Journal of Statistical Software

February 2005, Volume 13, Issue 9. http://www.jstatsoft.org/

Some Notes on the Past and Future of Lisp-Stat

Luke Tierney
University of Iowa

Abstract

Lisp-Stat was originally developed as a framework for experimenting with dynamic graphics in statistics. To support this use, it evolved into a platform for more general statistical computing. The choice of the Lisp language as the basis of the system was in part coincidence and in part a very deliberate decision. This paper describes the background behind the choice of Lisp, as well as the advantages and disadvantages of this choice. The paper then discusses some lessons that can be drawn from experience with Lisp-Stat and with the R language to guide future development of Lisp-Stat, R, and similar systems.

Keywords: Computing environment, dynamic graphics, programming languages, R language.

1. Some historical notes

Lisp-Stat started by accident. In the mid-1980's researchers at Bell Labs (Becker and Cleveland 1987) started to demonstrate dynamic graphics ideas, in particular point cloud rotation and scatterplot matrix brushing, at the Joint Statistical Meetings. Their proprietary hardware was not widely available, but the Macintosh had been released recently and seemed a natural platform on which to experiment with these ideas. MacSpin (Donoho, Donoho, and Gasko 1988) was developed as a tour de force of what can be done with point cloud rotation and became a successful commercial enterprise for a while. I was interested in going beyond point cloud rotation alone and experimenting with linked brushing along the lines of the interactive scatterplot matrix as well as linking of other displays. I started by implementing scatterplot brushing as a stand-alone Macintosh program. This worked well but quickly raised the issue of how to more easily prepare data for display. Integrating the stand-alone program into a language that could be used for data manipulation seemed like a natural next step. I happened to notice an early version of David Betz' XLISP interpreter posted on `usenet` at the time, and that seemed like a reasonable place to start.

The idea of using a full programming language as the basis for a statistical system was not

new. The *Brown Book* version of the S language (Becker and Chambers 1984) had become quite popular in the academic community and the *Blue Book* version, the New S Language (Becker, Chambers, and Wilks 1988), was under active development. The basic functionality provided by S of the time seemed a reasonable starting point and motivated much of the statistical functionality included in Lisp-Stat. Just as research interests at Bell Labs led to the inclusion in S of certain functionality that, at least at the time, was viewed as non-standard, my interests in Bayesian computing motivated including some tools for approximate Bayesian Inference in Lisp-Stat.

One difficult decision I had to make was whether to build my system around the Lisp language itself, i.e. have Lisp be the language of user interaction with the system, or whether it would be better to implement a more traditional syntax, perhaps one based on the S language. I experimented with a front end based on S-like syntax but found that certain things would still have to be programmed in Lisp. This closely mirrored my experience with the Macsyma symbolic mathematics system, where many basic operations could be coded in the more traditional infix Macsyma language but more sophisticated tasks required moving to the underlying Lisp level. This dual language situation seemed less than ideal, so I decided to see if a single language approach based on Lisp could be made to work.

2. Lisp as a language for data analysis

Incorporating a high level language within a data analysis framework allows for basic operations such as arbitrary data transformations. It also allows one to easily apply one analysis to results, such as residuals, from another analysis, and it allows a complex sequence of analysis steps to be abstracted into a unit that can be easily applied to new data. In addition, such a language provides a framework for developing new methods ranging from minor variations useful for a particular analysis to entirely new algorithms.

Most advanced data analysis environments include some form of language for expressing transformations or repeated analyses. In many cases these languages have been developed from scratch. In other cases existing general purpose languages have been adapted and extended to support statistical computations. Basing the language for a statistical environment on a high level general purpose language has a number of advantages. Such languages have a literature of their own, providing general introductory material to the language as such that can be used as a reference for statistical users. General purpose languages also have often gone through a more extensive design process, or at least design evolution, that can lead to a cleaner language than is likely to arise in a language developed from scratch. General purpose languages also tend to have a range of implementations, often including implementations based on compilation.

APL is one general purpose language that has been used as the basis for statistical computing environments (Anscombe 1990, e.g.); another is the Lisp language. Lisp-Stat was by no means the first effort to build a statistical computing environment on the Lisp language. Other efforts include work of McDonald and Pedersen (1988), Oldford and Peters (1988), Stuetzle (1988), and Buja, Asimov, Hurley, and McDonald (1988), among others. Most of these were originally developed on Lisp machines and not widely distributed, but all made important contributions to the design of statistical languages.

