). The specialization to images is largely in choosing the appropriate object for the result. Binary operators treating two images require that the dimensions of the images match: the widths and heights must either correspond or be equal to 1.

One point to keep in mind when using these functions is that a byte image will generally be turned into a numeric image by them. These numeric images will then need to be re-converted to byte form for graphical display or for use of an image-processing operation that requires byte data. The rescaling will be performed automatically, but a warning may also be generated, since there is no single "correct" rescaling in general. Applications that do know how to rescaling is to be done can convert the numeric image back to byte explicitly, by calling one of the conversion functions on page 12.

What about other numeric computations? Those that are done by composing or building on numeric results of the functions defined for images should go through directly. The `Summary` group of functions produce an ordinary S vector, so functions that depend on image arguments only through these functions will always be ok. The other two groups return image objects as their value. In general, they will compose with one another but not necessarily with other computations. For example the following simple rescaling function works for all size images:

```
to.ZeroOne <- function(x) {
    r <- range(x)
    (x - r[1])/diff(r)
}
```

The function `range()` is in the `Summary` group, so it works for images and returns an ordinary S vector of length 2. The expression on the second line involves two arithmetic operators and other expressions not having an image objects as their arguments. The result will be a numeric image.

3.4 Other Functions Defined for Images

Images can be generated by taking subsets of other images, using the matrix or vector form of the `"["` and `"[["` operators. If `x` is an image, then

```
x[i, j]; x[i]; x[]
```

will extract images or, if used on the left of an assignment, will alter some of the data in the image. As with matrices either `i` or `j` can be omitted, implying all rows or columns respectively. Notice that vector-style subscripts, or no subscript at all, can be used as well. The replacement version of these operators behaves like a matrix for matrix-style replacements. In particular, if `xx` is a 2 by 3 matrix:

```
small.image[1:2, 1:3] <- xx
```

puts the elements of the matrix in the corresponding place in the image. On the other hand, vector-style replacement stores the new values in raster-line order, since this corresponds to how the image is stored.

The mode of an image can be changed, while leaving the object as an image:

```
mode(x) <- newmode
```

For images, only `"logical"` and `"numeric"` make sense for `newmode`. The external data will be recopied as necessary to represent the new mode. Note the contrast between this computation and the use of `as.vector()`, which converts the image into an ordinary S vector.

A slightly different issue is conversion of numeric images between the two external forms, `"byteplain"` and `"dbleplain"`. Conversion from double to byte is required for graphical display, and may also be needed to use an interface to an image-processing package. Sometimes the data is in principal already defined on the range 0-255, with only a conversion of the external storage mode required: the function `dble2byte()` handles this case. In other cases, the numeric data is to be translated linearly to the byte form, mapping the range of the data to the range 0-255: function `fitchar()` does this. A third interpretation is that the numeric data is only defined up to monotone transformation, so that successive discrete values should be mapped into successive numbers in the byte range: `discretize()` is the function for this purpose. All these functions use on a single method, `cut.plain(x, breaks)`, which produces a byte external object using the supplied breaks. This method can be used directly for more specialized conversions. See the detailed documentation for the interpretation of `breaks`.

As mentioned previously, a variety of matrix-like operations work on images. Two common functions that work in general are:

```
t(x); diag(x)
```

The function `t()` has an explicit method for images, but `diag()` comes for free, since it is defined in terms of subset operations. Most other matrix-like functions will work by coercing the image to a matrix and therefore may be slow for large images.

4 Interfaces to Image Packages

There exist many useful packages, systems and collections of subprograms for dealing with image data. None of the packages can be said to dominate all the rest; in any case, the relevance of operations from the various packages will vary depending on the application. An interface to such a package is useful, since the underlying techniques are sometimes challenging to implement. As in similar situations, the philosophy of S is that an interface that models the communication between S objects and a particular image package is an effective way to make use of the package. Designing such an interface is an interesting exercise in itself, and many of the design considerations will be similar for most packages.

We present an example of such an interface, first describing a set of specific functions using the interface and then discussing how the interface itself is designed. The functions supplied should be of some interest, and additional functions using the same interface should be easy to write. If an interface to another package is needed, this should be designed using the example in section 4 for hints.

