Computing with Data: Concepts and Challenges

John Chambers
Bell Labs, Lucent Technologies

Abstract

This paper examines work in "computing with data"—in computing support for scientific and other activities to which statisticians can contribute. Relevant computing techniques, besides traditional statistical computing, include data management, visualization, interactive languages and user-interface design. The paper emphasizes the concepts underlying computing with data, with emphasis on how those concepts can help in practical work. We look at past, present, and future: some concepts as they arose in the past and as they have proved valuable in current software; applications in the present, with one example in particular, to illustrate the challenges these present; and new directions for future research, including one exciting joint project.

Contents

1 Introduction

Our use of computing, as statisticians or members of related professions, takes place in a broader context of activities, in which data is acquired, managed, and processed for any of a great variety of purposes: the term *computing with data* refers to these activities. Traditional statistical computing is an important part, but far from the whole.

This paper examines concepts, both in statistical software and in other areas, that can contribute to computing with data and thus to the activities that it supports, including science as well as industry, government and many others. This paper is based on the Neyman Lecture presented at the 1998 Joint Statistical Meetings, at the invitation of the Institute for Mathematical Statistics. The Neyman Lecture is intended to cover some aspect of the interface between statistics and science. Computing with data is an appropriate topic for the lecture, since the ability to define and execute the computations that we really want is often the limiting factor in applying statistical techniques today. The paper presents personal reflections on the topic, based on a long career in many related fields. I hope that users of current statistical software will find the concepts useful, and especially that the paper will contribute to discussion of how future software can improve its support of statistics research and applications.

The plan of the paper is to sandwich two general sections, on concepts and on challenges, between three sections looking at more specific examples in the past, the present, and the future. Some glimpses at the past will help motivate the concepts, an application in the present will help introduce the challenges, and possible directions for the future, including an exciting new joint project, will suggest responses to the challenges.

2 The Past

A coherent history of computing with data would require much more space and attention to diverse issues than we can spare here. As an alternative, we will take brief glimpses, first of some relevant general programming languages in 1963, and then of two activities in statistical computing that took place in 1965 and 1967. The choice, particularly in the case of statistical computing, is admittedly personal, with the excuses that it provides some history that has not previously been published and that the specifics will lead us to useful general concepts.

2.1 Programming Languages in 1963

What programming languages might catch the attention of someone interested in computing with data around 1963? In that year I was a beginning graduate student in statistics at Harvard. Those of us interested in computing occasionally wandered down the river to MIT, sometimes to hear about new developments, sometimes just to play "space war", perhaps the world's first computer video game. Here are three documents we might have seen (at any rate, all of them found their way into my collection soon after):

- an IBM manual for the Fortran II language;
- the MIT press publication of the *Lisp* manual (McCarthy 1962);

- the book *A Programming Language* by Kenneth Iverson (1962);

Fortran would have been visible around Harvard; to learn about the other two at that time, one would likely have needed to be someplace such as MIT more in the thick of computing. Each of the three introduced some concepts worth noting; together, they give a broad though not complete idea of the background for thinking about computing for statistics during that period.

Fortran (*FOR*mula *TRAN*slator), introduced in 1958, brought the notion that formulas, the symbolism by which scientists and mathematicians described their ideas, could be translated into computer instructions that produced analogous results. The formulas were the standard "scientific notation" for arithmetic and comparisons (slightly altered to accommodate contemporary character set limitations). Also, and perhaps more importantly, it used the "functional" notation that called for evaluating a function by writing down its name followed by a list of its arguments:

```
min(x, y, 100)
```

The concept of formulas and function calls to communicate computations may seem obvious, even trivial. But when I encountered Fortran first as an undergraduate in 1961, the concept was revolutionary for someone who had learned computing via machine language and wiring boards. An important step had been taken in seeing computing as a central tool for science.

The Lisp language, like Fortran, remains an active part of the computing scene today. Indeed, while Fortran is far from dead, its relative importance has diminished over the last twenty years or so. Lisp, on the other hand, started out as a specialized and rather academic language for manipulating list structures. Since that time it has vastly expanded its applications, to support software such as the *emacs* system for editing and programming, and the Lisp-Stat statistical system (Tierney 1990). Two concepts closely associated with the development of Lisp are particularly relevant. First, Lisp emphasized the ability to define serious computations from a simple, clearly defined starting point. The *recursive* definition of structures and computations was the key mechanism. Second, Lisp, especially in its later versions, developed the concept of an *evaluation model* for the language. That is, the language not only allowed users to specify what computations were to be done, it gave them a definition of the meaning of those computations, based on a model for how the evaluator operated (the λ calculus, taken from early theoretical work on computing).

Most readers will have heard of both Fortran and Lisp. Rather fewer perhaps will recognize the title of Iverson's book. The book did not, in fact, describe an existing programming language; rather it presented a system of notation to describe certain computations. Soon after, however, it was decided to implement the notation as an actual language. To name the language, for want of any other ideas perhaps (I can testify that naming programming languages is a challenge), the authors just took the initials of the book's title: *APL*.

APL, like Fortran, was a language using a version of scientific formula notation to express computations. In APL, in fact, this was the *only* way to express computations. All computations were built up from what would now be called unary and binary infix operators, parentheses and square brackets. APL had many operators, using non-alphanumeric characters, greek letters, and special symbols. For example, the question mark indicated random operations:

```
4 ? 4
```

evaluates to a random permutation of the numbers 1,2,3,4. Typing the expressions required a special terminal interface (originally a special type ball on an IBM electric typewriter). APL developed an intensely loyal user group, for statistical as well as other applications; the book by Anscombe (1981) presented a detailed approach to statistical programming in APL. For non-believers, the expressions could be of daunting obscurity; for example,

```
M ← ( ι 4 ) ϕ 4 4 ρ 'ABCD'[ 4 ? 4 ]
```

This assigns to `M` a random symbolic 4 by 4 Latin Square (Anscombe 1981, pp 44–47). Devoted use of APL continued for many years, but its reputation for obscurity and inefficiency discouraged wider adaptation. In retrospect, however, the language was a bold and important contribution, if admittedly as idiosyncratic as its author.

Two additional concepts are introduced by APL, the first being an interactive user interface: users typed formulas, APL evaluated them and, by default, printed the result. The user becomes part of the computing model. Interactive interfaces existed contemporaneously in a few specialized systems, but APL brought the concept that a general programming language could also be an interactive, user-friendly system.

A second new concept appeared in the treatment of the data, the operands appearing in the formulas. The data corresponding to operands could be scalars, vectors, or multiway arrays. But they were, in fact, *dynamic, self-describing objects*, to use modern terminology. Assignment operations created objects in the working area by storing the results of a computation. The user was not responsible for figuring out the dimensions of an array result; instead, APL stored the structure information with the object, and provided operators to let the user/programmer retrieve this information.

2.2 Statistical Computing: Bell Labs, 1965

Over the next few years, a number of projects were begun in providing computing for statistical applications. In this section and section 2.3 we will look at two somewhat unusual efforts, a design developed at Bell Labs for a statistical system and a working group in England looking at possible steps to encourage the use of computers in statistics. Neither of these resulted in a specific software package, but both involved thinking hard about the computing needs of statistics. Concepts and goals were laid out that remain relevant today, although the overall computing environment has of course changed radically. The first effort and some aspects of the second have never been described in print before, and both have some interesting anecdotes.

