# DRAFT
# A Surfer's Guide to Lisp-Stat

John A. Scales & Martin L. Smith

Samizdat
Press

# A Surfer's Guide to Lisp-Stat

John A. Scales & Martin L. Smith

Samizdat Press Golden · White River Junction

Published by the Samizdat Press

Center for Wave Phenomena
Department of Geophysics
Colorado School of Mines
Golden, Colorado 80401

and
New England Research
76 Olcott Drive
White River Junction, Vermont 05001



# Contents

# List of Figures

# Chapter 1

# Writing Beautiful Programs and Having Fun

The goal of this course is to learn about a programming language. We hope that in so doing you will also learn something about the way computers work and the way humans interact with them. We also want everyone to have fun. Computers should be fun. Admittedly it is often more difficult to communicate with a computer than with a person; computers require a precision of expression that humans do not. But writing computer programs should not be a drudgery, a seemingly endless cycle of design, code, debug. We will use a programming language, called Lisp, that encourages rapid, interactive code development. Partly this turnaround speed is due to certain features of the language (e.g., garbage collection and run-time typing[1]) that free the programmer from much of the nitty gritty associated with languages such as C and Fortran. Partly the speed is due to the intrepreted nature of the language–when you type code into the interpreter, you get immediate feedback as to its correctness. And partly the speed is due to the "functional" nature of the language which allows you to isolate the effects of bugs. At this point all these claims may seem incomprehensible. We will explore these issues in detail during the course of the semester. For now we should focus on the larger issues at stake.

In his book *ANSI Common Lisp* Paul Graham makes an intriguing analogy:

> Strange as it sounds, less planning can mean better design. The history of technology is full of parallel cases. A similar change took place in painting during the fifteenth century. Before oil paint became popular, painters used a medium called *tempera* that cannot be blended or overpainted. The cost of mistakes was high, and this tended to make painters conservative. Then came oil paint, and with it a great change in style. Oil 'allows for second thoughts.' This proved a decisive advantage in dealing with difficult subjects like the human figure .... The medium did not just make painters' lives easier. It made possible a new and more ambitious kind of painting.

## 1.1 Languages, languages and more languages

There are hundreds of different computer languages in widespread use. If you think that the world is limited to a few (Fortran, C, Basic, Pascal, perhaps) then take a look at the *Usenet*

[1]We will define these two terms in due course.

"lang" groups. Here are just a few: ada, apl, awk, basic, beta, c, c++, clarion, clipper, clos, clu, cobol, crass, dylan, eiffel, forth, fortran, hermes, icon, idl, java, lisp ... and dozens more.

Why this plethora of computer languages? Partly this is because most languages arose to fulfill specific needs. Fortran, the first high-level programming language, was invented in the 1950s to aid in "**for**mula **t**ranslation"; i.e., translating scientific and engineering expression into a machine readable form. Before Fortran, if you wanted to program a computer to perform some task, you had to use the computer's own native language (assembly language), which was painful, slow and prone to errors.

## 1.2 Lisp

At the same time Fortran was being developed, John McCarthy (then at MIT) saw the need for a different language to support research in artificial intellegence. (The very first organized study of AI was the 1956 Dartmouth Summer Research Project on Artificial Intelligence.) According to McCarthy

> My own research in artificial intelligence was proceeding along the lines that led to the Advice Taker proposal in 1958 ... This involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate - it still is - and a list processing language also seemed appropriate for programming the operations involved in deduction - and still is.[2]

Lisp (a contraction of list processing) is the language that grew out of this milieu. In contrast to a language that results from a formal design effort, Lisp has evolved to meet the demands of users. It is remarkable that after all these years Lisp is still at the forefront of programming technology. Partly this must be due to the *extensibility* of the language. By that we mean that you can use Lisp to define new Lisp operators. "As new abstractions become popular (object-oriented programming, for example) it always turns out to be easy to implement them in Lisp. Like DNA, such a language does not go out of style" [Gra96]. To hammer home this point Graham implements a simple Smalltalk-like object-oriented version of standard lisp in *8 lines of code*. Try doing that with C or Fortran.

In addition, this extensibility of the language allows (encourages in fact) a style of programming that is bottom-up. Instead of writing a program that works its way down to the language, you build up the language to the program. This bottom-up approach has enormous implications for software development.

Now the best way by far to learn any programming language is to write programs. So let's jump right in. We will come back to these larger issues later but for now let's get on with it.

[2]John McCarthy has a web page at `http://www-formal.Stanford.EDU/jmc/`. Here you will find a treasure trove of useful information on Lisp, computation, programming languages, artificial intellegence, including copies of many of McCarthy's seminal papers.

# Chapter 2

# Lisp-Stat: The Lisp Interpreter We Will Use

There are a number of Lisp implementations available for free or commercially. For example, at the Linux archive site

`ftp://sunsite.unc.edu/pub/Linux/devel/lang/lisp`

you will see (at the time of this writing) some eight different Lisp interpreters/compilers. Some of these are able to compile into native machine code (via C) for increased speed. All eight of these packages can be freely downloaded and used straight away.[1]

By far the most common dialect of Lisp is *Common Lisp*. Further, most of these Lisp implementations support the Common Lisp Object System (CLOS – pronounced either c-loss or claws). We will discuss CLOS later when we talk about object-oriented programming in detail. At that time we will also discuss why for our work a different approach to OO design than that adopted in CLOS is useful. The particular object system we will use is implemented in a version of Lisp called XLISP-PLUS developed by David Betz, Thomas Almy and others. Because of the advantages that XLISP offers for interactive graphical and numerical calculations, it was chosen by Luke Tierney for his implementation of Lisp-Stat. This may be a bit confusing now, but to set the heierarchy straight: the software package we will use is called Lisp-Stat, it is implemented using XLISP-PLUS, which is a dialect of Lisp. Much of the code we write will run under any Common Lisp interpreter.[2] However, Lisp-Stat gives us a huge body of graphical, numerical and statistical code so that we'll be able to write programs at a very high level.

## 2.1 Using Lisp-Stat

Our implementation of Lisp-Stat, called *Xlisp-Stat* is available for Unix, Macintosh and Windows machines. Once it's installed all you have to do is type `xlispstat` in a Unix shell and you should see

[1]Be careful to read the copyright or license agreement on code that you download off the net. Some packages place restrictions on distribution. You are OK if you see words to the effect "distributed under the terms of the Gnu Public License (GPL)."

[2]An interpreter (or evaluator) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression. In other words, the interpreter, which determines the meaning of expressions in a language, is simply another program [AS96].

```
4 slate> xlispstat
XLISP-PLUS version 2.1g
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT 2.1 Release 3.44 (Beta).
Copyright (c) 1989-1994, by Luke Tierney.

>
```

To end an Lisp-Stat session type `(exit)` at the prompt

```
> (exit)
```

## 2.2 Getting Lisp-Stat

Before we start using Lisp-Stat, you might be interested in getting a copy of it for your home machine. It is freely available in source form or, for a few computers, as pre-compiled binaries via the World Wide Web. Point your WWW browser to

`ftp://stat.umn.edu/pub/xlispstat/`

and look for *mswin*, *macintosh*, etc. At the time of this writing, the latest version of of Lisp-Stat is 3-44. The Unix source code is contained in the file xlispstat-3-44.tar.gz. Once you have this you can uncompress, using

```
gunzip -c xlispstat-3-44.tar.gz | tar -xvf -
```

This will unpack the source code in a directory called xlispstat-3-44. Follow the directions in the INSTALL file. The directions for installing the Macintosh and Windows versions are in README files in the specific ftp directories.

Lisp-Stat was developed by Professor Luke Tierney at the University of Minnesota. If you like it, use it, and appreciate the fact that Tierney is giving it away for free, you might consider dropping him an email thanking him at:

`luke@stat.umn.edu`

Tierney's book [Tie90] is *the* definitive Lisp-Stat reference. It is also an excellent introduction to the Lisp programming language. You should be aware that more recent versions of the Lisp-Stat system have appeared since this book was published. A comprehensive and inspiringly well-written guide to Common Lisp is Paul Graham's book [Gra96].

## 2.3 Invoking Lisp-Stat

These notes are being written by someone who is sitting in front of a generic PC running a version of the Unix operating system called Linux. Some of the discussion may be slighly different if you're using the Macintosh or Windows implementation, but to actually verify such details would require the author to actually use one of these other operating systems.

1. From a Unix shell, type xlispstat. You'll get a response like the one above followed by the prompt.

```
>
```

The `>` is the prompt.

2. You can fire up xlispstat in a separate xterm.[3] For example, at the Unix prompt, you could type: `xterm -e xlispstat &` and an xterm will pop up with xlispstat running in it. The `-e` says to *execute* the command in the xterm window.

3. You can invoke xlispstat from within emacs.[4] This is definitely the best way to do it, since it gives you a nice interface much like a Mathematica notebook, with command line editing, history, scrolling, multiple windows, etc. You can edit a lisp file in one window and execute commands in another. Emacs interaction with Lisp-Stat is illustrated in Figure 2.1.

## 2.4 A Few Simple Examples

### 2.4.1 The Lisp Interpreter

The Lisp interpreter uses `>` for a prompt. If you type something and press the *enter* key (what we old fogeys call a carriage return), the interpreter evaluates whatever you typed. Numbers evaluate to themselves:

```
> 1
1
```

Symbols evaluate to whatever value is associated with the symbol, thus

```
> pi
3.141592653589793
```

but

```
> z
Error: The variable Z is unbound
```

since, in this example, the symbol $z$ has not been defined. To define a symbol, we could do something like this, where we define `e` to be $e^1$

```
> (def e (exp 1))
E
> e
2.718281828459045
```

Here we have used `def` to define a global variable `e` to have the value `e`. The first "E" is echoed by the interpreter, which is case-insensitive. At the next prompt, we typed the symbol, and the interpreter responded by printing its value. The logical values `t` (true) and `nil` (false) also evaluate to themselves. Nil comes from the Latin *nihil* meaning nothing.

[3]xterm is a terminal emulator for X windows. If you type `xterm` at a Unix prompt you will get a new window with a Unix shell running inside it. To find out more about xterm or other Unix/X commands, type `man` followed by the command name. Try `man man` for starters.

[4]emacs is usually described as an editor, but it's really an extensible environment for doing everything from reading your mail and news, to editing/compiling/debugging programs, formatting documents, and lots more. Emacs was originally developed by Richard Stallman, who began the Free Software Foundation. Emacs should especially appeal to Lispers since it's largely written in and extensible from Lisp.

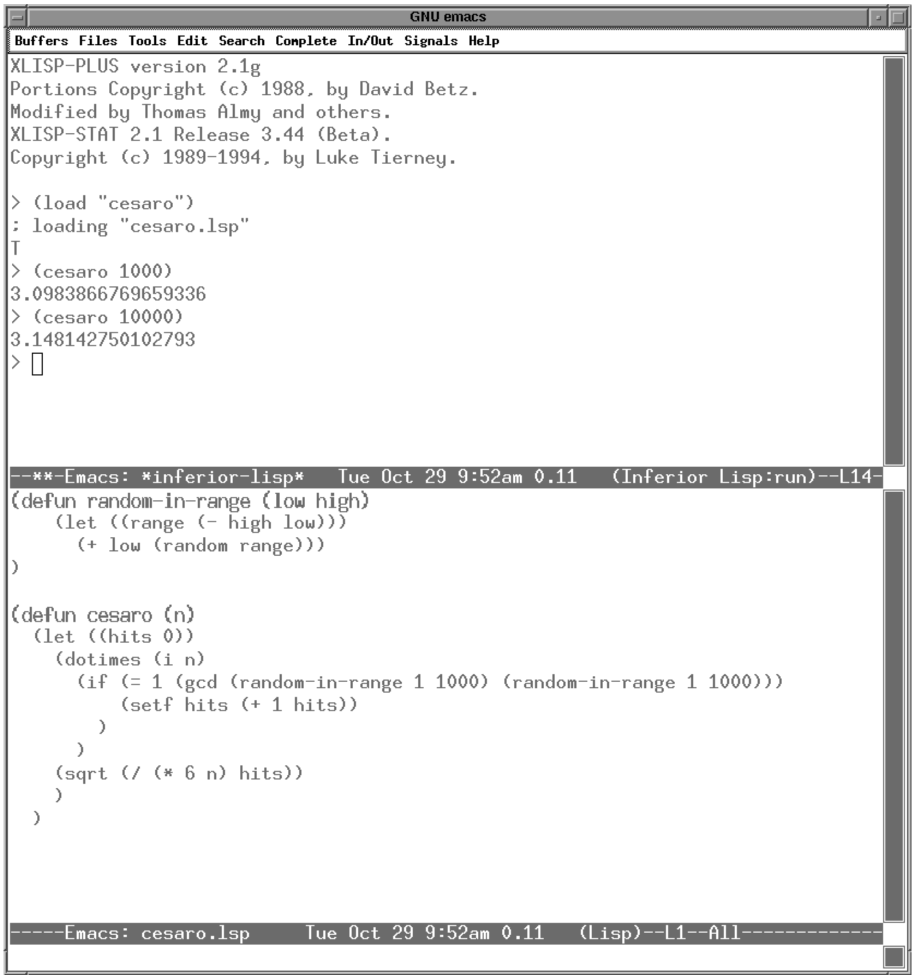


Figure 2.1: GNU Emacs as an interface to the Lisp-Stat interpreter. We can edit lisp programs in one buffer while interacting with the interpreter in the other. This lets us take advantage of all the power of Emacs when talking to the interpreter.

