Classes and Methods in S
I: Recent Developments

John M. Chambers[1]

[1]AT&T Bell Laboratories, Murray Hill, New Jersey 07974 USA

Summary

This paper, divided in two parts, represents talks given at the Second International S Workshop in Toulouse, France. The general theme of the paper, the use of the *class/method* mechanism in the S system, is developed by describing, in this part, some history of S and recent projects. Some of the projects discussed, such as database management software and image analysis, are particularly relevant for those who might build specialized extensions of S. The discussion here updates that in the two books on S. The second part describes planned directions for future work on S.

Key words and phrases: Programming, Languages, OOPS.

1 Introduction

This paper has as its general theme the use in S of a *class/method* mechanism for organizing programming. From the view of S as a programming environment, the most substantial recent change has been the gradual absorption of this mechanism, with explicit class attributes for S objects and with methods chosen automatically based on those classes. Many recent projects, including all those discussed in section 2, have incorporated this mechanism into their design. Planned changes in the mechanism will be a major feature of the next version of S, along with several other major revisions, all closely interrelated. The class/method mechanism represents an approach to ideas from what is usually called object-oriented programming, customizing those ideas to the overall goals and style of S.

The theme of the paper is elaborated by discussing some recent developments in S, with particular emphasis on topics not covered, or not in detail, in the two books, Becker, Chambers and Wilks(1988) and Chambers and Hastie (1992), all of the topics being substantially influenced by using the class/method mechanism. Part II of the paper discusses future directions planned for S, with emphasis on the rôle of classes and methods.

Although it may appear that emphasizing programming as an activity is to select a small fraction of the use of S, this is not the case. From its beginning, S has emphasized non-routine, creative use of the system. As a result, most users are in fact involved in programming, whenever they write or modify S functions. Making this activity productive therefore has always been a major goal of the system.

1.1 A Brief History

S might be defined as the software side effect of research at AT&T Bell Laboratories in statistical computing — most of S resulted from that research and most of that research tends to be reflected eventually in software in S. S has evolved in a rather different way from most statistical software. Table 1 summarizes the stages in the evolution.

Version 1	1976-1980	Honeywell GCOS, Fortran-based
Version 2	1980-1988	Unix; Macros, Interface Language
	1981-1986	QPE (Quantitative Programming Environment)
	1984-	General Outside Licensing; Books
Version 3	1988-1993	C-based; S functions & objects
	1991	Classes & Methods; Statistical Models
	1992	Extensions; Performance Improvements

Table 1: *Stages in the Development of S. Version numbers are intended to demark major revisions, usually including some substantial break with past designs.*

After some initial attempts in 1965 at a language design (Becker and Chambers, 1989, section 2), never implemented, our statistical computing for the next 10 years centered around subroutine libraries (including graphics) and ad-hoc programming, mostly in Fortran. Version 1 of S was largely seen as a way to provide a high-level interface to algorithms, in the form of subroutines, with additional features of data management and mechanisms for extending the system. Moving the software from a non-portable, proprietary operating system to the UNIX system became attractive when the latter was implemented on machines suitable for scientific computing. Version 2, the first UNIX-based version, carried over the general ideas of the first version, with extensive internal revisions. General distribution of this version and the two books describing it (Becker and Chambers 1984; Becker and Chambers 1985) broadened the population of users greatly, and led us to re-think the purposes of the software correspondingly.

The re-thinking led initially to a separate project of mine, termed the Quantitative Programming Environment (Chambers 1987), which threatened for a while to go off in a separate direction from S. Instead, the ideas in that project were integrated into a re-designed version of S that kept nearly all the specific capabilities of Version 2, while replacing much of the underlying implementation. From the user's viewpoint, the main change was the use of function objects as the single main tool for defining computations, replacing a combination of macro language and special-purpose Fortran preprocessor used in Version 2 (Becker, Chambers and Wilks 1988, Appendix 4). Internally, the base language for the S executive and low-level code changed from a preprocessed Fortran to a reasonably standard form of C, augmented by C structure definitions and preprocessor macros that defined the internal form of S objects. Uniformity, first-class objects for functions and expressions, extended abilities to define data structures, and an explicit concept of interfaces to the operating system and to other languages were the main themes of Version 3.

An important, but upwardly compatible, extension to Version 3 came in 1991 and is largely described in Chambers and Hastie (1992). The two main features, statistical models and a class/method mechanism, will be summarized in section 2, along with some other developments not described in detail in Chambers and Hastie (1992). This was again revised in 1992, mainly to improve performance.

