Testing Software for (and with) Data Analysis

John M. Chambers
AT&T Bell Laboratories, Murray Hill, New Jersey
Statistics Research Report No. 107

October 9, 1992

Contents

Abstract

As software systems become more ambitious and extensive, testing them for correctness or performance becomes more challenging, especially when the systems themselves provide a general computing environment, not just a few narrowly defined services. This paper describes some testing experience with S, a large system for data analysis and scientific computing. Included are:

1. comparisons and contrasts with general software testing issues;
2. description of tools and techniques for testing S;
3. examples drawn from recent experience.

The paper discusses issues in software testing, with particular reference to testing languages and systems for data analysis. Conversely, some of the examples argue for the importance of flexible techniques for data generation and analysis as support for testing.

1 Introduction.

This paper discusses some tools and techniques used in testing the S system. The tools test the integrity of the system; i.e., whether it performs particular computations as expected. They also measure the performance of the system in terms of the computer time and memory required. The discussion brings in some related topics as well, such as similarities and differences compared to the standard software testing literature.

Such testing should be contrasted to software *evaluation*, as that term is usually applied. Software evaluation usually compares two or more systems, from the user's perspective. Sometimes there is no actual testing involved, particularly if a large number of systems are being compared (e.g, Francis, 1973). If testing is done, the tests will be chosen to represent some common subset of tasks applicable to all the systems. In contrast, the testing discussed here is usually designed by the creators of the software or by a testing group working on that software, and represents an attempt (never entirely successful) to test the system completely.

A discussion of testing for data analysis software is worthwhile at this time, for a number of reasons. Software for data analysis is widely used and continually evolving: testing facilities are essential in this context and need to be described and improved. So far, to my knowledge, little or nothing has appeared from owners of ambitious systems of this type to describe the testing process. The tools and techniques for testing can benefit not only

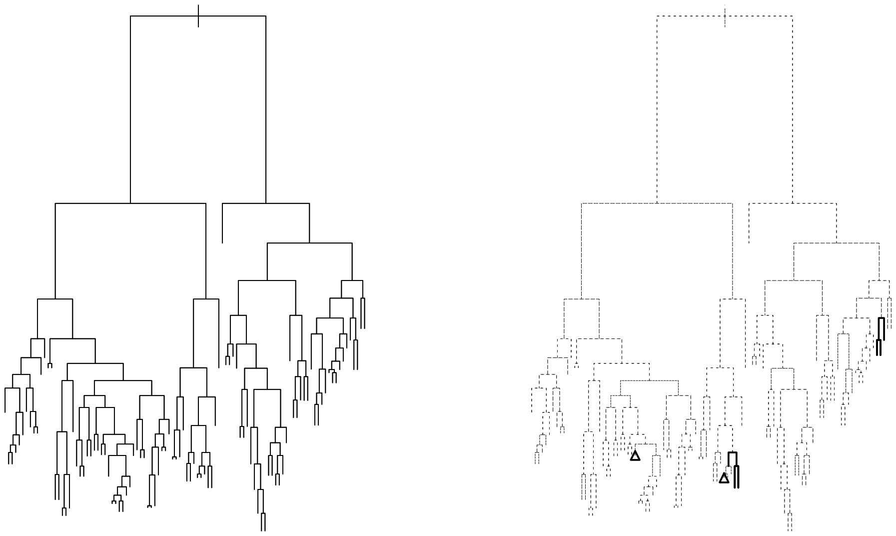

Figure 1: *The left panel shows a tree-based model (specifically, the example shown on page 397 of Clark and Pregibon (1992)), but computed on a machine with a different model for floating-point arithmetic from that used in the reference. The tree shown in dotted lines on the right panel is the reference. Heavy lines show subtrees in the reference not reproduced on the left panel and the two triangles mark terminal nodes differing in the two trees.* *See page 27 for more details.*

system developers but also users who find themselves building substantial software on top of the original system, as is often the case with S. The techniques may also have some useful applications to testing systems other than those for data analysis. Lastly, a number of interesting technical questions arise, many of which could benefit from further research.

An example of such a question is: how should the results of a highly non-trivial statistical computation be tested for equality when carried out on two different computer systems? As will be seen, this question is central if testing is to be extended in a general way as the software system evolves. For example, Figure 1 compares a tree-based model computed on two different machines by the same software. The results are very similar but not identical. Are the two computations equivalent, up to expected numerical differences? Can we hope to test this equivalence in general? Sections 3.2 and 4.2 will discuss this issue: for tree-based models we do not yet have a completely satisfactory answer; even for linear least-squares models the problem is not trivial.

The plan for the rest of the paper is as follows. After discussing approaches to software testing in section 2, we describe the approach taken for testing S in section 3, as it has evolved in recent work. Section 4 presents some examples and section 5 some general comments.

2 Testing Software Systems.

For obvious reasons, discussions of software testing tend to focus on systems with millions of lines of code, involving hundreds of people in design, implementation, and testing. Statistical systems tend to be of a smaller scale; S has less than 150,000 lines of code and a small, mostly part-time group of designer/programmer/testers. However, testing such systems is no less important to their use, and the very general way in which users can apply such systems presents special challenges in testing.

2.1 Testing Large Systems

Testing large systems is generally accepted to be very important and very difficult. Marks (1992) gives a readable account of current practice, with an admirably realistic emphasis. It will be useful to summarize that practice, both as a background to issues specific to statistical software and also to suggest areas where improvements can be made.

Testing is usually described as a process carried out, in parallel with the evolution of the system, by a separate testing organization. Viewed longitudinally the process involves three main phases: planning for testing, executing the testing, and evaluating the testing. Viewed in terms of the content, most current practice can be summarized in four parts:

- testcases, the data and other input that defines what is tested;
- procedures, including tools used for testing and for reporting and documentation activities;
- evaluation, especially in terms of "metrics";

- management, both within the testing organization and between that organization and higher management.

Marks comments several times that testing organizations are understaffed and perhaps under-appreciated relative to the importance of testing. He also rightly emphasizes that the tools for testing benefit from good software design technology in making the testing organization more productive. Some other technology could be applied more than it is currently, however, as some examples below will suggest. The evaluation phase particularly makes little use of data analysis or even of computing techniques. For the most part it continues to be described in terms of printed reports and rather insensitive metrics, perused by the testing personnel to come up with analysis of the testing results. Experience with S certainly suggests that both computational tools and data analysis are important in reducing the human effort involved. For example, we will see that the use of object-oriented tools in software for evaluating tests is key, just as it is in the system itself. Also, some simple data analysis that collects and summarizes comparative results under varying test conditions is essential in evaluating proposed changes to a system.

Testing general software systems is usually described as part of the "development life cycle" of the software. In particular, modifications to the system should be integrated into the testing process as well. Current practice in very large software systems usually does this by written reports (software modification requests) which are shared by the software development and testing organizations. In the much less formal work on S, the importance of this integration is at least as great. Some examples below will illustrate the value of quick feedback from tests as guidance while software is being modified. The computational and data-analytic tools for test evaluation come particularly into their own here, where feedback with little human effort is crucial.