4.1 Functions Using the `pbmplus` Package

The image library includes as its example interface a function, `.pbmplus()`, implementing an interface to the `pbmplus` package [5]. This is a collection of stand-alone executable programs. The programs to which we interface are image-processing commands that read and write specially formatted files, consisting of a header followed by packed byte-coded image data. The S image library has functions to invoke a number of image operations from this package.

```
pgm.flip.image(x, . . .) # various transposes, etc.

pgm.bentley.image(x) # vertical smearing
pgm.edge.image(x) # edge dectector
pgm.enhance.image(x,level) # contrast enhancement
pgm.oil.image(x, nbdsize) # smearing

pgm.hist.image(x, . . .) # histogram of image values
```

The first function performs various reflections and rotations. The next four are various visual enhancements based on neighboring values. All the functions return a new image, except for `pgm.hist.image()`, which either returns or plots a histogram of the pixel values in the image. For the meaning of the functions and their optional arguments, see the corresponding online documentation.

Two additional functions convert between S image objects and the file format for `pbmplus` commands:

```
get.pgm.image(file)
put.pgm.image(x, file)
```

to read a `pbmplus` file and to write an S image to such a file. These functions are appropriate if you want to do some extended processing in the package, outside of S, or if you have a file produced by the package that you want to study in S.

The `pbmplus` package has three levels of image formats, dealing with bit, single-channel (gray level), and three-channel ("pixmap") images. Our interface is to the second level, commands for which all begin with the letters `"pgm"`. Note that all the `pgm` commands work only on byte data. Numeric images will be converted to byte form before calling the interface to `pbmplus`. The conversion assumes that the numeric data are already scaled into the range 0 to 255; an error will occur otherwise. Other conversions may be desirable (see page 12); these need to be embedded in some code that converts the data to byte coding, invokes the relevant image-processing function, and if necessary converts the results back. In any case, of course, some precision may be lost in representing numeric image data on the restricted byte scale.

4.2 Writing New Functions Using `pbmplus`

The `pbmplus` package and, even more, the specific functions above, are intended as a paradigm for such interfaces, to be adapted and extended according to your interests and needs. All the functions above call the function `.pbmplus()`, the interface to the `pbmplus` package. This function has three arguments, two required and one optional:

- `command`, the command to be executed in the package, as a character string including all options but not the input image

- `x`, the input image object;

- `output`, an optional argument, only needed if the command does not produce an image as its result.

If the command has no data-dependent options, this argument will be a constant; otherwise, the `paste()` function can usually construct it suitably. The `pgmbentley` command, for example, needs only an input image and returns an output image, so the entire function to invoke the command is:

```
"pgm.bentley.image"<- function(x)
  .pbmplus("pgmbentley", x)
```

The `pgmoil` command takes an option, the neighborhood size over which the smearing operation operates. This option is also an optional argument to the S function, and the function is defined as:

```
function(x, neighborhood = 3)
  .pbmplus(paste("pgmoil", "-n", neighborhood), x)
```

In general, the function using the interface will check values for the option and paste together the appropriate command, excluding the information about the input image itself.

The interface in `.pbmplus()` knows how to read the image produced by the command and convert it to an image object in S; in particular, the width and height of the output image need not be pre-computed by the S function invoking the interface.

Occasionally, as with the histogram command, the output produced by `pbmplus` is not an image. In this case, it is the responsibility of the function to interpret the output of the command. This is done by supplying the third argument. Usually, this argument is a function object. The `pbmplus` output will be written to a file, and `.pbmplus()` will call the function supplied as the `output` argument, with the name of the file as its argument. The user-supplied function typically reads this file using `scan()`. The value of the `output` function is then the value of the call to `.pbmplus()`.

Consider the histogram example. The `pgmhist` command produces a file consisting of two lines of formatting information, followed by lines for each of the 256 possible values in the image. Each line has two numeric fields, one for the value and one for the count of how often that value appears in the image. Some other information appears on these lines, but we want to skip this. So the function we supply should skip two lines, and read the two fields from the remaining lines, which can be done using `scan()` as follows:

```
function(file)
   scan(file, skip=2, what=list(values=0, counts=0), flush=T)
```

This S expression is an in-line S object; we could assign it, but it's cleaner when the expression is simple to supply it directly as an argument. The call to the `pbmplus` interface is then:

```
.pbmplus("pgmhist", x, function(file)
    scan(file, skip=2, what=list(values=0, counts), flush=T))
```

If there is no need to process the output file by an S function, the call to `.pbmplus()` can be made to simply return the output, as a character vector with one element per line, working the same way as the `unix()` interface:

```
.pbmplus("pgmhist", x, FALSE)
```

would return the output of the command as a character vector.

Using the `output` argument to the interface requires some knowledge of the command's output format, but it is only needed for nonstandard commands. By far the majority of commands will produce an image as output; for these, you only need to construct the options, if any, that the command requires.