The Lisp language, in particular Common Lisp (Steele 1990), offers many advantages as the

basis of a language for data analysis. As a result of many years of refinement the language is powerful yet has a very clean design. Lisp supports high level programming in which succinct programs can express very elaborate computations. Nevertheless, the evaluation model is such that resource usage is easy to predict and control, even in the presence of garbage collection, and it is thus possible to write programs that are efficient in memory usage and computational speed. Some implementations of Common Lisp provide compilers capable of producing code that is competitive with optimized C and Fortran compilers. Lisp also provides excellent support for developing domain specific languages through the definition of suitable sets of macros (e.g., Graham 1994; Norvig 1992). Finally, Lisp is considered one of the best general purpose languages for *experimental programming* in which code is continually modified to address evolving problem understanding. Experimental programming is thus very similar to the kind of computing needed to support good data analysis.

Lisp also has some drawbacks—some perceived and some actual. The Lisp language has been in existence for many years and a substantial folklore has grown up around it. Some of the myths about Lisp include that it is inherently slow and memory hungry. Both may have been true at one point in time, but that time is long past. High performance Lisp systems are competitive with compilers for C and Fortran, and memory requirements of well written Lisp programs are on the same order as requirements for comparable C programs using manual memory management. Lisp surface syntax is noticeably different from the surface syntax of most other languages. This can be an impediment for users of more traditional languages who want to learn Lisp. It need not be an impediment for novice users unless they are told to perceive it as such. Unfortunately there are many statements in the literature and on web pages that tell potential users that Lisp syntax is hard to learn, and this becomes a self-fulfilling prophecy. A careful examination shows that, for most tasks, Lisp syntax is little different from procedure call syntax used in C or Fortran. The only point where there is a significant difference is in arithmetic expressions and in array indexing. In these areas Fortran or S language syntax is closer to standard mathematical notation and hence more natural to most scientists.

While surface syntax is the most obvious way in which Lisp differs from languages such as C or Fortran, there are also differences in the programming style that is most effective. Most good Lisp programs are developed from the bottom up by designing and developing small functions that solve specific problems and then building additional layers of functions on top that work towards solving the larger problem. This style is not unique to Lisp. Most high level languages, including the S language, are best used in this functional style. The familiar syntax of the S language can make C style programming seem reasonable, and one can write programs using this style, but the resulting programs are not very efficient and they are often as difficult to debug and maintain as their C counterparts. Except for surface syntax the R dialect of the S language in particular is closer to Common Lisp than it is to any other language. The primary semantic differences are lazy evaluation of function arguments and atomic vector data.

One criticism of Common Lisp that is relevant in the present context is that Common Lisp on the one hand includes an enormous amount of functionality that in other languages is placed in libraries, and on the other hand does not provide a well defined mechanism for extending the language with domain-specific libraries. The power to create an infrastructure to support such extensions is certainly present in Common Lisp, but there is no standard mechanism in place. Java, Perl, and Python have all been designed to have a mechanism for managing a

set of standard libraries, and that same mechanism can be used for developing and managing new functionality. A major reason for the success or R is the package mechanism that is used to provide both a standard library of packages and for managing a now huge collection of user supplied packages available from CRAN (R Core Development Team 2004) and other repositories.

For a number of reasons Lisp's popularity has seen a decrease over the past decade. The number and economic viability of Lisp vendors has declined significantly while the price of high performance Lisp systems has remained prohibitive. Some high quality open source implementations are available, notably CMUCL, but the prospects for the future of these implementations are unclear.

3. Object-oriented programming

During the 1990's object-oriented programming came into the mainstream, or perhaps even became the mainstream, of programming practice. The object-oriented approach is particularly well suited to user interface design but is also valuable in many other areas of programming. It's relevance to statistical computing is hardly surprising given its roots in the simulation language Simula.

Many approaches to the design of a system for object-oriented programming are available. Systems can be classified according to whether they are pure in the sense of all values being objects, whether they permit single inheritance or multiple inheritance, and whether they use a message-sending single dispatch approach or generic functions and multiple dispatch. Another possible variation is to use a prototype-based approach or an approach based on formal classes and instances.

During the development of Lisp-Stat the Common Lisp Object System CLOS was in final stages of standardization. Lisp provides an excellent framework for experimenting with different approaches to object-oriented programming, and many different experimental systems were developed during this period. After some research and experimentation I decided to use a prototype-based system allowing multiple inheritance but based on single dispatch.

The use of a prototype based system was motivated by ideas from the Self language in particular (Smith and Ungar 1995), but also by the object system used in early versions of Coral Common Lisp on the Macintosh. The prototype-based approach is well suited to a usage model in which there is no formal separation between programming and use. An interactive plot, for example, can be assembled based on one particular data set, and new plots can then be constructed as being like the original except for the particular data used. A prototype approach is also well suited to programming by direct manipulation. Self and Omega (Blaschek 1994) provide examples. While the prototype approach has not been widely adopted it remains a focus of active research (Noble, Taivalsaari, and Moore 1999).