### 2.4.2 Run-time Typing

In languages such as Fortran and C you must declare (or rely on implicit declarations of) the type of every variable used. Statements such as `Real*8 x` or `float y`, force the programmer to define the type of variables. The advantage of this so-called *strong typing* is one of efficiency. An operator such as `+` uses different code depending on whether it's adding real or complex numbers, integers, etc. In Lisp, you can define the type of a variable at run time. If you know in advance the type of a variable, then you can fix it and gain some efficiency. But when developing code it is extremely handy to be able to do things like

```
> (def e 1)
E
> e
1
> (setf e (uniform-rand 3))
E
> e
(0.4982946009067598 0.6092169488583029 0.14338857176804926)
```

without having to worry about whether `e` is an integer or an array of floating point numbers. In this example `setf` replaces `e` with the list returned by `(uniform-rand 3)`.

### 2.4.3 Lisp Comments

A Lisp comment begins with a semi-colon. Thus

```
> ; This is a Lisp comment
```

is ignored by the interpreter.

### 2.4.4 Quotes

```
> (quote x)
X
> ´x
X
```

both return `x` without evaluating it. For instance

```
> (+ 1 2)
3
> ´(+ 1 2)
(+ 1 2)
```

And how about ´´x?

```
> ´´x
(QUOTE X)
```

Strings are enclosed in double quotes:

```
> "strings are enclosed in double quotes"
"strings are enclosed in double quotes"
```

Notice that

```
> ´()
NIL
```

so `nil` is also the empty list.

### 2.4.5 Life is a Function

Lisp makes heavy use of functions. There are zillions of them, but once you know a few, you can write very compact code. Here are some examples of the arithmetic functions:

```
> (+ 1 4)
5
> (- 1 4)
-3
> (* 1 4)
4
> (/ 1 4)
0.25
> (* 2 (/ 1 2))
1.0
```

Notice that the function is delineated by matching parentheses and functions can be nested within functions. Technically functions evaluate their first argument. So although `def` looks like it might be a function, in fact it's not because it does not evaluate `a`. However this distinction is not important to us; for more information see the discussion in [Gra96] about macros and special forms.

### 2.4.6 Prefix vs. Infix Notation

Lisp uses "prefix" notation. Functions all begin with a left parenthesis followed immediately by the symbol representing the function. A reasonable first impression is that writing `(+ 1 2)` is less natural than writing `1 + 2`. True enough, but prefix notation has some powerful advantages. For instance, anything between the operator name ("+" in this case) and a matching right parenthesis is automatically an argument to the function. Thus we can easily do things such as

```
> (+ 1 4 9 8 1 0)
23
```

In Lisp it is easy to write functions that accept a variable number of arguments. There is never any ambiguity as to the scope of a Lisp operator, since it's always delimited by parentheses. And in case any of the arguments are themselves function calls, they are evaluated according to the same rules. E.g., when

```
(+ 7 (- 10 3))
```

is entered, the following sequence of evaluations occurs. First `7` evaluates to itself. Next `(- 10 3)` evaluates to `7`. Finally these two values are sent to the function `+` which returns `14`:

```
> (+ 7 (- 10 3))
14
```

Evaluation Rule

Most operators in Lisp are functions and all function calls are evaluated in this way. **Arguments are evaluated left-to-right and their values are passed to the function, which returns the value of the whole expression.**

In our work we will frequently need arrays of numbers. We can define them by hand as, e.g.,

```
> (def my-data (list 22 34 30 24 55))
MY-DATA
> my-data
(22 34 30 24 55)
```

Fortunately many of Lisp-Stat's functions are "vectorized", that is, they operate on entire arrays (or vectors). Thus

```
> (/ my-data 2)
(11 17 15 12 27.5)
```

results in the division of each element of `my-data` by 2. A slightly more interesting example, would be

```
> (- my-data (mean my-data))
(-11.0 1.0 -3.0 -9.0 22.0)
```

Here we found the mean of the `my-data` array, and then subtracted this element-wise from the data. `mean` is a built-in Lisp-Stat function. Notice that we didn't have to tell the `mean` function how many elements there were in `my-data`.

### 2.4.7 A Picture is Worth a Thousand Words

Lisp-Stat's plotting capability is excellent. One-dimensional arrays of numbers can be plotted with the `plot-points` and `plot-lines` functions. If we try something like

```
> (plot-points my-data)
Error: too few arguments
Happened in: #<Subr-PLOT-POINTS: #c01c8>
```

we get an error message. So let's see what that's all about. Let's get some help

```
> (help (quote plot-points))
PLOT-POINTS                                              [function-doc]
Args: (x y &key (title "Scatter Plot") variable-labels point-labels
symbol color) Opens a window with a scatter plot of X vs Y, where
X and Y are compound number-data. VARIABLE-LABELS and POINT-LABELS,
if supplied, should be lists of character strings. TITLE is the window
title. The plot can be linked to other plots with the link-views
command. Returns a plot object.
```

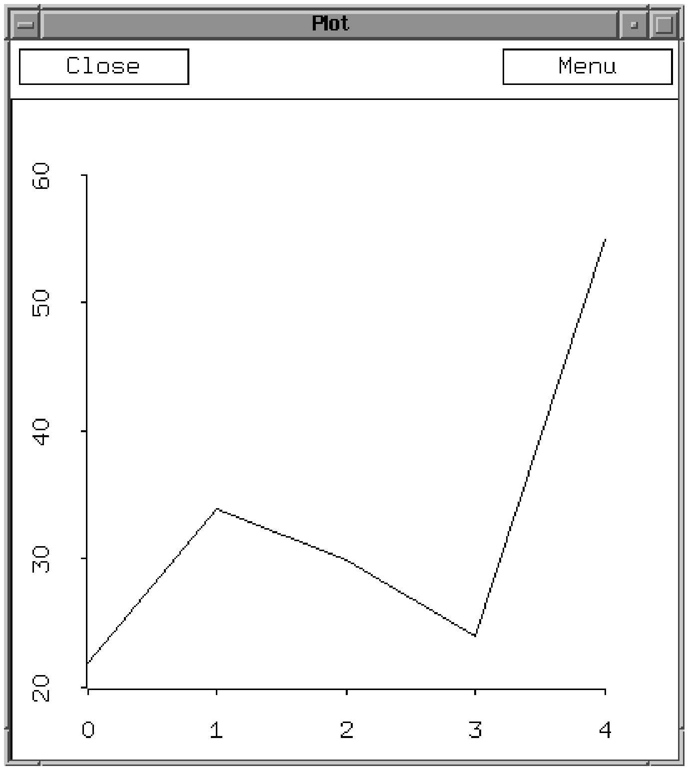


Figure 2.2: The function `plot-lines` applied to my-data.

Everything past the `&` in the line beginning `Args:` represents an option, so let's postpone discussing these for a bit.[5] Right away we see that the `plot-points` function is expecting both an ordinate and an abscissa. So we can fake an abscissa just by defining a list of integers

```
> (plot-points (iseq (length my-data)) my-data)
#<Object: 1412952, prototype = SCATTERPLOT-PROTO>
```

or

```
> (plot-lines (iseq (length my-data)) my-data)
#<Object: 1438744, prototype = SCATTERPLOT-PROTO>
```

The latter result is shown in Figure 2.2. The function `iseq` simply creates an integer sequence of the given length. The odd looking line beginning `#<Object:` can be used later to identify plots. For now we're not going to worry about this; henceforth these messages will be suppressed.

In these last few examples we have used an array that we typed in by hand. More often we will read a file in from disk or create some array with one of Lisp-Stat's functions. For example, let's create three arrays according to three different pseudo-random number generators.

```
>  (def uniform (uniform-rand 100))
UNIFORM
> (def gaussian (normal-rand 100))
```

[5]The `quote` function (equivalent to `(help 'plot-points)` in this example) is necessary since we want to send the interpreter the symbol itself and not its value.

```
GAUSSIAN
> (def chisquared (chisq-rand 100 5))
CHISQUARED
```

Here we have defined three arrays, each of length 100 containing pseudo-random samples drawn from, respectively, the uniform, normal and $\chi^2$ distributions. The statistical interpretations are not important now; here we're just generating some numbers to play with. Now we can plot them. For example, we could

```
> (plot-lines (iseq 1 100) uniform)
```

as we did before. But let's suppose that these three arrays are somehow connected. They could represent three different data sets which might somehow be "correlated", such as (let us suppose) daily rainfall, pollen count, and automobile miles driven. If we wanted to see how these data were connected, we might want to plot rainfall versus pollen count, or pollen count versus miles driven, etc. These are called "scatter-plots". Lisp-Stat let's make scatter-plots of any number of different data sets. We'll use this function in shortly to look at some real geophysical data, but for now, let's just pretend our pseudo-random numbers are interesting data and call the `scatterplot-matrix` function:

```
> (scatterplot-matrix (list uniform gaussian chisquared)
     :variable-labels (list "Uniform" "Gaussian" "Chi-squared") )
```

generates the scatter plot matrix shown in Figure 2.3. While

```
> (spin-plot (list uniform gaussian chisquared))
```

generates a spin plot that can be rotated interactively–just click on one of the six buttons at the bottom of the display (two each for pitch, roll and yaw). This is shown in Figure 2.4.

### 2.4.8 Input and Output

Suppose we have some data in an ascii file of the form

```
2387.75
2433.34
2480.71
```

etc, representing some laboratory P-wave velocity measurements, and a similar file with density measurements. We would like to be able to read into Lisp-Stat and do some calculations and make some plots.

```
> (def vp (read-data-file "pwavelog") )
VP
> (def rho (read-data-file "densitylog"))
RHO
> (def imp (* rho vp))
```

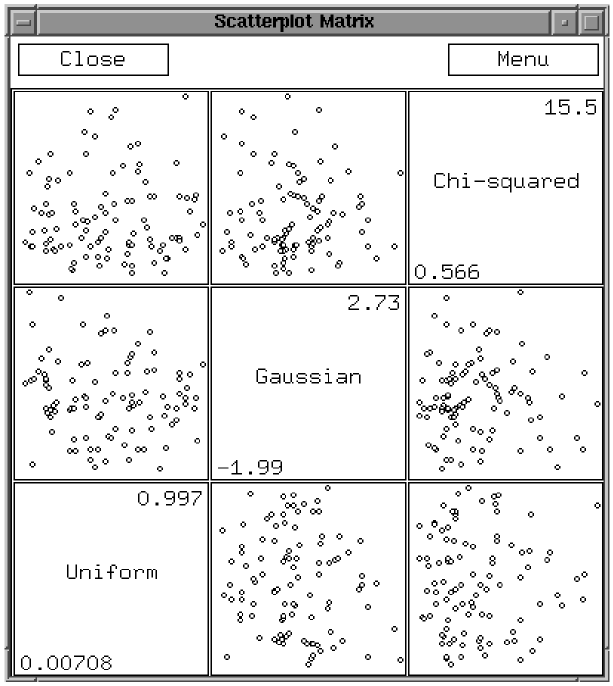


Figure 2.3: A scatter plot matrix. In this case, since the pseudo-random numbers have no connection to one another, we see no obvious pattern when they are plotted one agains another. Later we will see an example in which scatterplots of multi-dimensional data bring out certain correlations that would be difficult to visualize otherwise.

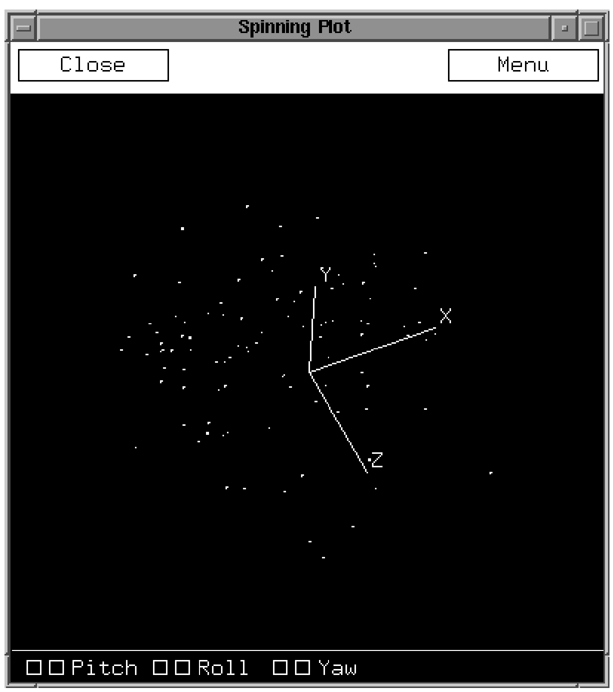


Figure 2.4: A spinning plot.

Here `pwavelog` and `densitylog` are the names of two files in the current working directory.[6] Notice that there are no presumed file extensions. So when we say the file is called `densitylog`, we really mean it. These data files are available on the WWW page under "code/logdata". There is also a little Lisp-Stat program for displaying the data.

And while we were at it, we computed the impedance (density times velocity) with a simple vectorized multiplication of the velocity and density, stuffing the results into an array called `imp`.

Now we can look at, let's say, a scatter plot matrix of velocity, density, impedance against depth (the latter we'll make up with `iseq`)

```
> (scatterplot-matrix (list rho vp imp (iseq 1 (length imp)))
        :variable-labels (list "Density" "Velocity" "Impedance" "Depth"))
```

which produces the results shown in Figure 2.5. The function `iseq` returns an integer sequence of length equal to the array `imp` beginning with 1.

