Variables in Data Frames

John M. Chambers
AT&T Bell Laboratories

October 19, 1993

Contents

Abstract

Models in S make use of `data.frame` objects to store a variety of variables involved in the models: numeric variables, factors, and other special classes of variables. This paper presents some new software that revises the treatment of such variables from that described in *Statistical Models in S.* The changes have three main goals:

1. to allow new classes of variables to be added smoothly to the software;
2. to make the treatment of variables more logical and consistent over the S functions involved;
3. to use the class/method mechanism more effectively in carrying out the computations.

The changes are carried out by new and revised S functions; once these revisions are made, the new computations can be used with the existing, 1992 release of S. In addition, the changes are designed to mesh with the future Version 4 of S.

1 Introduction.

Statistical models in S make use of the `data.frame` class of objects to represent the data used. Data frames contain a number of named variables, all representing some data indexed by the same set of observations. Thus data frames resemble matrices, with variables taking the role of columns and observations the role of rows. Unlike S matrices, however, the individual variables in a data frame can in principle be anything, so long as the indexing concept is well defined. At the same time, data frames are often used to define evaluation frames in S; that is, to evaluate S expressions in which the individual variables are referred to by name.

The software for statistical models in S uses data frames in several ways. Normally, users store the data for the models in data frames, and specify models in fitting by formulas that use the variable names as objects. The fitting software constructs a special *model* data frame, in which the formula is used to construct the numerical data required for fitting. Other computations, such as prediction, may also construct data frames for their own purposes. Users can construct and manipulate data frames with a variety of S functions and methods. Some of these use the analogy with matrices, for example to extract and replace portions of data frames referring to rows and columns. The functions `data.frame()` and `as.data.frame()` construct data frames and the function `read.table()` reads them in from table-like text files.

Since there has been no flood of complaints, the software provided presumably is fairly adequate for most applications. There are, however, a number of known deficiencies in the existing treatment.

- The full generality of variables is not so easy to achieve. In particular, `data.frame()` makes some assumptions internally about the kinds of variables it will handle. There is an "escape" via the `I()` function, but this is not entirely effective.

- As a result, it is difficult to introduce new classes of variables into data frames, making extension to new kinds of models more difficult than it should be.

- The different methods operating on data frames are not consistent in their treatment of variables. For example, the same data included in a data frame via `data.frame()`, `cbind()`, or the methods for replacing columns could produce different results.

- The treatment of classes and methods for data frames and their variables developed along with the current approach to classes. Not surprisingly, the treatment is somewhat inconsistent. Future revisions of classes and methods in S will require a cleaner approach.

One simple change (see section 3) organizes the data frame software around a consistent set of methods that can be extended by users to introduce new classes of variables.

Most users will not need to take explicit account of the changes. Section 2 outlines the rules resulting from the new treatment. These are mostly what would have been expected from the description in chapter 3 of *Statistical Models in S* [1] and section 2 of this paper may be considered as an addendum to section 3.2 of that book. Section 3 discusses the mechanism for introducing new classes of variables into the computations on data frames.

2 Treatment of Variables in Data Frames.

New or modified variables are introduced into data frames by the expressions:

```
data.frame(...); cbind(...)
as.data.frame(object)

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

The arguments to the functions and the right side of the replacements can be objects of various classes, including numeric vectors, factors, matrices, lists, and other data frames. Conceptually, each such expression is converted individually to a data frame, and the resulting data frames are then merged or used to provide replacement data.

The rules for introducing variables into data frames can therefore be presented by saying what happens when a particular class of object is coerced to be a data frame; that is, by defining the effect of `as.data.frame(object)`, summarized in Table 1. In this table, the column "Method" implies a function whose name is `"as.data.frame.`*Method*`"`; the entries in the column "Classes" are classes in the extended sense that will apply in future versions of S[2], roughly equivalent to the value of `data.class()` in the current version.

3 New Classes of Variables.

New classes of S objects can be included in data frames by defining a method for the generic function `as.data.frame()` for that class. The method should return a data frame with the appropriate number of observations. In most cases, one of the methods in Table 1 can be used directly or called from the new method. This saves effort and makes the new method robust to any changes in details of the data frame functions.

Two common cases serve as examples. Suppose a new class operates like a vector, in that it wants to contribute a single variable to a model. In this case, the `vector` method can be appropriated. For example, suppose class `"chron"` objects are to be treated like vectors. The S source for the new objects should then contain the line:

Method	Classes	Result
`vector`	`numeric, complex, factor, ordered`	The object produces a data frame with one variable. Names, if any, become row names.
`character`	`character, logical, category`	The object is converted to a factor and the `vector` method then applied.
`matrix`	`matrix`	The data frame has one variable for each column. The `dimnames` if present define the names of the data frame, from the column labels, and the row names, from the row labels.
`list`	`list`	Each element becomes a variable, by applying these rules recursively. All elements must correspond to the same set of observations.
`model.matrix`	`model.matrix`	The object becomes a single variable. Row labels, if present, define row names.
`data.frame`	`data.frame, design`	The object remains a data frame with the same variables and row names.

Table 1: *The result of coercing an object to be a data frame, by using a method for* `as.data.frame()`.