Many object-oriented languages provide some form of multiple inheritance. A language allowing multiple inheritance is more expressive but also more complex than one restricted to single inheritance. The complexity that is added is that the ancestor precedence relationships can become ambiguous. Different languages have taken different approaches to address this issue, usually involving some arbitrary decisions to induce a linear ordering among ancestors. In most situations these approaches behave reasonably, but most approaches lead to surprising and sometimes undesirable results in some unusual settings. The difficulty in arriving at

satisfactory solutions is illustrated by the history of developing an approach to linearize the ancestors in the Common Lisp Object System and Dylan (Barrett, Cassels, Haahr, Moon, Playford, and Withington 1996). Lisp-Stat uses an approach based on the linearization algorithm used in CLOS.

Like multiple inheritance, multiple dispatch is strictly more expressive than single dispatch. However, in contrast to multiple inheritance, which can essentially be ignored in situations where only single inheritance is needed, multiple dispatch represents a significant conceptual change since it means one can no longer think of there being a single object that is receiving a message. As a result, languages supporting multiple dispatch usually use a generic function model in which methods become associated with their generic functions, and method selection can be based on one or more arguments. In addition to this added conceptual complexity, there are again potential ambiguities that need to be resolved and the implementation of efficient dispatch becomes significantly more difficult. Successful design of a language supporting multiple dispatch requires a careful balance of achieving reasonable semantics while at the same time allowing for an implementation with satisfactory performance. Kiczales, des Rivières, and Bobrow (1991) discuss these issues at length in the context of CLOS.

As a result of these considerations there are far fewer languages that support multiple dispatch. Common Lisp and Dylan may be the only general purpose languages with a significant user base that support multiple dispatch. Cecil (Chambers 1995) is a research language based on multiple dispatch and prototypes that also incorporates other interesting features such as predicate-based dispatch. Predicate based dispatch means methods can be defined for objects, or sets of objects, that satisfy a predicate. A simple example is a method for a square matrix, i.e. a matrix for which the number of rows equals the number of columns. A system supporting predicate dispatch would allow a method for a square matrix to be defined without the need to introduce a formal square matrix subclass. Millstein (2004) discusses some recent work in this direction.

The S4 formal class and generic function mechanism for the S language introduced in Chambers (1998) supports multiple dispatch, multiple inheritance, and several other advanced concepts. A new implementation that addresses some shortcomings of the original approach is available in the **methods** package for R. The design is very ambitious and potentially very powerful, but the intended semantics are not yet sufficiently clear to allow a careful assessment of the design or to be sure that a correct and efficient implementation is feasible. This is an area of continuing research.

One issue with object-oriented languages in general and perhaps the generic function approach in particular is that developers who use a set of generic functions originally designed in a particular context expect these functions to cooperate in certain ways. A simple example is the expectation that if `length(x)` returns 10 then `x[i]` will do something reasonable for `i=1:10`. Most languages do not provide a way of registering or documenting this sort of collaboration expectation, never mind checking for it. Java interfaces address this to a degree, but at the cost of requiring a static type system. This is an issue that merits further investigation.

The Java language has become one of the most used languages in introductory programming courses. It is nearly pure, and becoming more so with the most recent language revision. It is class-based and supports only single inheritance and single dispatch. This rather limited design has the benefit of simplicity, both for implementation and for user understanding, but

it does represent a significant restriction over more advanced object-oriented languages. A benefit of the popularity of Java and of C++ is that object-oriented programming is now a familiar term. A drawback is that there is a wide-spread mind-set that only the form of object-oriented programming supported by Java is "true" object-oriented programming. This unfortunate closing of minds is reflected in the object-oriented design and analysis literature which has unified around the UML approach (Object Management Group 2004). It is nearly impossible to find references to other approaches, such as CLOS, in this literature; in contrast, earlier literature was more open to alternatives (e.g., Booch 1995).

As with the choice of basic programming language, a statistical language needs to balance the benefit of expressive power against simplicity and familiarity in choosing an appropriate object-oriented programming infrastructure.

4. Graphical user interfaces and dynamic graphics

A major objective of the design of Lisp-Stat was to provide a framework for experimenting with dynamic graphics ideas. The design of the graphics system was guided both by limitations of the time and by a desire for simplicity. One example of the limitations of the time is color. Color was not available in the early stages of Lisp-Stat development and, once it became available, the number of colors that could be used was limited and available colors needed to be managed carefully for a number of years thereafter. Current hardware provides nearly unlimited color availability as well as options such as alpha blending. A revision of the Lisp-Stat color support would need to take these new capabilities into account.