2.2 Testing Statistical Languages and Systems

The example used in Marks (1992) was a hypothetical order-processing system. This and similar systems respond to a variety of tasks supplied to them. In testing, attention is generally focused on whether they produce the requested results and maintain integrity of the overall state of the system (its databases), as well as possible issues of real-time performance and the interface for data input. The testing of general purpose languages and systems for data analysis and for related computations shifts emphasis somewhat and introduces some important issues considered at most marginally in the standard testing literature. Every discussion of software testing notes that complete testing of a large system is impossible, at least within economic constraints. A general language is impossible to test completely in a stronger sense: No fixed test process could anticipate all the programs that users could write in the language, especially if the language operates on objects whose size and structure are not restricted a priori. We need to rethink the choice of testcases and particularly the evaluation of testing if we hope to extrapolate from the test results to a much broader range of user interaction with the language.

For some views of programming languages, on the other hand, the testing process might appear to be narrower, not broader. Testing of compilers and support libraries for programming languages has tended to consist of verifying correct compilation of the basic syntactic range of the language and correct behavior (usually judged subjectively by the installer) on a fixed, fairly small set of test programs. Useful statistical software, however, supplements the language itself with a large system. Testing the language is only a small, though important, part of testing the overall system.

Statistical software is also harder to test than, for example, an order-processing system because of what it does. The correct state of the latter system after it processes a given customer request can be specified with some precision, and that correctness can be verified unambiguously. Statistical software centers on at least three kinds of computing that make such verification more difficult:

- numerical computations, often highly nontrivial;
- displays, printed and especially graphical;
- user interaction.

For these computations the question of a correct result raises some difficult and interesting issues, as some examples will show below. Furthermore, S and other statistical systems aspire to portability, over a variety of computer hardware, display device, and perhaps even operating system. Making the testing process adequately portable as well intensifies the difficulties.

Last, but clearly not least, is the question of performance. Tests of whether the system gives valid results for a particular computation must be supplemented by measurements of performance — the time and memory the system uses to do the computation. Given that the range of computations users can hand to the system is unbounded and unpredictable, and that performance results depend also on local conditions such as computing hardware, overall system load, performance tests are even less amenable to a pass/fail result than integrity tests, but they are key elements in evaluating a system. All serious testing likely includes performance testing, but in many accounts, no explicit distinction is made from integrity testing. Real-time systems build performance requirements into the system design and therefore provide performance guarantees as part of integrity testing. General languages cannot have such precise requirements for performance. As a result, much of the information in performance testing comes from comparisons between different versions of the system or different computations of a particular result. Much of the information also amounts to a fairly optimistic extrapolation from some fixed suites of test results.

All the same, the ability to test performance flexibly and easily is an essential for tuning any system design. Having good tools for testing and for summarizing the results of performance tests helps elevate performance issues to the prominence they deserve.

2.3 Testing S

The rest of this paper summarizes the approach to testing taken with the S system. S is an interactive language and system for data analysis, graphics, statistical modeling, and other computations. Use of the system is described in two books: Becker, Chambers and Wilks (1988) and Chambers and Hastie (1992). For this paper, we need a brief description in terms of what S is, rather than how to use it, to put the testing issues in context.

S is implemented as a combination of compiled code (from source mostly in C with some Fortran) and functions written in S itself. The source for the 1992 version consists of some 140,000 lines of which about three-quarters is the C and other compiled code, with the rest being S functions plus a relatively small amount of administrative code implemented as command scripts or other UNIX tools.

Is this a small, medium, or large system? Marks (1992, page 9) defines medium systems to have more than 500,000 lines and big systems to have more than 3 million lines. Certainly, S is small by this criterion; other measures such as programming staff size would agree with this. The "programming staff" at AT&T Bell Laboratories for S has consisted essentially of the authors of the two books referred to, and then only when work on

S is part of their current research. Viewed externally then, S and similar statistical systems are small software projects. Indeed, as we shall see, the testing procedures used are more automated and require less time to run than would (perhaps) be reasonable to expect with millions of lines of code.

On the other hand, the alternate definition given by Marks is "A small software system provides a single service ...". The implication is that such systems are easy to test because of their narrow functionality. Is this true of S or other statistical systems? If they do, in some definition, provide a "single service", that service is capable of unboundedly different use, as noted in the previous section. The code and the programming staff for an order-processing system might be orders of magnitude larger, but the range of applications is more restricted.

These two different views of size — the internal size versus the range of use — have implications for what is easy and difficult in testing a system like S. On the one hand, the testing process can take advantage of the order-of-magnitude smaller source code, to use testing more flexibly and to integrate it closely with system design and modification. Such techniques, as described in the following sections, may or may not have useful analogues for the design of conventionally big systems. On the other hand, an important caution remains: we test the facilities of the system only over some range of S expressions. This describes very imperfectly what *users* will do with the system and any extrapolations of the test results to how the system will behave for users must be made cautiously. To support such extrapolations at all, the testing must include parametric studies and sufficient analysis to give a quantitative summary of performance. Much important research needs to be done to guide performance testing of statistical software.

Sections 3 and 4 outline some of the testing techniques used for S and discuss some examples. A number of the techniques and most of the examples were created during a period of several months of intensive testing preceding the 1992 release of the system. Section 3 discusses the testing methodology. Included are tools within the language to assist in performing automated tests, to generate special-purpose test output, to evaluate test results, and to extend testing to the issues of performance and portability described earlier. Section 4 gives a brief narrative of the testing period prior to the 1992 release and includes some examples illustrating points of interest.

2.4 The Role of Data Analysis

This paper is mainly concerned with testing software for data analysis. Before getting on with that topic, here are a few general comments on the relationship in the other direction: the role of data analysis in testing any software. Good testing generates data, sometimes in large quantities. Success in analyzing the data sensibly and quickly will make the testing process work better, and as a result benefit the software. In many respects, the process is the same whether the subject for study is a software system (for whatever purpose) or some other equally complicated application process. Three main steps are involved.

- *data preparation*: everything needed to generate and verify datasets, and put them into a useful form.
- *data presentation*: the summary of the data, as tables, graphs or whatever format lets us look at what is happening.
- *data analysis*: (in the more restrictive sense) the taking apart of the data to uncover specific patterns, model behavior quantitatively, or test for particular effects.

The examples below will show how such steps arise in our software testing. Some general comments can be made.

The order in which the steps are presented above is also the order of importance for many applications, software testing data included. Unless data is collected, organized usably, and verified to be what was supposed to have been measured, anything done with the data will be compromised if not totally useless. Once the data are prepared, good presentation will often make detailed analysis unnecessary. For example, if one is asking whether a proposed software change gives a performance improvement, graphical displays often give a clear answer.

Further analysis might quantify the answers, and such quantification may be important in some cases. For a general-purpose system such as S, however, such quantification needs to be approached cautiously. The tests, however well designed, are shaky predictors of the pattern of future use, still less a basis for some "utility" model of the system's performance. The point of the test data is usually not an explicit prediction but rather an attempt to test a variety of vaguely typical computations, in the hope that all or nearly all will give a consistent answer to the questions of interest. Quantitative models *can* be essential when one has more quantitative questions; for example, to choose default values for some system parameters.