Or we can make a spin plot of density, velocity and depth:

```
> (def depth (iseq 1 (length imp)))
DEPTH
> (spin-plot (list rho vp depth))
```

as shown in Figure 2.6.

### 2.4.9 More Input and Output

The `print` function prints to your screen. Thus typing

```
> (dotimes (k 5)
        (print k) )
0
1
2
3
4
```

results in the integers 0 through 4 being printed to the terminal. Notice that Lisp loops begin by default at 0, like C/C++, but unlike Fortran.

I/O to files is handled nicely with streams.[7] We can define a file stream as

```
> (def myfile (open "myfile" :direction :output))
MYFILE
>(print "this is a test" myfile)
"this is a test"
> (close myfile)
T
```

[6] You can change the current working directory from within Lisp-Stat with `set-working-directory`.

[7] For our purposes a "stream" can be thought of intuitively as a flow of information. In this case, the information is flowing out from the code to a file.

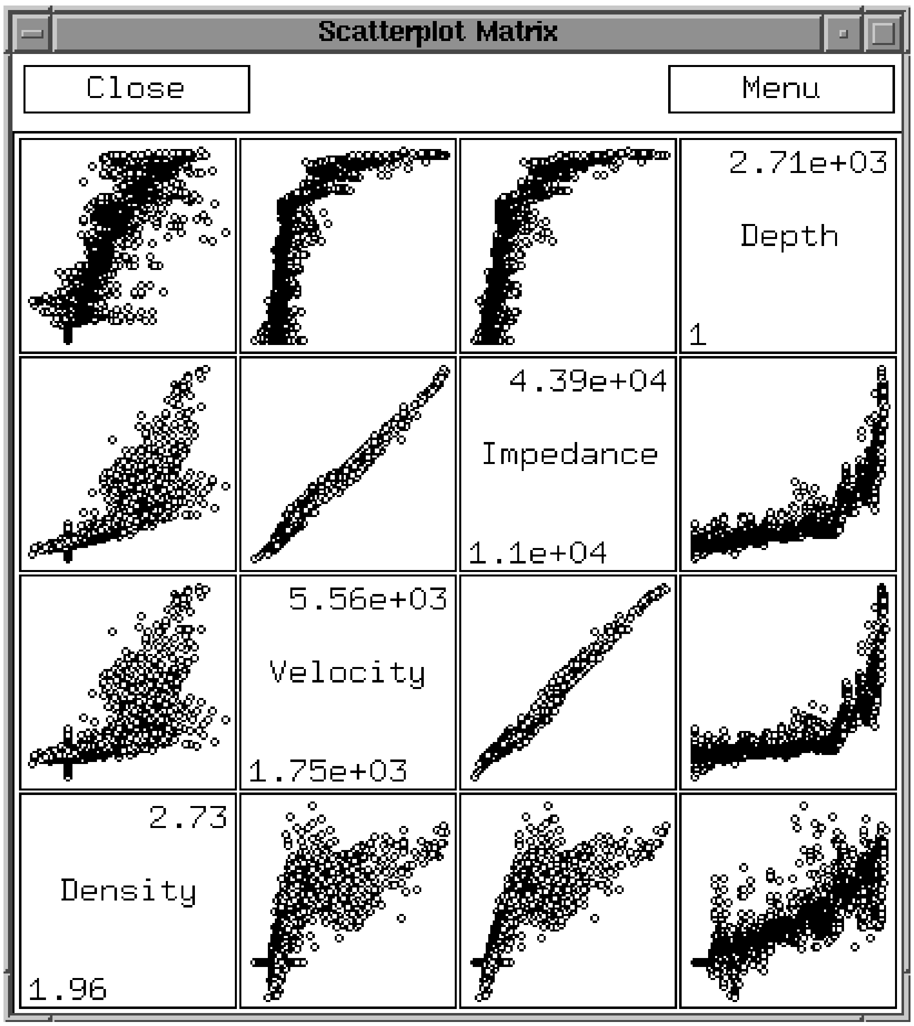


Figure 2.5: A scatter plot of the velocity and density logs along with impedance and depth.

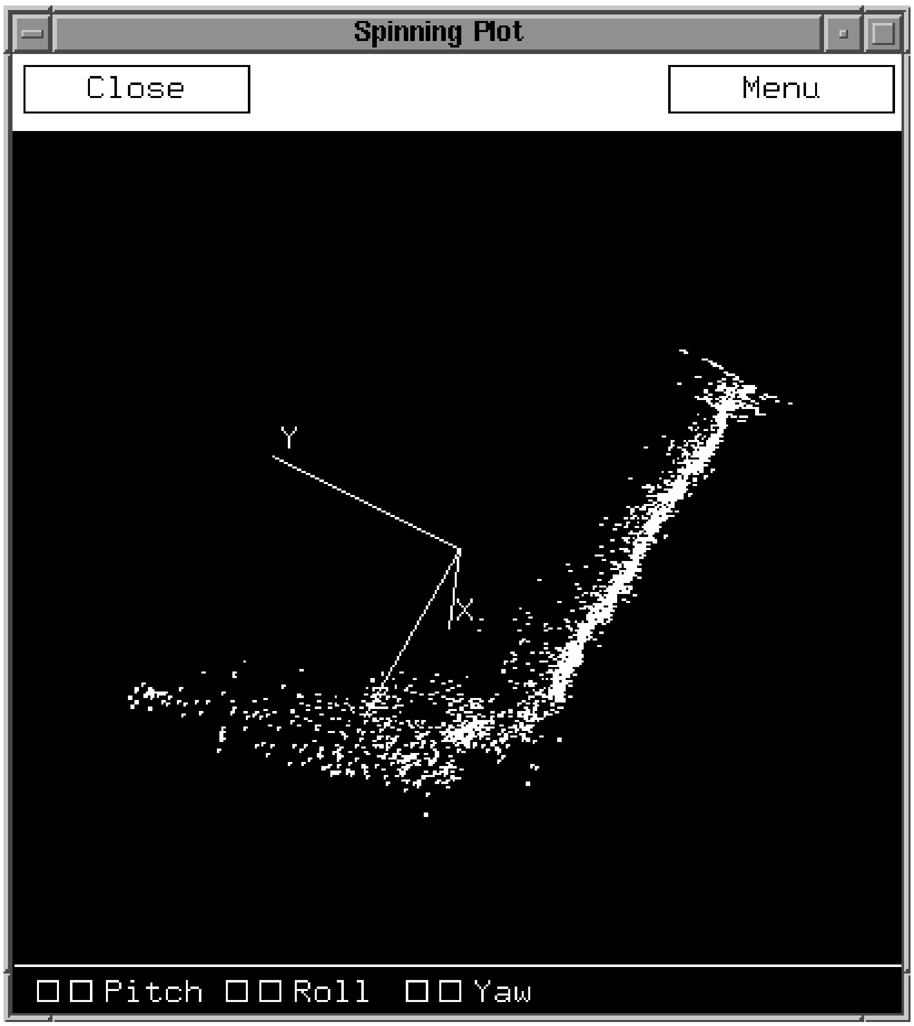


Figure 2.6: A spinning plot of the velocity and density logs along with depth.

Arguments that begin with a colon (such as direction and output) are "keywords". We will have more to say about these later, but keyword symbols evaluate to themselves and so don't need to be quoted. The direction is input by default. This means that if we had simply said

```
> (def myfile (open "myfile"))
```

we would have been able to read from the file but not write to it. This code opens a file called *myfile*, writes a string to it and then closes the file. Formatted I/O is covered in Chapter 4 of [Tie90].

### 2.4.10 Making Your Own Functions

We have seen a number of examples of Lisp-Stat's built-in functions, now let us see how to define our own. Functions can be made with `defun`. For instance

```
(defun pythag (x y)
   (sqrt (+ (* x x) (* y y)))
   )
```

defines a function that returns the value $\sqrt{x^2 + y^2}$.

```
> (pythag 1 1)
1.4142135623730951
```

Lisp functions can be recursive, that is, a function can be defined in terms of itself. For example,

```
> (defun factorial (n)
    (if (= n 0)
        1 (* n (factorial (- n 1))))
    )
```

### 2.4.11 A History Mechanism

Lisp-Stat uses the same history mechanism as Common Lisp. The symbols `- + ++ +++ * ** ***` have the following meaning:

`-` The current input expression

`+` The last expression read

`++` The previous value of `+`

`+++` The previous value of `++`

`*` The result of the last evaluation

`**` The previous value of `*`

`***` The previous value of `**`

Thus

```
> (def x 1)
X
> (*)
1
> (/ * 10)
0.1
> (* * 10)
1.0
> (* ** 10)
1.0
```

Actually the first one is a trick. Notice that, for example, the symbol * can be both a function and a variable. If we ask for help about this symbol

```
> (help ´*)
*                                                          [function-doc]
Args: (&rest numbers)
Returns the product of its arguments. With no args, returns
1. Vectorized.
*                                                          [variable-doc]
The result of the last evaluation
```

we get both its definition as a function and its binding as a variable. And in this particular case * always returns 1 if it has no argument.

### 2.4.12 An Example of an Object: Dynamic Plotting

Lisp-Stat has a special data type known as an *object*. Later we will have a lot more to say about objects, since one of our goals is to learn about "object-oriented programming." Lisp-Stat objects are similar to structures in C, except that in addition to holding data in *slots*, they can also respond to certain messages. The function `send` is used to send messages to objects.

First let's make an object.

```
> (def myplot (histogram (normal-rand 100)))
MYPLOT
```

How do we know this is an object? Lisp-Stat has a predicate function `objectp` which tells us:

```
> (objectp myplot)
T
```

In other words, the answer to the question: "Is `myplot` an object?" is "true".

The simplest message to send to our object is `:help`.

```
> (send myplot :help)
ABLINE ACTIVATE ADD-FUNCTION ADD-LINES ADD-METHOD ADD-MOUSE-MODE
ADD-POINTS ADD-SLOT ADJUST-POINTS-IN-RECT ...
```

and much, more.

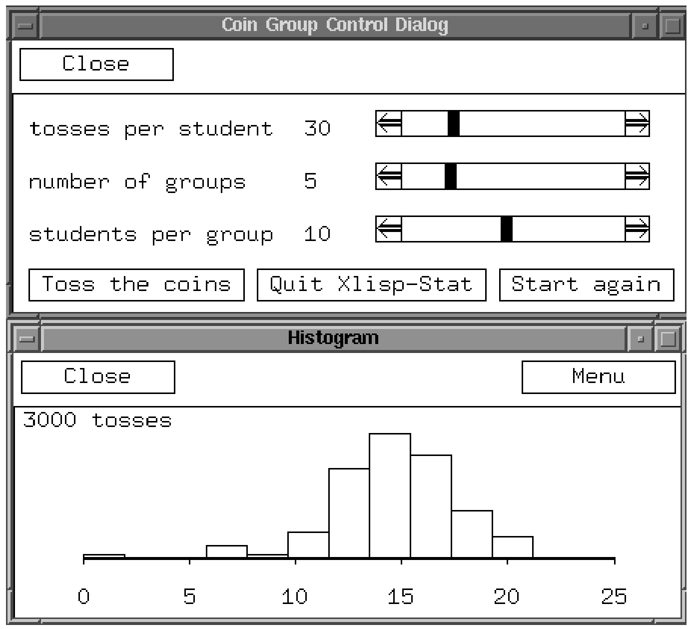


Figure 2.7: A dynamical Monte Carlo simulation. Here 50 students toss a fair coin 30 times per experiment. (The fact that the students are divided into groups is an irrelevant detail of how we actually do this in the classroom.) To repeat the experiment all we have to do is click a button and the histogram evolves before our eyes as data points are added. Lisp-Stat makes it amazingly easy to write such interactice graphical simulations. We will see how to do this later in the course.

```
> (send myplot :help :add-points)
ADD-POINTS
Method args: (points (draw t))
Adds points to plot. POINTS is a sequence or a list of sequences.
If DRAW is true the new points are added to the screen.
```

Thus we can add some points to our histogram without recalculating it from scratch:

```
> (send myplot :add-points (normal-rand 100))
```

results in a histogram for 200 points.

### A Dynamical Graphics Example

In the next section we will do an extended exercise involving two random experiments, tossing coins and needles. We will see how even tossing a coin results in the appearance of the magical bell-shaped curve if you toss enough of them. We can simulate this on the comuter dynamically by sending messages to a histogram plot. An example of such a simulation is shown in Figure 2.7. Later in this course we will see how easy it is to code graphical user interfaces such as this. For now just keep in mind that you will be doing stuff like this in no time at all.

# Chapter 3

# Lisp-Stat: Exercise I

Probability and statistics are mathematical disciplines for which experimentation is easy and fun. In this exercise we will explore two experiments that can easily be done in class, at home, in groups or alone. The first is tossing a fair coin $n$ times. If the coin is fair, then the probability of getting a head or tail in a single toss is, of course $1/2$. However if we toss the coin $n$ times, then we can ask for the probability of getting $m$ heads, where $m \leq n$. If we repeat this experiment over and over again, making a histogram of the results, we should see a bell-shaped curve appear, peaked about $n/2$ heads/tails. The fact that this bell-shaped curve appears from something as simple as a fair coin is one of the most important ideas in all of probability. The general result, known as the *Central Limit Theorem* can be found in any book on probability or statistics. To compute the probability of any given number of heads/tails analytically all we have to do is count up the number of ways of getting $m$ heads and divide by $n$. For instance there is only one way to get 0 or $n$ heads, but there are $n$ ways of getting 1 since we could get a head on any of the $n$ tosses.

