Some New Developments in Lisp-Stat

Luke Tierney*
School of Statistics
University of Minnesota
Minneapolis, MN 55455
luke@umnstat.stat.umn.edu

Abstract

Lisp-Stat is a statistical computing environment based on the Lisp language. In addition to standard numerical methods, Lisp-Stat includes a dynamic graphics system for use on bitmapped graphics displays. Dynamic plots provided include histograms, scatter plots, rotatable plots, and scatter plot matrices. The graphics system is designed to be extensible using the Lisp language and an object-oriented programming system. This paper provides a brief introduction to the Lisp-Stat system and reviews some recent additions, including byte code compilation and interfacing to foreign code. Plans for future developments, including a new virtual machine, multithreading support and possible revisions to the graphics system are outlined as well.

1 Introduction

Lisp-Stat [8] is a statistical computing environment based on the Lisp language (a subset of Common Lisp). Lisp-Stat includes vectorized arithmetic operations, basic statistics and linear algebra functions, and a window system interface. The window system interface provides access to standard user interface tools such as menus and dialogs, and forms the basis for dynamic graphics tools. The built-in dynamic graphs are extensible via an object-oriented programming system.

Originally Lisp-Stat was designed to allow it to be implemented on top of any Common Lisp system. However, it has proved more effective to instead use the XLISP language as a base, adding new Common Lisp features to this base as needed.

The basic system is used trough a command line interface in which Lisp expressions are typed in to the system, evaluated, and the result is printed:

```
> (+ 1 2)
3
> (def abrasion-loss (list 372 206 ...))
ABRASION-LOSS
> (mean abrasion-loss)
175.4667
> (standard-deviation abrasion-loss)
88.12755
> (median abrasion-loss)
165
> (log abrasion-loss)
(5.918894 5.327876 5.164786 ...)
> (- abrasion-loss (mean abrasion-loss))
(196.5333 30.53333 -0.4666667 ...)
```

The mean and median functions are examples of statistical functions available in the system, logarithm and location shift transformations illustrate the vectorized arithmetic operations.

Basic graphs available in the system include histograms, scatter plots, scatter plot matrices, and three-dimensional rotatable plots. Plots can be linked so that points selected in one plot are also selected in all linked plots. Figure 1 shows an example of exploring a three-dimensional relationship in a data set on abrasion loss in rubber samples by selecting, or conditioning on, low hardness values in the histogram and observing the corresponding points in the scatter plot of the two other variables, tensile strength and abrasion loss.

Lisp-Stat also provides tools for fitting linear, generalized linear, and nonlinear models, maximizing functions, and approximating posterior distributions. These tools are also based on the object-oriented programming system.

A major objective of Lisp-Stat is to provide a framework for using and experimenting with dynamic graphical ideas. The Lisp language and the object-oriented graphics framework can be used to assemble new dynamic graphs from the building blocks provided. As a simple example, Figure 2 shows a normal probability plot of the residuals from a regression. The slider controls the power parameter in a power transformation

*Research supported in part by grant DMS-9303557 from the National Science Foundation.

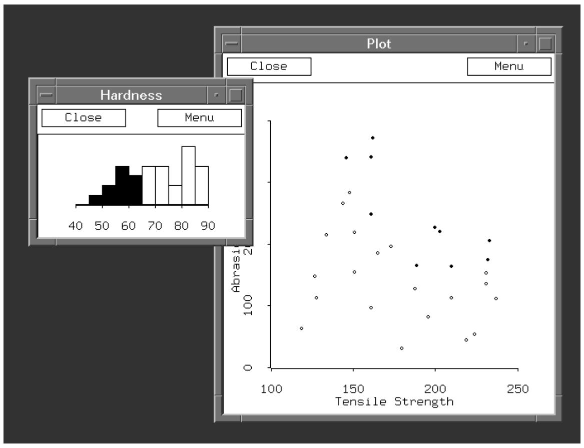

Figure 1: Linked histogram and scatter plot.

of the response variable. The code used to produce this example is shown in Figure 3.