In summary, the activities of data analysis in its broadest sense are crucial: collecting useful data that is properly prepared and organized; presenting quick, clear views of the relevant aspects of the data; analyzing the data to provide quantitative answers to quantitative questions. That conventional statistical theory is not the most important tool in most applications should challenge statisticians but not discourage them. One final comment: it will be clear in the rest of the paper that S was intimately involved in analyzing its own tests. Without a powerful set of data analysis tools and plenty of computing power to support them, the period of intensive testing of S would have been much less productive.

3 Tools and Techniques.

Software testing in S uses shell commands, S functions, audit files, extensions to the underlying C code, and special programs. Taken together, these are designed to help at several stages.

- When S is installed, the administrator is encouraged to run a series of tests to verify the integrity of the system.

- The same tests are used to check portability, either when S has changed or when a new machine or operating system is being added.

- The tests and even more the individual tools are essential during the evolution of the system, to test the integrity of proposed modifications and to compare the performance of alternative computations. This applies to changes in either S functions or underlying low-level code in C or Fortran.

The testing facility is designed to carry out extensive integrity tests of the system automatically, with little human intervention afterwards (for example, the tests try to be silent so long as no problems are found). At the same time, the design allows new tests to be added easily by developers or by moderately advanced users. The testing tools can also be used to study performance, once integrity is verified. Such testing is an important part of system modification and tuning; many of the examples below arose in the course of performance tuning for the 1992 release of S.

The installation tests of S are black-box tests, in the sense that they are applied to the system without customizing it. Other important testing tools modify S itself to generate special-purpose data either to track down bugs or to measure proposed changes. Such techniques require ways to alter the system and suitable places to put the data for later analysis. S has several possible approaches to both.

3.1 Test Scripts and Functions

Integrity testing of S (testing that the system behaves as expected after installation or modification) depends on three main sets of tools:

1. test scripts, in the form of files containing S expressions;
2. S functions, some to be used in the scripts and some to evaluate the scripts;
3. shell commands to run the tests, in various ways, and to generate relevant output from them.

The base for integrity testing is a set of files containing scripts of S expressions. The key testing convention in writing the scripts is that each expression in each file should evaluate to a logical `TRUE`. The shell commands and related S functions are designed to evaluate the expressions silently in this case. The shell commands are all used in much the same way, but run the tests under different conditions and produce different results, depending on the purpose of the testing. All the tools are kept in the administrative directory of S, `adm/cmd` under the S home directory. The notation `$A/what` refers to file `what` in that directory.

The standard and most commonly used testing command is:

```
$A/TEST file
```

This runs the expressions in `file`, and reports on standard output the name of the test and some memory and timing information. Any other output indicates a discrepancy from the asserted behavior. An example of nominal behavior would be:

```
$ $A/TEST tests/lm
------- Test file: tests/lm ---------
Tests completed; Memory, user, sys, clock time= 3024940 31.79 10.18 58
```

There are a variety of options to `$A/TEST`, some of which will be discussed when we talk about issues of error testing and special data generation. If no file argument is given, all the test scripts in the local subdirectory `tests` will be run. (This is the standard mode for testing a newly compiled version of S.) Invoking `$A/TEST` generates a call to the S function `do.test()` for each file. Given the name of the file as an argument, this function parses the contents of the file and evaluates each resulting expression. If the value of the expression is not `TRUE`, `do.test()` prints both the expression and its value. The effect is then to evaluate all the expressions on the file in a single top-level S expression. The memory and times reported are the values returned by the S functions `memory.size()` and `proc.time()`.

Two alternative commands are `$A/BTEST` and `$A/PTEST`, for batch-mode and performance-mode testing. In batch-mode testing, each expression in the file is evaluated as a separate top-level S expression. Another function, `btest()`, operates like `do.test()` in reporting an expression and its value if that value is not `TRUE`, but operates on a single expression, not a source file. Batch-mode testing may be preferable to standard testing for two reasons. Because each expression is done separately, the demands on memory will be substantially less. Machines with limited memory may be able to run some of the test scripts through `$A/BTEST` but not through `$A/TEST`. Also, some aspects of S can only be tested in separate expressions. For example, S data management includes a strategy for commitment and backout (Chambers, 1992) that operates only at the close of a top-level expression.

Performance-mode testing is designed to produce detailed memory and timing results. Running `$A/TEST` gives the final memory size and the total user, system, and elapsed time of the process. These are useful to compare changes internal to S that tend to affect many different computations. Using `$A/PTEST` instead gives somewhat analogous results on a per-expression rather than a per-file basis. The results of the tests in this case are converted directly to S objects, specifically to a three-way array whose dimensions index repetitions of each expression, the memory and timing variables, and the expressions within a file.

Files of test scripts are provided with S to test the core functions and the statistics library. These scripts have evolved with S itself and will continue to do so, supplying regression tests for future changes. The testing methodology as it has recently developed is intended to be used with these scripts but also more generally, as part of any large-scale software project using S. One lesson from the development of large systems is that testing needs to be integrated into the overall evolution of the system. In the case of S, the strategy is to provide a battery of tools that can be applied easily in a variety of ways. The test scripts, the S functions they use, and the shell commands to run them form the core. A number of issues arise in applying the tools, and these need to be considered next.

3.2 Integrity Testing

Turning from the mechanism for integrity testing to the tests themselves, we need to consider first the testing of numerical results. Traditional tests of numerical software have used indirect assertions, such as that the sum of residuals from a least-squares fit is zero or that the product of a matrix and its inverse should be the identity matrix. A limitation of such tests, particularly when portability of the software is claimed, is that the assertions can only be approximately true. Differences can arise from a variety of causes, but one which is of special concern to statistical software is a difference in the numeric computations on the two machines, whether from hardware floating-point differences or software incompatibility. A more fundamental limitation from our view is that such tests are necessary conditions for verifying integrity, but far from sufficient. As the computations get more sophisticated, the indirect assertions typically are harder to find and less useful as a thorough test of the software.

In the current test suites for S, we have chosen to verify integrity directly as well as indirectly. Most of the tests for the statistics library take a *target* object representing the result of some computation, compute a *current* object for the same computation, and test whether the two objects are entirely equal, up to expected numerical differences. This approach is particularly natural for S, where the result of a statistical computation is nearly always represented as a single object. (Section 3.3 discusses some exceptions.)

The key advantages of this approach are that it applies broadly to all computations and that it tests thoroughly. A disadvantage is that the target objects must be computed and shipped with the software to allow testing, adding substantially to the size of the shipped system. Another disadvantage is that changes to the software are likely to change some details in the target objects. This entails a potentially laborious checking of the new objects to verify that the differences are as expected and not bugs in the new software. A third disadvantage, or at least a serious challenge, is to figure out *how* to test general equality of the target and current object. Notice that the target object is not the "correct" result, whatever that would mean, but an object that we accept as the result of the intended computation, carried out on the originating computer system. We need to account for rounding errors or other limitations in the target computations as well as in the current computations.