### 3.0.13 Part I

Here is a factorial function:

```
(defun factorial (n)
    (if (= n 0)
        1 (* n (factorial (- n 1))))
    )
```

Now write two additional functions. One should have two integer arguments n and k and return the combinatorial coefficients

$$\frac{n!}{k!(n-k)!}$$

These are also called the binomial coefficients because they appear in the binomial probability below. The term "combinatorial" refers to the fact that these coefficients tell how many combinations can be made when $k$ objects are chosen from $n$. With combinations, these selections are unordered (i.e., the order doesn't matter). This is in contrast to *permutations*, where the order does matter. Call this function

```
(defun combin (n k) ...
```

The second function should have three arguments, n, k, and a probability of p. Call this one

```
(defun binomial (n k p) ...
```

It should return the binomial probability, i.e., the probability of getting k heads out of n tosses given that the probability of a head is p. binomial should call combin, which in turn should call factorial. The algebraic expression for the binomial probability is:

$$\frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}.$$

The Lisp function for exponentiation is expt. Thus

```
> (expt 2 4)
16
```

Next, here is a function that does a coin toss experiment.

```
(defun cointoss (n)
   (let* ((results (make-array n)))
     (dotimes (i n)
        (setf (elt results i) (binomial n i .5))
     )
    results)
)
```

This function performs a couple of tricks we have not covered in class yet, so don't worry if you don't understand all that it does yet. However, the `let` function is described in detail below.

Now define a function

```
(defun plotresults (n) ...
```

that will call cointoss and plot the results. Use this function to make a plot for n=10 and see how this compares to your observations. You will probably want to use the function `plot-lines` to display your results. Remember `plot-lines` works as follows

```
(plot-lines (iseq 10) (uniform-rand 10))
```

plots 10 pseudo-random numbers. The x coordinates are $1-10$.

```
(plot-lines  '(1 2 3) '(1 2 3))
```

plots a straight line through the three points $(1,1)$, $(2,2)$ and $(3,3)$. So `plot-lines` expects two lists as arguments, first the $x$-coordinates, then the $y$-coordinates.

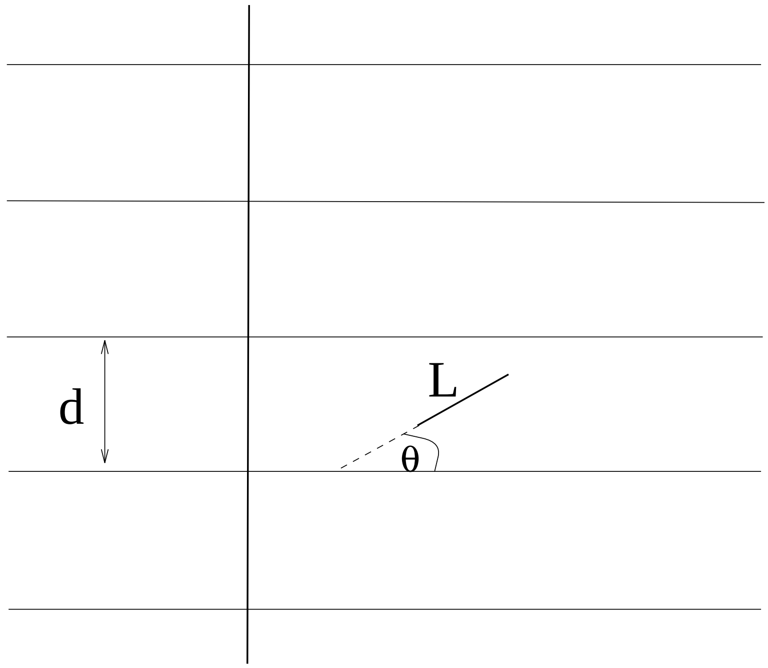


Figure 3.1: The probability of a needle of length $L$ intersecting parallel lines $d$ apart is $2L/\pi D$ provided $L \leq d$. It is possible to estimate $\pi$ to roughly 10% in 100-200 tosses.

#### 3.0.14 Part II

The Comte de Buffon (1708-1788) was a French aristocrat and prolific naturalist who studied probability. Buffon showed that the probability of a needle of length $L$ intersecting the lines of a ruled surface (as in Figure 3.1) was $2L/\pi D$ provided $L \leq D$ so that the needle could not intersect more than one rule at a time. This result is not too hard to show with elementary calculus: try it. However, the interesting thing is that Buffon used this result to estimate $\pi$

The problem here is to simulate the needle toss experiment. Write a function `buffon` such that

```
(buffon n)
```

tosses a needle of length $L$, $n$ times onto an imaginary floor whose lines are $D$ apart and returns an approximation to $\pi$. According to the theory, the probability of the needle hitting a line is $2L/(\pi D)$.

To help you get going, here are two functions that perform a similar Monte Carlo calculation. A theorem of Cesaro (described in [AS96]) says that the probability that two randomly chosen integers will have no common multiples (i.e., a greatest-common divisor *gcd* of 1) is $6/\pi^2$. First we define a function that returns random integers in a specified range.

```
(defun random-in-range (low high)
    (let ((range (- high low)))
      (+ low (random range)))
)
```

Next we use the built-in gcd function to decide whether the two integers have a gcd of 1. If they do, we count that as a hit. After n samples, we return the estimate of $\pi$. So your job is to make a function buffon that performs similarly.

```
(defun cesaro (n)
  (let ((hits 0))
   (dotimes (i n)
      (if (= 1 (gcd (random-in-range 1 1000)
                    (random-in-range 1 1000)))
             (setf hits (+ 1 hits))
                  )
                     )
                      (sqrt (/ (* 6 n) hits))
                       )
    )
```

These two functions use `let` to define local variables. In the first case "range" is defined to be high - low. Once the function is finished, "range" goes away. Similarly in cesaro, "hits" is used as a counter variable. By using let, we can limit the scope of this variable to within the function. Here are some examples of the use of let.

```
> (let ((x 2)
        (y 3))
    (print (+ x y)))

5
```

You see that this has the form of `let` followed by one or more assignments enclosed in parenthesis followed by some evaluation using these assigned variables. In this case we begin by assigning $x$ the value 2 and $y$ the value 3. Make sure you understand why the parenthesis are where they are. The first and last one enclose the `let`, and each individual local assignment is enclosed within parenthesis. Then we print $x + y$.

`let`s are very handy. However we have to be careful about one thing. Try this code:

```
> (let ((x 2)
        (y x))
    (print (+ x y)))
```

What happens? You'll get an error about "X" being unbound. The reason for this is that `let` makes its assignments in parallel. Thus $x$ is assigned to the value 2 at the same time $y$ is assigned to 3 in the first example. Thus, in the second example, we cannot assign $y$ the value of $x$ since $x$ has no value yet. Since it would obviously be useful to be able to use local variables as if they were assigned sequentially, there is an alternative form of `let` called `let*` that does this:

```
> (let* ((x 2)
        (y x))
    (print (+ x y)))

4
```

Finally `dotimes` is used for looping. Thus

```
> (dotimes (k 5)
    (print k))

0
1
2
3
4
```

# Chapter 4

# A Few Miscellaneous Lisp Tools

## 4.1 Array Access

In scientific programming we use arrays routinely. We have already used arrays (or lists) of numbers such as

```
> (setf x (uniform-rand 3))
(0.4931160313514962 0.3147434134687622 0.08523929735726343)
```

We can access individual array elements with the `elt` function. To access the first (i.e. 0) element we could

```
> (elt x 0)
0.4931160313514962
```

We can modify individual elements of the array in the same fashion

```
> (setf (elt x 0) pi)
3.141592653589793
> x
(3.141592653589793 0.3147434134687622 0.08523929735726343)
```

This says to replace element 0 of `x` with $\pi$.

We can also make arrays with the `make-array` function. The following example makes a one-dimensional array of length 5 and then loops over these elements assigning them to be the value of the loop index.

```
> (setf my-array (make-array 5))
#(NIL NIL NIL NIL NIL)
> (dotimes (i 5)
    (setf (elt my-array i) i)
    )
NIL
> my-array
#(0 1 2 3 4)
```

By default, `make-array` doesn't initialize the contents of the array, so until something is put in these array elements are `nil`. We can use the `:initial-element` or `:initial-contents` keywords to stuff (respectively single or multiple) values in. For instance,

```
> (make-array 5 :initial-element 1)
#(1 1 1 1 1)
> (make-array 5 :initial-contents (normal-rand 5))
#(0.6194290 -0.5988862 0.18704214 0.18428428 0.5885177)
```

The reason for the # in front of the printed list is that `make-array` has actually returned a *vector* rather than a *list* as we have become used. Tierney ([Tie90], 4.4.5) says that random access to vectors is more efficient than random access to lists. And here is proof. Try running the following two pieces of code.

```
(def dimen 1000)
(def trials 100)
(def my-list (iseq dimen))
(time
 (dotimes (i trials)
   (dotimes (j dimen)
     (setf (elt my-list (random dimen)) (random 10))
     )
   )
 )

(def my-vector (make-array dimen :initial-contents (iseq dimen)))
(time
 (dotimes (i trials)
   (dotimes (j dimen)
     (setf (elt my-vector (random dimen)) (random 10))
     )
   )
 )
```

All these codes do is randomly access and change elements of a length 1000 array and vector. The run times reported by the Lisp `time` function on a 66 Mhz 486 running Linux are 72 seconds for the list and 24 seconds for the vector. However, for our purposes we can ignore the distinction between vectors and lists.

We can construct vectors easily with the `vector` function or its shorthand `´#`

```
> (vector -1 2 1 1 0)
#(-1 2 1 1 0)
> ´#(-1 2 1 1 0)
#(-1 2 1 1 0)
```

where, as you might have expected, we back-quote the # to prevent the vector being evaluated.

We can make higher dimensional arrays in the same fasion. Thus with `make-array` we can say

```
> (def my-matrix (make-array ´(2 3) :initial-element 0))
MY-MATRIX
> my-matrix
#2A((0 0 0) (0 0 0))
```

This makes a two by three array and initializes the contents to zero.[1]

Two things to note here. First we needed to give `make-array` a list containing the dimensions of the matrix to be made. We could have done the same thing with the one-dimensional array if we had wanted to, i.e.,

```
> (make-array ´3)
#(NIL NIL NIL)
```

And secondly, the `2A` says that this is a rank 2 array (i.e., a matrix). A vector has rank one. It is just a list. A matrix, however, can be thought of as a list of lists. I.e., each row (or column) is itself a list. We can continue this line of reasoning and think of objects that are lists of lists of lists. For instance, suppose that we have $2 \times 2$ matrix, each element of which is itself a vector. Here is an example of such a beast. It is a $2 \times 2$ matrix each element of which is a length 2 vector:

```
> (make-array ´(2 2 2) :initial-element 0)
#3A(((0 0) (0 0)) ((0 0) (0 0)))
```

The *rank* of this thing is 3.

For arrays of rank 1 (vectors) we used `elt` to access the elements. For arrays of higher rank we need the function `aref` function. This works the same way, you just need to give it more index information. E.g.,

```
> (def x (make-array ´(3 3)))
X
> x
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
> (dotimes (i 3)
    (dotimes (j 3)
      (setf (aref x i j) (+ i j))
      )
    )
NIL
> x
#2A((0 1 2) (1 2 3) (2 3 4))
> (aref x 0 0)
0
> (aref x 1 2)
3
```

This works for higher rank arrays too:

```
> (setf x (make-array ´(2 2 2) :initial-element 0))
#3A(((0 0) (0 0)) ((0 0) (0 0)))
(setf (aref x 0 0 1) pi)
3.141592653589793
> x
#3A(((0 3.141592653589793) (0 0)) ((0 0) (0 0)))
```

[1]To print matrices in a pretty fashion, try the `print-matrix` function.

## 4.2 Document Strings

Immediately after the arguments in a function definition you can (and likely should) add a string giving documentation. For instance

```
> (defun p-norm (x p)
    "Args (x p):  computes the p-norm length of a vector x."
    (expt (sum (expt (abs x) p )) (/ 1 p) )
    )
P-NORM
> (p-norm ´(1 1) 2)
1.4142135623730951
> (p-norm ´(1 1) 1)
2
> (help ´p-norm)
P-NORM                                                         [function-doc]
Args (x p):  computes the p-norm length of a vector x.
```

Notice that in this example, the function `p-norm` is perfectly happy if you give it a number instead of a vector:

```
> (p-norm 2 2)
2.0
```

## 4.3 Formatted I/O

We've already seen examples of using the `print` function. Common Lisp supports a C style formatting. Here are a few examples.

```
> (format nil "test number ~d:  use d to print an integer" 1)
"test number 1:  use d to print an integer"
> (format nil "test number ~d:  use f for real numbers such as ~f" 2 3.14159)
"test number 2:  use f for real numbers such as 3.14159"
> (format nil "test number ~d:  use e for exponentials such as ~e" 3 3e+4)
"test number 3:  use e for exponentials such as 3.0E+4"
> (format nil  "test number ~d:  use s for strings such as ~s" 4 "this)"
"test number 4:  use s for strings such as \"this\""
```

and so on. This is as fancy as we will get in this course. For more details see [Tie90]. The first argument to the `format` function is either `t` or `nil` depending on whether you want `format` to return a value to standard output or a file stream. For most of our purposes it won't matter whether you use `t` or `nil`. Writing to file streams was described on page 13. In place of `t` or `nil` we could insert any open file stream.