Another area where technological change has had an impact is in animation speed. With current hardware most graphs can be rendered fast enough to require a pacing mechanism to insure that each rendered frame stays on screen for an appropriate length of time. The animation model used in Lisp-Stat did not allow for this pacing as it was not needed on hardware of the late 1980's and early 1990's. The current release of XLISP-STAT has some `pause` commands inserted at appropriate places to insure, for example, that point cloud rotations behave reasonably, but a revised model based on a program controlled frame redraw rate is needed to handle this properly.

Yet another area in which hardware has improved significantly is in support for 3D graphics. This also would need to be taken into account in a revision of the basic graphics system.

One example of a case where the Lisp-Stat design chose simplicity over flexibility is in the identification of plots with the top level windows containing the plots. This allows a single object to be used as the target of operations changing the content of a plot and also as the target of commands for positioning and sizing the window on the screen. An alternate design would have allowed for nested plot frames within top level windows. With more recent concepts in object-oriented design it might have been possible to develop a scheme that preserves this simplicity in most instances but nevertheless allows the flexibility of separating content from top level frame. This should be investigated in a revision of the Lisp-Stat graphics system.

The Lisp-Stat graphics system has been reasonably successful in providing a framework for statistical researchers to experiment with new ideas. This is shown by larger projects such as the Arc code of Cook and Weisberg (1999) and ViSta (Young and Bann 1996), and by smaller examples, such as an enhanced linking mechanism described in Tierney (1996) and

many others. The system has been less successful in providing data analysts with a framework for quickly building a custom interaction appropriate for a particular data analysis context. A system that allows custom interactions to be built from a graphical, direct-manipulation interface would in many cases be a better approach. The current graphics system does provide sufficient tools in principle for developing such an interface, but the design of an effective visual language, even with a fairly limited domain, is a daunting task and remains an opportunity for future research.

Both Arc and ViSta make extensive use of the menu/dialog facilities of Lisp-Stat as well as basic graphics primitives to develop a graphical user interface for the system. Through this interface users can use standard graphical interactions to request computations without the need to be familiar with, or even aware of, the underlying programming language. This significantly lowers the entry barriers for new and casual users, though at the cost of making the full power of the system less accessible. This tension is found in many statistical systems, and a long term goal of research on statistical environments is to produce a framework in which a smooth transition form a comfortable and familiar graphical interface for simple operations to the full power of the underlying system is possible.

While current user interface design is still far from this goal, it is quite adequate for settings in which sophisticated analyses or tools for carrying out specific analyses are to be delivered to end users who have no need for the full power of a data analysis system. Such analysis applets, as one might call them, can be constructed with the basic Lisp-Stat infrastructure, but further enhancements of that infrastructure would make this easier and more effective. For such applets to be immediately usable they need to fit in with the computational framework familiar to their users. This means providing a standard native look and feel and taking advantage of standard forms of user interface items, such as tabbed dialog boxes or toolbars with tooltips. Direct manipulation interface construction tools would be very valuable in this context, though constructing tools that can adequately handle differences between the major user interface flavors, such as Windows and MacOS, is a significant challenge.

5. Challenges for Statistical Languages

Lisp-Stat has been used in a number of innovative ways over the years. Recently R has become the primary open source statistical environment, with an explosive growth in use and in user contributed software. Experience with the use of Lisp-Stat and R has shown that a programmable system can and will be used in ways that may surprise a system's designers. Examining these uses can help in guiding system evolution and in the development of new systems.

5.1. Language design and programming infrastructure

Language-based statistical systems provide users who have mastered the system with a powerful tool for sophisticated data analysis that allows data analysis tools to be adapted to a particular problem instead of having to force a problem into a fixed menu of available approaches. Language-based systems also allow the development of new statistical methodology. Mastering a language-based system involves learning a computing language, which requires a certain investment of time and effort. In some situations, such as use in an introductory statistics class or for providing a particular analysis to non-statistical scientists, this invest-

ment may not be appropriate. For these situations it is useful to be able to provide a simple graphical interface. A major open question is whether such a simple interface can be provided while at the same time making available the full power of the language. Showing the corresponding language commands as operations are performed using menus and dialogs is one approach taken, for example, in the **Rcmdr** package (Fox 2004). Other approaches worth exploring range from developing automated graphical interface generation tools to graphical programming languages in which programs are designed by direct manipulation interfaces (Burnett, Goldberg, and Lewis 1995). Developing a complete graphical version of Lisp-Stat or R, or developing a new graphical language of comparable power, is a daunting task; a language for more limited domains, such as graphics or model construction, may be more attainable. The graphical interface to model construction provided in WinBUGS (The **BUGS** Project 2004) is one example of a possible approach.