1.2 General Comments

As noted, S developed from a long-term concern for statistical computing among the research statisticians at AT&T Bell Laboratories. Computing is important to us, and to most statistical research groups, in three main ways:

1. as research on new computational techniques relating to the preparation, presentation, and analysis of data;
2. as support for all the research of the group, including that in statistical theory or in methods of data analysis;
3. as a medium (nowadays the most important medium) for the transfer of our technology, the results of items 1 and 2, to our colleagues at AT&T and to the scientific community.

In the past at least, the *absence* from this list of large scale, routine data processing applications was also relevant to our evolution. The relative importance of non-repetitive tasks meant that early-generation packages and languages played a correspondingly small part in our activities. An emphasis on interaction and experimentation characterizes our approach to statistical software.

Over the evolution of S, the design of the system and its implementation have changed radically, and further dramatic change is likely in the near future. The underlying goals of the software have changed much less. Essentially, we are still trying to provide an effective interactive computing and programming environment for those involved in learning from data. Advances in computing hardware and software and our own work have allowed us to be much more ambitious in the way we attack those goals.

Another aspect of S that has changed relatively little is the syntax — the appearance of the language — particularly in direct, interactive analysis. Expressions such as

```
plot(x,y); hist(rnorm(100))
```

are unchanged since the first version. S began as an interactive language with a generally C-like syntax, using data that was allocated dynamically as needed, without type declarations. It retains these features and will continue to. While there are several details that we might like to change if allowed to rewrite our history, none of these is critical enough to justify the pain inflicted on users by making such changes.

Evolutionary change becomes more evident as soon as we consider the *programming* use of S. The tools, the organization, and the potential scope of such programming have changed several times and will change again. Future planning is especially concerned with making S a better programming environment for the increasingly diverse community of users.

2 Recent Developments

This section discusses some aspects of recent changes in S; specifically, changes since the 1988 release of Version 3, described in the "blue book", Becker, Chambers and Wilks(1988). All the projects described in this section made use of the class/method mechanism, which forms our unifying theme.

2.1 Classes and Methods

Version 3 of S introduced notions of functions (and expressions) as objects and emphasized the ability to design software around "classes" of objects, using the term in an informal sense (Becker, Chambers and Wilks 1988, section 5.5 and chapter 8). Classes of objects were defined by their representation. Some number of functions would use this representation to generate or modify objects, and to extract information from them. In principle, these functions would then provide the interface for other software using the class of objects, hiding the details of the object representation.

One tool of object design was particularly distinctive and important; namely, S *structures*, objects using a notion of data plus attributes (Becker, Chambers and Wilks 1988, section 5.6).

An S object is a structure if it consists of a vector of data values (numbers, character strings, or anything else), which are *structured* in some way by additional attributes, with the key property that the structure is independent of the individual values. Element-by-element computations on structures change the data while leaving the structure attributes unchanged. Many common S objects are examples of structures. Matrices have their values arranged in a two-way layout; general arrays have values arranged in a general multi-way layout; time series have values laid out at equally-spaced positions on some continuum.

Not all objects are naturally viewed as structures. The other main model for object design in Version 3 of S was that of a list or list-like object with a set of components (elements with fixed names). An example is the object that represents the matrix decomposition used in linear fitting, the `qr` object (Becker, Chambers and Wilks 1988, section 8.2). This has four fixed components, representing all the information needed to define the decomposition. The two techniques of attributes and components can be combined or used recursively; for example, one of the components in the `qr` object is a matrix.

These techniques and the common examples of them in S will be useful comparisons as we discuss approaches based on more formal definitions of classes, particularly in talking about future plans in Part II. In particular, both attributes and components could be subsumed in the notion of *slots* commonly encountered in object-oriented programming systems (Keene 1989, page 230). Slots correspond directly to components; S structures can be included also by requiring that a structure have a *data* slot, to which element-by-element functions can be applied without altering the other slots (the attributes).

The evolution of S since 1988 has seen a gradual exploration of more explicit use of classes as an organizing mechanism for programming. There are many programming systems using such concepts, generally lumped together under the term object-oriented programming; two widely-used examples are C++ (Ellis and Stroustrup 1990) and the Lisp-based CLOS system (Keene 1989). We could have swallowed object-oriented programming whole by converting S to such a system. For reasons of both principle and practice we have chosen instead to take a small dose first, designing our approach to combine attractive features of existing systems plus ideas of our own that are appropriate to our applications. As a matter of principle, adopting an existing system would require either dropping or seriously compromising some existing features of S (this was particularly true when we first considered the question), imposing a penalty that was considered too high. A practical consideration was that a complete conversion would have required an essentially full re-implementation, at a time when we had a sizable and growing user community as well as a number of other projects we wanted to pursue.