## 4.4 Reading Multi-column Data

In one class, 16 pairs of students tossed needles on the gymnasium floor. The results were organized in three columns (number of hits, misses and tosses):

```
57      46      103
95      88      183
59      48      107
53      47      100
75      60      135
63      61      124
65      36      101
61      39      100
50      50      100
80      46      126
56      47      103
51      51      102
65      31      96
59      48      107
66      50      116
95      45      140
```

Since it is so common to have column-oriented ascii data, Lisp has a built-in function to accommodate this called `read-data-columns`. Suppose we had an ascii file with the above numbers in our current directory called *needle-toss-data* and wanted to read in the numbers. Here is how we could do it.

```
(setf data (read-data-columns "needle-toss-data" 3))
```

What the interpreter returns is a list of lists, each sublist being one of the 3 columns of data. To extract the first we ask for the first, to extract the second we ask for the second, and so on. We need only the first and third, so:

```
(setf hits (first data))
(setf total (third data))
(setf prob (/ (sum hits) (sum total)))
```

Very simple. The first column corresponds to the number of needles that crossed one of the lines ("hits") and the third corresponds to the total number of tosses for each pair of students. Thus we *estimate the probability of a needle hitting the line by computing the frequency of hits*. In this case that's the total number of hits divided by the total number of tosses. Here is the rest of it.

```
(setf L 1.3125) ; our needles
(setf D 1.5) ; our boards

(setf our-pi-min (/ (* 2 L) (* prob D))) ; our estimate of pi
(format T "lower bound on pi ~f ~%" our-pi-min)
```

We said "lower bound" since it looked like some of the gym boards might be about 1/16 inch shorter than 1.5 inches. If we were to suppose that perhaps 20% of the boards were short, then we might be inclined to estimate $\pi$ as

```
(setf error (/ 1 16))
(def perc-short-boards .25)
(setf our-pi-likely (/ (* 2 L) (* prob (- D (* perc-short-boards error)))))
(format T "our best guess of pi ~f ~%" our-pi-likely)
```

Similarly, we could get an upper bound by assuming (unrealistically) that all the boards were 1/16 short. In any case what we get is

```
lower bound on pi 3.0716666666666668
upper bound on pi 3.2052173913043482
our best guess of pi 3.104
```

So it looks like Buffon may have been on to something here but no fancy $\pi$-computing program has anything to worry about from needle tossing.

## 4.5 Predicates and Conditional Evaluation

Lisp has all the usual algebraic predicates such as `< > <= >=`[2] and so on. They return true or false depending on the evaluation of their arguments. Here are some examples.

```
> (> 1 2)
NIL
> (> 2 1)
T
> (> 2 1 0)
T
> (> 2 1 2)
NIL
```

In the last two examples you see that `>` can take more than two arguments. It returns true or false according to whether or not its arguments are in descending order. The inequality is strict. Thus

```
> (> 2 1 1)
NIL
> (>= 2 1 1)
T
```

We have already seen examples of how to use `if` to do conditional evaluation. Lisp has another useful function called `cond` which allows you to make any number of conditional evaluations at once. For instance we could define a function that returns the sign of its argument if the argument is nonzero and zero otherwise.

[2] I.e., less than, greater than, less than or equal to and greater than or equal to respectively.

```
(defun sgn (x)
  (cond ( (> x 0) (format nil "+") )
        ( (< x 0) (format nil "-") )
        ( (= x 0) (format nil "0") )
        )
  )
SGN
> (sgn -1)
"-"
> (sgn 23.1)
"+"
> (sgn 0)
"0"
```

You see that `cond` works its way through the *predicate expressions* until it find one that evaluates to true, then it evaluates the corresponding *consequent expression*. The consequent expression must be written such that it should evaluate to `nil` for false or any non-`nil` value for true.

### 4.5.1 And and Or

The syntax for `and` and `or` are as follows: `and` evaluates in order from left to right. If any argument evaluates to `NIL`, it returns immediately with the value `NIL`. Otherwise it returns the value of the last argument. Similarly `or` evaluates arguments in order from left to right. If any argument evaluates to non-`NIL`, it quits and returns that value. If the last argument is reached, `or` returns whatever value it returns.

Some examples. First we define a function to tell us whether a given number `x` is in the closed interval `[a,b]`.

```
> (defun in-range (x a b)
    (and (>= x a) (<= x b))
    )
IN-RANGE
> (in-range 1 2 3)
NIL
> (in-range 2.1 2 3)
T
>
```

We could easily define a `not-in-range` function along the same lines, but it's much easier simply to look at the `not` of the `in-range` function, as in

```
> (defun not-in-range (x a b)
    (not (in-range x a b))
    )
NOT-IN-RANGE
```

```
> (not-in-range 1 2 3)
T
> (not-in-range 2.1 2 3)
NIL
>
```

# Chapter 5

# Functions

## 5.1 Optional and Keyword Arguments

We can define two kinds of additional arguments to functions: keyword arguments and optional arguments. For instance, using the `p-norm` function as an example we could define a keyword `verbose` that would cause the function to be more prolix in its output:

```
> (defun p-norm (x p &key verbose)
    (let ((norm (expt (sum (expt (abs x) p )) (/ 1 p) )))
      (if verbose (format t "l-~d norm = ~f" p norm) norm))
   )
```

The term `verbose` is what *we have chosen* to call the keyword. So now if we call the function with the keyword verbose being true, we get

```
> (p-norm ´(1 1) 2 :verbose t)
l-2 norm = 1.4142135623730951
```

whereas if we do not use the keyword we get the default behavior

```
> (p-norm ´(1 1) 2)
1.4142135623730951
```

which is the same as

```
> (p-norm ´(1 1) 2 :verbatim nil)
1.4142135623730951
```

There are also *optional* arguments in Lisp that do not require that a `:keyword` be specified. But, not surprisingly, this requires that you give the optional arguments in the precise order they were defined. With a `:keyword`, the keywords can be in any order. Otherwise optional arguments are quite similar to keyword arguments. Consult [Tie90] (section 4.2.2) for more on optional keywords.

## 5.2 Symbols

A Lisp symbol is more complicated than symbols in languges such as C and Fortran. In Lisp a symbol has four parts:

- a print name
- a value cell
- a function cell
- a property list

We will not discuss property lists at all. So let us look at the other parts. For instance, if we

```
(def f 10)
```

then the symbol `f` has the value 10. We can see this either by asking the interpreter to evaluate `f`

```
> f
10
```

or by using the `symbol-value` function

```
> (symbol-value ´f)
10
```

Now the same symbol can represent a function too. We haven't given `f` a function cell yet, so if we ask for it, we get an error:

```
> (function-value ´f)
Error: The function F is unbound.
```

So let's make `f` a function:

```
(defun f (x)
    (* x x)
)
```

Thus we can

```
> (f 5)
25
```

and

```
> (symbol-function ´f)
#<Closure-F: #80efc88>
```

We will have more to say about "closures" in just a bit, but for now notice that by giving `f` a function cell we didn't disturb its value cell:

```
> (symbol-value ´f)
10
```

This is why we can get away with doing something like

```
> (sin 1)
0.8414709848078965
> (setf sin 1)
1
> sin
1
> (sin 1)
0.8414709848078965
```

## 5.3 What Functions Return

Pay close attention to what your functions return. For instance, notice that if we say

```
> (print 5)
5
5
```

You see 5 twice. The first 5 is a result of the `print` itself, while the second is the value returned by the `print` function. We could capture this returned value if we like via

```
> (setf x (print 5))
5
5
> x
5
```

whereas

```
> (dotimes (i 5) (print i))
0
1
2
3
4
NIL
```

because (as you will see if you type `(help ´dotimes)`) the `dotimes` function returns `NIL` by default.

## 5.4 Functions as Arguments

We have a symbol `f` with which we have associated both a value and a function. If we want to obtain the function associated with a symbol we use the function `function`. A typical scenario would be that we want to make a plot of `f`. We can't just call `plot-function` with `f` as a argument, we must give `plot-function` the function associated with `f`:

```
> (plot-function (function f) 0 10)
```

This code will plot the function $f(x) = x^2$ on the interval $(0, 10)$.[1] There is a shorthand notation for `(function)`. In this example it would be:

```
> (plot-function #'f 0 10)
```

## 5.5 Bindings, Environment and Scope

If we use `def` to assign a value to a variable, then the variable is said to be *global* in that once defined it remains in existence until it is redefined (or undefined) or until you quit the interpreter. Thus

```
> (def x 1)
X
```

*binds* the variable `x` to the value 1. We make *local bindings* of variables in functions when we call the function with an argument. Thus

```
> (def x 1)
X
> (defun f (x) (+ x 1))
F
> (f 99)
100
```

Here is a slightly more complicated example.

```
> (def y 101)
Y
> (defun f (x) (+ x y))
F
>(f 99)
200
```

In this example `y` is called a *free variable* since it is not bound to any of `f`'s arguments. In order to be able to evaluate the function the interpreter must get the value of `y` from the *global environment*.[2]

But suppose we define a new function (say `g`) which calls `f`. Such as

```
> (defun g (y) (f 99))
```

How will the interpreter evaluate `(g 1)`? Well, 1 is locally bound to `y`. So one possibility is to use this local binding when `f` is evaluated. This would cause `(g 1)` to evaluate to 2. Another possibility is to use the binding for `y` that was in effect when `f` was bound. Both approaches are used. The former is called *dynamic scoping*, while the latter is called *static* or *lexical scoping*. By default Common Lisp uses lexical scoping, so in the above example `g` evaluates to 200 no matter what its argument. You can use dynamic scoping if you want by defining *special variables* but we will not do so in this course.

[1] Functions of two-variables can be plot with `spin-function`.

[2] An environment is collection of variable bindings in effect at a given time.

## 5.6 Anonymous Functions

If we ask you to write down a function that squares its arguments, you are likely to do something like:

$$f(x) = x^2$$

by which you mean $f$ is the name of a function of one variable $x$ that returns the square of $x$. And the function that cubes its arguments?

$$f(x) = x^3.$$

Now the $f$ is pretty much along for the ride. We could have called our function anything but it's convenient to call it something. Well, sometimes it's convenient to call it something. But sometimes it's a pain. For instance, if all we wanted to do is plot the function that squares its arguments, wouldn't it be nice if we could dispense with the superfluous naming?

Here is how we can do this in Lisp:

```
> (plot-function #'(lambda (x) (* x x) ) 0 10)
```

Or if we wanted to evaluate "the function that squares its arguments" when that argument is 5 we would type

```
> ( (lambda (x) (* x x)) 5)
```

What we've done here is invented a new symbol $\lambda$ to be used to refer to these generic (or anonymous) functions. I.e., function where you don't care what the function is called, you just want to use it. This $\lambda$ *calculus* was invented by the logician Alonzo Church. John McCarthy, the original developer of Lisp, was a student of Church's. A mathematical expression such as

$$\lambda(x) = x^2$$

should be read: *the function that returns $x^2$ for the argument $x$.*

Related to the `function` function, is the very useful `funcall`. `funcall` can be used to apply a function to given arguments. For example[3]

```
>(defun sample-function (the-function samples sample-interval)
  "ARGS: the-function samples sample-interval. Produces a sampled
   version of a given function. Eg., (sample-function 'sin 8 (/ pi 8))"
  (let* ((output (iseq samples))
         (output (* sample-interval output)))
    (funcall the-function output)
    )
  )
SAMPLE-FUNCTION
> (sample-function 'sin 3 (/ pi 8))
(0.0 0.3826834323650898 0.7071067811865475)
```

[3]Contributed by Prof. Hadsell

# Chapter 6

# Lisp-Stat: Exercise II

The description of this exercise is very long. But that's because we spend a lot of time going over the basic Lisp-Stat that you will need. So think of this as an extension of the lecture notes. We will spend several weeks going over these ideas.

In the end you will be turning in a Lisp-Stat code that will read in data, plot it, annotate the plots properly (title, axis labels), do some statistical analysis of the data and plot this information as well. We will summarize exactly what you must turn in in the end. But for now let's go through the exercise step by step.

### 6.0.1 The Basic Problem

In this exercise you will get some experience analyzing data. The first step is to download the data. The Arctic System Science (ARCSS) Data Coordination Center

`http://arcss.colorado.edu/`

maintains a repository of data related to climate, ocean/atmosphere/land interaction, human and physical geography of arctic regions. If ftp to

`ftp://arcss.colorado.edu/pub/projects1/icecore/ecm/`

you will see numerous files containing electrical conductivity measurements taken on ice cores removed from two boreholes drilled into the Greenland Ice Sheet. We won't look at all of these, but download the following one or two files such d1500.ecm and d1550.ecm. Together these two files contain 100 meters of measurements. Also get the file aaaaa.ecm, which describes the measurements. If you are using Netscape, you can select "save as" under the *file* menu and store these files in a convenient location.

### 6.0.2 Reading the Data

You'll need to read the data files into Lisp-Stat. The `read-data-columns` function is perfect for this. If the data files are not in the current working directory, you can give the full pathname in quotes when you read the data:

```
> (def data (read-data-columns "/home/myname/datafile-name" 2))
```

which assumes that a file named `datafile-name` is in the directory `/home/myname` and has 2 columns of numbers.[1]

### 6.0.3 Plotting Data and Interacting with the Plots

In this exercise you will analyze these data. You will need to write various functions as well as use some of Lisp-Stat's functions to do the analysis. However for starters let's talk about plotting the data and how we can interact with the plots via message passing, once the plots have been made.