For developers of new statistical methodology good language design and support tools can significantly enhance the development process. The high level language itself is the most important component in the development framework. A well designed high level language can make the developer much more effective by allowing a few simple language constructs, such as a vectorized arithmetic expression or a built-in matrix operation, to succinctly express very complex computations. This feature alone tends to greatly reduce the development effort compared to using a more traditional low level language. Additional tools that can increase developer productivity are good debugging frameworks and testing harnesses.

One of the reasons R has been extremely successful is that it provides a very good mechanism for developing and organizing add-on packages. The basic R package mechanism provides tools for building packages: with a single command R code is pre-processed, C or Fortran code is compiled into dynamically loadable libraries if necessary, and different forms of documentation are created. R also provides a package checking mechanisms that runs all examples in the documentation, along with additional test code if provided, and also checks that all visible functions in a package are documented, and that basic usage documentation for these functions is consistent with their definitions. More recent developments include mechanisms for discovery of packages available on the internet as well as a name space mechanism for managing which definitions in a package are intended to be publically visible and which are intended to be private to the package.

Basic profiling tools already available for R provide a useful means of assessing where performance bottlenecks are located. Enhanced tools that allow more detailed viewing of profiling information are currently under development. Work in progress for R also includes code analysis tools to look for common coding errors and inconsistencies, and for byte code compilation to improve performance. Byte code compilation and the associated code analysis for Lisp-Stat proved to be valuable both for improving performance and for detecting errors and thus making the code more robust. Preliminary work suggests that similar results will be achieved for R (Tierney 2001).

Basic language design can have a significant effect on how easy it is to develop high performance code. One design feature of the S language family, and R in particular, that makes it difficult to predict and control performance is the decision to make all vectors atomic in the sense that there are no R language level operations for destructive modification. Thus after the sequence of expressions

```
x <- c(1,2)
```

```
y <- x
x[1] <- 3
```

the variables x and y have different values. In contrast, after the seemingly analogous Lisp expressions

```
(setf x (list 1 2))
(setf y x)
(setf (elt x 0) 3)
```

the variables x and y will have identical values. The R approach has the benefit of eliminating a class of possible errors caused by inadvertently modifying data that is shared by several data structures, though experience with Lisp-Stat indicates that this error occurs very rarely in code using the functional style encouraged for both Lisp-Stat and R. But the cost is a potential drop in efficiency since the R approach requires copying the entire vector. R tries to avoid unnecessary copying by keeping track of some sharing information and internally performing a destructive modification when no sharing is possible. However, certain operations can make it impossible to guarantee that destructive modification would be safe, and in these cases data structures must be copied. The cost of unnecessary copying is often negligible in computations involving small amounts of data, but for larger data sets it can be prohibitive. Unfortunately it is very difficult for a developer to predict in what circumstances copying will occur, and as a result it is difficult to accurately predict the cost of basic operations in different circumstances. The Lisp approach allows a much simpler evaluation model in which accurate performance prediction is possible. An open question is whether it is possible to develop a language that combines Lisp's predictable performance behavior with R's protection against unintended modification. Allowing data structures to be created modifiable for initialization and then locked against modification once initialized is one approach that may be worth investigating. The *freezing* mechanism in Ruby (Matsumoto 2001) is an example of this approach.

A simplified and more predictable language semantics can also aid in the development of a compiler for further improving performance. R is an extremely dynamic language, and this flexibility, while very valuable at times, severely limits the amount of optimization a compiler can legitimately carry out. Allowing some aspects of a program to be frozen, for example allowing certain local variables to be declared immutable once initialized, can help both in clarifying the intent of the code and in allowing a compiler to take advantage of this knowledge in performing optimizations.

Experience with Lisp-Stat and R has shown the importance of an expressive language but also the value of language features that aid in verifying the correctness and predicting the performance of code. These experiences will be valuable as these languages evolve and new languages are developed.

5.2. Managing long-running computations

On hardware available today many statistical computations are virtually instantaneous. But some computations can take minutes, hours, or even days. This range of possible task durations raises a range of issues for a statistical computing environment.

For computations taking tens of seconds to several minutes it is important for an environment to provide some indication that a computation is in progress. Ideally, there should be

some means of inspecting the progress of the computation or the ability to carry out other tasks while the computation completes. A minimal requirement is to be able to interrupt a potentially long running or even non-terminating computation without losing data. An environment with a graphical user interface should ideally be able to at least perform housekeeping tasks, such as updating windows, while a computation is in progress. To adequately manage these sorts of concurrent activities requires some form of thread support.