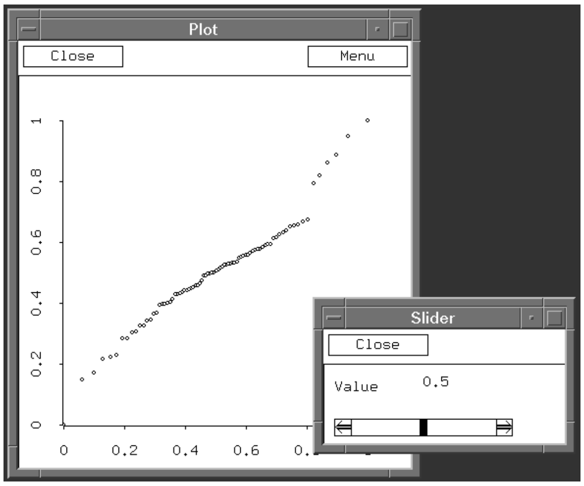

Figure 2: Animated power transformation.

2 Additions Since First Release

This section outlines features that have been added since the first release and are now part of the standard distribution. Some of the new features include special variables, multiple return values, packages for name space management, a byte code compiler, a new generational garbage collector, ratios and unlimited precision integer arithmetic, new random number generators, and Common Lisp conditions for structured exception handling. Byte code compilation, packages, and new random number generators are described in more detail in the following subsections.

```
(defun bcr (x p)
  (if (< (abs p) 0.0001) (log x) (/ (^ x p) p)))
(defun sc (x)
  (let ((min (min x))
        (max (max x)))
    (/ (- x min) (- max min))))
(setf mussels (regression-model (list height shell) mass
                                :print nil))
(defmeth mussels :simple-bc ()
  (let* ((n (send self :num-cases))
         (nqr (sc (normal-quant (/ (iseq 1 n) (+ 1 n)))))
         (idx (iseq n))
         (r (send self :residuals))
         (p (plot-points (select nqr (rank r)) (sc r))))
    (defmeth p :change-power (pow)
      (send mussels :y (bcr mass pow))
      (let* ((r (send mussels :residuals))
             (r-nqr (select nqr (rank r))))
        (send self :point-coordinate 0 idx r-nqr)
        (send self :point-coordinate 1 idx (sc r))
        (send self :redraw-content)
        (pause 2)))
    (let ((s (interval-slider-dialog
              '(-1 2)
              :action #'(lambda (pow)
                          (send p :change-power pow)))))
      (send s :value 1)
      (send p :add-subordinate s))))
```

Figure 3: Code for animated power transformation.

2.1 Byte Code Compiler

A byte code compiler translates a complex high-level language like Lisp into a very simple language that can be interpreted by a very fast byte code interpreter, or virtual machine. The internal representation of this simple language is a string of bytes, hence the name byte code. The compilation process eliminates a number of costly operations the interpreter has to perform, including variable lookup and setting up exits for nonlocal transfers of control. It also performs some inlining and makes the language tail-recursive. This means that tail calls are compiled as jumps and therefore iterations can be implemented using recursion.

As a simple example, consider a function to add up the elements of a list.

```
(defun f (x)
  (let ((s 0))
    (dolist (y x s)
      (setf s (+ s y)))))
```

Applying the interpreted function to a list of 1000 numbers produces

```
> (def x (iseq 1 1000))
X
> (time (dotimes (i 10) (f x)))
The evaluation took 2.10 seconds
```

With compilation we get

```
> (compile 'f)
F
> (time (dotimes (i 10) (f x)))
The evaluation took 0.20 seconds
```

This is an improvement of a factor of 10 in speed. Optimized native C code using a native C array of integers can be another 100 times faster, so there is room for further improvement, but the byte code compilation has made a substantial difference. There are examples where byte code compilation improves performance even more, but typically improvements are a bit more modest, on the order of a factor of two to five.

In any high-level language for scientific and statistical computing there will always be a need to implement some code in a lower level language like C to obtain acceptable performance. But developing code in the higher language itself is usually much faster and the resulting code is typically easier to understand and maintain. It is therefore advantageous to move the point where performance demands use of a low level language as far down as possible. Byte code compilation helps significantly in this respect.

It is possible to translate the byte codes produced by the compiler into C code, which can then be compiled with a native C compiler and linked into the system dynamically or statically. The naive approach does indeed produce additional speedup of about a factor of two by eliminating the byte code interpretation component, but there is a significant cost: the resulting object code is huge. As a result this option has not been pursued. However a more sophisticated approach based on a higher level intermediate representation appears to be promising and will be considered as part of the development of a new virtual machine discussed in Section 4.1.