Since the data file is two-column, we can plot it straight away without having to worry about generating the x-values.

```
> (plot-points data)
```

or

```
> (plot-lines data)
```

if you want to joint the points with straight lines. But as with the data themselves, it is a good idea to assign the plot object to a symbol. E.g.,

```
> (def my-plot (plot-lines data))
```

Now we can talk to `my-plot` by sending it messages. This is such a useful concept that you will be sending messages all over the place once you get the hang of it. So **please slow down and go over this carefully**. The first message you might send to an object is help. Our object is now called `my-plot`, so we can do this as follows:

```
> (send my-plot :help)
SCATTERPLOT-PROTO
Scatterplot
Help is available on the following:

ABLINE ACTIVATE ADD-FUNCTION ADD-LINES ADD-METHOD ADD-MOUSE-MODE
```

and a whole lot more in addition.

The `SCATTERPLOT-PROTO` message will be explained later when we talk about object-oriented programming. Each of the keywords beginning with `ABLINE` refers to a message that you can send your plot object. You can get help on each of these keywords by giving it as an argument to help:

```
> (send my-plot :help :add-lines)
ADD-LINES
Method args: (lines &key type (draw t))
or:          (x y  &key point-labels (draw t))
Adds lines to plot. LINES is a list of sequences, the
coordinates of the line  starts. TYPE is normal or dashed. If
DRAW is true the new lines are added to the screen. For a 2D
plot LINES can be replaced by two sequences X and Y.
```

[1] If we had used `setf` instead of `def` the numbers themselves would have been spewed to the screen. `def` simply returns the variable name. `setf` actually does more than this (see the `help` on it), but for our purposes we can use it and `def` interchangeably.

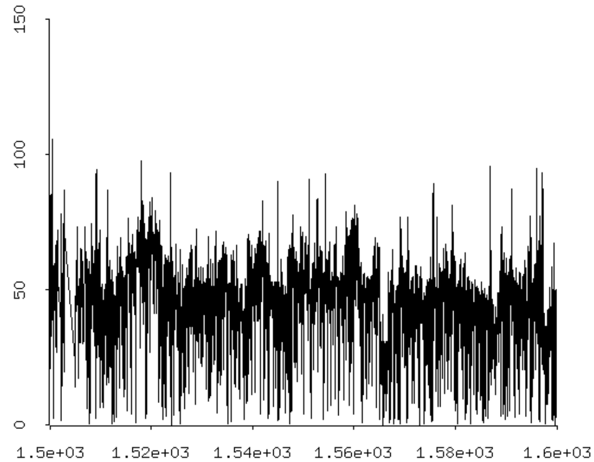


Figure 6.1: A small portion of the electrical conductivity data from ARCSS.

A *method* is a piece of code that is executed in response to a message. So the method for the `add-lines` message can be used to add more lines to a plot. In the present example suppose that we read the first ECM data file into `data` and the second into `data1`

```
> (def data1 (read-data-columns "d1550.ecm" 2))
```

Then we can add the new points to `my-plot` by sending the following message to `my-plot` (assuming `my-plot` already contains `data`):

```
> (send my-plot :add-lines data1)
```

Now if you try this you won't see all the new data at first, but that's because `my-plot` is not automatically re-scaled when you send it new lines or points. But there is a method for this:

```
> (send my-plot :help :adjust-to-data)
ADJUST-TO-DATA
Method args: (&key (draw t))
Sets ranges to the actual range of variables in the original coordinate
system. If DRAW is true sends :RESIZE and :REDRAW messages.
```

So all we need to do is

```
> (send my-plot :adjust-to-data)
```

The result should look like Figure 6.1.

There are a number of other messages you will probably want to be able to use. Since reading Lisp-Stat's help is sometimes like drinking from the proverbial fire-hose, below are *some* useful scatterplot methods you may wish to know about:

```
ADD-FUNCTION
ADD-LINES
BACK-COLOR
CURRENT-VARIABLES
```

```
DRAW-STRING
LINE-WIDTH
POINT-COLOR
SHOWING-LABELS
TITLE
VARAIABLE-LABEL
```

Adding labels with the `VARIABLE-LABEL` method is slightly tricky. For your plot there are two variables. You can discover this by sending the following message:

```
> (send my-plot :current-variables)
(0 1)
```

Variable 0 is the x-axis and variable 1 is the y-axis. By default the variables are empty

```
> (send my-plot :variable-label 0)
""
```

but you can set this string to something easily, e.g.

```
> (send my-plot :variable-label 0 "X-axis")
"X-axis"
```

One minor hitch: before the labels will actually appear on the plot you have to set the `showing-labels` state to true. So send this message with the `t` argument or pull down the menu and select this item.

Later on, you will be adding lines to your plots; lines computed either by smoothing the data or by doing a *regression*.[2] For instance, here is how you could add a straight line to your plot:

```
> (send my-plot :add-lines ´((1500 1600) (50 50)))
```

Thus, we've given the method for `add-lines` a list of x and y coordinates. So the line will connect the two points $(1500, 50)$ and $(1600, 50)$.

### 6.0.4 First Task

The first task is to write two function. Let's call the first `running-mean` and the second `running-std`. The function `running-mean` should take a vector and an integer and return a running average of the vector. The integer will be the length of the averaging window. This works as follows. Suppose the window length is `m`. Then take the first `m` points of the vector and compute their mean. Let these be the first element of a new vector (the one that will be returned by your function). Now slide the window down your vector one slot. The *window* is now sitting over the points from 1 to `m`+1. (Remember, Lisp indexes everything from 0.) Take the mean of these `m` points and stuff that in the second slot of the new vector. Continue on in this fashion until you reach the end of the vector. There is a tricky bit at the end having to do with how you treat the last `m` points of the vector, but since there is no *best* answer to this, make a reasonable choice on your own. To compute the mean of a set of points you can use the built-in Lisp-Stat function `mean`.

The net result of this will be a vector representing a smoothed version of the original data. This is illustrated in Figure 6.2.

[2]Fitting a straight line. We'll define this later.

running average of length 3

data (1, 2, 5, 1, -2, 4, 5, 2, -7, 0, 12, 3, 4, 1)

$$\frac{1+2+5}{3}, \frac{2+5+1}{3}, \ldots$$

smoothed data (8/3, 8/3, 4/3, 1, .... )

Figure 6.2: Running average of the ECM data.

The degree of smoothness will depend on the length of the window. If $m = 1$, then your not doing any smoothing at all. As `m` gets bigger and bigger, you should see your data get smoother and smoother.

**Note:** `data` and `data1` are two-dimensional lists, having both the x- and y-values. If you want just the x-values you extract the first member of this list of lists as:

```
> (def x-values (first data))
```

and if you want just the y-values you can do

```
> (def y-values (second data))
```

Next write a second function, exactly the same as `running-mean` except that instead of taking the mean of the window, take the standard deviation. The standard deviation represents the *spread* of data. (Our *data* are the y-values.) It will be explained in detail in the class. You can do this with the built in `standard-deviation` function of Lisp-Stat. So here the window size represents a sort of *sub-population* of the original data in which you are computing the spread. If you plot this `running-std` you will see the standard deviation of your data as a function of the depth along the log.

If we regard the rapid fluctuations we see in the data as being associated with measurement errors, then the standard deviation as a function of depth represents the uncertainty of the measurement at that depth. The standard deviations are also related to what people call the *error bars* of the measurements.

### 6.0.5 Analyzing the Data

Once you've figured out how to smooth one-dimensional arrays of numbers you can subtract the smoothed version of the ice core data from the original to see the fluctuations about this smooth trend. These fluctuations should look more or less bell-shaped when you make a histogram. So if you do something like the following

```
> (def ecm-data (append (second data) (second data1)))
ECM-DATA
> (def fluctuations (- ecm-data (running-mean ecm-data 20)))
FLUCTUATIONS
```

Then `fluctuations` is now a vector containing the difference between the raw measurements and the measurements smoothed over a window of length 20. The function `append` constructs a new list by concatenating its arguments. You can make a histogram of the fluctuations with the Lisp-Stat `histogram` function. Does it look like a bell-shaped curve? [3]

### 6.0.6 Regression

Now when you plot your data you will see several things immediately. First the data are rather *noisy*, i.e., there are lots of high-frequency fluctuations. Secondly there are some longer scale variations, on the order of 10-30 meters. And finally there is an overall linear shape to the curve. A common issue in the analysis of data is whether a given curve "fits the data." In this case we might ask whether a straight line reasonably fits the data. We, being probabilists, must address the issue at a more fundamental level than simply eye-balling a line through the data. Fortunately for us, Lisp-Stat is well suited for this task. What we are going to do is use the built-in `regression-model` function. We will be explaining the probability theory behind this function during the class. For now we will be a little loose with definitions.

Recalling that we can extract the x and y values from the data via

```
(def x-values (first data))
(def y-values (second data))
```

let us call Lisp-Stat's `regression-model` function and see what we get. We need to assign the resulting object a name so that we can send it messages so we might see somthing like the following:

```
> (def regmod (regression-model x-values y-values))

Least Squares Estimates:

Constant                    161.555      (6.37038)
Variable 0                 -7.276398E-2  (4.173528E-3)

R Squared:                  6.613157E-3
Sigma hat:                  12.3536
Number of cases:              45662
Degrees of freedom:           45660

REGMOD
```

We will explain in class what these messages mean, but for now let's simply extract the information we need and plot it.

```
> (def my-fit (send regmod :fit-values))
MY-FIT
> (def my-straight-line (list x-values my-fit))
```

[3]A bell-shaped curve is also known as a Gaussian or normal distribution. The *standard normal distribution* is a normal distribution with zero mean and unit standard deviation. You can always convert a normal distribution to standard form by subtracting the mean (and thus making the result have zero mean) and dividing by the square of the standard deviation (thus making the result have unit standard deviation).

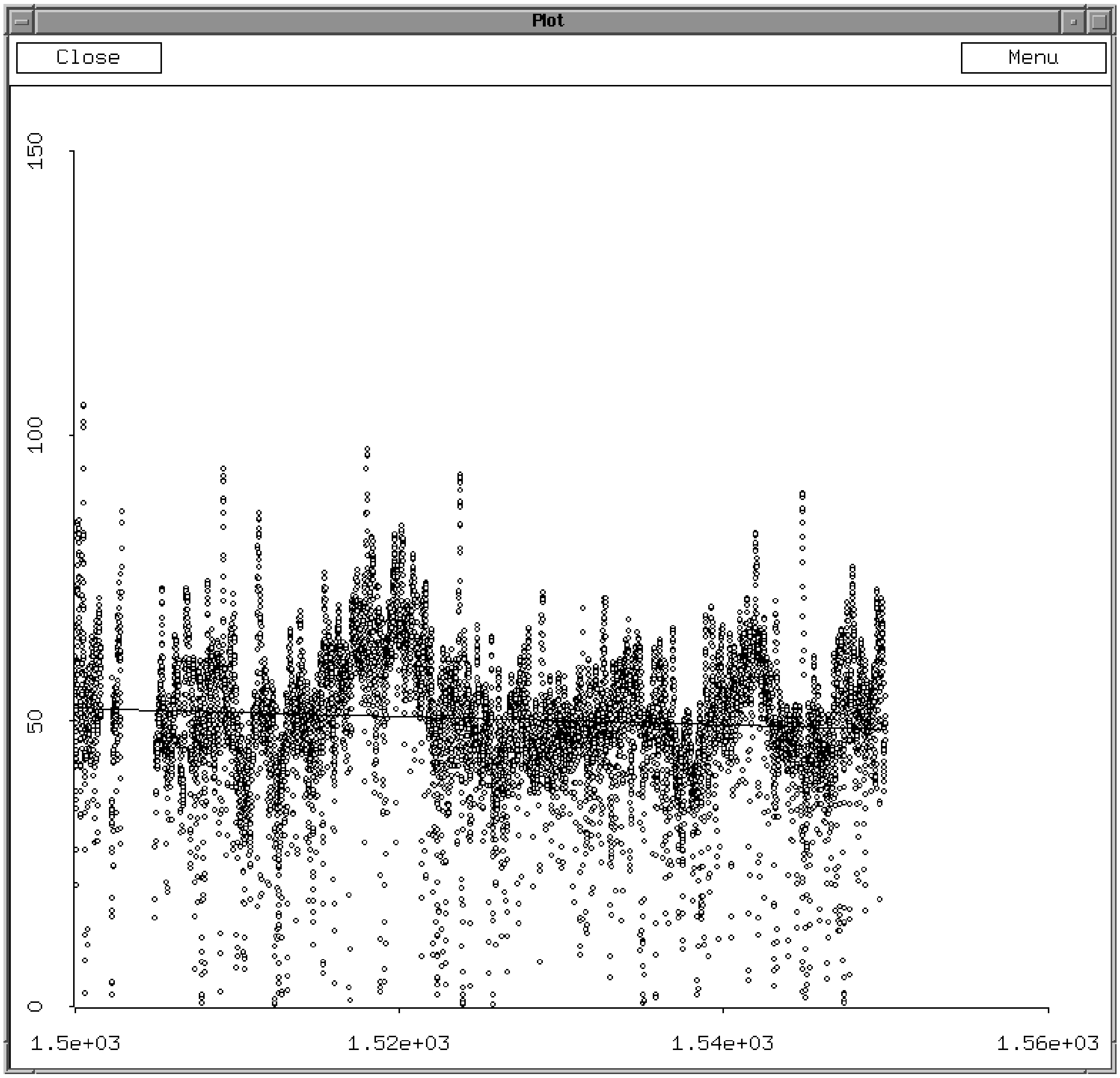


Figure 6.3: Best least-sqruares fitting straight line.

```
FITTED-VALUES
> (send my-plot :add-lines my-straight-line)
```

The first line extracts the fit values from the regression object. These are the y-values of the best-fitting straight line. And the second defines a new straight line with these y-values and the original x-values (the depths of the measurements). Then we simply use the `add-lines` message to add this fitted straight line to the original data. The result will be a plot showing the raw measurements as a function of depth and a straight line that is optimal in a sense that will be explained in class. The results of this regression, when applied to the first of the ice-core files `d1500.ecm` is shown in Figure 6.3.