During 1965, a group at Bell Labs discussed a possible system to support data analysis and statistical research. There were already a number of statistical systems and programs by this time. The BMD and P-Stat programs were in use, and unknown to us at the time, John Nelder and Graham Wilkinson were then developing the first version of GenStat (Dixon 1964, Buhler 1965, Nelder 1966). The Bell Labs situation was, then as now, characterized by the combination of some large-scale, real problems with the freedom to deal in long-term research that was itself often stimulated by the applications. The question "What should the computer do for us?" then needed to be answered for both purposes.[1].

During a series of meetings and by writing design documents and a little prototype software, we designed a statistical computing system based on an extension of the PL-1 language. The system was never implemented, for several reasons, mostly *not* related to the feasibility or merits of the proposed statistical system itself. The system had been predicated on a general operating system, Multics, which itself was a bold joint venture, but which Bell Labs dropped because of delays (it went on to be created largely at MIT and to hold on to a small but faithful user community for about twenty years). The concepts behind the statistical system, however, and even more the motivation for the concepts, are worth reviewing. The discussions included John Tukey, Martin Wilk, Ram Gnanadesikan, and Colin Mallows, and so brought substantial collective insight into the needs of statistical data analysis.

A number of unpublished documents record our thinking from 1965; two brief excerpts will show some key concepts. On February 2, 1965, we had a planning meeting and general discussion. The next day, John Tukey wrote an informal memo, Tukey (1965), to the participants. Figure 1 is a diagram from that memo.

Tukey's contributions to statistics need little summary here, but his relation to this particular project does: he divided his time between Princeton University and Bell Labs, and tended to appear at the Labs on occasion to attend meetings or seminars, to discuss current interests, and to advise management. His appearance at

[1]Large-scale applications did exist, even at this time; the analysis of the data recorded by TelStar, an early communications satellite, involved tens of thousands of observations and challenged contemporary computing technology, (Gabbe, Wilk & Brown 1965)

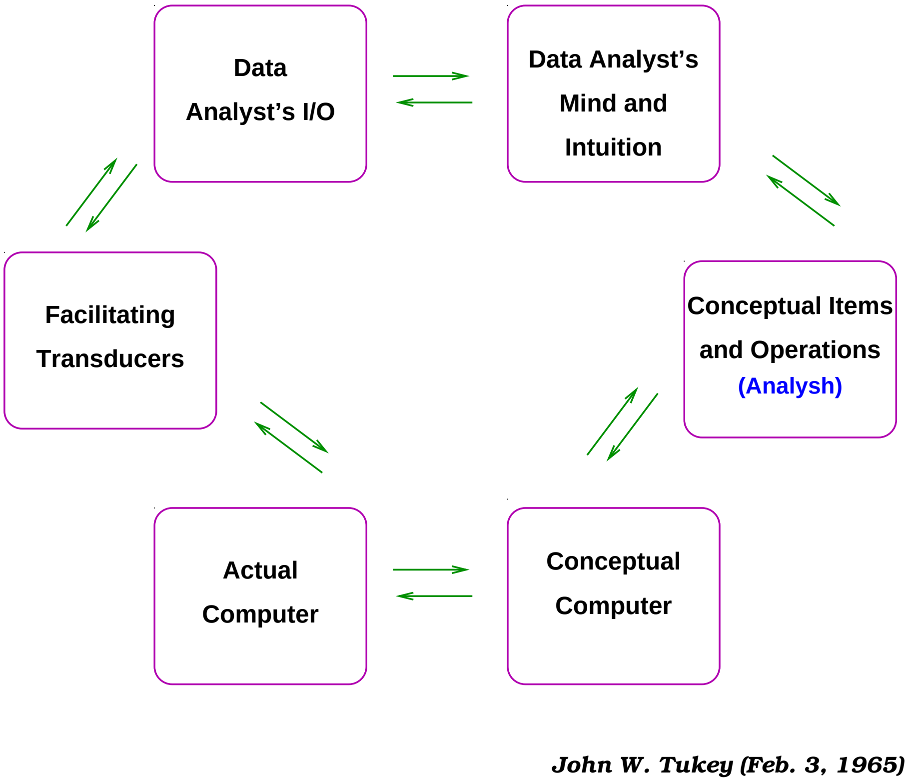

Figure 1: *Concepts for a Statistical System, from John Tukey.*

meetings often resulted in one of his inimitable memos, full of new ideas and newly invented terminology. Interpreting Tukey's ideas was a fruitful if risky activity; Figure 1 provides a good example. Let me try my hand at explaining it, with the admission that thirty-three years of hindsight may affect the results. Keep in mind that this was written within a day of the meeting that stimulated it. We should be looking for the concepts it reveals, not picking at details.

Follow the arrows clockwise from the *Mind and Intuition* block. Tukey's notion is that data analysts have an arsenal of operations applicable to data, which they describe to themselves and to each other in a combination of mathematics and (English) words, for which he coins the term *Analysh*. These descriptions can be made into algorithms (my term, not his)—specific computational methods, but not yet realized for an actual computer (hence the *conceptual* computer). Then a further mapping implements the algorithm, and running it produces output for the data analyst. The output, of course, stimulates further ideas and the cycle continues.[2]

On August 3, 1965, an extended group met to review some of the work so far. The following paragraph comes from opening remarks (Wilk 1965), at that meeting made by Martin Wilk, who was then head of the statistics research department at Bell Labs (emphasis added):

> What do we want? We want to have *easy, flexible, availability of basic or higher level operations*, with convenient data manipulation, bookkeeping and IO capacity. We want to be able easily to modify data, output formats, small and large programs and to do all this and more with *a standard language adapted to statistical usage*.

The first emphasized passage in this paragraph defines a goal expressed thirty years later in the book *Programming with Data* (Chambers 1998*b*) by the slogan:

To turn ideas into software, quickly and faithfully.

The operations must be *easily* available so we can go quickly from the data analyst's concepts to working software. They must be *flexible* so that the software can faithfully reflect those concepts, as we work to refine the first efforts. The term *higher level operations* refers to efforts to find close computational parallels to the data analyst's essential concepts, another aspect of easy use of computing.

The second emphasized passage reflects another general principle, that we need to embed our statistical computing in a general environment, not restrict it to some pre-defined scope. The approach planned in 1965 to attain this goal was to embed the statistical operations directly in a general programming language. Later on, the same goal would be approached via *interfaces* between a statistical language and other languages and systems (see section 3.3 for interfaces and section 6.2 for a possible combination of both approaches).

The documents cited and the project itself were essentially forgotten after Bell Labs dropped out of the Multics operating system development. Eleven years later, a different group met again at Bell Labs with a similar charter; the resulting software eventually became the first version of S. In 1976, we did not make any explicit use of or reference to the previous work and I was the only participant common to both efforts. Only when re-reading the older documents in preparing the Neyman lecture was I struck by the key concepts that were later re-expressed in different terms and finally implemented.

[2]The "facilitating transducers" I interpret to mean software that allows information to be translated back and forth between internal machine form and forms that humans can write or look at—a transducer, in general, converts energy from one form to another. So parsers and formatting software would be examples.

2.3 Statistical Computing: England, 1967

In December, 1966, a meeting on statistical computing was held in Cheltenham, England, organized by Brian Cooper and John Nelder. Papers were presented describing current systems and other topics. Some of the papers appeared in *Applied Statistics* in 1967, (Cooper & Nelder 1967). I was able to participate, since David Cox had invited me to spend a post-graduate year teaching at Imperial College; my contribution was a general review, (Chambers 1967*b*), since the 1965 Bell Labs effort was not going ahead.