The crucial tool in integrity testing for S applies the class/method paradigm in the language to the testing process. Integrity tests are largely built around one function:

```
all.equal(target, current)
```

returns `TRUE` if the two arguments are judged effectively equal. Otherwise it returns a character vector describing the differences detected. The function is designed to be used by the testing tools of the previous subsection. Many lines of the testing scripts are simply calls to `all.equal()`. Essential flexibility in testing comes because `all.equal()` is a *generic* function. Methods for `all.equal()` can be written as needed to test various classes of S objects (Chambers and Hastie, 1992, Appendix A). As S develops and expands, these methods can be kept up to date by those who understand the individual classes of objects. The testing procedures themselves will not need to grow in complexity by absorbing such information.

It might appear initially that methods for `all.equal()` would be rather trivial. This is far from the case, as a few examples will show. The general design is to write methods for some basic components of results; for example, `all.equal.numeric()` tests equality of numeric vectors. The default method for `all.equal()` for non-atomic objects such as fitted models calls `all.equal()` recursively for the components and attributes of the objects.

Consider what `all.equal.numeric()` should do. The comparison needs to consider the the two vectors may come from computations that are exactly identical, either because they the same computations were done on different machines, or because two expressions used slightly different, though theoretically equivalent, methods. Numeric computations on the machine will at best be accurate to some small multiple of the relative precision of the numeric representation used. This information is available in S as the special object `.Machine`, whose components give the hardware characteristics of the current environment. The characteristics are computed automatically when S is installed by an extension of the method of Cody (1988). Because the computations involved in statistical computations may be rather extensive and because we want to avoid too many false positives, the testing tools use a rather permissive tolerance, $\sqrt{\epsilon}$, where ϵ is the larger of the relative precisions for the target and current computations. Given a chosen tolerance, two vectors are compared in terms of their relative difference,

```
mean(abs(current - target))/scale
```

where `scale` is the scale of the target value. Absolute (L_1) norms are used rather than squared (L_2) norms to avoid undue influence from single large values.

In section 4.2, we will take such considerations considerably further in comment on suitable methods for testing equality of model objects. The key points of general methodology are two:

1. The essential technical task is to design methods based on knowledge of how a particular class of objects was computed and what the objects represent. Without the class/method organization of the software, this aspect of testing would be very messy indeed.

2. Even so, our experience is that the goal of fully black-box testing is unlikely to be attained, for a system as varied and as evolutionary as S, when ported over a wide range of computer systems.

3.3 Errors, Output and Interaction

The integrity testing described so far amounts to generating objects in the system and comparing them to known or precomputed target objects. In a system like S, such tests cover a large range of the system's design goals. Everything in S is an object, including functions, user expressions, and the results of all analyses. Testing and especially the automation of testing are made easier by this property. Nevertheless, some activities are not so obvious to capture this way:

1. Error behavior: this includes verifying that known errors are detected and that error handling behaves as asserted. Signals, such as floating-point exceptions, need to be tested similarly.

2. Input and output: this includes reading data from files, printing to the display or to a file, plus more specialized operations of which the major example is output from a graphics device, such as `postscript()`.

3. User interaction and control: this includes specialized input (e.g., menu selection or graphic input), plus facilities that allow the user to control the system, such as facilities for tracing and debugging..

The rest of this section describes briefly some issues in testing each of these, with examples of techniques currently used for S.

Error Behavior

Automatic testing of error behavior in S is based on an evaluation model defined in S and accessible to ordinary S functions (Becker, Chambers and Wilks, 1988, Chapter 11). Users can specify error actions, exit actions for individual functions, interrupt actions, and a restart switch so that the evaluator re-calls a particular function rather than terminating the top-level expression after an error. The last is used to automate error checking in the integrity tests. An S function, `error.check(expression)` arranges to return `TRUE` if evaluating `expression` produces an error (which the function catches), and `FALSE` otherwise. Calls to `error.check()` then can be part of the scripts used in automatic testing.

Error detection is now verified; it remains to verify error actions. An error in S results in a message written to standard error and an action, usually to save a trace or dump as an S object. The action is set by a controllable option. To verify that the action was called, `error.check()` sets the option so that the action assigns in a known place a randomized value chosen by `error.check()` in each call. A simple and secure verification is then made that the error option was effective and that the action was carried out. The shell commands for automatic testing redirect standard error (including the error messages) to a file. This file is compared (character-by-character) to a target version shipped with S. Provided that the error behavior is as expected and that the content of the error messages remains the same on all machines, the error checking is entirely silent. Without silent error-checking, too much human intervention will be required to bear extensive verification of the system's error handling; fortunately, portability of error messages has not so far raised some of the problems encountered in other areas (see the comments below).

Output Checking

Aside from the standard assignment of objects, output to the operating system mainly involves three activities: printing for the user to look at, graphic devices that write character output to a file or process, and dumps of S objects to text files. The third of these is used for transporting S itself between machines and gets very heavily tested. Special tests are provided for the other two. The relevant expressions are run in a supposedly correct version, with the output captured on files. The files become the target version, shipped with S. The test scripts include comparison between the contents of current output and target files.

The limitation of this method, of course, is that the comparison is done character by character. Any machine dependency in the output will generate an exception. The human tester then needs to study the reported differences to judge whether they are likely to represent actual problems. Our experience with these tests has been mixed. Most of the printing tests remain silent over a wide range of machines. The tests of graphics devices have been more likely to show up small differences, typically slight differences in plotting co-ordinates. It would be possible to develop tools to compare the output with some tolerances for numerical differences, but the prospect is daunting. As the discussion in section 3.2 suggested, even for S objects the definition of the appropriate methods is challenging. The graphics device output is more loosely structured and devising suitable tools to decide on approximate equality has appeared to us to be more work than it would be worth.

User Interaction

Automatic testing of computations involving user interaction (menu selection or interactive browsing, for example) requires special techniques. Since automatic testing is by definition done *without* human interaction, the input must be presented in canned from, from script files. Testing tools must be able to put the input in the right place, relative to the expressions that need it; some of our shell commands for testing are adapted to this, but not all. There must also be controls in the system to simulate interaction. For example, S provides options to control whether the system thinks it is running interactively and to specify prompt strings for user input. (The latter are made empty strings to keep the system silent when it would normally be prompting for expressions from the user.)

Assuming the technical details can be handled, automatic testing of interaction still requires prepared scripts that include expressions and input. Keeping such scripts in good order and expanding them to include all the important, testable interactive activities in the system requires extensive, picky work from the testers. If testing of interaction were a key to the system, the designers would do well to write a monitor that could update scripts, including the interactive input, by monitoring human interaction.

3.4 Performance Testing

Performance testing measures the resources needed to carry out tasks in the system. Processor time used and memory occupied are the aspects of the computation that can be readily recorded. The underlying goal is, or should be, to measure how well the system can respond to heavy use. For systems with a fairly narrow range of task definition, such as an order-processing system, there is some hope of modeling this directly, by subjecting the system to an increasingly heavy input of artificial but assertedly realistic jobs. As noted earlier, conclusions drawn from a fixed suite of problems must be treated very cautiously in extrapolation to the unlimited variety of user programs in S. Any performance testing must be recognized as a non-random sampling from an unknown pattern of future use. All the same, performance testing is important and useful, especially in the design and tuning of the system. Two approaches have proven valuable:

- Benchmark tests under varying conditions. A fixed set of tests are applied under alternatives to compare versions of the system, design alternatives, or values of system parameters.

