

The Language of S (or R)

Jane L. Harvill
First Edition, Fall, 2003

Contents

1	Introduction	3
2	Everything is an Object	4
3	Functions in S	5
3.1	Using Intrinsic Functions	6
3.2	Getting Started	7
3.3	Generating Data	7
4	Functions of Random Variables	15
4.1	The Cumulative Distribution Function	16
4.2	The Probability Density Function	18
4.3	The Quantile Function	19
4.4	Generating Random Variates	20
4.4.1	Estimating a Density with a Histogram	21
4.4.2	Estimating a Density with a Kernel Estimator	23
5	The weibmle Function	25
5.1	Constructing a Function in S and R	25
5.1.1	The Estimators	26
5.2	Logicals and Loops	29
5.2.1	Loop Statements	30
5.2.2	Logical or “Test” Statements	30
6	References	31
A	Example of Function weibmle	32

This document contains a brief description of programming in the language known most commonly as *S*. A free version of the *S* language, called *R*, is available for download from the website <http://www.r-project.org>.

1 Introduction

The statistical analysis package known as *S-Plus* is available to all students in the Mathematics & Statistics department's computer classrooms (currently located in Allen 414 and Allen 14). When you start the *S-Plus* program, you will see a fairly typical interface for *Windows* based programs. *S-Plus* has a spread-sheet type data entry system, a graphical user interface (of buttons, menus, and dialog boxes) for manipulating data, computing statistics, and producing graphs. Using this part of the *S-Plus* package is fairly straight-forward. In fact, it is the “*Plus*” in the name *S-Plus*. The “*S*” in the title, refers to the original software that was released in the early 1980s. The preface of book *The New S Language*, by Becker, Chambers, and Wilks, gives an idea of what the *S* part of *S-Plus* is all about.

S is a language and an interactive programming environment for data analysis and graphics. The *S language* is a very high-level language for specifying computations. The language is a part of an *interactive environment*. *S* encourages you to compute, look at data, and program interactively, with quick feedback to enable you to learn and understand.

The primary goal of the *S* environment is to enable and encourage good data analysis. The facilities in *S* are directed toward this goal:

- *S* is about *data*: it provides general and easy-to-use facilities for organizing, storing, and retrieving all sorts of data.
- *S* is about *analysis*: that is, computations you need to understand and use data. *S* provides numerical methods and other computational techniques.
- *S* is about *programming*: you can write functions in the *S* language itself. These functions can build on the power and simplicity of the *S* language. Because *S* is highly interactive, new functions can be designed and tried out much faster than with most languages. *S* also provides simple interfaces to other kinds of computing, such as commands from the UNIX system or to C and FORTRAN routines.
- Especially, *S* is about *graphics*: interactive, informative, flexible ways of looking at data. The graphics capabilities of *S* are designed to encourage you to create new tools and try out new ideas.

This document will not describe how to use graphical user interface that makes up the statistical analysis package *S-Plus*. Instead, it will focus on programming in the original language. The reasons for this are two-fold. First the graphical user interface makes using

the *analysis* part of *S-Plus* through the buttons, menus, and dialog boxes fairly straightforward, and something most graduate students should be able to pick up on their own, with little help. Secondly, the functions you build yourself add to the power of the package as a whole, enabling you to perform analyses that are beyond the “standard” package.

2 Everything is an Object

The language of *S* belongs to a class of languages referred to as “object-oriented” programming languages. Basically speaking, that’s because everything that you have access to through the *S* language is called an “object.” Consider for example, a plot created using the *S* language. A snap-shot of such a plot is seen in Figure 1. Everything you see in this plot