4.3 Implementing other Interfaces

The interface to `pbmplus` is made up of three essential steps:

1. convert the S image object into an image as understood by the package;
2. construct and invoke a `pbmplus` command as requested by the function calling the interface;
3. optionally, read the output of the command and interpreting it as an S image.

The first and third steps were relatively easy when working with `pbmplus`, because the format used related simply to the `byte` version of S image data. The individual commands use their own conventions about options, as usual with shell commands in UNIX. This means that the interface itself does not need to have a syntax for describing such options; indeed, it could not reasonably have such a syntax. This makes writing `.pbmplus()` easier, but of course shifts the burden of encoding options to the individual functions calling it. Fortunately, the options on the commands used so far are simple to encode, as the example of the `pgmoil` command illustrated.

Our hope is that interfaces to other image-processing packages can be designed analogously. If the interface uses the three steps as above, one hopes that the process will not be too laborious. Considerable interest exists currently in conversion among image formats ([4], e.g.). Progress in this area should help in the first and third steps. The question of how commands are to be generated could in principle require a more sophisticated model of the image-processing language. In practice, however, many of the more useful commands are likely to be also the simpler commands to express.

A serious concern in the use of interfaces, as with all the image software, is performance. The `pbmplus` interface currently copies images both in and out. If it turns out that this aspect of the computations slows the computations too much, two possibilities can be considered:

- use a more sophisticated interface, such as pipes, to write the data directly to the commands;
- allow the S images to be stored in the `pbmplus` format, optionally.

The second idea could be implemented by having a further class of `pbmplus` images. This is made more practicable because the `pbmplus` format differs from the S format only by having a header, which to the S software amounts to an offset. Whether either of these ideas is worth implementing will require more experience with the software.

5 How Images are Implemented; External Objects.

This section describes how the image class of objects is implemented. It is not required reading in order to use the image software, but will be useful for those who want to extend the software in various ways. In particular, if you would like to create image objects that are organized in a somewhat different way or to create image methods for new generic functions, the explanations in this section may be important. In addition, the idea of external S objects has a number of other potential applications besides images. Anyone wanting to develop software for such applications should start by reading the discussion of external objects here.

5.1 External Objects

Images are implemented in part as an application of *external* S objects. External objects are an approach to handling objects, potentially many kinds of objects, that share the property that they are not stored as ordinary S objects and that one would prefer not to bring the entire object into memory at once. Generally, this will be because the object could be very large. Besides images, other applications include objects kept in external database management systems or objects kept in special form (e.g., compacted) to reduce file space requirements.

External objects are implemented using the class/methods mechanism; specifically, they inherit from class `"external"`. This is a special sort of class, however, in that no object can inherit *only* from class `"external"`, for reasons that will become clear shortly. Methods for this class are built around one simple assumption: while bringing in the entire object may be prohibitive, some methods will exist for extracting or replacing a contiguous chunk of the data in the object. We call such a contiguous chunk a *span*. Methods for external objects all revolve around two functions, to get and replace spans:

```
spans(object, descriptor)
spans(object, descriptor) <- value # "spans<-"()
```

To make use of external objects for some other class of objects, methods for these two functions must be written. The external class itself does not define how spans are extracted or replaced, this is the essential additional information that must be associated with any other class using external objects. In the next subsection, we will see how images supply this information.

Once the appropriate spans methods are defined, methods can be inherited from the external class for a variety of generic functions, including all the basic S functions in table A.2 of [2], page 749. These inherited methods all tend to have a similar structure. They usually contain a loop that extracts successive spans of an object, does some calculation on that span and then uses the result, perhaps by replacing a span in another external object. For example, a call to any of the functions in group `Summary`, such as `sum()` or `range()`, will use the following method if `x` inherits from `external` and there is no overriding method ([2], p. 472).

```
Summary.external <- function(x) {
    n <- length(x)
    m <- min(n, blocksize(x))
    sp <- make.spans(s = 1, w = m)
    f <- get(.Generic, mode = "function")
    results <- NULL
    while(all(sp > 0)) {
        results <- c(results, f(spans(x, sp)))
        sp <- shift.spans(sp, m, F, n)
    }
    f(results)
}
```

The object `sp` is a descriptor of the spans to be extracted. It is initialized by `make.spans()` and shifted along by `shift.spans()`. A spans descriptor is a two-row matrix, the first row giving starting locations and the second row the lengths of spans to be extracted or replaced. When the descriptor has been shifted beyond the end of the object, the corresponding element in the second row is set to 0. The function `blocksize()` computes an appropriate number of elements to be brought in from the external object at one time. By default, it is one percent of the maximum object size (see the documentation for `options()`). Methods can be written for this function if some other block size would be better for some classes of objects.