- Parametric tests. Some specific computations are run varying one or more parameters, usually defining the size of some input objects, to study the dependence of performance on problem size.

The statistics library regression tests have been used in S as benchmarks. They deliberately include some applications of model-fitting and related computations to large datasets. The time and memory requirements for these have proved to depend substantially on implementation details. Much of the tuning of S during this period has used this set of benchmarks to compare alternative approaches. As shown above, the `$A/TEST` shell command reports memory required and total time used at the end of the test run. Running each of the ten statistics tests gives ten sets of memory and time figures for each set of conditions. The numbers are in no sense independent, identical replications. However, the different tests span a fairly wide range of computing styles and their behavior has turned out to be usefully different. For a particular comparison, some of the tests will usually prove to be more sensitive than others.

The `$A/PTEST` shell command generates more detailed information on the timing and memory used by individual expressions rather than the total over the whole test script. For finer tuning this may be essential, at least for time measurements. So far, our experience is that during re-design gross measurements are more useful. By definition, the designers are looking for substantial improvements that will show up in the large as well as the small. Measuring memory usage in the small is difficult even in principle, for a system that handles all objects dynamically, as S does. The pattern of memory allocation depends on the whole history of the session. Fragmentation from re-allocation and from the need to allocate some permanent space (e.g., for database dictionaries) means that the actual memory pattern before and after carrying out a computation is not simply a function of that computation. So far, gross measurements of memory use have been more useful than detailed data in assessing design alternatives.

Parametric tests measuring performance against some size variable offer quantitative information. Intuition often suggests that performance should have a fairly simple relation to problem size. For example, an iterative computation might be expected to increase in memory and time requirements linearly with the number of iterations, other things being equal. If the performance relates to problem size in a reasonably smooth way, we will have some grounds for predicting behavior under actual use. (See Section 4.4 for an example.)

Three important cautions have to be kept in mind. First, any simple predictions are only valid for a limited range of inputs. Extrapolation to arbitrarily large computations will eventually break down for a variety of reasons; for example, large memory requirements will eventually cause dramatically super-linear time consumption as page-swapping overhead comes to dominate all other computation. Second, the chosen set of parametric studies has to be an even more biased representation of actual system use than the benchmark studies, because many computations cannot be re-sized arbitrarily in a meaningful way. Any conclusions from such studies need to be quickly tested more broadly before committing system design to choices based on the conclusions. Third, quantitative conclusions from the parametric studies are only valid for a particular computer environment. They do change, sometimes radically, when the study is moved to a different combination of processing hardware, memory resources, and operating system.

A single run of a parametric study can also be treated as a benchmark test, to be applied under different conditions. The overall pattern may help display the performance improvements in at least a semi-quantitative way. Where a parametric curve is an adequate summary of the performance behavior, comparison of the fitted curve under the different conditions can give a quantitative benchmark applicable to a range of computations.

3.5 Generating Special Test Data

The tests described so far are "black-box" tests in software-testing terminology. Integrity and performance are measured in response to requests, using no explicit knowledge of system internals. For large systems, black-box tests are often recommended (e.g., Marks, 1992, page 36) on the grounds that the system is so extensive and complicated that any attempt to test based on understanding how it works will vastly increase testing time. However, such tests may be too noisy to provide sufficient guidance in design or re-design of the system. The designer may have to make some untested assumptions in order to link the result of a black-box test to a particular design change being considered. A better approach in such cases is to extract some data directly related to the change.

A lesson learned in seeking quality in manufacturing applies to software as well: *It is crucially important to generate relevant data and to analyze it simply.* The habit of asking questions, generating data, and looking at the results is essential to improving the quality of software. This habit, in turn, needs to be supported by tools for generating data that are easy and general, as opposed to complicated but relatively narrow monitoring schemes. For example, S has an audit file on which a record is kept of user expressions and database activity (Becker and Chambers, 1988). The audit file was not intended for testing studies, but it provides a convenient place to generate special test data. The special data will automatically be interleaved with the record of expressions being evaluated, so that it can be related to the test computations. One simple tool is sufficient—a C function that sends output to the audit file. As an example, at one point in the re-design of S several ideas for improving memory compacting in loops were being considered. Black-box testing was too crude to answer questions such as the overhead involved in compacting or the percent of memory recovered. The approach taken was to record on the audit file some relevant data by a temporary modification of the system. About 10 lines of C code were added, typically looking like:

```
MESSAGE "start_compact %g %ld %ld\n",
 proc_time(), Nframe, nbytes
AUDIT;
```

The `MESSAGE` and `AUDIT` macros are S tools; the rest of the statement will be recognized by C programmers as arguments to functions to `printf` to produce output. In this case, we print the accumulated process time, the

S evaluation frame and the storage in bytes. The total modification was completed in half an hour, used to generate relevant data, and then thrown away. The data generated was edited into a file suitable for reading into S. Simple analysis answered the crucial questions, in this case confirming that compacting was relatively cheap and that substantial memory could be saved by an adaptive strategy to compact more frequently based on the fraction of space recovered.

Simplicity is the key. If the designer needs to generate some special data, having to learn a complicated testing mechanism will be discouraging. Anticipating the need to generate data is essential; trying to pre-define in detail the form or content of the data is usually counter-productive.

3.6 Other Tools

The use of the audit file above illustrates the importance of tools that might not have been initially designed for testing, but that fit in well with the testing scenario. Another such tool for S is a visual monitor (Becker, Chambers and Wilks, 1993). This is a system that generates data on memory allocation during an S session. The data may be written to a file (let's call it the *memory* file) which is later read by a display program; alternately or in addition, the memory information can be piped directly to the display program. The display shows each allocated block of memory, its position and the function call or permanent data frame it belongs to shown by the horizontal and vertical placement on the display. Data is seen dynamically allocated and released, and user interaction can be applied to move the display forward or backward through the session.

Diagnosis of S was a major motivation for the visual monitor, but the monitor is not part of the routine testing process for several reasons. The memory file is very large. Even a moderate-sized test of S can generate tens of megabytes of data. At the same time, the communication of so much data to the display program slows down S and the display lags considerably behind the time taken to evaluate the S expressions — the monitor cannot be used in real-time. Even if it could, the displayed pattern for a complicated S expression is itself complicated. Sorting it out visually remains challenging (in defense of the monitor, however, sorting out so much data by any technique *other* than dynamic graphics is likely to be harder still).

Nevertheless, the video monitor has proved useful in debugging and testing in at least two situations. The memory file can be used to track down very specific questions about allocation, since it records essentially all the allocation information. The technique mentioned on page 20 to write to the audit file has an exact counterpart for writing special messages to the memory file. We have used this to mark special situations (for example, stages in storage compaction), allowing us to find problems such as memory leaks or invalid freeing of space during the special situation. Bugs of this sort are very difficult to track down by conventional debuggers. On a few other occasions, the monitor has been useful directly to detect and pin down memory allocation problems at a fundamental level, where the whole point is to study the behavior of S on simple expressions. An example is mentioned in section 4.3.