After the meeting, a number of the participants spontaneously formed a "British Working Party on Statistical Computing". Members included some of the main practitioners of statistical computing in Britain at the time. The goals of the group were to promote statistical computing and start some initiatives to encourage research and co-operation in the field. Supported initially by the U.K. Science Research Council, the Working Party was, I believe, the first professional organization in statistical computing; it later metamorphosed into the computing section of the Royal Statistical Society. A description of the Working Party later appeared in the *American Statistician* (Chambers 1968); experience with the Working Party encouraged some of us to support early efforts towards a computing section for the American Statistical Association.

The main concrete achievement of the Working Party was the algorithm section of the journal *Applied Statistics*. We had some other, more ambitious goals as well; for example, to encourage data exchange and standardization among statistical systems. No explicit proposal to this effect ever resulted, but at least two memoranda and some additional correspondence provide another interesting example of concepts that were to recur later.

John Nelder wrote a description of data structures, drawing both on the recent GenStat work and on an earlier paper related to the structure of variates in experimental design. Partly in response to this and partly reflecting the earlier Bell Labs effort, I wrote a memo (Chambers 1967*a*) titled *Self-Defining Data Structures for Statistical Computing*. A thirty-one-year-old document with that title must be interesting, one way or another.

The memo contained "some ideas concerning self-defining data structures, with particular reference to data matrices". The underlying concepts, which had evolved from discussions of the Working Party, were in effect a proposed standard for organizing statistical data on external media (although the word "standard" never appeared in the memo). What did "data matrix" mean? "Conceptually, a data matrix consists of a two-way array of data." However, the term "array" was used in a more general sense than in Fortran or similar languages. In later terminology, it corresponded closely to a *data table* in a spreadsheet or relational database system, or to a *data frame* in S.

The memo defined a data matrix abstractly by twelve functions that would have values for a data structure "if and only if the structure is a data matrix". For example, `datum(i,j)` returned the value of the `j`-th variate for the `i`-th observational unit. Individual data elements could have numeric, string, or logical values, or the values MISSING or UNDEFINED (the distinction being that MISSING values were for observations undefined for `(i, j)` in the range of the data matrix dimensions, and UNDEFINED for observations outside the range. Other functions provided various labelling information. There was additional discussion of possible physical storage layouts and extended structures, relating to what we would now consider different experimental designs.

The concept of self-defining data structures was implicit in APL. Here it is explicit and in some respects more general; at any rate, the data matrix structure as here defined is closely related to later data tables in statistical and database software. Self-defining data structure is key to *object-based* computing with data (section 3.2).

2.4 Statistical Computing: Thirty Years Later

Programming languages of thirty-five years ago had introduced some key ideas. Concerned professionals in statistics were starting to address important issues such as the real needs for serious computing with data and the concepts underlying such computing.

Some natural questions then arise. If all this was going on over thirty years ago, why do we still need to discuss the same topics now? And did the aborted efforts at that time represent missed opportunities for the statistics profession?

The first question is not at all impertinent; most of the rest of the paper will amount to considering partial answers to the question. What current concepts have we in fact got right, more or less? What are the key challenges still facing us? Are there opportunities in the near future to get some things more nearly right than we have managed in the past?

As for missed opportunities, some measure of regret is indeed inevitable. The Working Party discussion of data matrix structure had many correspondences to the relational data model, which was then about a decade in the future and which now dominates most database management software. There were far too many things wrong with the data matrix proposal as it was left for it to have made a positive contribution, but had we pursued our ideas together, statisticians might well have exerted more influence on data definition in computing, to the benefit of both the statistics and computing communities.

Only gradually, and along with enormous changes in the general computing environment, would the concepts recognized to some extent thirty years ago become practical parts of the statistician's computing toolkit.

The key concepts identified in programming languages of 1963 were not so obvious then, and practical issues about languages like Lisp and APL would inhibit their use for some time. The concepts for statistical computing, for example of data structure, would need rethinking and refinement. Some important computations, such as graphics, lacked the general formulation thirty years ago that would arrive a few years later. Non-interactive languages (such as our proposed extension of PL-1) would not give sufficiently easy access to computations, and general interactive languages such as APL would not prove to be flexible enough, in the opinion of most potential users. That "easy and flexible" combination would remain difficult to attain.

All the languages in section 2.1 belonged to what was then beginning to be called *scientific* computing, in contrast to *business* computing. They shared the common notion that some form of symbolic formulation (function, list, or operator) was a reasonable way for the human to communicate to the computer. Business-oriented languages rejected this as too abstract for their users, in favor of various specialized syntaxes, often verbal in some form of English-like layout. This bias, for example, continued into systems for database management, including the SQL standard for relational databases. Integrating such systems with languages similar to those discussed here remains a major challenge. Modern facilities such as the JDBC interface to database software, (see, for example, Hamilton, Cattell & Fisher (1997)), are an improved approach to this interface, but much remains to be done.

3 Concepts

With reflections on the past as background, let us next attempt to characterize what makes computing with data effective. Our emphasis will be on defining some key concepts, and relating them to practical software. Three concepts in particular will help:

1. *language*, the ability to express computations;

2. *objects*, the ability to deal with structure;

3. *interfaces*, the ability to connect to other computations.

All of these arose in our discussion of past work; over time, they have evolved and become even more important

One additional general point needs emphasis: in modern computing, there should not be a sharp distinction between *users* and *programmers*. Most programming with statistical systems is done by users, and should be. As soon as the system doesn't do quite what the user wants, the choice is to give up or to become a programmer, by modifying what the system currently does.

Such user/programmers then naturally go through stages of involvement. In the first stage, the user needs to get that initial programming request across to the system, quickly and easily. Later, the user needs the ability to refine that first request gradually, to come closer to what was really wanted. Good software for computing with data should support all such stages smoothly.

3.1 Language

The role of *language* in programming is to prescribe (to the computer) and to describe (to ourselves and other humans) requests for computations. In a fundamental sense, what we can describe is what we can compute. For computing with data, in the sense we are using the term—the organization, visualization and analysis of processes involving significant quantity and complexity of information—the languages available determine what we can learn and show about the data.

Over the thirty or thirty-five year history of languages for statistical computing, has some definable progress been made? I think the answer is yes (an answer of no would certainly be discouraging) and that a little further examination of the role language plays will help. Certainly what makes a "good" language is both subjective and empirical. Every user or observer of a language will justifiably form an opinion; and empirically a good language is one with which users can produce good results.

To be a little more specific about the assessment, however, I think we can isolate two goals already implicit in the discussion.

- Users should be able to program in the language quickly and conveniently.
- Users should be able to say what they mean accurately in the language.

This is another version of the slogan in section 2.2:

To turn ideas into software, quickly and faithfully.

The relevant sense of time in measuring "quickly" should be the clock on the wall, not the computer processor time consumed. Human time is the scarce resource, for many reasons. Most users have many demands on their time; learning from data has to compete for that time, and if a new idea takes too long to express, some alternative to gaining the knowledge will be used instead. How can languages reduce the human time expended to express a new idea?

1. Users should be able to capture what has already been done and just change it a little.

2. The language and the system should give the user a high-level view, doing as much as possible automatically.

The first goal needs some tools for easy definition and modification of software in the language; see, for example Chambers (1998*b*, pp 19-21). This helps the user already working in a language; not so well served currently is the user who starts from a graphical interface; we need to develop techniques to capture such interaction and convert it cleanly to language form.

The second goal is one area where real progress has, I think, been made. Languages such as APL and data structure discussions such as the Working Party memorandum had, indeed, grasped some key concepts. Modern languages, however, have a much more advanced set of concepts. Invoking a function or operation on a self-describing, general object, particularly in a language using the class/method paradigm, is much more likely to produce an expression that visibly "says what it means". There is some tension between this goal and principles, such as strong typing, proposed for general programming languages. Section 6.2 will suggest ways to resolve such tensions.