The first step in our incremental approach was to add *class* as an optional attribute to S objects. A small change to the S executive then allowed an S function to call explicitly for the selection of a *method* based on the class of an object; for example,

```
UseMethod("plot", x)
```

would examine the object `x` for a class attribute, typically a character string. If there existed a function object named `plot.`*class* corresponding to the class of `x`, that function would be used as the method, replacing the current function call. If no such method exists, `plot.default()` will be called.

This was certainly minimal-dosage object-oriented programming, deliberately so. The mechanism was first released with the 1991 revision and described in Chambers and Hastie (1992, Appendix A). By then it had been somewhat expanded to include inheritance, a `NextMethod` mechanism, and the concept of methods for groups of functions. The internal shift in the implementation of S, however, remained small. The effect on programming style, in the work on statistical models and in other projects, has been dramatic.

Conceptually, the most important effect is the separation of the *purpose* of the computation, as expressed by the generic function, from the *implementation* of the computation in particular, specialized circumstances, as expressed by the methods for that generic. This separation makes the programming process simpler for both the user of the generic function and the implementer of the underlying methods. The user calls the generic without having to pre-judge how the purpose of the function will be achieved in each circumstance. The implementer can partition the design according to the relevant classes of objects; in addition, commonality as expressed by inheritance has allowed re-use of code in many instances. Our experience is that these advantages are the key contributions to effective programming in our range of applications. S comes from a rather long evolution, combining a number of ideas and philosophies. Any profound changes need to be taken carefully, obtaining the most from the new ideas without sacrificing a productive general approach.

There has been some objection to the approach taken in this version of S from others involved in applying object-oriented programming to statistics, largely on the grounds that we were failing to take full advantage of this style of programming. While, as the comments above indicate, this criticism somewhat misses the point of our approach, it has nevertheless some justification. Having shown that a nearly minimal dose of the programming style has been invigorating and, on the whole, beneficial, we are now prepared to assess what additional ingredients would be of further value. Version 4 of S will contain a much extended use of the class/method mechanism, as will be discussed in Part II.

2.2 Statistical Models

The work that eventually produced Chambers and Hastie(1992) and the software it describes started in 1988 as a broader proposal: "We and our colleagues are planning another book on *statistical software* in S" (Becker, Chambers and Wilks, 1988, page x, emphasis added). The increased computational discussion in Becker, Chambers and Wilks (1988) compared to Becker and Chambers (1984) pushed out much of the more statistical discussion (which was, in any case, fairly limited). In addition, two other forces were at work. Users were pressing for more software in some of the traditional areas of statistics, such as analysis of variance. At the same time, several of us had either software in development or ideas for software that would extend S in such directions. The initial plan was for a set of chapters, fairly independent though written in a consistent style, covering whatever topics in statistical software seemed important in S.

In the event, something related but significantly different appeared. The focus was narrowed to a variety of statistical modeling techniques and the treatment was much more strongly unified in content. There were losses in coverage (e.g., a simulation chapter) and the project took much more time and effort than initially anticipated. In compensation, the project produced a more integrated and powerful computational approach to a relatively wide variety of models than had been previously possible. It also stimulated some research in statistical computing, both generally in its use of classes and methods and more narrowly in the approach to computing for models.

Our computations for statistical models are organized around three main aspects:

1. the data, specifically the `data.frame` class of objects;
2. the structural model, specifically the `formula` class of objects;
3. the statistical treatment of the models — fitting, summaries, diagnostics, and other operations.

The first two aspects allowed the strongest unification. Some relatively simple ideas supported extensive sharing of computational techniques.

The data frame is, as implemented, a list of named variables whose values all correspond to the same set of observations. The user's view of it is more as a matrix, with the observations acting like rows and the variables like columns. Unlike ordinary S matrices, the columns need not be the same kind of data. In principle, they can be anything, so long as they are indexed by the same set of observations. In practice, the statistical models in Chambers and Hastie(1992), and the software for them, restrict the variables to be numeric vectors, numeric matrices, or factors (including ordered factors). The dual nature of a data frame as a matrix-like and list-like object makes the use of the class/method mechanism important. Methods for the subset operator, for example, allow users to treat the data frame like a matrix when extracting subsets of rows and columns. Because the subset operator is a generic function, such code is kept separate from the implementation of the operator for more traditional S objects.