A second example of general tools that can be adapted to testing are *profiling compilers*, which provide non-selective data generation, usually more difficult to mold to specific questions but sometimes useful. A profiling compiler generally produces output that measures the time spent in individual routines and/or portions of routines. The measurement process is not trivial nor entirely precise; the profiler we have used produces counts of pseudo-instructions as an indirect but precisely measurable surrogate for processor time. With a large system, application of a profiler generates a mountain of data. Shell tools and editors can process the data if relevant questions can be formulated. Sometimes, individual routines can be found that use up a large fraction of the time, or at any rate much more time than seems reasonable. For example, profiling all the statistics benchmark tests at one stage in our re-design showed that dynamic storage management and specifically copying of character string data took a substantial fraction of the time in some of the tests. This led to a study of the relevant code. By moving data rather than copying when possible, by reducing the duplication of labels, and by other internal changes string copying was made substantially more efficient. Performance testing identified this as an area of the code worth the effort of tuning.

In other cases, attention may already be focused on a small, key portion of the system. Comparative timings from the alternative approaches can then be filtered out of the mass of profiling data and analyzed fairly easily.

4 Examples.

The examples to be considered come from a period of re-design between the 1991 and 1992 releases of S. The 1991 release represented a substantial extension of the system, including a general approach to statistical models.

The number of functions in the system was nearly doubled, with many of the new functions being heavy computations. At the same time, a class/method mechanism was introduced that altered the programming style of the language fairly radically. All this meant that the system was a moving target for testing. Not much time was left over for testing, either, beyond the essentials of verifying the integrity of the system.

After the 1991 release, a period of serious evaluation was in order, while holding the basic user view of the system constant. In particular, the 1991 extensions to the system often involved more extensive computations, with large demands on time and memory. A goal of the work during this period was to study possible internal re-design to improve performance.

The following examples are a very small selection of activities during this period. Each was chosen to illustrate a moral or an issue.

4.1 An Algorithmic Example

This example begins with a report from one of our users that the `sort()` function in S seemed to be failing on certain datasets. Specifically, for some datasets the function "appears to go into an infinite loop". The datasets in question were fairly large (tens of thousands of values). All the same, the behavior was puzzling. S used an adaptation of the `qsort` subroutine in the UNIX operating system, which in turn is a version of the *quicksort* algorithm, an extremely longstanding, well-regarded method, believed to sort n values in order $n \log n$ comparisons. What was more, we could sort equally long vectors ourselves in a perfectly reasonable time.

The first step in solving the mystery was to get a copy of the offending data. When plotted, it looked something like Figure 2 The data showed a saw-tooth pattern, with long stretches already sorted or nearly so. The `qsort` algorithm is known to behave well on randomly ordered data and on data that is perfectly pre-sorted. Was the problem the saw-tooth pattern, occasional departures from it, or something else entirely?

At this point, a few minutes reflection on the algorithm and the local C subroutine implementing it suggested a tentative hypothesis. The algorithm works by choosing a single element, called the fence. Each value is compared to the fence and the dataset is then divided into those values that fall on either side of the fence. The two subsets are then sorted by applying the algorithm recursively to each. A crucial point in the algorithm is the choice of the fence. The overall behavior will be of order $n \log n$ unless a very few values consistently fall on one side, with nearly all the values on the

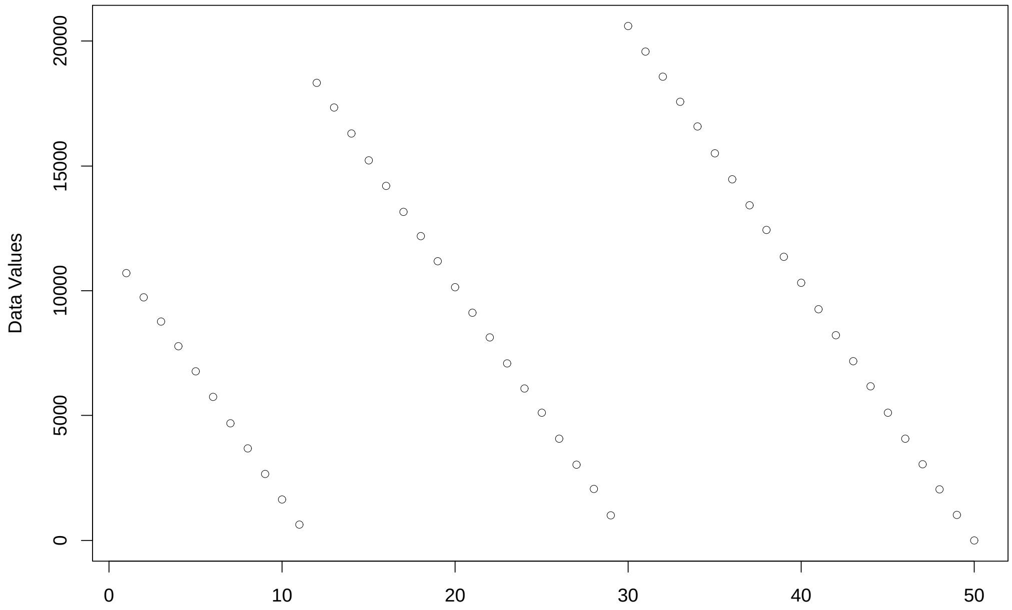

Figure 2: *A dataset showing a saw-tooth pattern that causes some versions of the* `qsort` *routine problems. The actual dataset had nearly 200,000 numbers.*

other. In that case, nearly n recursive levels can be needed, each requiring n comparisons. This gives n^2 behavior, and with $n \simeq 200,000$ that will be slow indeed!

The local implementation picked the middle value of the data as the fence. It seemed on reflection that this choice could do poorly for saw-tooth data. We quickly verified this by generating data of the form:

```
c(1:100000, 100000:1)
```

which behaved just as badly as the original sample. After a few more experiments of this sort we were convinced that this was the problem. For practical purposes, the solution was simple: if it happened that the simply-chosen fence gave a bad split, we just chose a new fence "at random" (using a random number generator) and went on. This involved only a few lines of change to the underlying C routine. The slow behavior on the original dataset and on our manufactured examples disappeared. Only a few hours of our time and a small change to the S code were needed to solve the problem.

This example leaves a nice warm glow. We spent a modest amount of time, found an interesting problem, and fixed it easily. One of our computer science colleagues, Jon Bentley, subsequently did some further analysis of similar issues in the `qsort` algorithm (Bentley, 1992). One might feel that this is what software testing *should* be like. However, general experience with large software systems suggests that this example is highly untypical in several respects.

- It turned out not to be a problem with the system as such, but an algorithmic issue.
- Application of "classical" computer science (what students are taught in courses) turned out to be just the thing to find and fix the problem.
- The problem was driven by a very special input situation. The unusual nature of the data was easy to see.

These characteristics endear such examples to computer scientists. There is the prospect for an elegant presentation of the problem with discussions of performance formulas and clearly comprehensible solutions. In practice, few of the serious problems to be solved in improving performance have much to do with basic algorithms or the contents of typical computer science courses.

Also, the third characteristic of the example is often a hint that the *overall* benefit to the system may not be large. The discussion began with a single, specific user problem as opposed to a more general study of system behavior. The particular user will of course be helped by solving the problem. When the fix is essentially cost-free for other users (as it was in this case), the system has been improved overall. Still, just how much the improvement will matter is undefined. That the problem only arose after very extensive use of the `qsort` subroutine in and out of S is evidence against the problem being very common.