Here then is a working regression example using synthetically generated "data."

```
(def data (list (iseq 1 100) (uniform-rand 100)))
(def my-plot (plot-lines data))
```

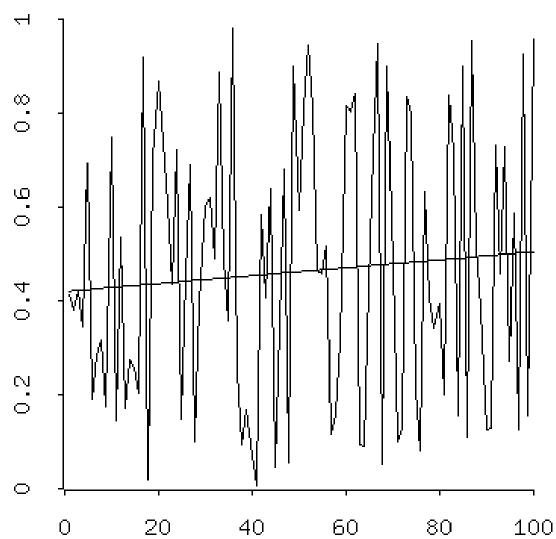


Figure 6.4: Regression fit to height/weight data.

```
(def regmod (regression-model (first data) (second data)))
(def my-fit (send regmod :fit-values))
(def my-straight-line (list (first data) my-fit))
(send my-plot :add-lines my-straight-line)
```

With the result that

```
Least Squares Estimates:

Constant                  0.465094      (5.688591E-2)
Variable 0                 7.430068E-4  (9.779612E-4)

R Squared:                 5.855523E-3
Sigma hat:                0.282299
Number of cases:               100
Degrees of freedom:             98
```

And the plot that is produced is shown in Figure 6.4.

This plot has no title, axis labels or other information. You'll want to put this sort of information on your plots.

What you should see is that the slope of the best-fitting straight line is pretty small. There is a noticeable slope but is it large enough to be consider significant? In other words, suppose someone stood up and said: the slope of this line is essentially zero. Would it be possible for us to refute this claim statistically? We are not quite in a position to be able to do this, but after the next class we will be.

## 6.1 Summary of What You Need to Do

You need to turn in a file with all the code necessary to do the following tasks. Annotate your code with comments.

- Download the two ECM datasets. If it turns out that dealing with both datasets is too slow on your computer, just use one.
- Write functions that compute running means and running standard deviations of arrays given a window length. These should be, more or less, of the form `(defun running-mean (x m) ...` where `x` is a vector and `m` is the window length.
- Plot the smoothed data and raw data together.
- Subtract the smoothed measurements from the raw. These are the fluctuations in the data. Make a histogram of these fluctuations. Does it look like a bell-shaped curve?
- Fit a straight line to your unsmoothed data using `regression-model`. Plot the straight line on top of the data. You can also plot the straight line and the smoothed data for comparison.

## 6.2 Reression Example

Here is an example of regression applied to a problem described in the splendid "Cartoon guide to statistics" by Gonick and Smith [GS93]. The code does the problem two ways: by hand, and using Lisp-Stat's built-in `regression-model`.

```
; define some data
(def height ´(60 62 64 66 68 70 72 74 76))
(def weight ´(84 95 140 155 119 175 145 197 150))

; make a plot
(def my-plot (plot-lines height weight))

; do a linear regression
(def regmod (regression-model height weight))
; extract the fit values from the regression
(def my-fit (send regmod :fit-values))

; make a straight line from the original x-values and the new
; regression values
(def my-straight-line (list height my-fit))
; send this new straight line to the plot
(send my-plot :add-lines my-straight-line)
```

Here is what is returned by Lisp-Stat Least Squares Estimates:

```
Constant                  -200.000      (110.690)
Variable 0                 5.00000      (1.62312)
```

```
R Squared:                 0.575484
Sigma hat:                  25.1453
Number of cases:                  9
Degrees of freedom:               7
```

I.e., height = -200 + 5 weight

Now we can do the calculation by hand. Let's define x and y variables so as to keep in mind which is the dependendent and which is the independent variable.

```
(def x height)
(def y weight)
(def xbar (mean x))
(def ybar (mean y))

(def SSxx (sum (^ (- x xbar) 2)))
(def SSyy (sum (^ (- y ybar) 2)))
(def SSxy (sum (* (- x xbar) (- y ybar))))
(def b (/ SSxy SSxx))
(def a (- ybar (* b xbar)))

(format nil "line is y = ~f x + ~f" a b)
(format nil "or")
(format nil "line is weight = ~f height + ~f" a b)

; now we already have the fit values from our call
; to regmod, but let´s continue to do things by hand
(def my-fit (make-array 9 :initial-contents (+ (* b x) a)))

; analysis of variance
(def SSR (sum (^ (- my-fit ybar) 2)))
(def SSE (sum (^ (- my-fit y) 2)))

; squared correlation
(def R2 (/ SSR SSyy))
```

Notice that `R2` equals `(- 1 (/ SSE SSyy))`. This is not a coincidence. R2 measures the goodness of fit. R2 = 1 corresponds to a perfect fit. In this particular case it turns out that R2 = .5755 We interpret this result as saying that 58% of the variation in weight is explained by height. The other 42% is "error". Some people use the correlation coefficient `(def r (* (signum b) (sqrt R2)))` which in this case turns out that r = .7586 What does signum do?

## 6.3 A Few Useful Tools

Here is a simple function that takes a list of paired data, like like the ice core measurements `data` above, and returns a list containing every other sample.

```
(defun decimate (data)
"returns ever other element of (X,Y) compound number-data"
    (let* ((y (first data))
           (x (second data))
           (decimated-data (list (select y (which (evenp (iseq (length y)))))
                                 (select x (which (evenp (iseq (length x)))))
                                 )
             )
           )
      decimated-data)
    )
```

Here we used `select` to select a subarray from a larger array. The `(which (evenp`... says to choose only the even-numbered elements. We can call `decimate` repeatedly to make smaller and smaller subsets of the data. For instance

```
(def newdata (decimate (decimate (decimate data))))
```

# Chapter 7

# Object-Oriented Programming

## 7.1 Creating Objects

We have already taken advantage of Lisp-Stat's *object-oriented* features. In the last exercise we focused attention on sending messages to objects returned from the plot and regression functions. The ability to receive and act upon messages is one of the defining features of an object. We call the piece of code that responds to a message is a *method*. In addition to methods, objects contain information specific to the object in named locations called *slots*. For instance let us create an object

```
> (def myplot (histogram (uniform-rand 100)))
MYPLOT
```

We can check that this is an object with the `objectp` predicate:[1]

```
> (objectp myplot)
T
```

The name of any object's slots can be returned with the `own-slots` message.

```
> (send myplot :own-slots)
(HARDWARE-ADDRESS SUBORDINATES GO-AWAY SIZE LOCATION TITLE
INTERNALS MENU-TITLE HAS-V-SCROLL HAS-H-SCROLL BLACK-ON-WHITE
MENU SLICERS MOUSE-MODE FIXED-ASPECT SHOWING-LABELS VARIABLE-LABELS
NUMBER-OF-VARIABLES MENU-TEMPLATE OPTIONS-TEMPLATE SCALE-TYPE
OVERLAYS HISTOGRAM-INTERNALS)
```

But how did `myplot` get these slots? The answer is that it *inherited* them from `histogram` somehow. As we shall see in this chapter, Lisp-Stat uses two kinds of objects. The first is just like `myplot` above. If we made 10 histogram plots, we would have 10 instances of the same type of object. However, Lisp-Stat also uses a more fundamental kind of object, called a *prototype* object, the main purpose of which is to create other objects. In this case, `histogram` is a prototype object (or simply a prototype) from which we created `myplot` and from which `myplot` inherited the slots mentioned above. This sort of prototype-based object system has

[1] In Lisp it is traditional end the name of logical predicates with a `p`.

some significant differences from the object systems used in CLOS, C++, and many other OO languages. CLOS and C++ are examples of *class-based* languages. Classes are specifications of the object in software. Objects themselves are *instances* of classes, which can be declared at run-time. But the classes themselves are fixed when the source code is compiled and cannot be changed dynamically.

If we type the objects name into the interpreter, we get a printable form of the object. For instance if we type the name `myplot` into the interpreter we get

```
> myplot
#<Object: 135147096, prototype = HISTOGRAM-PROTO>
```

`HISTOGRAM-PROTO` is the name of the histogram prototype and the number is used by the object system to refer to the particular object. We can ignore these numbers.

But let us begin with a much simpler object than `myplot`. Let's build an object from scratch using the most basic of all object prototypes, simply called `*object*`.[2]

```
> (setf x (send *object* :new))
#<Object: 135291288, prototype = *OBJECT*>
```

The `:new` message tells the object to create a new object with itself as the prototype. Thus, having created `x`, we could

```
> (setf y (send x :new))
#<Object: 949496, prototype = *OBJECT*>
```

At this point `x` has no slots of its own

```
> (send x :own-slots)
NIL
```

but it can respond to messages for which methods have been defined for `*object*`.

```
>(send x :help)
*OBJECT*
The root object.
Help is available on the following:

ADD-METHOD ADD-SLOT DELETE-DOCUMENTATION DELETE-METHOD
DELETE-SLOT DOC-TOPICS DOCUMENTATION GET-METHOD HAS-METHOD
HAS-SLOT HELP INTERNAL-DOC ISNEW METHOD-SELECTORS NEW
OWN-METHODS OWN-SLOTS PARENTS PRECEDENCE-LIST PRINT PROTO
REPARENT RETYPE SHOW SLOT-NAMES SLOT-VALUE
```

[2] *Global variables* are accessible everywhere except in expressions that use the same name to create a local variable. To avoid such collisions it is conventional to give global variables names beginning and ending with asterisks. `*object*` would be pronounced "star-object-star."

## 7.2 Adding Slots to Objects

We can add as many slots as we like to `x`. Let's begin by giving `x` a title and some data. We add slots with the `add-slot` message

```
(send x :add-slot ´title "a time series")
(send x :add-slot ´data)
```

In the first case we not only added a slot called `title` but we assigned some data to this slot, namely the string `"a time series"`, by giving this string as an optional argument to `add-slot`. In the second case, we simply defined a slot called `data` with nothing in it. We could also given our `data` slot some numbers at the same time via

```
> (send x :add-slot ´data (uniform-rand 5))
(0.17167213074158214 0.7431496373080966 0.35840563962669636
0.48164240825885024 0.4559074508644821)
```

or, having defined the slot name as before, we could send `x` a message to stuff something in this slot:

```
> (send x :slot-value ´data (uniform-rand 5))
(0.33076277705879775 0.7202696423942395 0.47190002868488945
0.14929668590608047 0.6913648353595061)
```

## 7.3 Adding Methods to Objects

But slots are static things. They hold information: data, strings, whatever. The don't *do* anything. That is what *methods* are for. To define our own methods we use `defmeth`. For example, suppose we want to give `x` the ability to make a histogram of itself. We'll define a method that responds to a `:plot` message:

```
(defmeth x :plot ()
  "make a histogram of the numbers in the data slot"
  (histogram (send self :slot-value ´data))
  )
```

Here we see that the arguments to `defmeth` include, in order: the name of the object (`x`), the *selector* or message name (`:plot`), a list of parameters (here empty, about which more later), an optional documentation string (any string before the body of the method is automatically taken to be a documentation string), and finally the body of the method (in this case a call to `histogram`). So now we can

```
> (send x :help :plot)
PLOT
make a histogram of the numbers in the data slot
```

or

```
> (send x :plot)
#<Object: 135399368, prototype = HISTOGRAM-PROTO>
```

Here is an example of a method that uses an optional parameter.

```
(defmeth x :summary (&optional (stream t))
  (format stream "this is a ~a~%" (send self :slot-value 'title))
  (format stream "the sample mean is ~f~%"
          (mean (send self :slot-value 'data)))
  (format stream "the standard deviation is ~f~%"
          (standard-deviation (send self :slot-value 'data))))
```

Which we can use by passing a `:summary` message

```
> (send x :summary)
this is a a time series
the sample mean is 0.5783423181318775
the standard deviation is 0.25118173474736666
```

or, if we want to write the summary to a file we can open a stream as we did in Chapter 2:

```
(def myfile (open "summary-file" :direction :output))
```

and then

```
(send x :summary myfile)
```

Notice that at this point (under Unix in any case), if you look at the file you just wrote (`summary-file`), it will probably have nothing in it. The reason for this is that Unix I/O is buffered. The operating system doesn't bothering doing file-writes one bit at a time, it saves up a whole bunch of them in a chunk of memory called a buffer and then writes the whole buffer out when it gets full. This is a lot more efficient, but it means that before you can look at the contents of `summary-file` you may need to close it with a call to `close` such as `(close myfile)`.