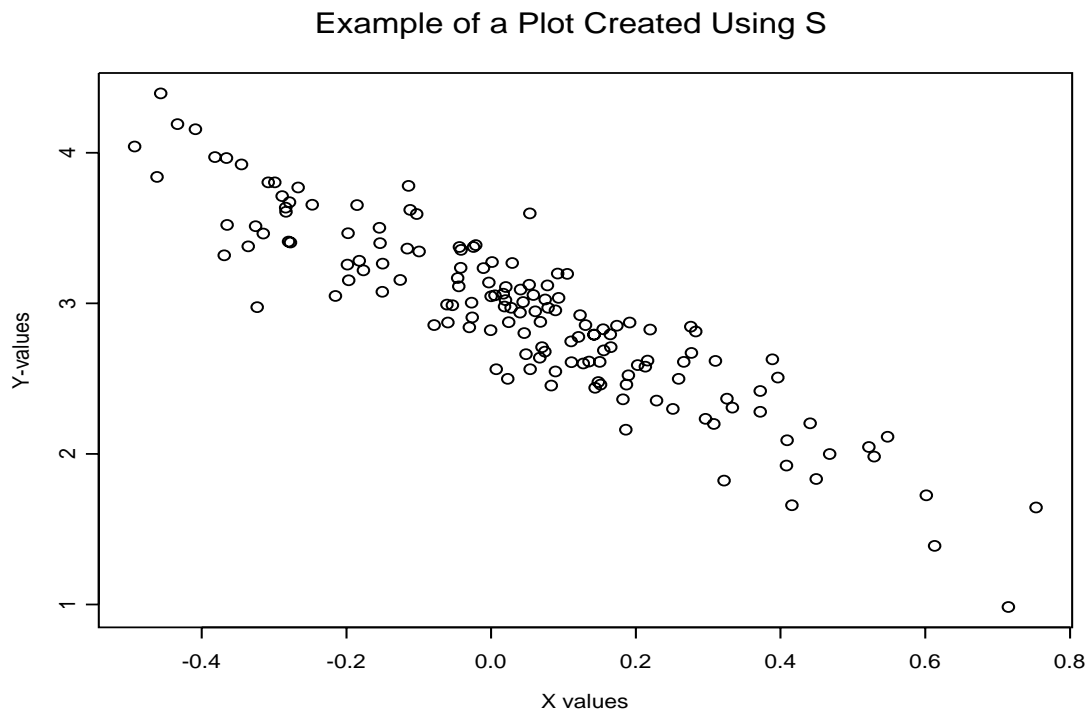


Figure 1: Plot Created Using *S-Plus*

is referred to as an “object”; specifically,

1. Each of the four edges of the border around the plot;
2. The major tick marks on the plotting axes (the bottom and left axes);
3. The numbers labeling the tick marks,

4. The minor tick marks (which are not seen in this plot), and the labels associated with the minor tick marks;
5. The label on the lower horizontal axis (which reads “X-values”);
6. The label on the left vertical axis (which read “Y-values”);
7. Each of the letters in the labels,
8. The plotting symbols (in this case small open circles);
9. The area inside the axes (called the plotting area);
10. The area outside the axes (called the margin);
11. The border of the plotting area;
12. The border of the margin;
13. The title of the plot;
14. The letters of the title of the plot;
15. ...

Now this seems like a long list of objects for a single plot, but – believe it or not – many of the objects of a plot are left out of the list! Through the language of S , you have access to each of these objects and can modify them so that the results is as customized as you need it to be. Everything in S is an object that can be manipulated through the S language to suit the purposes of the problem at hand. This makes S a very powerful tool in the hands of an experienced user.

What else can be considered an object in S ? A vector is one answer to that question. But just as a plot is a object made up of objects, so is a vector an object made up of objects. Each value in the vector is an object, and so is the length (dimension) of the vector. In the remainder of this document, we’re going to see how to create and access objects

3 Functions in S

All objects in S can be accessed in some manner through what is called a *function*. The best way to get an idea of how a function works, we’ll look at a specific example, given in detail in Appendix A. As we go, we’ll also learn about other functions related to, but not used in that example.