Two characteristics of the example *are* typical:

1. The ability to analyze relevant data describing the example is crucial, especially the easy generation of relevant graphical displays.
2. Studying the performance of software is itself a computing-intensive activity. Its practitioners need lots of computer power themselves.

The overall goals are to improve the software system, to do so quickly, and to make effective use of human effort in doing so. Data analysis and computing power are important resources.

4.2 Testing Models for Equality.

At the end of the discussion of methods for testing object equality, on page 15, we had defined a test for numerical equality, based on tolerance and scale parameters, and had assumed that general comparison of more structured objects would be carried out by applying the equality tests recursively. This is not sufficiently flexible for many situations where the computations can produce more complicated patterns of departure from strict equality. In particular, more specialized methods are needed for statistical computations involving additive representations of data,

```
y ~ fitted + residual
```

To begin with, it is possible that either the residuals or the fitted values are, in principle, zero. The computed values will be small but usually non-zero, in both the target and the current objects. The relative difference as measured by a standard use of `all.equal.numeric()` can be large if the computations were done on different machines. The scale computation in this case has to be done outside `all.equal.numeric()`. Only at a higher level is enough information known to choose the scale. The solution is to allow `scale` to be supplied as an argument, typically using the size of `y`.

Here the use of classes and methods helps greatly. The top-level objects for comparison are often members of special classes. For example, many of the fitted models with an additive representation as above inherit from the class of `lm` objects. A method for this class for the `all.equal()` function can establish a value for the scale and pass the value down. A simplified version of the method would be:

```
all.equal.lm <- function(target, current, ...,
    scale = mean(abs(target$fitted + target$residual)) {
    NextMethod("all.equal")
}
```

The `scale` argument defaults to the estimated scale of the response. Two advantages to working with classes and methods arise here. First, none of the test scripts need to know anything about the special scaling for `lm` objects. Invoking the generic `all.equal()` function gets the method automatically.

Second, the same strategy is inherited automatically by other models, such as generalized linear models and generalized additive models, that inherit from the class of linear models.

There is more, however, to comparing linear models. We need to compensate for an indeterminacy in the algorithm that is essentially harmless but can again give false error indications. The standard fitting algorithm used, an extension of that in Dongarra et al. (1979), creates an orthogonal decomposition from successive reflections. Each reflection has a choice of the $\pm$ direction. The algorithm makes a particular choice, designed to reduce cancellation error. Under certain circumstances, this choice amounts to choosing the sign of a number that is zero except for rounding error and therefore it can well differ between target and current. There is no significant effect on the model, but the signs of some effects will change and this can induce an erroneous report of differences. The `all.equal.lm()` method detects such sign mismatches and adjusts the objects accordingly.

The cause of the problem here was a discrete result (the sign of the effect) chosen as a function of a continuous value. Whenever this happens, we can expect problems in testing. A much less tractable example arises in testing tree-based models (Clark and Pregibon, 1992). The fitting algorithm here proceeds by splitting the observations recursively, according to a criterion based on one of the variables. Each split is chosen to give the best performance at the current stage. Then the two subsamples are examined the same way to choose the next split, and so on until no further splitting is possible.

The problem for testing is that two possible splits may be equally valid or nearly so. Unless all the variables are factors, the splits will depend on numeric computations; therefore, ties may be broken differently for the target and current computations. If so, the entire subsequent course of the two computations may be different. Since the splitting algorithm does not iterate to achieve any optimality criterion, one of the fits may be non-trivially better than the other, although both are valid applications of the algorithm. In this situation, the `all.equal()` method can only hope to detect the node on the tree where divergence occurs, remove the subtree below this node from the two trees, and try again.

This was the problem leading to Figure 1. The method, `all.equal.tree`, used to compare the current and target tree, found four nodes at which different splits occurred. These were the two subtrees shown in the figure, plus two terminal nodes, shown by triangles. Not all the differences produced alterations in the split: in some cases, two variables may give an identical split, so that the appearance of the tree remains the same. Other differences may produce different splits that are nearly or exactly equivalent, with the choice being flipped by small differences in numerical computations. In either case, one would prefer the `all.equal.tree` to remain silent if it judges that the two trees produced are "equally reasonable", or at least to report the differences in a gentler way. In the 1992 version of S, this sorting out is left to the human, using tools such as the `burl.tree()` function (Clark and Pregibon, 1992, p. 402), which compares alternative splits at a particular node. In the example of Figure 1, study convinced us that the differences were plausibly explained by different arithmetic on the two machines.

4.3 Memory Problems

During the tuning of S, we were concerned about memory growth in S expressions that involve large amounts of data, extensive iteration, or just a large amount of computation. These issues are related to problems of memory leaks; that is, allocation of dynamic memory that is not subsequently freed. Most memory growth problems are not leaks, strictly speaking, although we will give one example below. The most important problems treated were excessive intermediate memory growth and excessive computational overhead in dealing with expressions that needed a substantial amount of memory.

Intermediate growth of memory in S arises largely from the way it treats data objects. S functions deal with objects — they get objects as arguments and return objects as their value. Part of the functional language nature of S is that objects are passed around, not pointers to objects. The conceptual clarity of the language is helped by this view, and users benefit from not having to think in terms of pointers, addresses and so on. At the same time, S objects are dynamically allocated and of arbitrary size and structure. This also leads to a simpler model of computation.

For large computations, this approach to objects has produced some performance difficulties. There is a great deal of copying of objects, sometimes large objects, as S functions are called and return. This has always been the case, but as more complicated use of the language became common, and as new facilities were added, the inefficiencies became more obvious. From the users' viewpoint, the most obvious performance fix in the period under consideration was the reduction in memory and time required for substantial iterative calculations. The benchmark tests of the statistics library, though not all that extreme as examples, illustrate the results: memory requirements were reduced by as much as a factor of three, accompanied by a reduction of about 25% in computing time. Another outcome of this intensive study was at least as important. The study confirmed an intuition that further major improvements would not come from tuning the current implementation, but only from a fundamental redesign. This redesign began after the 1992 release and is being guided in part by the experience gained in extensive performance testing. We have also had the opportunity since this release to work with a software tool, Purify (Hastings and Joyce, 1992) that aims directly at finding memory leaks and access errors. We expect future use of such tools to be a substantial help in testing future versions of S.

The improvements came from many tests and much study of the code; as is typically the case, no single magic fix provided the answer. Essentially all the techniques described in section 3 were used, plus a number of others. The typical procedure was to identify a possible source of inefficiency. If a fix could be tried easily, it was. Test data would be generated before and after, prepared for analysis in S, and displayed. Usually the success or otherwise would be clear from tabular or graphical display. By the standards of large software systems, the whole procedure was minimally regimented. As noted, S is still small by the scale of such systems and being the property of a small research group, the designer, tester, and implementer for a particular "modification request" is often the same person.

From the many studies of memory-related fixes, one example might be singled out for discussion. At one stage, users of our local test version of S reported that memory seemed to expand gradually throughout the S session, more than would be expected. Some gross tests confirmed this. At first, the problem seemed to be associated with large-scale computations but closer study suggested that such computations just made the growth faster and therefore more obvious. Finally, we convinced ourselves that even fairly simple iterative calculations, if carried out long enough, caused memory leakage.