Given these definitions, the operation of `Summary.external()` should be clear. The object `.Generic` contains the name of the particular summary function. This is copied to `f`, and `f()` is applied to each span in turn, collecting the results. The method calls the function one more time at the end to get the overall summary. This relies on a form of idempotent behavior of the summary functions; for example,

```
sum(c(x1, x2)) == sum(c(sum(x1), sum(x2)))
```

Otherwise, some special calculations would have been needed in the last step of the method.

Because there must always be some other class preceding external, external methods can never generate a new object directly. Instead, they rely on a generic function `copy()`. Classes inheriting from external must have or inherit a method for this function. Again, as with `spans()`, there is no `copy` method for class external itself. So, for example, a method for the `Math` group of functions might contain the following code:

```
result <- copy(x)
n <- length(x)
m <- min(n, blocksize(x))
sp <- make.spans(s = 1, w = m)
while(all(sp > 0)) {
    spans(result, sp) <- f(spans(x, sp))
    sp <- shift.spans(sp, m, F, n)
}
result
```

The only differences from the previous example are that a new external object is created by `copy()`, and that the partial results are stored back in `y` using the spans replacement method.

The actual method for this group is slightly more complicated for two reasons. Part of the complication comes from the existence of functions in this group that can have an optional second argument. This is handled by treating these functions like operators. More relevant for designers of classes is that the object `result` is treated as an optional argument. The method only uses `copy()` if the argument is not supplied. Classes inheriting from `external` will often know more about what sort of object should be used for the result. If so, a method for the inheritor class can be written to generate and pass down the result object. An example is the method for `Ops` for image objects. The method arranges to extend 1-row or 1-column images when they are used with other compatible images.

To summarize, designers of new classes inheriting from `external` must supply at least three methods, for `spans`, `"spans<-"`, and for `copy`. In addition, they will need the usual functions for generating the objects and methods for printing and/or plotting. Most such applications will also have to have some approach to handling *temporary* objects as well. This last subject we will leave for section 5.3.

5.2 Images and Plain External Objects

The class of image objects is designed around some assumed properties of images. Images are data observed over some two-dimensional grid of points. They have attributes `width` and `height` and the length of the object is always the product of these. We make two assumptions about storage: first, that the data are stored, left to right, by raster line so that all the `width` points in one line are contiguous; and, second, that the rasters are stored from top to bottom. These assumptions are mainly relevant for plotting and for coercing the images to matrix form. Also, of course, they must be matched to the assumptions of any image package in designing an interface.

Images, like external objects, do not of themselves assume anything about how the data are stored or retrieved. There needs to be a third class, between `image` and `external`, providing the specifics of the storage. In principle, there could be several alternative choices for this class. Currently, however, we only supply one, the `plain` class. This class provides for storing data in raw, packed form on a file. The data may be represented either as bytes (unsigned integers in the range 0 to 255), or as double-precision floating-point numbers. The function `external.mode(x)` codes this choice, returning or setting the external mode of `x` as either `"byte"` or `"double"`.

The methods for extracting and replacing spans read and write files of either byte or double data. Since these computations are at the heart of any methods for plain external objects, they have been written specially in C, with some extra care taken, for example, to reduce the number of times a file needs to be opened.

Further methods for plain objects largely center around the two external modes. The interface to the `pbmplus` package and the graphics computations for images are designed for byte data. Most image data arrives in this form and, of course, large images take 8 times less file space if stored in byte form. On the other hand, standard S numeric computations tend to produce double-precision numeric output. One would not want them to produce byte data in any case, since 256 values are inadequate to represent numeric results. As a result, the issue of which form of data is needed for a particular computation arises frequently. The decision depends on the application. For numeric computations, byte data must always be converted to double. The conversion is automatic when using the methods supplied; if you need to do it explicitly,

```
external.mode(x) <- "double"
```

causes the conversion to take place.

The opposite conversion, from double to byte, is not so straightforward. Clearly, information can be lost going in this direction, so it is important to know what purpose and assumptions underlie the computation. As discussed on page 12, conversions can try to preserve the exact values of the numeric data, can apply a linear transform to the range 0 to 255, or can try to preserve as many discrete values as possible.

Finally, some computations may be willing to return either byte or double external data, whichever is needed to represent the results. Such functions are handled automatically, without explicit conversion. The method for replacing spans of an object of class `plain` calls some C code to determine whether it needs to promote a byte object to double, when the values to be inserted are numeric. Therefore, a function or method that uses the style for external object methods shown in the previous section should work correctly.