In mathematics, it is often that we have a function $w = f(x, y, z)$, say, that we wanted to evaluate at specific values of x, y , and z . The “name” of this function is f , and the

Table 1: Partial Listing of Intrinsic Data and Arithmetic Functions

<code>c(...)</code>	<code>seq(from,to,by)</code>	<code>rep(x,times,each)</code>
<code>matrix(data,nrow,ncol)</code>	<code>array(data,dim)</code>	<code>dim(x)</code>
<code>sin(x)</code>	<code>cos(x)</code>	<code>tan(x)</code>
<code>asin(x)</code>	<code>acos(x)</code>	<code>atan(x)</code>
<code>sinh(x)</code>	<code>cosh(x)</code>	<code>tanh(x)</code>
<code>complex(real,imaginary)</code>	<code>Re(z)</code>	<code>Im(z)</code>
<code>Conj(z)</code>	<code>Mod(z)</code>	<code>Arg(z)</code>
<code>exp(x)</code>	<code>sqrt(x)</code>	<code>log(x)</code>
<code>logb(x,base)</code>	<code>log10(x)</code>	

“arguments” of the function are x , y , and z . Now, suppose, for example, that $w = f(x, y, z) = 3x - 2y + z$, and we wanted to evaluate f at $x = 0$, $y = 2$, and $z = -1$. Then we would write

$$w = f(0, 2, -1) = 3 \cdot 0 - 2 \cdot 2 + (-1) = -5.$$

Functions in S work very much the same way: they have names and arguments. The name of the function defines precisely what operation(s) we want to perform, and the values of the arguments of the function specify the conditions under which they operation(s) are to be performed.

3.1 Using Intrinsic Functions

S has a large collection of intrinsic functions; too numerous to even begin to enumerate here. Hopefully, as we learn a little about some of the functions, you will see a fairly that function names are fairly intuitive. Additionally, S -Plus and R have an extensive on-line help system and are very useful in finding function names in instructing in the use of each function.¹ A partial listing of some intrinsic data and arithmetic functions in both S and R is given in Table 1. For most of the functions, the list of arguments is only partial. For detailed help on how to use any of these functions, from the command line prompt, call the `help` function. The argument of the `help` function should be the name of the function (from Table 1) for which you want help. For example, to get help on the `sin` function, at the command prompt, type

```
help(sin)
```

¹If you are running R , an excellent introduction to R can be found by calling the function `help.start()`. This will open an internet browser window with links to documentation on the R system.

In S -Plus, the on-line help is invoked by selecting **Help** from the menus. The most useful guide for beginners is found by selecting **Online Manuals**, and then **User’s Guide**. S -Plus on-line manuals require Adobe Acrobat reader.

Table 2: Partial Listing of Intrinsic Statistics Functions

<code>dnorm(x)</code>	<code>pnorm(q)</code>	<code>qnorm(p)</code>	<code>rnorm(n)</code>
<code>dt(x,df)</code>	<code>pt(q,df)</code>	<code>qt(p,df)</code>	<code>rt(n,df)</code>
<code>dchisq(x,df)</code>	<code>pchisq(q,df)</code>	<code>qchisq(p,df)</code>	<code>rchisq(n,df)</code>
<code>df(x,df1,df2)</code>	<code>pf(q,df1,df2)</code>	<code>qf(p,df1,df2)</code>	<code>rf(n,df1,df2)</code>
<code>dpois(x,lambda)</code>	<code>ppois(q,lambda)</code>	<code>qpois(p,lambda)</code>	<code>rpois(n,lambda)</code>
<code>mean(x)</code>	<code>median(x)</code>	<code>var(x,y)</code>	<code>stdev(x)</code>
<code>colMeans(x)</code>	<code>colSums(x)</code>	<code>rowMeans(x)</code>	<code>rowSums(x)</code>
<code>mad(x)</code>	<code>cor(x,y)</code>	<code>lsfit(x,y)</code>	<code>hat(x)</code>
<code>plot(x,y)</code>	<code>lines(x,y)</code>	<code>points(x,y)</code>	<code>hist(x)</code>

Table 2 contains a partial listing of functions (with a partial listing of arguments of those functions) for some common statistical procedures. As before, to get help on any of them, use the `help` function.