Next, the "faithfully" part of the slogan. For a language to reflect faithfully the ideas its users want to express, it needs first to be expressive and flexible. Ease of expression, which we need to say things quickly, is equally important in saying things accurately. A long, convoluted piece of software is unlikely to be clear, probably not even to its author and certainly not to anyone else. Unclear software is also unlikely to be accurate, particularly as time goes by and it is modified for various reasons.

Just how best to encourage and assist faithful programming of ideas remains controversial. Many principles have been proposed, including structured programming, functional languages, object-oriented programming, strong typing, and many more. The notion that users become programmers and, in particular, that they do so by moving from interactive use of a system to increasingly serious programming, introduces some special considerations. As a general position, I believe that languages for computing with data should include a language that encourages initial programming steps by the user of an interactive systems, but which also allows the programming to migrate smoothly to more precise and detailed control. The current version of S (Chambers 1998*b*) is an attempt to implement such an approach, but new and better efforts are definitely a hope for the future, as section 6 discusses. Interestingly, some recent discussion of programming languages in general makes a related distinction between interactive "scripting" languages and "system programming" languages (Ousterhout 1998). The ability to combine both, however, is perhaps the most important future goal.

An appendix to the web version of this paper (Chambers 1998*a*) suggests some specific principles for languages to support computing with data. To summarize, it recommends a combination of functional programming, dynamic and untyped object references, and class/method-based software for further refinement. A second appendix discusses the relation between graphical user interfaces and languages; while sometimes cast as opponents they should ideally work together, particularly by defining elements of the interfaces in the language.

3.2 Objects

What are the main useful concepts in organizing data for our computations?

1. The data we deal with exist as dynamic, self-defining *objects*.

2. A fundamental organizing principle for objects is that they belong to, or have as an attribute, a *class*, which determines the information they contain and the methods or operations that are defined for them.

3. The information content of classes can be defined *recursively*, starting from some simple base classes.

4. Everything is an object. Computations can examine the definitions of classes (and other constructs in the language) by using suitable objects (*metadata* objects) containing that information.

These concepts are almost exactly those in the discussion of data structure for the version of S described in Chambers (1998*b*). Very encouragingly, though, they also apply, with a few changes, to current versions of Java and to the language underlying the CORBA standard for distributed computing (of which we shall say more in the discussion of future projects), as well as to other modern languages to a greater or lesser degree.

3.3 Interfaces

In contrast to the biblical account of human languages, languages for computing never had a period before the tower of Babel. As soon as the concept of a programming language or system started to crystallize, different realizations of the concept began to appear. By thirty-five years ago, there were many languages (relative to the amount of computing in the world, at least as many as today), with recognizably different purposes.

For our discussion, the term *interface* means the ability of software in one language to invoke operations or methods in other languages. The need for interface computation was recognized very early, but only later was this concept explicitly implemented.

The *everything is an object* principle and a functional programming style suggest that an inter-language interface should be a function call returning a self-describing object. This is the model that S and some other statistical languages use currently for interfaces to subroutines in languages such as C and Fortran in the same process, and to subprocesses such as shell commands. The function defining each interface uses a model for computations in the other language; for example, mapping basic objects in S into arrays in a Fortran subroutine call.

What about general communication with other processes, perhaps running on other machines? The technique of *remote procedure calling* implements this style of interaction, for a single language; for example, current versions of Java implement *remote method invocation*, again between two Java applications.

This model for communication will only work between different languages if (at least) one language knows enough to construct the necessary data and perform the request in a form meaningful for the remote program in the other language. The limitation of such approaches is that each pair of systems will need a separate interface. Just as the data-structure proposal of thirty years ago hoped to do with data matrices, so it would be better if some single form could be devised to mediate both the requests and the objects. The prospects for better approaches in the future are promising, as discussed in section 6.1.

4 The Present

A few general reflections and a look at one substantial current project may help assess the impact of past work on computing with data and the challenges ahead.

4.1 How are We Doing?

This question paraphrases former New York mayor Ed Koch. The answer really comes from the voters, or in our case the users, but some general reflections are possible. The concepts outlined in the previous section have matured over the decades and been incorporated in various ways into systems and languages for computing with data. Users do get a real boost in their ability to express their ideas in software.

For example, consider a typical linear regression expression in S:

```
lm(Fuel ~ Weight + Type, auto98)
```

Assuming the user is comfortable with functional notation, and a little familiar with models in S, the expression says fairly directly what the user wants: a linear model that fits the variable `Fuel` to a predictor containing terms in variables `Weight` and `Type`, using the dataset named `auto98`. The language, based on users' reactions, seems to express the ideas reasonably well, contributing to turning ideas about models into software quickly.

Less overt but equally important is the role of objects, particularly the *everything is an object* philosophy. The formula expressing the model is itself an object, meaning that the software implementing the model fitting can examine its own call to find the variables in the model and construct from that information the specific data needed. The user is *not* required to extract the specific variables, only to identify them. The variables are themselves self-describing objects, meaning that they can be interpreted according to their structure. For example, `Weight` is a numeric variable and `Type` a categorical factor; in the model, these will contribute suitable, different predictors to the fit. The value of the call to `lm` is also a self-describing object. Rather than pre-specifying all the information wanted from the fit, the user constructs the fitted model as an object and then studies that object in an open-ended, interactive series of computations.

The effectiveness of the computation also benefits from the use of the interface concept. While the management of the model specification and the construction of the object resulting from the fit are naturally part of a statistical language, the fundamental numerical computations in this case fall in an area long and skillfully worked by numerical analysts and implemented in program libraries, traditionally in Fortran. A natural interface from the statistical language to Fortran allows these computations to be used, essentially unchanged, without much concern on the statistical programmer's part about the numerical details.

The features cited are largely "obvious" by now for users of modern statistical systems, but they represent efficiency gains in the use of the scarcest resource, skilled human time. So we have done pretty well in a number of ways in improving the environment for data analysis. However, the challenges arguably loom at least as large now as ever, for several reasons. We will look at these in section 5, but an underlying theme is that the general environment has been changing around us at least as quickly as our own software has been evolving. Both new opportunities and tougher challenges have resulted. A look at one current project will illustrate.

4.2 An Application

To focus our understanding of where we are and what challenges we face in improving our computing with data, we will benefit from looking at a current, serious, and quite successful project. The computations involve the most voluminous data source in the communications industry: the detailed information about telephone calls, or *call detail records* in the jargon of the industry. Over a period of several years a group of research statisticians and others looked at statistical and computational techniques using this data, with the specific goal of identifying possible fraudulent use of the system; for example, calls charged to a customer account but not authorized by the customer. The project began at AT&T Bell Labs and continued, after the split between AT&T and Lucent Technologies, as two projects at AT&T Research and at Bell Labs (the details below follow the Bell Labs version). Both projects have been interesting and successful; in particular, AT&T believes it has made substantial savings by applying the techniques.

Why, on the other hand, is this a useful example for discussing computing with data? There are several reasons: first, the computational problems are very serious, just at the edge of what is currently possible; second, the success of the project came with a great deal of human effort, so that we can reasonably ask whether improving the computing environment might make similar projects easier in the future; and, third, although the project is unique, as are nearly all serious efforts, it demonstrates a number of similarities with other major applications of statistics to science, business, and society. Elucidating these common features will help to illuminate some challenges and suggest directions for future work.