The formula objects used to define the structural part of the model follow a style of compact description, popularized by Wilkinson and Rogers (1973) for analysis of variance models, in which operators, such as `"+"`, `"*"`, `"/"`, and `"-"`, are overloaded to refer to terms appearing in a model. In the reference, operands would be names of variables, typically factors in an experimental design, and the algebraic formulas constituted a special language, which in addition to the operators above allowed parenthesized expressions and two additional operators, `"-*"` and `"-/"`. For linear models and extensions of them, formulas can dispense with any notation for the coefficients themselves, since knowing what terms are included implies what coefficients can enter the model.

Two features in S make formulas particularly attractive. First, since expressions are S objects, the formulas need not use any special language either for "parsing" or for manipulation. All computations can be done directly in S; in addition, the operands to `"+"`, `"*"`, etc. can be completely general expressions, not just variable names, without any special computations. Other operators, special functions and conventions can be added with little effort. We added the convention that the name `"."` expanded into all the variables not otherwise appearing in a particular context[2]. One operator was added specially for formulas, `"~"`, whose left side is the response (not expressible in the algebra of Wilkinson and Rogers (1973)) and whose right side is the formula predicting that response. Adding this formula only involved extending a table in the lexical analyzer. We overloaded the operator `"^"` to define formulas including interactions up to the order given by the power. Using these conventions,

```
y ~ . ^ 2
```

says to fit `y` to all the other variables in the data, including main effects and second order interactions.

A second advantage is that the class/method mechanism allows formulas to have a number of the desired effects without users needing to call special functions. This is important since we need to avoid interpreting the operators in the formula in their default, numeric sense. Methods are defined for `plot()`, `print()`, and `update()`, among others. Additional methods are defined for the related class `terms` discussed below.

[2]Wilkinson and Rogers (1973) used `"."` as an operator, following GENSTAT syntax; we used the operator `":"` instead, since names in S can include the character `"."`. In practice, this operator rarely appears explicitly in formulas.

Formulas are most heavily used in analysis of variance (aov), generalized linear models (glm), and generalized additive models (gam). However, they are worth having even in models, such as local regression (loess) or tree models, where the special kinds of terms the operators describe are nearly or entirely absent. Formulas still provide a compact and readable description of the structural part of a model. Even in nonlinear models, where formulas must include coefficients explicitly, we retained the general form, which we found useful to clarify thinking and to emphasize the common ingredients in the modeling process.

Aspect 3, fitting and summarizing, is the least unified part of the software. While linear models, aov, glm, and gam to some extent share computational steps in fitting, each has a degree of differentiation. The models farther removed from conventional linear models, such as loess and tree models, depart more rapidly from the common path, while nonlinear models have only a faint resemblance.

However, a unified approach and the class/method mechanism are still useful. Fitted models are objects from classes corresponding to the type of model (`lm`, `glm`, `loess`, etc). Plots, printing, summaries, and other computations use methods that trigger from the particular classes. Natural inheritance (for example, the `glm` class inherits from `lm`) allow methods to be reused (the `lm` methods for predicting and updating are used for other classes of models as well).

Also, some of the intermediate objects produced during the fitting are shared among a number of types of model. Except for nonlinear models, all the models use the `terms` and `model.frame` objects. The former object expands the implications of the formula, providing explicit information on all the terms in the model, what variables they contain, etc. The latter is a specialized data frame, defined by the terms and perhaps by optional arguments to the fitting function. It contains exactly the variables and observations to be used in fitting the model. Since these two intermediate objects are important to so many types of model, time spent on their implementation was justifiable. Both ended up being generated in part from specialized C code. The `model.matrix` objects are another class of object that is useful in a variety of model types, though fewer than in the case of terms or model frames. Model matrices contain the numeric variables to be used in the actual fitting for linear, glm, and gam models. They also contain information to interpret the variables, such as an assignment attribute matching columns of the matrix to terms in the model. Models such as loess and tree models do not use a model matrix, but they maintain a useful parallelism of design by generating similar matrices corresponding to the form in which they need to represent data in the fitting process.

Throughout our work on the model software, we emphasized that we intended to provide a base for further, more detailed or specialized software. Hopefully, considerable analysis can be performed directly from the software described in Chambers and Hastie(1992). However, our organization aims to encourage extensions and revisions, since the overall field is clearly very large and since different applications can lead to substantially different views of the same class of models (compare, for example, the use of aov in industrial designed experiments, in biology, and in the social sciences). The organization in S, particularly the use of classes and methods, lends itself to this approach. As usual, feedback from users has stimulated some extensions. The 1992 release of S, for example, adds a contrast attribute to the model matrix from which computations may recover the contrasts used in forming the model matrix. This information is needed to have a general facility for showing effects of given linear combinations of factor levels.