3.2 Getting Started

Now that you see a very small part of all that's at your disposal, let's get started actually learning how to use all of this! If you haven't already done so, start *S-Plus* or *R*. If you're using *S-Plus*, you'll need to be sure a **Command Window** is open. (The **Command Window** will be a window that has as its title **Commands**, and which will be inside the *S-Plus* program window.) If a **Command Window** is not open, you can open one by clicking on the button almost in the center of the screen that has two greater than (>) signs in a column the left side, and to the right of bottom > is an `x|`. The > are intended to represent the *S-Plus* command prompt. The analogous window in *R* is the **R Console Window**. The **R Console Window** will automatically open when you start *R*. In both *R* and *S-Plus*, the command prompt is represented by a greater than (>) sign.

3.3 Generating Data

To run any kind of procedure, we first need some data. So let's investigate first how to get data into a variable name. There are several data structures in *S* (or *R*). Before we start, there's one important thing you need to know to avoid. The letter `c` is the name of a function in both *R* and *S* that is used to create a vector. So you should **avoid at all costs naming any variable with the single letter c**.

1. **Scalars:** A *scalar* is simply a single number. Let's assign the scalar value of 3 to the variable name `a`. At the command prompt, type

```
a <- 3
```

and strike the return key. To see the value contained in the variable name `a`, simply type `a` at the command prompt. The value

```
[1] 3
```

is printed to the window. The `[1]` means there is one dimension for the variable `a`. The `3` following the `[1]` indicates the value stored in the variable `a` is `3`.

Before we continue, you probably noticed that an equal sign (`=`) was not used, but instead, a backwards arrow (`<-`) using the less than and minus sign was used. This is a convention in both *S* and *R*. It is best read “assign the value `3` to the variable `a`.” The `=` is used for other purposes that we will discover later.

2. **Vectors:** A vector is a one-dimensional array of numbers. There are a variety of ways to form a vector in *R* or *S*. A vector is used to store data from a single sample of size n , a constructed sequence of numbers, or repeated values of the same number. In the items below, we’ll investigate several ways to form a vector.

(a) `c`: The `c` function (already mentioned above on page 7) is the simplest way to create a vector of numbers. The arguments of the function are simply the values to be contained in the vector. For example, type

```
c(1,2,3,4,5)
```

and strike the return key. The output is the vector `(1, 2, 3, 4, 5)`. The `[1]` indicates there is one dimension to the value returned. To assign this vector to a variable name, use the `<-`. Try this:

```
x <- c(1,2,3,4,5)
```

and then type `x` at the command prompt.

(b) `rep`: The `rep` function will create a vector of repeated values of the same number. At the command prompt, type the following and strike the return key:

```
rep(0,10)
```

You should see the following output:

```
[1] 0 0 0 0 0 0 0 0 0 0
```

The `[1]` indicates there is one dimension. Count the number of zeros. You get 10. The usual call to the `rep` function uses this syntax (`rep(x,times)`), where `x` is replaced by the specific value to be repeated, and `times` is the number of times the value is to be repeated. Now try this

```
rep(c(1,2,3,4,5),10)
```

(or if you want `rep(x,10)`). Now try each of these examples: `rep(x,each=10)`; and then `rep(x,10,each=10)`. The `rep` function has other arguments. You should experiment with it to see how they work.

Before we continue, you may have noticed we're now using an `=`. Equal signs are reserved for setting arguments of functions equal to specific values. If you type `help(rep)` at the command prompt, the documentation will tell you that the usage of the function `rep` looks like the following

```
rep(x,times=<<see below>>,length.out=<<see below>>,
    each=<<see below>>)
```

In this description, notice that the argument `x` is not followed by an `=`. This means the user **must** specify a value for `x`. In the *R* and *S* help files these arguments are called **required arguments**. The values for the arguments `times`, `length.out`, and `each` are described in the documentation below. To call the `rep` function, and not use the argument `times`, the call must look like the following

```
rep(c(1,2,3,4,5),length.out=15)
```

In other words, if all arguments are not assigned a value, then once an argument is skipped (like the argument `times` was in this example), the remaining arguments must be specified using their name (like the argument `length.out=15` was in this example).