Figure 2 illustrates the process for the application. All billable calls generate information (the call detail records) needed for billing and other transaction requirements. The data includes the time and duration of the call, the calling and called number, and special codes for different kinds of call. The transaction manager program makes this information available to the fraud detection software. As calls come in for a particular customer, the software updates a *signature*, a statistical characterization of that customer's calling pattern. We all use the phone system for diverse purposes, so one call may differ from the next in many ways; however, in a statistical sense the distributional properties of our usage tend to have distinctive features, differing among customers but changing only in fairly slow or regular ways for an individual customer. Designing and computing signatures to characterize both normal and fraudulent use is the art behind this approach to fraud detection. When a customer's calling pattern starts to differ significantly from the signature, and perhaps starts to resemble signatures typical of fraudulent use, the software will at some point raise an event (in computing terminology) that triggers an intensive followup procedure, involving human intervention. Additional data, such as customer records, may be used in the followup.

The volume of data flowing through the transaction manager can be very large, from millions to hundreds of millions of calls per day depending on the application and on where the data is being drawn off for analysis. The software must update the signatures very quickly, putting strong time constraints on the form of the signature.

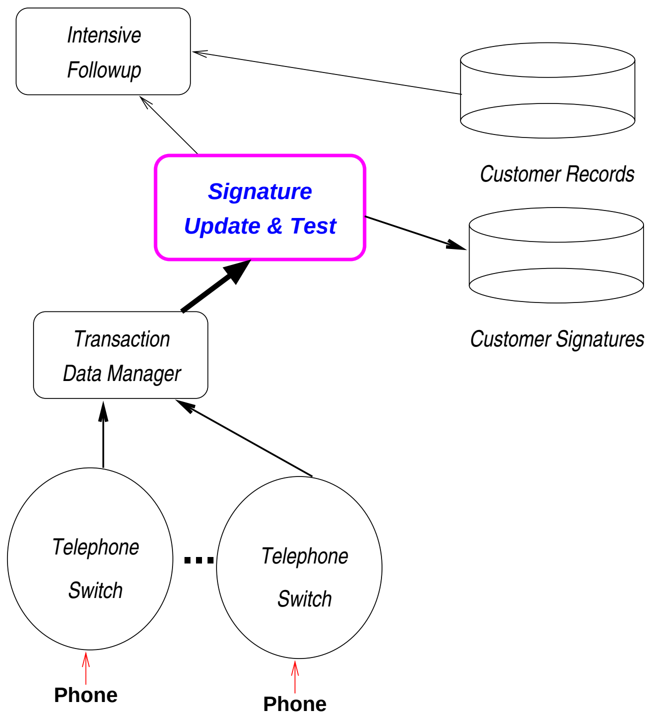

Figure 2: *Processing of telephone call detail records to update signatures characterizing customer behavior, and to test for possible fraudulent usage.*

Also, the computations must be designed both to maintain a database of customer signatures, when behavior seems normal, and also to provide as accurate a discriminator as possible for fraudulent behavior. The classification of calling patterns as possibly fraudulent can never be sure; hence, the need for a followup with human intervention. But the enormous volume of data means that the signatures must do a good job of selection, since otherwise the followup will be swamped by false detections (and cost more than it saves) or will miss a substantial amount of fraudulent calling. Clearly the design, implementation, and testing of the signatures presents a major challenge.

As a technical and statistical effort, the application has many fascinating aspects. We have introduced it here to examine computational questions, but before moving on to that, a few brief comments on the application generally are needed to put the computations in perspective. What contributed to the success of the application? As usual, good ideas (such as the signature notion) and much hard work were the main ingredients. In addition, however, there was a willingness to engage in the *process* underlying the technical problem, and to take on challenges in areas such as data management that were not obviously part of the statistician's job, but were essential to success. There was also a willingness to rethink the statistical methods in a form that made sense and was implementable under the constraints.

On the other hand, one might ask whether the statistician brings some special value to the application, beyond what might be achieved by techniques broadly defined as "data mining", based more on algorithmic than data analytical perspectives. Two aspects seem to support the value of statistical insights. First, the notion of estimating and comparing distributions, particularly in an adaptive way, turns out to be an important insight, and such thinking comes naturally from a statistical perspective. Second, this application does not lend itself to completely automated solutions: the best that a procedure is likely to hope for is to help the human in the followup, not by any means to declare automatically that a fraud situation exists. In this context, the data analyst's skills at summary and visualization are key, particularly if we can combine them with a quality interface for the followup, as we will suggest in the next section.

The signatures have to be computable very quickly, so the production versions need to be implemented directly in an efficient form, with relatively little flexibility. In *designing* the signatures, however, the statisticians will benefit from "easy and flexible" access to any helpful tools. The current technique uses tools such as shell and `awk` scripts. The tools may themselves be manufactured or augmented by other programs, and the results of testing them will be assessed by reading the results into a statistical system (S) for display and summary. A production system will be implemented by another group, in a language that can be added to the transaction manager in the switch.

It takes nothing away from the achievement here to note that the programming for the computations is very labor intensive and requires the participants to link the tools together in an *ad hoc* way. Also, the databases and the user interface for followup are not currently part of the design process, but should be. In the next section, we will consider how we might modify the computing environment to make better use of the designers' time and effort.

5 Challenges

The position of statistics as a profession, relative to science in general, to applications in industry, business, and government, or to overall academic activities, has made substantial advances over the period surveyed by this paper. Research and applications in industry, for example, have scored a number of successes such as the example in the previous section. (It can be added that advances in the computing support for statistics have contributed to many of these successes.) There remains a fairly widespread sense that our technology should be applied and/or appreciated more in many areas. Hahn & Hoerl (1998) and their discussants provide some views on the subject, in the context of industrial applications.

As suggested in the example, much of the challenge can be summarized by saying that we need to be more involved in the *process* behind the data. From the perspective of the present paper, this leads to the question: What aspects of computing can contribute to broadening our involvement in the process, and are there some specific new directions in statistical computing that would help?

A prescription to the statistics profession that it should be involved in more of the process will not be very helpful if it does not go beyond just encouraging students to learn more and practitioners to work harder. In particular, from the computational perspective, if the advice is only to learn database management languages, user interface programming, and various other systems in addition to the statistical software already needed, we are unlikely to see much progress.

So the overall challenge presented here translates into the challenge for computing of making this increasingly diverse range of tasks accessible to practitioners *without* forcing them to learn many new languages or to abandon the computing environments that, as we have noted, have largely proven helpful to the profession.

Fortunately, new technology in computing may help us to meet this challenge: section 6 will look at examples. First, though, let us make the nature of the challenge more concrete. To do so, suppose we revisit the application in section 4.2, where call detail data was used to detect possible fraudulent calling behavior.

As we noted in that section, the tools used to test out possible signature computations were to be reprogrammed later in another language; the database requirements for managing the signature databases likewise were specially crafted; results from the testing were imported manually into a statistical system for display and summary. The design of software for the followup (in particular, of a user interface for the human intervention required) was not included in the study.

Some improvements in the computing environment would help both the statistical work and the reimplementation that follows. If the computations could be *distributed* transparently among different languages and systems, better use could be made of software specialized to tasks such as data management and user interface programming. Software *reuse* in the reimplementation would also be improved if the initial design and experimentation could use some or all of the same systems as would be appropriate for the final procedure. In other words, we should be able to choose high-level, appropriate languages for the various tools, and link them together easily and flexibly.

Figure 3 suggests one possible configuration to support the application in this style. By comparison with Figure 2, we have essentially blown open the single box for computing and testing signatures into a co-ordinated collection of diverse tools.