2.3 Data Management

S has always included the idea of separate, persistent object assignment; that is, an assignment expression in S can create something associated with a name that will be accessible in later S sessions and will persist until explicitly removed. Regular S users will hardly consider this a radical idea, but it contrasts with a number of other languages (particularly so at the time when S was first implemented). It raises a number of problems and opportunities that, in our typical fashion, we have only grappled with gradually in successive versions. Most importantly, having separate, persistent assignments means that S must provide the facilities of a database management system to store, retrieve and organize named objects.

The initial approach was to let somebody else do it. Since most computer operating systems include some form of hierarchical file system, the natural first choice for S was to implement persistent assignment by writing the S object to a file of the same name, in a special directory that was typically a specific subdirectory of the user's current directory. Very much in the spirit of interfacing to other computer software to re-use other people's work, this choice was certainly a good starting point. File-system operations, in addition to writing and reading, allowed us to list the contents of a database and remove assigned objects. By Version 3, we had added the notion of a *search list*, consisting of all the databases currently attached to the running S process, important for users who wanted to customize S by including special libraries of functions or collections of data objects.

As S has evolved, however, both practical and conceptual reasons have arisen for separating the idea of an S database from its particular implementation via file-system operations. An S database is a persistent collection of named S objects. Objects assigned to a database remain there, with their current contents, throughout and between S sessions, until explicitly modified or removed. A database may be attached during a session, after which the name of an object on the database may be used to represent that object in an S expression. Any number of databases may be attached, with the S evaluator matching names according to its search list (Becker, Chambers and Wilks 1988, section 5.4). It is this notion of multiple, attachable databases that makes S different from many other systems. Lisp programs have a single address space of named objects, for example, and APL similarly uses a single workspace of objects as its fundamental organizational unit. Most compiled languages, such as C or Fortran, regard persistent data as outside the scope of the language. Of course, packages built upon any of these languages often do include some form of data management. S must integrate data management with the language and with its model for evaluation.

The work on statistical models introduced the first important generalization of databases, allowing a single S object with named components, typically a data frame, to be attached as a database. The components could then be referenced by name in an expression, in the same way as ordinary assigned objects. This extension was particularly attractive for modeling using formulas. The names in a formula might refer to separate objects, to variables in a data frame, or to a combination of the two. Allowing all the possibilities to be treated in the same way simplifies the use of the software substantially.

One related concept needs to be mentioned as well. Databases are persistent collections of objects; the S evaluation process also uses temporary collections of objects, which we now call *evaluation frames*. Just as with databases, named objects in evaluation frames can be referred to, and assignments can be made, but only for the duration of the frame during the evaluation of an S expression. Evaluation frames are created automatically to evaluate a call to an S function. The revisions of S in 1991 and 1992 extended the concept to allow the creation of an evaluation frame as a copy of an S object and the explicit control of the duration of the frame (Chambers 1991, section 3). The operating principle was that evaluation frames should parallel the semantics of databases, allowing for their different role as temporary constructs during evaluation. In particular, the model frame constructed during fitting was handled as an explicitly attached evaluation frame, with advantages of simplicity and efficiency.

Some other changes at this time removed restrictions on directories used as S databases. Restrictions placed by file systems on file names (sometimes these are limited severely in length; nearly always there are some prohibited characters, such as `"/"` for UNIX systems) are inconsistent with S and can be annoying — even occasionally disastrous, as when two distinct object names are mapped to the same file. The problem is particularly vexing for systems, like MS-DOS, that impose severe restrictions on file names. S now uses a simple name-mapping mechanism to ensure that any non-empty character string can be used as an object name. A number of other internal changes have been made, mostly to improve the reliability and/or efficiency of the database management operations.

The 1991 and 1992 versions of S actually introduced a much more general extension of databases, documented in Chambers(1991). Using the notion of classes, the S search list was re-interpreted as, conceptually, an object itself with components being the individual databases. Individual databases can be directories, S objects, a special compiled type of database, or can belong to user-defined *database classes*. The last is the most important conceptual extension. A class of objects is suitable for use as databases if methods are defined for a specific set of generic functions:

```
dbread(database, name)          # read 'name' from the database
dbwrite(database, name, object) # write 'object' as 'name'
dbremove(database, names)       # remove the named objects
dbobjects(database)             # the names of all the objects
```

What happens inside the methods is completely arbitrary so long as they produce the desired effect; once again, the separation of purpose from implementation opens important new possibilities.