Multi-threading is not about parallelism per se but rather about the disciplined management of conceptually concurrent activities. To reduce the time of long running computations requires true parallelism through the use of multiple processors. Dual processor workstations are now quite widely available, but few statisticians currently have regular access to facilities with significantly more than two shared memory processors. Shared memory multi-processing can be used in two ways in a statistical computing environment based on a high level language: by allowing multiple high level language threads to run in parallel or by allowing primitive vector and array operations to spread their work across several processors. Both are worth exploring. Several threaded BLAS libraries are available and can be used to improve performance of linear algebra computations (ATLAS Project 2004; Goto 2003). These ideas could be extended to all built-in vectorized operations.

To achieve significant speedups through parallelism requires a large number of processors. These are most likely to be available as a network of workstations or possibly in a computational grid (Foster and Kesselman 1998). These processors will not share memory and will need to be accessed using a message passing approach to parallel computations. Interfaces from R to the two main message passing libraries, PVM and MPI, are available (Li and Rossini 2001; Yu 2002). Higher level approaches that are available include SNOW (Rossini, Tierney, and Li 2003) and Parallel-R (Samatova 2004).

Enabling distributed parallel computation within a high level statistical system raises some interesting challenges that have not yet been fully resolved. One example is allowing a user to interrupt a parallel computation that has been started in a way that properly terminates all remote computations and allows a new parallel computation to be scheduled successfully. Another issue is how to deal with the range of different errors that can occur, ranging from errors signaled on some nodes within the high level language to hardware failures and network disruptions. These are already significant issues for computations on a cluster or network of workstations; they will increase in importance as parallel computations are scaled up to the grid level.

5.3. Data size, data storage, and data acquisition

Data set sizes can vary considerably among areas of applications. Very large data sets will need specialized algorithms and handling. In many cases storing the data in a data base and working on selected subsets of the data is the best approach. To support this approach it is important to have tools to conveniently communicate with data bases.

Some problem areas, microarray data analysis for example, lead to data sets that, while not huge, push the limits that can be handled by the address space of commodity 32-bit processors. 64-bit processors are now readily available at competitive prices, and most statisticians investing in new hardware would be well advised to choose a 64-bit machine. Both R and XLISP-STAT can be compiled on standard 64-bit configurations, though Win64 with the convention that a C `long` is 32 bits will cause problems for XLISP-STAT and perhaps for R as

well. Many commercial and public domain Lisp systems will need significant rewriting because assumptions about a 32-bit word size have been built into the memory management systems.

Several difficult low level design issues are raised by increases in accessible memory brought about by a move to 64-bit hardware. The internal representation of integers used by R for example, is 32 bits. This limits the size of a single vector object to $2^{31} - 1$ elements. Some careful thought and a significant amount of code rewriting will be needed to increase this limit. The obvious solution of using a 64-bit integer representation has the drawback that the largest representable integers can no longer be represented exactly as double precision floating point numbers, an assumption that underlies a number of operations. A possible option to explore is to use double precision floating point numbers to represent both integers and reals, with a marker on the vector object to distinguish intent. Limiting the permitted range of integers to $\pm 2^{53} \approx 10^{16}$ will then preserve the ability of all integers to be represented exactly in double precision floating point.

In addition to larger data sets, distributed data sets are becoming more common. Data stored at data acquisition sites that is only accessed as needed, perhaps via summaries prepared at the storage sites, is one example. Distributed meta-data, such as data on genes stored in online repositories, is also becoming increasingly important. A high level statistical language needs to provide useful primitives for network access and powerful error handling mechanisms for managing the wealth of errors and other unusual conditions that can arise in a distributed network environment. These primitives can then serve as the building blocks for higher level functionality that makes effective use of distributed data and meta-data.

Integration with persistent storage systems is another area where further work is needed. Both R and Lisp-Stat assume that working data is maintained in memory, but both require a form of specialized storage for preserving data across sessions. This storage mechanism is also used for managing the definitions of system functions. In R the mechanism for serializing objects into a stream of bytes for storage has been made explicit and has been used to allow the implementation of a deferred loading mechanism for function definitions (Ripley 2004). This can be viewed as an ad hoc persistent storage interface. A more formal structure in which different back end data bases can be used individually or in combination is worth exploring. A careful design, integrated well with existing SQL data base access approaches, could lead to a significant improvement in both maintainability and usability. Experience with the Root system (Brun and Rademakers 2004) may provide valuable guidance in this work.

New data structures may also help in developing programs that work well with distributed or large local data collections. Generators and iterators as provided by Ruby and recent versions of Python, together with enhancements of R's `for` loop and `apply` functions, may be a step in the right direction for R.

5.4. Dynamic graphics and graphical user interface support

The extensible dynamic graphics system remains the most important and distinctive feature of Lisp-Stat. Some interesting attempts have been made to add dynamic graphics support to R (Urbanek and Theus 2003; Lumley 2001) but to date no significant effort has been made to develop a framework to support the development and implementation of new dynamic graphics ideas.