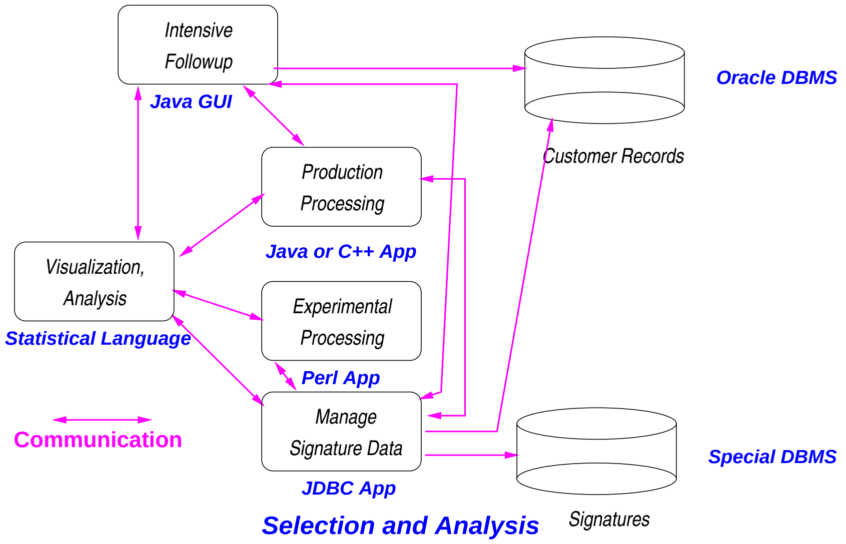

Figure 3: *Revised version of Figure 2, filling in the specific computations with a wish list, in which high-level systems take over important tasks. Arrows show communication of requests among the systems.*

Signature databases are now managed for us in a systematic way; as one helpful possibility, the figure shows an interface to the database management through the Java DataBase Connectivity (Hamilton et al. 1997). Many other possibilities exist, but JDBC is general, handles many database systems, and benefits from many features of Java.

Similarly, the followup box, which was unspecified before, is now assumed to be a graphical user interface, programmed at least in prototype form during the design phase. Bringing the GUI into the design phase is important because it allows experimenting with tools that might help the human in the followup to obtain more information, perhaps even by communicating with the statistical language to obtain visualizations and summaries. The prototypes of signature computations will still need to be done in some easily re-programmed form, but now we want to bring the corresponding language explicitly into the programming environment. Finally, there will need to be some more efficient version of the chosen computations for the production system. However, instead of assuming that these will be re-programmed from scratch we would much prefer for at least a close approximation to be produced during the design and analysis phase. That way, at least a good approximation to the performance of the final product can be obtained early on, instead of committing to an expensive reprogramming effort without direct evidence that the result will perform well enough.

Notice that relatively little of the technology we needed in the example falls under the typical definition of statistical computing. Database management systems, software for designing graphical user interfaces, and techniques for communicating computations among diverse systems are not found in the usual texts on statistical computing. Nor, to repeat the earlier point, is it useful to simply argue that they should be. Statisticians will need to understand some key *concepts* from these areas, to some extent, but the challenge is more to make the concepts accessible from good computing environments for data analysis, while hiding as much of the detail as possible.

The arrows in Figure 3 show requests that might likely be made from one of the systems to another. For example, if we assume the statisticians are working in their interactive environment, they will need to issue requests to the database directly to find some summary information about the signatures on the database. The prototype signature computations, of course, also need to communicate to the simulated transaction database to get input and to the signature database to update signatures. The statistical language needs to communicate to the prototype generator, both to re-program the specific computations and also to run and analyze simulations. The user interface for followup needs access to the customer database and as we mentioned plausibly to the statistical language or to some other graphical system for helpful visualization and summary. And so on, through a very extensive range of useful inter-connections.

Through all this we need to remember that statisticians and, even more, users of the followup interface should *not* be expected to master more than one or a very few of these systems in detail. It is the job of the general computing environment to provide “easy and flexible” communication among the systems. Surprisingly, perhaps, the prospects for such an environment are quite promising. Several routes might get us there; section 6.2 describes a new joint project in statistical computing that I regard as an exciting approach to this and other goals.

6 The Future

Those of us interested in computing with data have many opportunities and challenges. In particular we would like to use computing technology to help statisticians contribute more to applications. The good news is that there are some exciting opportunities for relevant research in these areas.

6.1 Distributed Computing with Data

A computing environment such as that suggested by Figure 3 combines traditional statistical computing, say through one or more statistical languages, with a rich variety of other tools. The statistician can adopt a broader role, becoming directly concerned with data management issues and with the design of user interfaces to handle the results produced by the statistical techniques. The languages and systems involved will need to communicate with one another, and with the end user.

All of this is a rich but complicated network of tools. To make things still more complicated, the users, the data, and the tools may be distributed geographically or over different computers and operating systems.

Neither the designers nor the users of tools in this environment should have to struggle with the details of all these systems. For the most part, each person will want to work with a single language or non-programming interface; the distribution of the actual computations should be transparent.

To deal with such an environment, we need some new programming concepts, not particularly associated with statistical computing in the past. Fortunately, a number of trends in general programming languages and systems are currently moving towards similar environments; after all, the need for the kind of distributed environment suggested by Figure 3 is felt broadly, not just in our own work.

Some of the key concepts, and some general programming tools related to them are as follows.

- Requests from one component of the system to another should appear local, with the communication dealt with for the programmer, not by the programmer. For example, Java provides the technique of *remote method invocation* in which an essentially ordinary-looking call to a Java method may in fact be evaluated remotely.

- Such requests, however, should ideally not be restricted to a particular language or operating system. There are many existing systems, certainly in our applications, that need to be accessible. The most developed standard and architecture for language- and system-neutral communication is CORBA, the Common Object Request Broker Architecture, (Pope 1998). To oversimplify, CORBA provides a mechanism to define methods offered by an application server in a form that can be translated into various supporting languages. CORBA also provides a variety of services that manage the details of communication.

- Although we want the details of control and communication handled automatically, the programming environment must include a variety of high-level ways to specify interactions between systems and with users. In particular, the handling of *events* is central to all user interface design and many other aspects of distributed systems as well. Both Java and CORBA provide mechanisms to define desired event management at a relatively high level of abstraction.

- Finally, all the information about methods, classes, and related objects must be available dynamically, so that the system itself can find the information for the programmer, rather than leaving the programmer to define the details of interfaces. This ability, variously called introspection, reflectance, runtime discovery, and self-defining metadata, is increasingly recognized as crucial to powerful distributed computing. Java, CORBA, and other systems provide such mechanisms.

To move towards powerful distributed systems, we need to incorporate the new concepts into our basic programming environment. The more modern statistical languages often include some such facilities currently, but none of them combines a full set of such features with a broad range of the basic necessities for computing with data.

Here then, is a challenge for the next generation of our software, along with some strong suggestions about useful computing tools to deal with the challenge. Where next?

6.2 A Co-operative Project in Computing for Statistics

A new project is starting that promises to bring many benefits to computing with data, including access to the distributed approach of the previous section. The project, called Omega for the moment, proposes a joint effort among a wide group interested in computing for statistics. The initial discussions included designers responsible for the current statistical languages S (John Chambers and Duncan Temple Lang), R (Robert Gentleman and Ross Ihaka) and Lisp-Stat (Luke Tierney). During and immediately after the conference “Statistical Science and the Internet” in July 1998, it became clear that we all have interests in new work that could benefit greatly from working together with a broad group of participants to produce some high-quality, open-source software.

The Omega web page, `http://omega.stat.wisc.edu`, documents the goals and activities of the Omega project. The present section outlines my personal view of the current, preliminary plans.