## 7.4 Functional Programming and the Curse of Assignment

Earlier we defined the factorial function as

```
(defun factorial (n)
    (if (= n 0)
        1 (* n (factorial (- n 1))))
    )
```

Apart from being a recursive definition (the function is defined in terms of itself), notice that we do not need to do any assignments. Consider an alternative.

```
(defun factorial1 (n)
    (def fact 1)
    (dolist (i (iseq 2 n))
      (setf fact (* i fact))
      )
    fact)
```

This function works and does indeed compute the factorial correctly, but it has an unfortunate side-effect. By defining `fact` as a global variable, we run the risk of this definition interfering with some other calculation which has fact within its scope. For instance

```
> (factorial1 10)
3628800
> fact
3628800
```

Thus `fact` is lurking in the shadows waiting to trip you up. **You should avoid such assignments whenever possible.** Assignments which *leak* outside the body of a function are called *side-effects* and one of the goal of the so-called *functional* style of programming is to avoid such side-effects. Local assignments made within `let` and `let*` have no such side effects. They exist only within the body of the defining `let` or `let*`. For instance, we could do

```
(defun factorial2 (n)
    (let* ((product 1))
    (dolist (i (iseq 1 n))
      (setf product (* i product))
      )
    product)
)
```

and then

```
> (factorial2 10)
3628800
> product
Error: The variable PRODUCT is unbound.
```

As Abelson and Sussman say ([AS96], Section 3.1.3): "In general, programming with assignments forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs." The alternative to functional programming is *imperative programming* in which assignments are made step-by-step. This may seem easier to grasp in the beginning but as Abelson and Sussman say "... it saddles beginning programmers with 'should I set this variable for or after that one' concerns that can complicate programming and obscure the important ideas.".

Here is another example. Consider the task of adding the integers from 1 to 10. In an imperative language such as C or Fortran, this would be expressed as a loop:

```
total = 0;
for (i=1; i<=10; ++i)
    total += i;
```

whereas in a functional language the same calculation would be done without any variable updates. For instance, in Lisp we could use evaluate the following expression:

```
(sum (iseq 10))
```

And again. Suppose we want to plot 10 random numbers in the range $[100, 200]$. Using our old friend the `random-in-range` function

```
(defun random-in-range (low high)
    (let ((range (- high low)))
      (+ low (random range)))
)
```

we might be tempted to begin by defining an array

```
(def random-numbers (make-array 10))
```

and then doing something like

```
(dotimes (i 10)
  (setf (elt random-numbers i) (random-in-range 10 100))
  )
```

This will work in the sense that the globally defined array `random-numbers` contains the results we're looking for, but the code is ugly on two counts. First, it's not general. We've hard-coded in values of the length of the array and the range of numbers. And secondly, it uses global assignments. We can avoid both these problems by defining a function that returns the required array and accepts all the parameters as inputs. For instance

```
(defun random-list (low high n)
  (let* ((my-array (make-array n)))
    (dotimes (i n)
      (setf (elt my-array i) (random-in-range low high))
      )
   my-array)
  )
```

When we call this function, the results are returned as an array

```
> (random-list 10 100 10)
#(91 44 91 12 78 56 78 56 13 56)
```

And we can capture these results in a globally defined array if we want

```
(def my-numbers (random-list 10 100 10))
```

or we can plot them directly

```
(plot-lines (iseq 10) (random-list 10 100 10))
```

The key point is, however, that since we wrote the function `random-list` with no global assignments, we can be sure that the code has no side-effects.

There is a *Usenet* group `comp.lang.functional` devoted to functional programming languages (such as Lisp). Here is the definition of functional from this newsgroups FAQ:

> Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values.
>
> A functional language is a language that supports and encourages programming in a functional style.

## 7.5 More Examples of Methods

You can add a method to any object. For instance

```
> (defmeth *object* :test () (print "it works"))
:TEST
> (send *object* :test)
"it works"
```

adds a method to the root object that simply prints a message.

We can change the contents of a slot with the `:slot-value` method, inherited from the root object, or we can define our own method to do the same. For instance, using `x` as above

```
(setf x (send *object* :new))
(send x :add-slot ´title "a time series")
```

we could define a `:title` method that returns the title if no argument is given, or sets the title if one is:

```
(defmeth x :title (&optional (title nil set))
   (if set (setf (slot-value ´title) title))
   (slot-value ´title)
)
```

Here we have used an *optional* argument called `title`. The *optional* argument allows us to set a default value for `title`, in this case `nil`. The `slot-value` function is one of several in Lisp-Stat that can only be used within the body of a method. We could also have used the `:slot-value` method within the body of our new `:title` method. In any case, the following illustrate the uses of the `:slot-value` method and our new `:title` method:

```
> (send x :title "a time series")
"a time series"
> (send x :title "newtitle")
"newtitle"
> (send x :title)
"newtitle"
> (send x :slot-value ´title "yet another title")
"yet another title"
> (send x :slot-value ´title)
"yet another title"
> (send x :title)
"yet another title"
```

## 7.6 Defining Your Own Prototypes

We can now add methods and slots to objects. But at some point we would like to be able to save ourselves the bother of building up our objects by hand. Elaborating on an example in [Tie90], let us define a time series prototype with slots for the data, title and type of measurement. We will define methods for plotting histograms of the data, making line plots of the samples (suitably annotated from the slot information), returning summary statistics; as well as methods for viewing and changing the slot information.

```
(defproto time-series-proto ´(data title value))

(defmeth time-series-proto :histogram (&optional (stream t))
  (histogram (send self :slot-value ´data))
  )

(defmeth time-series-proto :plot-lines (&optional (stream t))
  (let ((theplot (plot-lines (iseq (length (send self :slot-value ´data)))
                             (send self :slot-value ´data))))
               (if (send self :value)
                   (send theplot :variable-label 1 (slot-value ´value)))
               (send theplot :variable-label 0 ``sample number´´)
               (send theplot :showing-labels)
               (send theplot :adjust-to-data)
               )
  )


(defmeth time-series-proto :summary (&optional (stream t))
  (format stream "this is a ~a~%" (send self :slot-value ´title))
  (format stream "measuring ~a~%" (send self :slot-value ´value))
  (format stream "the sample mean is ~f~%"
          (mean (send self :slot-value ´data)))
  (format stream "the standard deviation is ~f~%"
          (standard-deviation (send self :slot-value ´data))))

(defmeth time-series-proto :title (&optional (title nil set))
  (if set (setf (slot-value ´title) title))
  (slot-value ´title))

(defmeth time-series-proto :value (&optional (value nil set))
  (if set (setf (slot-value ´value) value))
  (slot-value ´value))

(defmeth time-series-proto :data (&optional (data nil set))
  (if set (setf (slot-value ´data) data))
  (slot-value ´data))
```

Now let us make an instance of this time series prototype, initialize the slots and make a plot of the data.

```
(setf x (send time-series-proto :new :data (normal-rand 100)))
(send x :title "time series")
(send x :value "voltage")
```

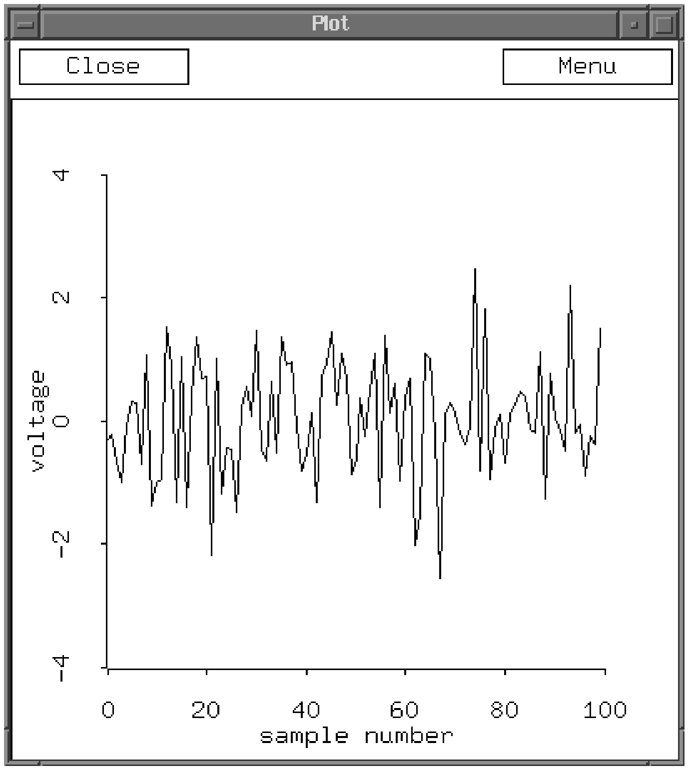


Figure 7.1: The plot-lines method in time-series-proto.

```
(send x :plot-lines)
```

## 7.7 Exercise 3

Define a prototype to represent paired data; i.e., lists of $(x, y)$ values. Call it `paired-data-proto`. It should have 5 slots: the x and y values, a title, and descriptions of what the x and y values measure. Give your prototype a plot method. This method can be as simple or complicated as you like and you can use any plot function to produce it.

Now show how you can use this prototype by creating an object from it, installing information/data in all the slots and then sending it a message to plot itself.

## 7.8 Inheritance

We can define prototype that inherit from other prototypes with `defproto`. For example, suppose we want to create our own generalized histogram prototype that has slots to hold the data and title:

```
(defproto myhistogram-proto ´(data title) () histogram-proto)
```

This new prototype inherits all the methods from `histogram-proto`, but now has two slots of its own. We can see the inheritance tree of any prototype by sending it the `precedence-list` message.

```
> (send myhistogram-proto :precedence-list)
(#<Object: 135153208, prototype = MYHISTOGRAM-PROTO>
#<Object: 135046280, prototype = HISTOGRAM-PROTO>
#<Object: 135061288, prototype = GRAPH-PROTO>
#<Object: 135067528, prototype = GRAPH-WINDOW-PROTO>
#<Object: 135027176, prototype = WINDOW-PROTO>
#<Object: 135039080, prototype = HARDWARE-OBJECT-PROTO>
#<Object: 135013464, prototype = *OBJECT*>)
```

This says that `MYHISTOGRAM-PROTO` inherits from `HISTOGRAM-PROTO`, which in turn inherits from `GRAPH-PROTO`, `GRAPH-WINDOW-PROTO` and so on, all the way back to the root object `*OBJECT*`.

This is an extremely condensed account of Chapter 6 in Tierney's book. Rather than simply reproduce Tierney's superbly clear explanations, you should consult his book for more details.

# Chapter 8

# Graphical User Interfaces

Graphical user interfaces (GUI's, pronounced gooey) are a perfect illustration of the advantages of the object-oriented philosophy. Graphical objects, such as windows, can be represented as software object to which messages can be sent and acted upon. Add a line, add a title, resize the plot, take some action when a button is clicked, etc. Here is a very simple window to which we have added a two-itemmenu.

```
(defun plot-it () (histogram (normal-rand 100)))

(setf simple-menu (send menu-proto :new "a very simple menu"))
(setf histogram-item (send menu-item-proto :new "make a histogram plot"
                       :action #´plot-it ))
(setf lineplot-item (send menu-item-proto :new "make a line line plot"
                          :action #´(lambda () (plot-lines (iseq 100)
                                                           (normal-rand 100))
                                      )))

(send simple-menu :append-items histogram-item lineplot-item)
(send simple-menu :install)
```

The basic structure of the code should be apparent, even without knowing precisely what these new methods do. Basically, we make a menu object by sending a `:new` message to the `menu-proto`. We add two items to this menu by sending `:new` messages to the `menu-item-proto`. In so doing we had to define the action to be taken when each menu item was selected. In both cases functions were called, but for the histogram item we defined a one-line function separately and called it by name, whereas in the lineplot item we used an anonymous function via a lambda expression. No good reason for doing this other than to illustrate the point that you can do this. Finally we send messages to our menu to add these menu items and install them.

And here is a simple example of using a slider bar.

```
; a simple scroll bar example
(setf length 1) ;set a default value of the scroll

; action taken when button is pressed
(defun runit ()
```

```
  (format t "the value of the slider is ~d ~%" length)
  )

; set up the scross item--label, text length, values, action, etc
(setf ok (send button-item-proto :new "Push Here" :action #'runit))
(setf quit (send button-item-proto :new "Quit Xlisp-Stat" :action 'exit))

(setf test-label (send text-item-proto :new "a scrollable item"))
(setf test-value (send text-item-proto :new " " :text-length 2 ))
(setf test-scroll (send sequence-scroll-item-proto :new
                   (iseq length 50) :text-item test-value :action
                   #'(lambda (x) (setf length x))
                   ))
(send test-scroll :value 0)

; create the window and install the widgets
(send dialog-proto :title "A Simple Scrollable Window")
(send dialog-proto :location 200 0)
(send dialog-proto :new (list
                   (list test-label test-value test-scroll)
                   (list ok quit)
                   )
      )
```

# Bibliography

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs.* MIT, 1996.

[Gra96] Paul Graham. *ANSI Common Lisp.* Prentice Hall, 1996.

[GS93] Larry Gonick and Woollcott Smith. *The cartoon guide to statistics.* Harper Perennial, 1993.

[Tie90] Luke Tierney. *LISP-STAT: an object-oriented environment for statistical computing and dynamic graphics.* Wiley, 1990.

# Index