2.2 Packages

High level languages like S, ML, or Lisp are most effective when they are used to build collections of small functions which are then combined and reused in building more sophisticated functions. A useful guideline is to keep each function small enough to fit in a single editor window. This principle was followed in developing the byte code compiler and resulted in a system of approximately 450 functions. If this system is to coexist with another system of similar size it is essential to be able to prevent accidental name clashes—some form of name space management is needed. Common Lisp uses packages for name space management, and the package mechanism has been added to XLISP.

The following example shows a simple file using the package mechanism.

```
(defpackage "MYPACKAGE" (:use "CL"))
(in-package "MYPACKAGE")
(export '(h k))
(defun g (x) (+ x 1))
(defun h (x) (+ (g x) 2))
(defun k (x) (h x))
```

Within the package all local symbols are available: `h` and `k` both use the unexported function `g`. To use the exported symbols outside `mypackage` either you need to import the symbols into your package, or you can refer to them by prefixing their names with the package name and a colon:

```
> (mypackage:h 1)
4
```

Since `g` is not exported it cannot be accessed in this way:

```
> (mypackage:g 1)
Error: external symbol not found - "G"
```

However Lisp philosophy is to not close off access to any internals unless doing so would allow significant performance improvements; therefore it is possible to access an internal symbol using a double colon:

```
> (mypackage::g 1)
2
```

Thus the package mechanism does not prevent access to internal symbols but it does prevent inadvertent access.

2.3 Random Number Generators

The random number generator mechanism has been changed to replace the original generator by more adequate ones and to support multiple generators. The mechanism represents a compromise between flexibility and performance. Generators are specified by an integer index. New generators can be added by adding to an internal C table. The currently available generators are

0 The original lagged Fibonacci generator.

1 L'Ecuyer's [4] version of Wichmann-Hill generator.

2 Marsaglia's Super-Duper, as used in S.

3 Combined Tausworthe generator of Tezuka and L'Ecuyer [5].

The default generator is generator 1. All three new generators have periods on the order of 2^{60}.

Random states are now printed as

```
> *random-state*
#$(1 #(2147483562 0 11716 54063))
```

A new random state for the default generator is produced by

```
> (make-random-state t)
#$(1 #(2147483562 0 11716 54088))
```

An alternate generator is specified by giving the generator index as a second argument. Thus a state for the Super-Duper generator is produced by

```
> (make-random-state t 2)
#$(2 #(2147483647 0 0 11715 0 54105))
```

3 Recent Developments

This section describes some more recent developments that are either available in the distribution or from the projects page [7]. These should be viewed as somewhat experimental and subject to change.

3.1 BLAS

Direct access to the basic linear algebra subroutines has been added to allow efficient numerical algorithms to be programmed in Lisp. This also required the addition of typed vectors.

As an example, if we construct two typed vectors of type C `double`

```
(def x (coerce #(1 2 3) '(vector c-double)))
(def y (copy-vector x))
```

then we can call the BLAS `ddot` and `daxpy` routines as

```
> (blas-ddot 3 x 0 1 x 0 1)
14.0
> (blas-daxpy 3 4 x 0 1 y 0 1)
NIL
> y
#(5.0 10.0 15.0)
```

All the BLAS routines represent vector arguments by a pointer and a step size. The Lisp interface uses the data, an offset from the start of the data, and a step size. Most of level 1 and some of levels 2 and 3 are supported.

This interface was developed before the package system was complete. It will most likely be modified to use a `BLAS` package, so that the `ddot` routine can be accessed as `blas:ddot`.

3.2 Shared Libraries

The ability to dynamically load code has been available since the first release but with some annoying differences between platforms and some inefficiencies. In the interim, shared libraries have become a part of all major operating systems. The dynamic loading mechanism is therefore being revised to use shared libraries and present a unified interface across platforms.

The new shared library mechanism consists of a low level and an intermediate level. The low level is based on the System VR4 `dlopen`, `dlsym` and `dlclose` interface. This interface is emulated on systems where it is not natively supported. Function pointers retrieved by `dlsym` can either use a calling convention in which they take pointer arguments and return a pointer value, or they can use the calling convention of the internal XLISP functions. This level allows the original foreign function calling mechanism to be emulated.

The intermediate level adds a simple versioning system, a mechanism for initializing constants and associating internal functions with symbols. The internal functions are assumed to use the internal XLISP calling convention. A reflection mechanism for examining library contents is also provided. This combination provides rudimentary support for a shared module system.