The expected results of the project can be viewed on three levels:

1. the definition of standard *modules*, to be define capabilities needed for statistical applications;

2. implementation of these modules in *packages* that provide the capabilities;

3. interactive *languages* that include access to the capabilities.

Nearly all details are subject to change at this time, but a general approach that could provide the three levels can be described now. If other implementations prove better, these can be chosen instead of or in addition to the description below.

The second level is perhaps the easiest to describe. While the implementation of packages is not *restricted* to a specific language, there is particular interest in the Java language and the virtual machine on which it is based. Java is designed around distributed computing and includes dynamically accessible definitions of available data structures and methods. A large and growing body of software in Java is available. Its use in programming user interfaces and graphical applications is well-known, but increasing attention is being paid to its use as a general programming environment, including progress in more efficient compilation of the code.

Although it includes dynamic facilities, Java grew from a more traditional, compilation-oriented view of programming. For our applications, this needs to be supplemented by a high-level but general interactive language—the "quickly" part of our goal for computing with data. In the Omega project, we anticipate providing several such interactive approaches. An interactive interface to Java itself has been developed; initially, it is relatively "pure" Java, but can now be extended in a number of ways to provide features of interest to us and our users. For example, some functional-language style of evaluating calls to function objects would mimic abilities in current statistical langugaes. A language based on Java or on its underlying virtual machine can access packages written in Java directly, using Java's reflectance capability; therefore, the interface issue, in the sense of section 3.3, largely goes away for such packages.

The other way to provide interactive use of packages is to add an interface between an existing statistical language (S, R, or Lisp-Stat, for example) and the packages. These interfaces would use dynamic access to module definition, say in CORBA, to request a service without intermediate compilation steps. Rather than a separate interface for each service, we only need to implement one client interface between each statistical system and CORBA. We can also implement a single server interface for each statistical system, to accept requests for methods and attempt to satisfy them dynamically in the statistical language.

We expect to implement both new languages and interfaces. The new languages are themselves interesting areas of research, in addition to the work on modules and packages. Interfaces to existing languages will be essential to providing a full range of computing to programmers and users, as facilities at the module and package level are developed in Omega.

The long-range value of the new project will rely on wide participation from the statistical community. We need to design and implement modules for new computing facilities (among many others, interesting possibilities include: general database access; new directions in modeling; Bayesian estimation; new visualization and graphics techniques). Our optimism that useful, high-quality software can come from such joint projects is based in part on existing successes in this style, such as Linux, and more directly on experience with statistical software, such as R. Here is another opportunity for us to work together to our mutual benefit.

7 Summary

The "peaceful collision of computing and statistics", as John Tukey described it in his 1965 memo, has profoundly changed the way statistics is done and how it is applied in science and other activities. The changes, on the whole beneficial, continue at an undiminished pace as we cope with new opportunities posed by changes in computing and in the applications we try to serve.

Over this long period, some concepts have been found that help produce useful software for computing with data. Languages that make it easy to express statistical ideas and that adapt smoothly as we refine the ideas allow us to turn those ideas into effective software. Powerful capabilities are created from true object-based computing, in which everything is an object and the language is capable of understanding and operating on its own classes, methods, and other components, as well as on an open-ended structure for statistical data. Effective interfaces are needed between languages and systems, so that computing with data is not restricted to a narrow scope; in the future, this needs to include computing distributed over geography, language, and computing environment.

Modern statistical software has made substantial strides in applying these concepts. Statisticians benefit from the advances in their own research and in developing tools for applications. The challenges remain undiminished however, since both computing technology and the scope of potential applications have changed at least as fast as we have. Fortunately, those changes have included some powerful new computing tools that we can use to move ourselves forward; in the process, we can enjoy some exciting research in software for statistics. One new project has been described that offers hope for interesting and useful joint work on a wide variety of topics. If we can work effectively together, the benefits for all of us could be substantial.

Acknowledgements

In terms of computing ideas, no small list could begin to enumerate contributions over this long period of time, but mention should be made of some of those involved in versions of S in the past (Rick Becker, Allan Wilks, Trevor Hastie, and Daryl Pregibon, for example) and in current plans (Duncan Temple Lang and David James, for example).

For the description of the fraud detection project I am indebted to discussions with both the Bell Labs group, including Diane Lambert, José Pinheiro, and Don Sun, and the AT&T Research group, including Rick Becker and Allan Wilks.

A Principles for Good Interactive Languages

No one can object to this endorsement of clear, compact, understandable programming, but what properties should languages have to encourage such programming? (Encourage is the correct verb; no language can prevent dreadful programming, much less ensure good programming.) There has, needless to say, been a generation or more of argument about such issues. Here are some personal opinions.

1. The effect of evaluating basic operations or function calls should be only to return an object, with no hidden side effects (the principle of functional languages).

2. The standard, default invocation of an operation should be simple and should then be simple to modify, for example through specification of optional arguments by name.

3. One should be able to build up extensive software by gradual refinement and extension. One important tool is a *general* way to define methods. Methods organize software in a conceptual table of generic functions by signatures for classes of the arguments to the functions.

4. Individual functions and/or methods should be local objects, organizable at one or more levels into packages. The language should contain simple mechanisms for persistent storage of objects, including function/method objects, along with the necessary database operations implied to access and manage such objects. Users should be able to include or attach such packages easily to provide tools for their own programming.

5. The language's syntax should be, to paraphrase Einstein, as simple as possible, but no simpler. In practice this means that the designer has the difficult

task of deciding how much syntactic sugar is good for the users. Extreme simplicity appeals to designers and theoreticians but has been shown to turn off users; conversely, sloppy or excessively elaborate languages can pretty convincingly be argued to lead to poor programming.

6. The most important principle for modern languages may be: “everything is an object”. The language must be able to compute on itself (see the example in section 4.1). There must be dynamic access to information about the language itself and about the available methods (see section 6).

No language would get perfect marks on these criteria, and the score for any particular language would likely be debatable.

The version of S described in Chambers (1998*b*) was designed with these ideas in mind, but constrained of course by back compatibility, limits on time and resources, and plenty of human frailty. My personal opinion is that Java is a general programming language providing a good base, in terms of these criteria, particularly if it is supplemented by interactive access to suitable packages, in order to improve the “quickly” part of the goal. The supplement needs to provide “functional programming” features, such as items 1–3 in the list; these have proven to be useful in interactive programming. The project described in section 6.2 addresses a number of these issues.

B Languages and GUIs

The fundamental goal of *language* in programming is the ability to prescribe (to the computer) and to describe (to ourselves and other humans) requests for computations. In a fundamental sense, what we can describe is what we can compute. For those needing to compute with data, in the sense we are using the term—the organization, visualization and analysis of processes involving significant quantity and complexity of information—the languages available determine what we can learn and show about the data.

It may seem obvious, perhaps even trivial, to assert then that languages, and in particular *good* languages, are the heart of successful computing with data. I believe the statement to be true, but it is not easy to defend explicitly, and much activity in computing with data (or equally in other kinds of computing) goes on as if it were not true. So before trying to outline some key concepts for languages, we need to examine some counter-indications.

The most obvious examples are *GUI*s: all those buttons, sliders, menus, flying icons, waving elves and other visual aids to computing. Don't they imply that for a large fraction of users, formal languages are one or more of irrelevant, incomprehensible, or not worth the effort? The busy executive, the non-specialist from another field, the non-technical person just needing a few simple answers—these users are much likelier to interact with a non-threatening, step-at-a-time visual interface than they are to learn the rules of a language. Even those of us who may be inclined to program and not particularly frightened by a little formal notation may still feel inclined to use a visual interface when operating outside the range of our usual computational tasks.