A goal of database classes has been to open S to the use of arbitrary database management software, when applications find such software useful. The four generic functions above provide a way to access databases, given that individual names in the database correspond naturally to ordinary S objects. One early use was to store S objects as *compressed* files, for a software system for visualizing data in a large network (Becker, Eick and Wilks 1991). *How* the data is stored is essentially arbitrary. However, if the *model* for data is essentially different, as for example when the relational model is applied to very large databases, some additional software will be needed. Combining external objects, to be discussed next, with the ideas of this section, offers a possible way to treat such databases in S.

2.4 External Objects and Images

An early design decision for S was that objects should be manipulated in program memory, with the entire object accessible at once, in contrast to a number of systems at the time, which defined operations in terms of iteration over objects stored on some external medium. Because S has always needed to support research on new techniques, the limitations imposed on possible computations by an external format were not acceptable. We have tended throughout the evolution of S to push hardware requirements somewhat, in the expectation that hardware advances will turn the marginally affordable into standard equipment in a few years, as has proved to be the case.

This is not the whole story, of course: users' appetites for larger problems and more computation grow at least as fast as computing capabilities. We would like to handle arbitrarily large objects in a more graceful way, if that can be done without sacrificing the essential style of programming in S. The major disadvantage of in-memory treatment of large objects in modern virtual-memory computing environments is that the time taken for a particular operation that is in principle linear in the size of the object will eventually increase much faster than linear. For example, at some point the overhead for paging to access the data will slow down computations substantially.

A first step towards treating arbitrarily large objects in S has been made, and applied to creating some software for handling image data. Neither the general software nor the image application should be regarded as production software at this point. Both have relatively sparse capabilities and are intended as research pointing towards possibly useful future capabilities. A research report, Chambers and Crellin(1991), outlines the techniques, with emphasis on the image application. In the present paper, the work will be reviewed briefly with more emphasis on the general goals.

Once again, the approach relies on the class/method mechanism. We will consider methods associated with the class of `external` objects. Conceptually, these are objects that may be stored externally, in some special manner. Reference to such an object will not cause the data stored in this form to be read in, complete. Instead, a special set of methods will provide controlled access to portions of the data at a time. If the user calls a generic function for which such methods exist, the computation will carry through automatically. In this way, a capability for handling external objects can be built up without restricting the overall organization of S or requiring users to adapt their computations explicitly. Of course, since the available methods cover only a subset of S functions the user will be brought up short upon trying an unimplemented function. A comment on this for the future appears at the end of this section.

Another design goal was to accommodate a diversity of applications. Rather than assuming that all external objects are handled the same way, we build the software on top of a few special access functions. Specifically, we assume that a contiguous subset of the data in an external object can be accessed efficiently if the subset is itself the size of a typical ordinary S object. Then we define a generic function, `spans()`, that takes an object, plus the starting elements and widths of one or more contiguous subsets, and returns the corresponding data (as an ordinary S vector):

```
spans(x, 1, 10000)
```

would return the first 10,000 elements of external object `x`. On the left of an assignment, the same expression would replace the corresponding data values in the external object. Methods for common S functions, such as the summary, math, or operator groups (Chambers and Hastie, section A.7), are then written in terms of the `spans()` function.

A programmer wishing to make use of the external object code needs to define some other class, inheriting methods from `external`. This new class must have or inherit methods for extracting and replacing spans, based on whatever scheme is used for storing the data. Besides methods for spans, an external class needs to be able to create and remove objects.

The application of external objects described in Chambers and Crellin(1991) is to software to create, manipulate, and display images. The image software builds two layers of additional classes on top of class `external`. The `image` class itself adds the two-way structure of an image, including the assumption that image data are stored by raster line, left to right and top to bottom. (This assumption permits image methods for matrix-like subsets using the `spans()` function and corresponding replacement.) On top of the image class, in turn, are classes that assume various specific protocols for storing image data. There are many such protocols, though often they are sufficiently similar to use all or most of the same access code to implement `spans` methods.

In the work reported in Chambers and Crellin(1991), we implemented a simple form of image object using packed byte or numeric data. With this as a base, functions to create, plot, and manipulate images were written. A number of generic functions were then obtained for free by using the `external` class methods. Additional matrix-like functions were also obtained, some of them for free since they were written using generic functions, such as `"["`, for which image methods existed. A single interface function was written to access data in an existing image package. This allowed fairly trivial implementation of S functions corresponding to commands in that package.