3.3 C Code Wrappers

To use the new intermediate-level shared library mechanism one has to write wrapper functions that map an internal XLISP function to a native C function. This is a simple but tedious task that can be automated. Several tools for automated interface generation exist; one of the most widely used is SWIG [1]. One disadvantage of SWIG is that it appears to be hard to interface it as closely to the garbage collection system as is desirable. Therefore a similar system based on Lisp was developed. Many ideas were borrowed from the design of SWIG.

As a simple example, a file containing

```
(in-package "POSIX")
(wrap:c-lines "#include <unistd.h>")
(wrap:c-function getuid "getuid" () :integer)
(wrap:c-function getlogin "getlogin" () :string)
```

defines an interface to the `POSIX` `getuid` and `getgid` functions. The wrapper generator produces a C and Lisp file. After compiling the C file into a shared library and loading the Lisp file, the functions can be used:

```
> (posix:getuid)
100
> (posix:getlogin)
"luke"
```

As a more complex example, the code

```
(wrap:c-lines "#include <pwd.h>")
(wrap:c-pointer (:struct "passwd")
                (:get pw-name "pw_name" :string)
                (:get pw-uid "pw_uid" :integer)
                (:get pw-gid "pw_gid" :integer)
                (:get pw-dir "pw_dir" :string)
                (:get pw-shell "pw_shell" :string))
```

specifies an interface to the structure returned by the `getpwnam` function. Similar expressions would be used to define interfaces to C++ objects.

This system is not yet complete; in particular it does not yet include a mechanism for specifying callback functions, automatic makefile/project creation, or FORTRAN and C++ wrapping.

3.4 Content Backgrounds

To date little has changed in the graphics system. This is likely to change in the near future. One small modification that has occured is a change to the redraw protocol for plots. Originally, the protocol called for a `:redraw-content` method to erase and redraw a plot's content. The erasing part has now been split out into a separate `:clear-content` method which is called before `:redraw-content`. Methods for `:redraw-content` should now assume that the content has been cleared. This change allows a plot to maintain a background image by defining a suitable `clear-content` method. As a very simple example, the definition

```
(defmeth p :clear-content ()
  (call-next-method)
  (let ((col (send self :draw-color)))
    (send self :draw-color 'red)
    (send self :paint-rect 50 50 100 100)
    (send self :draw-color col)))
```

maintains a red rectangle in the background of a plot. More useful examples might be to include a histogram, a function, or a map in the background.

4 New Directions

A major revision of Lisp-Stat is currently in progress. This revision will explore a number of possible changes, including a new virtual machine and runtime system, support for multi-threading and distributed computing, a revised graphics model, support for alternate languages, and new approaches to specifying statistical relationships among variables.

4.1 New Virtual Machine Design

The success of the byte code compiler described above suggests a reorganization of the basic system design. Currently the system is based on an interpreted framework with a byte code interpreter as an add-on. This will be changed to a design in which the virtual machine is at the core. Figure 4 gives a schematic representation of the design. In contrast to the current interpreter design, the virtual machine will be isolated from language-specific details as much as possible. This will make it easier to provide alternate syntaxes, such as an S-like syntax or a graphical syntax, for the system.

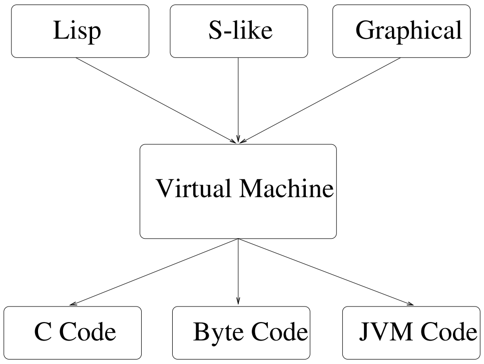

Figure 4: Outline of new virtual machine design.

The virtual machine will be designed as a conceptual interpreter for simple intermediate language that is at a slightly higher level than the current byte code language. The intermediate language will retain some additional structural information that will facilitate translation to several different execution targets. Possible targets will include a byte code language similar to the current one that is interpreted by an interpreter written in C, a direct translation into C, and the Java virtual machine. If the machine is designed properly it will be possible to mix code written with different syntactic front ends and compiled to different targets in one application.