5.3 Temporary External Objects

All ordinary S objects produced while evaluating an expression are dynamically allocated and automatically released, unless explicitly assigned. Usually, this happens when the evaluation of a function call is complete. The value of the function is returned to the calling frame and all other storage in the current frame is released.

External objects are implicitly different. By assumption, they always exist on some external medium such as a file. Whenever an external object is generated, it must refer to such a file, regardless of whether the external object is to be assigned permanently or only used temporarily in some S function. This file is not associated automatically with any frame of the S evaluator, so no current mechanism in S will cause the file to go away when a function returns. More seriously, there is no easy way to know whether the file *should* go away at that time, since an object that refers to the file may have been returned as the value of the function.

This sort of complication is an indication that external objects are violating the spirit of the language to some extent. Specifically, an external object contains in essence a *pointer* to the external file or other medium. Most S computations carefully avoid pointers—the language deliberately has no pointer mechanism. It is not surprising, then, that several touchy difficulties arise for external objects.

The current software for plain external objects and images treats these problems as follows:

- When a temporary plain external object is generated, the file is created in a special subdirectory. The underlying code arranges for this file to be removed automatically, but not until the end of the top-level expression.

- As a result, images and other plain external objects can be passed around and returned as values of S functions without taking any special precautions against the external file disappearing.

- When a plain external object is assigned permanently, the file is moved from the directory for temporary objects to another directory for permanent files. The directory for permanent files is parallel to the directory in which the object themselves are stored (actually, a subdirectory of it).

- In keeping with the bias against pointers, there should never be two permanent S objects referring to the same permanent file for a plain external object.

The current approach as not ideal, but its advantages are that it provides a safe, reasonably general way to use plain external objects without imposing too large an overhead or requiring special features in S itself.

Several disadvantages make it likely that the mechanism will be modified in the future. Most obviously, many temporary files can proliferate in the course of an expression. There is a mechanism (discussed below) to associate temporary objects with a particular frame, but the mechanism requires explicit action by all the functions generating temporary external objects to be effective. An subtler limitation is that the mechanism is tied to the notion of a directory as the S database in which the permanent objects are to be assigned. Some extension is needed for other kinds of S database. Finally, the assumption that the temporary external will be stored in a file may not be true for other kinds of external objects to be developed in the future. There needs to be a more open-ended provision for creating and managing the external medium associated with temporary external objects.

When a temporary external object is created, it is in fact associated with the particular evaluation frame of the function that created it. Normally, this fact is of no particular consequence. A special function exists, however, to use this information to clear out temporary external objects:

```
return.external(value, frame)
```

expects as its arguments an external object, `value`, and a number identifying a frame in the S evaluator. The frame associated with `value` is set to `frame` and all other temporary externals associated with the function that called `return.external()` are cleared out. This function provides the needed mechanism for passing back a temporary external object as the value of a function and removing any other temporary externals. At the moment, however, it is too clumsy to be used in every relevant method.

A long-range solution will likely involve recognizing the `return` operation as a generic, so that return actions can have methods. As its name suggests, `return.external()` is in effect a method for returning external objects. If that method were invoked automatically, the same semantics of storage release that apply to ordinary S objects could be provided for temporary external objects as well. The 1991 version of S does not provide methods for returning function values. Provided that the overhead of the feature can be kept low, it is likely to appear in a future version. Once that mechanism is in place, automatic removal of other temporary objects in an evaluation frame could also be done safely.

6 Acknowledgements

Thanks to David James, Jim Landwehr and Daryl Pregibon for help with the examples, to Bruce Wallace for assistance in digitizing, and to Allan Wilks for discussions of image graphics.

References

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

[2] John M. Chambers and Trevor Hastie (eds) (1991) *Statistical Models in S*, Wadsworth, Pacific Grove, California

[3] Comizzoli, R. B., Landwehr, J. M., and Sinclair, J. D. (1990) Robust Materials and Processes: Key to Reliability. *AT&T Technical Journal* **69** (No. 6), 113–128.

[4] David R. Nadeau, T. Todd Elvins, and Michael J. Bailey (1991) Image Handling in a Multi-vendor Environment. *Proc. Visualization '91*, IEEE Computer Society Press, Los Alamitos, CA.

[5] Jef Poskanzer (1991) The PBMPLUS portable bitmap toolkit. (At the time of writing, this is software is available free by anonymous ftp. For current status, contact the author: jef@well.sf.ca.us.)