Software designers dread memory leak problems, particularly in systems as complicated as S. If the offending operations are embedded in a very large number of successful allocation/de-allocation operations, one has a needle in a very large haystack. In our case, this turned out to be a job handled well by the dynamic video monitor. Running a simple loop in S through the monitor showed us directly that a block of memory was indeed being allocated but never freed. Furthermore, once this was detected visually, the information on the memory file allowed us to isolate the block being left behind. Using the techniques mentioned on page 22, the problem was further narrowed down and after relatively few hours of debugging was traced to a flaw in re-organizing chains of allocated memory after compacting.

This example reinforces several lessons. Users who put the system under stress are essential; in this case, the problem only became obvious with heavy use. Flexible tools are important, especially if they can generate relevant data. Visual display of such data is very helpful, even if the display cannot be applied to all problems.

4.4 Performance of a Linear Model Computation

This example illustrates the use of parametric tests. Specifically, we wanted to examine the performance requirements for a linear model computation using the model-fitting software. Figure 3 shows the time and memory used

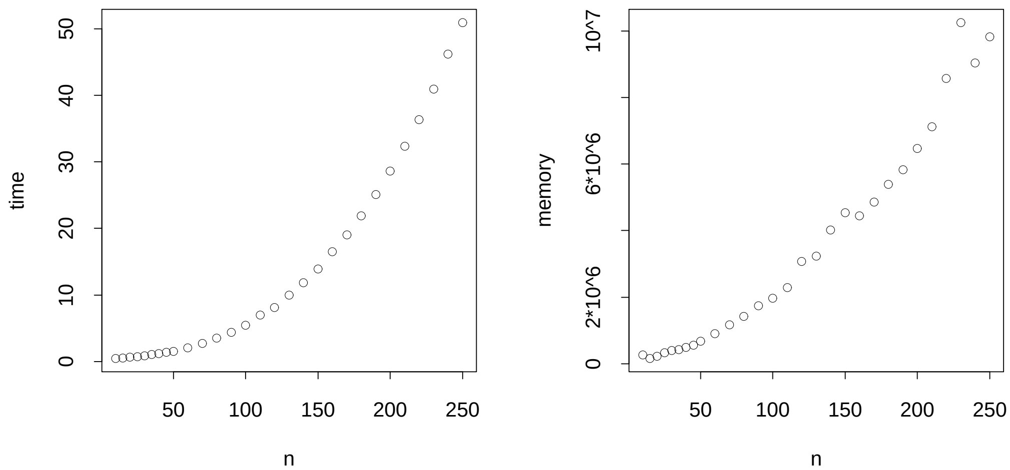

Figure 3: *Performance requirements of a parametrized linear model test. For each value of* `n`*, gaussian data with* `3*n` *observations of* `n+1` *variables were generated. A linear model with* `n` *predictors (*`y` *~.) was fitted. The left panel shows processor time, the right the increase in memory from beginning to end of the computation.*

in fitting a linear model as the number of observations and variables go up, proportionately. Some analysis of the data in Figure 3 suggests that the time goes up with n^3 for n large, consistent with the theory of linear models. The total space requirements should go up with n^2, which is also confirmed, though the measured memory use is more irregular than the processing time.

The data shown in Figure 3 look much like classical numerical-analytic studies of least-squares computations. The usefulness of the results in studying a system, however, comes from different applications than in studying an underlying numerical algorithm. We were more concerned to measure the overhead than the numerical computation, that is to look at the difference between the total computation requirements and those due to numerical calculations alone. A major goal during this period was to improve the efficiency of some of the computations contributing to the overhead. Data analysis of results such as those shown can give a rough but useful estimate of such overhead. Comparing results like those in Figure 3 to similar tests run on the 1991 release showed a substantial (and quantifiable) change in behavior. In a small way, such results allow us to predict performance improvements in other computations, but much more needs to be understood before such predictions are likely to be very successful.

5 General Reflections.

This paper has described some tools and techniques found useful in studying a general, interactive language and system. Much more research needs to be done on good ways to test integrity and performance in such systems, but some general lessons are clear. The examples in this paper and many others not reproduced here prompt some reflections on testing and improving complex software systems.

1. It's difficult: making substantial improvements in a large-scale system with diverse applications requires both skill and luck. Two characteristics of our situation in this respect contribute to the difficulty:

    (a) The input space is effectively infinite; that is, there is no way to supply a reasonably-sized set of test cases that will accurately predict performance in practice. For languages like S, the situation is even worse than that, as discussed on page 6. Not only is the space effectively infinite, it is effectively infinite-dimensioned since users can apply the language to arbitrary new projects, each of which can depend on any number of variable input parameters.

    (b) Tuning changes are likely to be intrinsically limited in how much improvement they can make. The most important result of testing may then be to suggest or re-enforce ideas for a more basic redesign.

2. Effective testing requires good data. The data must be easy to generate and relevant to the performance questions being considered.

3. Once collected, the data need to be analyzed, though often good graphical presentation techniques make more formal analysis unnecessary. It is *not* enough to process the data in pre-defined ways; one must be able to compute and display results relevant for the particular questions being asked, but not necessarily results that could have been anticipated in advance. The time taken to pose the questions and see the answers must be short, because many different questions will likely need to be asked.

4. Good data will be hard to generate unless the system design anticipates the need for data collection. It must be easy to turn on a data-generating mechanism that provides sufficient detail on the system's activity.

Acknowledgements

The work on testing S is part of the evolution of the system itself and much of it has been joint work with Richard Becker and Allan Wilks. Similarly, the evolution of testing for statistical model computations involved the ten authors in Chambers and Hastie (1992). Richard Becker, Daryl Pregibon, and Allan Wilks provided valuable comments and help with the current paper.

References

[1] R. A. Becker and J. M. Chambers (1988), Auditing of Data Analyses, *SIAM J. Stat. Sci. Comp.*, **9**, 747-760.

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

[3] R. A. Becker, J. M. Chambers, and A. R. Wilks (1993), Dynamic Graphics as a Diagnostic Monitor for S, submitted to *J. Comp. & Graph. Stat.*.

[4] J. Bentley (1992), The Trouble with qsort. *Unix Review*, vol. 10, no. 2 (February, 1992), 85-93.

[5] J. M. Chambers (1991), Data Management in S. *Stat. Res. Rep. No. 99*, AT&T Bell Laboratories, Murray Hill NJ.

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

[7] L. A. Clark and D. Pregibon (1992), Tree-Based Models, in Chambers and Hastie (1992), pages 377-420.

[8] W. J. Cody (1988), MACHAR: A subroutine to dynamically determine machine parameters, *TOMS*, vol. 14.

[9] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart (1979), LINPACK *Users' Guide* SIAM, Philadelphia.

[10] I. Francis (1973), *Statistical Software: A Comparitive Review*, North Holland, New York.

[11] R. Hastings and B. Joyce (1992), Purify: Fast Detection of Memory Leaks and Access Errors, *Proc. Winter Usenix Conf.*

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