4.2 Multi-threading

The new system will add multi-threading capability. Multi-threading is needed for two reasons. The first is that there are times when a long running calculation needs to be carried out, but the user interface should remain responsive during the calculation and perhaps be able to monitor the calculation. This can be accomplished within the current single-threaded event-driven model but is extremely awkward. It is conceptually much simpler to start a separate thread for the long-running calculation.

A second reason for the need for multi-threading is that systems with two or more processors are becoming more affordable and there is thus a need for software that allows experimentation with how to take advantage of this hardware in statistical computing.

A question that needs to be resolved is what synchronization mechanisms should be provided. For performance reasons there may be a need to provide low level primitives such as mutexes at least at the C level. But it may be advisable to use a higher level mechanism in the high-level language level. One possible approach is to follow the example of the Limbo [3] language and base synchronization on the channels mechanism from the communicating sequential processes (CSP) approach to parallel computing.

4.3 Revised Graphics Model

A final area of major change will be in the graphics system. Some of the changes will be simple in concept. These include adding hierarchical menus, hierarchical and nested window structures, geometry management, and support for using native widgets.

At a higher level, there is a need to redesign the basic model for statistical graphs. The current model represented a compromise of expressiveness and performance. Performance is still an issue but for different reasons than five years ago. Design changes need to take account of the fact that techniques may now be applied to data sets ranging in size up to hundreds of thousands of observations.

The primary reason for a need to redesign the graphical model is best illustrated by an example. The plot in Figure 1 can be enhanced by adding a smooth through the highlighted points in the scatter plot as shown in Figure 5. This is a simple and useful idea that could in principle be made available as an option for any plot. But there are many similar enhancements that could be made available, and doing so would create an unmanageable collection of options. A better approach is to provide a flexible set of primitive operations that allows an analyst to assemble this form of enhancement with a few operations taking a few seconds. Currently it is possible to add this enhancement with about a page of code taking an experienced user about an hour to write. This is adequate for a researcher in dynamic graphical methods to try out a new idea, but it is prohibitive for an analyst exploring a data set.

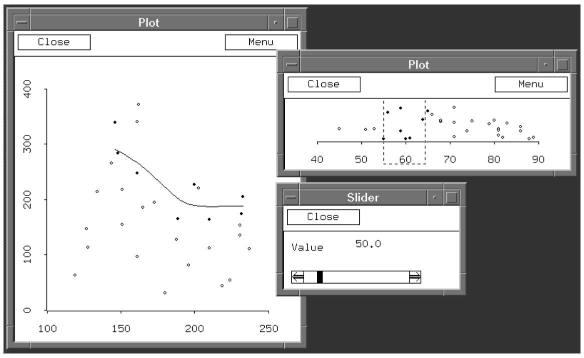

Figure 5: Plot linking enhanced by a smoother.

By collecting and analyzing a number of similar examples where enhancements should be possible with a few simple operations it is hoped that it will be possible to identify an effective set of primitives and a graphical model to support them.

4.4 Formulas as Closures

One final area that requires some attention is the representation of statistical models. Currently there is no analogue to the formula data used in S [2]. Such a mechanism needs to be added. But there is a weakness to the S mechanism that should not be duplicated. S code for fitting models described by formulas contains rather awkward and brittle constructs to locate the data a formula refers to. Since formulas are typically created by a user in a context where the data are available, it seems natural to use the concept of lexical scope and make a formula capture the relevant portion of the environment in which it is created, thus acting like a closure. This environment would then provide the default data values for fitting functions.

5 Further Information

Further information is available from the XLISP-STAT home page [6]. Recent developments are described on the projects page [7]. Two `majordomo`-managed mailing lists are available at stat.umn.edu: `stat-lisp-news` for general discussions and `stat-lisp-devel` for development issues.

References

[1] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual Tcl/Tk Workshop*, 1996.

[2] John M. Chambers and Trevor J. Hastie. *Statistical Models in S.* Chapman and Hall, 1993.

[3] Lucent Technologies Inc. Inferno home page. http://inferno.bell-labs.com/inferno/index.html.

[4] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31:742–749, 1986.

[5] S. Tezuka and P. L'Ecuyer. Efficient and portable combined Tauseworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1:99–112, 1991.

[6] Luke Tierney. XLISP-STAT home page. www.stat.umn.edu/˜luke/xls/xls.html.

[7] Luke Tierney. XLISP-STAT projects page. www.stat.umn.edu/˜luke/xls/projects/projects.html.

[8] Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* Wiley, NewYork, NY, 1990.