Overall, the work seems a useful feasibility demonstration for the style of programming. We would like to see users with specific needs in this area experiment further, particularly to assess the responsiveness of the general approach, in terms of efficiency and the ease of implementing new computations. Also, as noted, the evolution of Version 4 will almost certainly provide new possibilities for external objects. One of the more intriguing possibilities is some form of automatic sampling from external objects. The simplest idea would be to intercept an attempt to use an external object as an ordinary object (for example, an `as.vector()` call), and to substitute a random sample of chosen size. This idea could be improved on in several ways (retaining sample(s) to provide consistent results; applying experimental design techniques to improve on random sampling). There are some obvious objections, such as difficulty in general in assessing the effect of using the sampled data, but the great advantage would be to avoid or postpone devising `external` methods for all S functions.

2.5 Software Testing

Work on S between the 1991 and 1992 releases included a concentrated study of performance, looking for improvements in speed and memory usage. The practical results were likely of most interest to users, with substantial reductions in memory and time for large, iterative computations. Two other products of the work are perhaps of more lasting interest: inferences for future re-design and techniques for software testing in the context of S. The performance studies produced considerable improvement but re-enforced an intuition that further major gains would require a substantial re-design. This will include new internal structures, new memory management, and the ability to compile S functions into the base language.

The testing of substantial software systems, both for integrity (does the system do what it should) and performance (how efficiently does it work) is recognized as a difficult, interesting problem (Marks 1992, chapter 5; Chambers 1992). By some standards, S is not a large enough system to be challenging to test — it has only about 150,000 lines of code and definitely employs a small programming staff, compared to millions of lines and hundreds of programmers in conventionally large systems. This small size is, in fact, reflected in the relative ease of doing some kinds of automatic testing. On the other hand, a general language and system such as S has a much wider scope for use. Testing by running standard "jobs" does not capture the use of S in the way it would for, say, an order-processing system.

During this period, the testing facilities for S evolved into a general, extensible set of tools, making use again of the class/method mechanism. In Chambers (1992), this work is related to general issues in testing software for data analysis. The brief discussion here relates the work to the general style and evolution of S.

The software developed to test S is organized in three levels:

1. S functions, particularly methods for testing equality of two objects;

2. S source files containing assertions about S computations, often using the functions in 1;

3. shell commands that run the source files.

The commands are designed to run the source files silently unless possible problems are encountered. Each expression in the source file evaluates to `TRUE` if the computations are carried out as expected. If the expression evaluates to anything else, the shell command prints the result along with the expression. Since the introduction of Version 3, part of the S system generation process has been a recommendation that the installer run the testing shell commands on a standard set of source files. This verifies that the system has compiled and loaded correctly on the machine in question. On the whole, the process has worked fairly well, aside from some uncorrected but innocuous machine dependencies, chiefly in the graphics code (Chambers 1992, section 3.3).

The testing facilities needed to evolve for several reasons. The statistical modeling software, Chambers and Hastie(1992), needed a more systematic, adaptable base for integrity testing. The extensive tinkering for performance improvements also needed adaptable, flexible testing, both for measuring performance and for detecting and fixing any glitches introduced. At the same time, there was an opportunity to make testing available in a convenient form to users developing substantial applications in S.

This sort of testing is automated, in the terminology of the field (Marks 1992), in that it does not require detailed human intervention. That it is also silent when all works well is relatively unusual in the testing field; the scenarios described in Marks(1992) usually assume that some reports will be generated and reviewed by humans and this is indeed often the way of proceeding. Silent testing is clearly more efficient of human time. Its success requires valid assertions to test all the relevant software. Assertions are usually simple enough for operations that essentially only manipulate data; for example,

```
all(x[row(x)==col(x)] == diag(x))
```

tests the consistency of the definitions for the functions `diag()`, `row()`, and `col()`. Passing the test definitely does *not* guarantee the correctness of the functions, but hopefully will catch a number of possible glitches if the functions were to be revised. Failing the test almost certainly means trouble.