```
as.data.frame.chron <- as.data.frame.vector
```

This depends on `chron` objects looking sufficiently like vectors; specifically, `length(x)` should return the desired number. Typically, such classes of objects are structures that contain suitable vectors as data, along with some special attributes.

The second common case is that the new class of objects will be used to construct several numeric variables in the data frame, all of which are supposed to make up one term in the model. What is wanted in this case is a data frame that contains a matrix as its single "column". The modeling software converts factor variables into such matrices, one column for each degree of freedom. Special functions, such as `poly()` and various smoothers, also generate such terms. The suitable method from Table 1 for this case is `model.matrix`. Suppose the new class of objects has class `"mMatrix"` and contains a numeric matrix, presumably along with other attributes. We want the corresponding data frame to have one variable containing the entire matrix. The S source should then contain the line:

```
as.data.frame.mMatrix <- as.data.frame.model.matrix
```

As with the previous example, this only works if `mMatrix` objects can legitimately be used as numeric matrices with the intended number of rows.

If you really need to design your own method for converting the object to a data frame, here is how to do it. The arguments to the method must agree with those of `as.data.frame()`:

```
as.data.frame <-
function(x, row.names = NULL, optional = F)
    UseMethod("as.data.frame")
```

The first argument is the object to be coerced to a data frame, the second is an optional vector of row names (this argument is never used internally but can be supplied by a user), and the third is a flag, which is `TRUE` when the calling routine does not require `as.data.frame()` to generate non-trivial row names or variable names. The flag is used when `as.data.frame()` is called by other functions, such as `data.frame()`, that plan to generate row and column labels later on.

Even with specially-written methods, the usual technique is to construct an object suitable for one of the existing methods, and then to invoke that method directly. For example, the existing method for `character` objects generates a factor and then applies the existing `vector` method to that object:

```
as.data.frame.character <-
function(x, row.names = NULL, optional = F)
as.data.frame.vector(factor(x), row.names, optional)
```

It is essential to include all three formal arguments in the function and to pass the arguments down to `as.data.frame()` or its methods.

Finally, any existing function that constructs a data frame will work as a method for `as.data.frame()` so long as the arguments are accounted for. Suppose function `otherData()` constructs a data frame from an object of class `"other"`. The method just needs to take account of user-supplied row names:

```
as.data.frame.other <-
function(x, row.names = NULL, optional = F)
{
    x <- otherData(x)
    if(!is.null(row.names)) {
        row.names <- as.character(row.names)
        if(length(row.names) != nrow(x))
          stop(paste("provided", length(row.names), "names for"
           nrow(x), "rows"))
        attr(x, "row.names") <- row.names
    }
    x
}
```

This method ignores the argument `optional`, which may mean a slight loss of efficiency if `other` objects don't define row names or variable labels in an obvious way.

4 Consistency.

The second objective of the revised treatment of variables in data frames was to ensure that apparently equivalent ways of treating data frames produced equivalent results. For example the three expressions

```
currentFrame <- data.frame(currentFrame, var)
currentFrame <- cbind(currentFrame, var)
currentFrame[, ncol(currentFrame) + 1] <- var
```

should produce the same result when `var` generates a single variable in the data frame. In the standard release code, the three expressions did not examine the right side in the same way; in particular, a character variable supplied as the right side would be converted to a factor by the first expression, but not by the other two. The revised code treats all three expressions by applying `as.data.frame()` methods to the arguments or the right side of the replacement expression.

All three constructions now follow the rules in Table 1 — roughly the behavior of `data.frame()` in the original release. As a result, some computations with `cbind()` or with replacement operators will not be upwardly compatible with the release, but the earlier behavior seems indefensible as a feature.

One further incompatibility arises when replacing a column (variable) of a data frame with an object that generates several variables (for example, a matrix). In the 1992 release, the replacement could expand the data frame, so that the number of columns would be different afterwards:

```
currentFrame[, 1] <- myMatrix
```

would increase the number of columns of `currentFrame` by `ncol(myMatrix) - 1`. This behavior is inconsistent with the corresponding behavior of column replacement in matrices, and generally with the semantics that says that replacement of elements in an object should not change the size of the object. In the revised code, column replacement does not change the dimension of the data frame. If the right side implies more than one variable, only the first is used and a warning is issued.

5 Installation.

This software is available for access by mail or *ftp* from the *statlib* archive.

The revised treatment of data frames can be inserted on top of the 1992 release of S by installing revised versions for the functions `data.frame()` and `as.data.frame()`, and methods for `as.data.frame()`, for `cbind()`, and for subset (`[`) and element (`[[`) replacements.The revised software is provided as a single S source file. There is no change to C code or to functions other than these, all of which are in the statistics library. Installation of the revised software therefore amounts to assigning the new and revised functions in the statistics library. If you don't want to adopt the new versions unconditionally, at least until testing them out, create a new directory with an empty `.Data` directory and assign the new versions there. Attaching this directory (in front of the statistics library) will give the revised behavior.

The software is supplied as a shar archive containing the source, this document and some test source. A `README` file gives instructions. The supplied code includes revised methods for `rbind()` and `"[["`related to the issues discussed here, but likely to be relevant to users of data frame objects (see the `README` file).

Acknowledgements.

Bill Cleveland, Trevor Hastie, David James, Daryl Pregibon, and Scott Vander Wiel have provided valuable comments and suggestions, not least by examples in which the original 1992 release misbehaved.

References

[1] J. M. Chambers and T. J. Hastie (eds) (1992) *Statistical Models in S*, Wadsworth, Pacific Grove, California (currently distributed by Chapman & Hall, New York).

[2] J. M. Chambers (1993) Classes and Methods in S. II: Future Directions. *Computational Statistics*, vol. 8, pp 185-196.