One possible approach to adding programmable dynamic graphics to R is to interface R to XLISP-STAT itself or to the internal graphics engine. This may be feasible, in fact a pre-

liminary approach exists in the Omegahat project, but, as mentioned earlier, the Lisp-Stat dynamic graphics system, while basically sound, is showing its age. A major revision is needed that addresses advances in available color technology and availability of 3D hardware support. Higher level additions that are needed include support for frame rate control in animations, an infrastructure to provide effective support for representations of and interaction with aggregates, such as histogram bars or hexagonal bins, and support for multiple plots within windows. It may also make sense to revise the basic coordinate systems to use floating point representations and standard axis directions throughout, rather than integer screen coordinates for the lowest level.

The original XLISP-STAT graphics system was designed with use from other systems in mind, and a package called **iviews** based on that system that implemented linked plots was made available for S and the SunView interface on Sun workstations. A revision of the system should again be based on the goal of allowing use from within multiple systems to the greatest extent possible. The number of competing window systems is currently down to three: Windows, X11, and MacOS; nevertheless developing and maintaining three separate systems is a challenge. It is unfortunate that no completely satisfactory open source cross platform graphical user interface toolkit has emerged yet. Many do a good job on two of the three platforms but are less than satisfactory on the third.

6. Future directions

R has become the major focus of work on open source statistical software, both in terms of design and development and in terms of use. Lisp-Stat remains in use among a smaller community, in particular for its strengths in dynamic graphics.

One of the features of R that seems to be attractive is the surface syntax, which many users find more familiar and comfortable than Lisp syntax. This is a phenomenon that seems to go well beyond statistics, with newer texts on artificial intelligence, once essentially the exclusive domain of Lisp, now branching out into other languages such as Python (Russell and Norvig 2002). This is an unfortunate development. While R and Lisp are internally very similar, in places where they differ the design choices of Lisp are in many cases superior. The difficulty of predicting performance and hence writing code that is guaranteed to be efficient in problems with larger data sets is an issue that R will need to come to grips with, and it is not likely that this can happen without some significant design changes.

In my original design of Lisp-Stat I decided to avoid a two-language system by relying on Lisp both as the implementation language and the user language. It may now be time to revisit this approach. A goal I have been exploring in recent years is to factor out core functionality for statistical computation, including basic interactive graphics support, into an abstract statistical machine, analogous to the virtual machine structure of Java. Like the Java VM, this machine would be defined in terms of a form of assembly language, and higher level languages would then be compiled to this machine, either in advance or by a just-in-time strategy. Initially both R and Lisp-Stat would be supported as high level languages, thus allowing code written in both languages to inter-operate. But separating core functionality from language in this way should allow further experimentation with language design and the development of new and improved languages, while maintaining the ability to take advantage of the valuable code bases that have been developed over the years.

Acknowledgements

Work supported in part by NSF grant DMS 03-05226 and NIH grant 1R33 HG02708-01A1.

References

Anscombe FJ (1990). *Computing in Statistical Science through APL*. Springer-Verlag New York, Inc. ISBN 0387905499.

ATLAS Project (2004). "Automatically Tuned Linear Algebra Software (ATLAS)." World Wide Web. URL http://math-atlas.sourceforge.net/.

Barrett K, Cassels B, Haahr P, Moon DA, Playford K, Withington PT (1996). "A Monotonic Superclass Linearization for Dylan." In "Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications," pp. 69–82. ACM Press. ISBN 0-89791-788-X.

Becker RA, Chambers JM (1984). *S: An Interactive Environment for Data Analysis and Graphics*. Duxbury Press.

Becker RA, Chambers JM, Wilks AR (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth Publishing Co Inc.

Becker RA, Cleveland WS (1987). "Brushing Scatterplots." *Technometrics*, **29**, 127–142.

Blaschek G (1994). *Object-Oriented Programming with Prototypes*. Springer-Verlag, Heidelberg.

Booch G (1995). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition edition.

Brun R, Rademakers F (2004). "The ROOT System Home Page." World Wide Web. URL http://root.cern.ch/.

Buja A, Asimov D, Hurley C, McDonald JA (1988). "Elements of a Viewing Pipeline for Data Analysis." In WS Cleveland, ME McGill (eds.), "Dynamic Graphics for Statistics," pp. 277–308. Wadsworth Publishing Co Inc.

Burnett MM, Goldberg A, Lewis TG (eds.) (1995). *Visual Object-Oriented Programming*. Manning, Greenwich, CT.