Unfortunately, things get more complicated as soon as one introduces numerical computation. Numerical analysts have long struggled to provide useful guarantees or tests for numerical accuracy that would apply portably to important algorithms. The closest one can come is typically the assertion that the computed result will differ from a known, "exact" result by an amount that can be bounded in terms of some characteristics of the machine's arithmetic. For example, some standard matrix decompositions can be shown to correspond to the exact decomposition of a matrix that is close to the given matrix in a quantifiable sense. An example would be an assertion to test the singular value decomposition:

```
{ zz <- svd(x); xx <- zz$u %*% diag(zz$d) %*% t(zz$v)
    all.equal(x, xx) }
```

The function `all.equal()` is defined to return `TRUE` if its two arguments are entirely equal up to reasonable rounding error. The definition of `all.equal()` is the key problem to such general assertions — no adequate definition can be independent of the computation being tested. At first the situation appears hopeless in that the definition of `all.equal()` seems to reduce to the numerical testing problem we were trying to solve in the first place.

In fact, while no complete solution to such testing problems is possible substantial progress can be made by using the `all.equal()` function. Two helpful considerations apply. First, while precise numerical testing is important in some applications, our goals are in a sense more modest. We are willing to start with a computation agreed to have been "good enough" when carried out once, on a particular implementation. The goal is then to catch serious regression failures, either because the code has been revised or because the computation does not work when applied on another machine — where by not working we mean something more serious than rather small numerical differences.

Secondly, the dependence of the desired test on the particular application can be gracefully accommodated using the class/method mechanism. The function `all.equal()` is a generic function. Methods have been written for testing equality of classes of objects, such as some types of fitted models. These methods can and must take into account known indeterminacies in the computation. For example, the decomposition used in linear modeling has a choice of sign at each stage that can amount to flipping a coin. The method to test equality of `lm` objects allows for such differences.

In addition to these methods, default methods test numerical objects using a rather lenient allowance for rounding error, and test general objects by matching components and attributes, using `all.equal()` recursively. The result is a general, extensible style of testing that allows the developer to create a reasonably thorough test script with a moderate amount of effort. As in the previous sections, we see the essential role of the class/method mechanism here in improving the organization of the effort, usefully separating the purpose — testing objects for approximate equality — from the implementation details needed in particular cases.

The goal of reliable, portable testing remains to be achieved, however. The computations involved in fitting models, as noted, lead to possible ambiguities. Not all can be eliminated easily in the `all.equal` methods. Fitting tree-based models, for example, employs an algorithm that does not optimize any explicit criterion. It recursively splits subsets of observations, choosing the best split at each step. Frequently, more than one choice gives the best split, up to rounding error. In this case, differences of arithmetic between two machines may lead to different fits, not necessarily equally good, even though the algorithm has behaved correctly in both cases (Chambers 1992, section 4.2) gives an example).

The adoption of the IEEE floating-point standard (IEEE 1985) goes a substantial way towards alleviating problems of testing numerical results. While the standard is permissive enough that numerical differences could still arise, our experience has been that anomalous behavior is much more common for systems that do not conform to the standard. There is reason to hope that non-conforming systems will become increasingly rare, but the time has not yet arrived when we can assume the standard in our testing, or in the software itself.

Other major challenges exist in testing a system such as S, including tests of interaction, error handling and graphics. Some of these are discussed in Chambers (1992).

References

Becker, R. A. and Chambers, J. M. (1984) *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Pacific Grove, California.

Becker, R. A. and Chambers, J. M. (1985) *Extending the S Language*, Wadsworth, Pacific Grove, California.

Becker, R. A. and Chambers, J. M. (1989) Statistical Computing Environments: Past, Present and Future, *ASA 150 Proceedings Volume*, Wadsworth, pages 245-258.

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

Becker, R. A., Eick, S. G. and Wilks, A. R. (1991) R. A. Becker, S. G. Eick, and A. R. Wilks (1991), Basics of Network Visualization, *IEEE Computer Graphics and Applications*, **11**, pages 12-14.

Chambers, J. M. (1987) Interface for a Quantitative Programming Environment, *Comp. Sci. and Stat., Proc. 19th Symp. on the Interface*, pages 280-286, Elsevier Science Publ.

Chambers, J. M. (1991) Data Management in S, *AT&T Statistics Research Report No. 99.*

Chambers, J. M. (1992) Testing Software for (and with) Data Analysis. *AT&T Statistics Research Report.*

Chambers, J. M. and Crellin, N. (1991) Images in S *AT&T Statistics Research Report No. 100.*

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth, Pacific Grove, California

Ellis, M. A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading.

IEEE Standards Board (1985) *IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, New York.

Keene, S. E. (1989) *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*, Addison-Wesley, Reading.

Marks, D. M. (1992), *Testing Very Large Systems*, McGraw-Hill, New York.

Wilkinson, G. N. and Rogers, C. E. (1973) Symbolic Description of Factorial Models for Analysis of Variance. *Applied Statistics* **22**, 392–399.