The point-and-click, drag-and-drop, and generally non-verbal approaches to computing must be accepted as increasingly dominant, particularly for non-technical audiences. (And most particularly for the ”third society” of those who intensively use technology but generally without thinking about the underlying computations.)

This recognition is *not*, however, a reason for either despair or for downplaying the role of language in computing with data. The role changes, but if anything the importance of language, and especially of good language increases in the new context.

1. Non-verbal systems are worth much more if well-based on a language.

2. Languages play multiple roles in the new context.

Base the GUI on Clear Language Definitions

An interactive, non-verbal system for computing with data justifies itself by providing its users with an easy, effective way to interact with the application at issue. If we, as statisticians or others with some interest in the techniques being used, feel some uncertainty, the qualms are likely to be with the quality of what the user has learned. What in effect was the analysis implied by a particular session of pointing, dragging, sliding, etc.? Does the end result communicate a reasonable message to the user? What if the user really meant to ask something else, perhaps not easily “programmed” in this non-verbal form?

All of these questions will be better handled if the non-verbal actions map simply and directly into a good language, particularly one that is object-based and object-oriented, in senses made more specific on page 13. Just as a simple example to make the ideas concrete, suppose the user selects a button looking like Figure 4. We would probably guess that this produces a linear fit to some data. (That’s our professional bias showing; other audiences might have some other interpretation, but suppose we’re right in this case.)

The result is very context-sensitive; that is, the action performed in response to the event of selecting the button depends on both the button definition directly and on the sequence of previous events in the user’s session with the system. Most

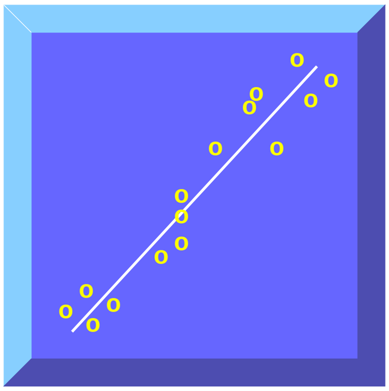

Figure 4: *A button from a hypothetical graphical user interface.*

importantly, what the user *perceives* as the result of the action depends on this context. Something happens to the state of the user's interaction with the system; perhaps the user sees some immediate visualization of the event such as a line on one or more scatter plots, a table summarizing the fit, at the least a message that computations were performed. Then future interactions with the system may depend on this event; the user may call up other displays, or further analytical buttons may use the results.

If, let's say, the effect of selecting the button is to perform a linear regression, then we're asking for the definition of that action. We can only satisfactorily answer any of the questions we raised if both the direct action in response to the button and also the relevant "state" we talked about have some clear definition. That clear definition can come from a language supporting the computing with data. If the language can represent clearly the regression action itself and also (this is the really important part) the objects that define the context of the regression, then we can say what the user's analysis means in a clear way. We can then hope to make some guesses about how meaningful the results would be (for example, did the user's previous actions create a dataset for the regression that might likely be misleading because of selections of variables or observations). Most of all, if the action represented by the sequence of user interactions has a clear language definition, the way is open to *modify* that action if it should be changed.

So the argument is that a clear language definition for the actions corresponding to a non-verbal interactive system greatly improves the quality of the system. I would also argue that the implementation of the system benefits as well, in a fairly obvious way. The development of the system, its debugging, and the process of refining it to be more useful will all require the developers to understand the system and the effects of various modes of user interaction with it. Developers will also benefit from the simplicity of changing the system. In both respects, basing the system on a good language and environment for programming with data will benefit, while programming the system in a low-level, obscure implementation will correspondingly defeat attempts to understand what's going on and will discourage efforts to change what needs changing.

Multiple Language Roles for GUIs

Issues of computing with data, however, are only part of the language role in any such non-verbal system. The presentation (*rendering* in computer graphic terms) of the relevant information to the user and the management of user-interaction events defines the user's actual view and feel of the system. Obviously, the quality of rendering and interaction greatly affect how happy the user will be with the system. With our background in mathematics and science, we may tend at first to minimize the importance of such questions. Any attempt to create a high-quality GUI should quickly disabuse us of that tendency, but may also generate a strong desire to have someone else do all the GUI programming.

Whoever does program the rendering and interaction, however, the *interface* between that programming and the computing with data is an essential part of the overall system design. The danger is sometimes that lack of a good way to program this interface causes the programming to be done in an language or an environment that is good for one side of the interface at the cost of the other side. To be specific, there is a temptation to either add the GUI programming to the statistical system (usually producing a clumsy and/or ugly result), or to program the system in a visual language with crude or inflexible interface to the computing with data.

The good news here is that current developments in programming environments promise a better approach. Programming environments that provide good support for GUI programming (Java, for example) can exist in a distributed system with environments that provide good support for computing with data. The interfaces between such systems can be defined in clear, object-based terms and can be implemented conveniently by the programmer.

References

Anscombe, F. (1981), *Computing in Statistical Science through APL*, Springer-Verlag.

Buhler, R. (1965), P-stat; an evolving user oriented language for statistical analysis of social science data, *in* 'Fall Joint Computer Conf.'.

Chambers, J. M. (1967*a*), Self-defining data structures for statistical computing, Technical report, Bell Laboratories. (also a report to the British Working Party on Statistical Computing).

Chambers, J. M. (1967*b*), 'Some general aspects of statistical computing', *Appl. Statist.* **17**, 124–132.

Chambers, J. M. (1968), 'A British working party on statistical computing', *Amer. Statistician* **22**(April), 19–20.

Chambers, J. M. (1998*a*), Computing with data: Concepts and challenges, Technical report, Bell Labs. (`http://cm.bell-labs.com/stat/doc/Neyman98.ps`).

Chambers, J. M. (1998*b*), *Programming with Data: A Guide to the S Language*, Springer-Verlag.

Cooper, B. E. & Nelder, J. A. (1967), 'Prologue *and* epilogue', *Appl. Statist.* **16**(2), 88 *and* 149–151. (Pages 89-148 contain papers and discussion from the December, 1966 meeting.).

Dixon, W. J. (1964), *BMD: Biomedical Computer Programs*, Health Sciences Computing Facility, University of California, Los Angeles.

Gabbe, J. D., Wilk, M. B. & Brown, W. L. (1965), An analytical description of Telstar 1 measurements of the spatial distribution of 50-130 Mev protons, Technical report, Bell Laboratories.

Hahn, G. & Hoerl, R. (1998), 'Key challenges for statisticians in business and industry', *Technometrics* **40**, 195–213. (with discussion).

Hamilton, G., Cattell, R. & Fisher, M. (1997), *JDBC Database Access with Java: A Tutorial and Annotated Reference*, Addison-Wesley.

Iverson, K. E. (1962), *A Programming Language*, Wiley.

McCarthy, J. (1962), *Lisp 1.5 Programmer's Manual*, M.I.T. Press.

Nelder, J. A. (1966), General statistical program (GENSTAT IV): User's guide, Technical report, Waite Institute, Glen Osmond, Australia.

Ousterhout, J. K. (1998), 'Scripting: Higher level programming for the 21st century', *IEEE Computer*. (web version at `http://www.scriptics.com/people/john.ousterhout/scripting.html`).

Pope, A. (1998), *The CORBA Reference Guide*, Addison-Wesley.

Tierney, L. (1990), *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley.

Tukey, J. W. (1965), An approach to thinking about a statistical computing system, (unpublished notes circulated at Bell Laboratories).

Wilk, M. (1965), Introductory remarks, (unpublished notes for a meeting).