Chambers C (1995). "The Cecil Language Specification and Rationale: Version 2.0." Technical Report.

Chambers JM (1998). *Programming with Data: A Guide to the S Language*. Springer-Verlag Inc.

Cook RD, Weisberg S (1999). *Applied Regression Including Computing and Graphics*. Wiley, New York.

Donoho AW, Donoho DL, Gasko M (1988). "MacSpin: Dynamic Graphics on a Desktop Computer." *IEEE Computer Graphics and Applications*, **8**(4), 51–58. ISSN 0272-1716.

Foster I, Kesselman C (eds.) (1998). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.

Fox J (2004). "R Commander." World Wide Web. URL http://CRAN.R-project.org/.

Goto K (2003). "High-Performance BLAS." World Wide Web. URL http://www.cs.utexas.edu/users/flame/goto/.

Graham P (1994). *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall, Englewood Cliffs, NJ.

Kiczales G, des Rivières J, Bobrow DG (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

Li MN, Rossini A (2001). "**RPVM**: Cluster Statistical Computing in R." *R News*, **1**(3), 4–7. URL http://CRAN.R-project.org/doc/Rnews/.

Lumley T (2001). "Orca [R [RJava]]." In K Hornik, F Leisch (eds.), "Proceedings of the 2nd International Workshop on Distributed Statistical Computing," URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/.

Matsumoto Y (2001). *Ruby In A Nutshell*. O'Reilly.

McDonald JA, Pedersen J (1988). "Computing Environments for Data Analysis III: Programming Environments." *SIAM Journal on Scientific and Statistical Computing*, **9**, 380–400.

Millstein T (2004). "Practical Predicate Dispatch." In "OOPSLA 2004: Proceedings of the 2004 ACM Conference on Object-Oriented Programming, Languages, and Applications," pp. 354–364. ACM.

Noble J, Taivalsaari A, Moore I (eds.) (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, Singapore.

Norvig P (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, CA.

Object Management Group (2004). "Unified Modeling Language." World Wide Web. URL http://www.uml.org/.

Oldford RW, Peters SC (1988). "**DINDE**: Towards More Sophisticated Software Environments for Statistics." *SIAM Journal on Scientific and Statistical Computing*, **9**, 191–211.

R Core Development Team (2004). "The Comprehensive R Archive Network." World Wide Web. URL http://CRAN.R-project.org/.

Ripley BD (2004). "Lazy Loading and Packages in R 2.0.0." *R News*, **4**(2), 2–4. URL http://CRAN.R-project.org/doc/Rnews/.

Rossini AJ, Tierney L, Li N (2003). "Simple Parallel Statistical Computing in R." Under review.

Russell S, Norvig P (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition.

Samatova NF (2004). "The Parallel-R Project for High-Performance Statistical Computing." World Wide Web. URL http://www.aspect-sdm.org/Parallel-R/.

Shalit A (1996). *The Dylan Reference Manual*. Addison Wesley, Reading, MA.

Smith RB, Ungar D (1995). "Programming as an Experience: The Inspiration for Self." In "ECOOP '95 Conference Proceedings," .

Steele Jr GL (1990). *Common Lisp the Language*. Digital Press.

Stuetzle W (1988). "Plot Windows." In "Dynamic Graphics for Statistics," pp. 225–245.

The **BUGS** Project (2004). "The **BUGS** Project: Bayesian Inference Using Gibbs Sampling." World Wide Web. URL http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml.

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

Tierney L (1996). "Dynamic Graphics in Lisp-Stat." In F Faulbaum, W Bandilla (eds.), "SoftStat '95: Advances in Statistical Software," pp. 21–28. Lucius and Lucius, Stuttgart.

Tierney L (2001). "Compiling R: A Preliminary Report." In K Hornik, F Leisch (eds.), "Proceedings of the 2nd International Workshop on Distributed Statistical Computing," URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/.

Urbanek S, Theus M (2003). "**iPlots** – High Interaction Graphics for R." In K Hornik, F Leisch, A Zeileis (eds.), "Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)," URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

Young F, Bann Carla M (1996). "**ViSta**: A Visual Statistics System." In R Stine, J Fox (eds.), "Statistical Computing Environments for Social Research," pp. 207–236. Sage Publications.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL http://CRAN.R-project.org/doc/Rnews/.

Affiliation:

Luke Tierney
Ralph E. Wareham Professor of Mathematical Sciences
Department of Statistics and Actuarial Science
University of Iowa
Iowa City, IA 52240, United States of America
E-mail: luke@stat.uiowa.edu

Journal of Statistical Software *Submitted:* 2004-12-11
February 2005, Volume 13, Issue 9. *Accepted:* 2005-01-04
http://www.jstatsoft.org/