(c) : and `seq`: There are two ways to create a sequence of numbers.

1. : (colon): If the sequence is simply to be a set of numbers where each successive number is the previous number incremented by one; that is, $a_n = a_{n-1} \pm 1$, a colon (`:`) separating the minimum and maximum sequence values can be used. For example, to create the sequence from 1 to 5, you can type `1:5`. Try it and see. Try these other examples to see the result.

a. `5:1`

b. `-1:-10`

c. `b <- 1.3:6`. Strike the enter key and type `b`.

As before, to assign any sequence to a variable name, use the `<-` (like in the last example).

2. seq: The `seq` function has the syntax seen below. This is only a partial list of arguments. For other uses of `seq`, type `help(seq)`.

```
seq(from,to,by=<<see below>>,length=<<see below>>)
```

A value from `from` and for `to` must be specified. Then a value for either `by` or for `length` may be specified. If no value for `by` or `length` is specified, then the value of `by` is assumed to be 1, and the value of `length` is inferred. Try the following examples to see how the `seq` function works.

a. `seq(1,5)`

b. `seq(-1,-5)`

- c. `seq(1,5,by=0.5)`
- d. `seq(1,5,length=9)`
- e. `d <- seq(1.3,6)`, strike the enter key, and type `d`.

3. **Matrices:** A matrix is a two dimensional array. The `matrix` function is used to put values into a matrix. Elements are entered by-column, unless you specify to enter them by-row. The syntax for the matrix function is

```
matrix(data, nrow=<<see below>>, ncol=<<see below>>,
       byrow=F, dimnames=NULL)
```

The argument `data` is a vector containing the data values for the matrix. It is required; that is, there must be some value specified by the user. The argument `nrow` must be an integer containing the number of rows. Similarly, the argument `ncol` must be an integer containing the number of columns. You can specify either `nrow` or `ncol`, or both. If you specify neither, then `nrow` is chosen to match the length of the data and `ncol` will be one. The argument `byrow` is a logical flag. If it is set to `T` (or `TRUE`), then the data values in `data` are assumed to be the first row, then the second row, etc. If `byrow=FALSE`, the values are assumed to be the first column, then the second column, etc. Finally, the argument `dimnames` is a list of length 2 giving a `dimnames` attribute for the matrix. Each component must either have length 0 or be a vector of character strings with length equal to the corresponding element of the `dim` attribute of the result. Now that all that has been explained, let's look at some examples. Try each of the following to see the result.

- (a) `matrix(0,5,2)` returns a 5×2 matrix of zeros.
- (b) `matrix(1:10,5,2)` returns a 5×2 matrix of the integers 1 through 10. Note that the first column contains 1 through 5 and the second column contains integers 6 through 10.
- (c) `matrix(1:10,5,2,byrow=T)` also returns a 5×2 matrix containing the integers 1 through 10, but now the matrix was filled by row instead of by column, so that row one contains 1 and 2, row two contains 3 and 4, ..., and row five contains 9 and 10.
- (d) `matrix(1:10,5,2,byrow=T,list(NULL,c("Odds","Evens")))`. This last example returns the same thing as in the previous example, but notice that the columns are labeled `Odds` and `Evens`. The last argument `dimnames` is given the value `list(NULL,c("Odds","Evens"))`. The value `NULL` is the way to tell `R` or `S` that you do not want the 5 rows to have a label. The vector `c("Odds","Evens")` gives the name `Odds` to the first column, and `Evens` to the second column. Note that, if we had wanted to name the rows, `NULL` would need to be replaced by a vector of length 5, where each of the elements of that vector specifies the name for the corresponding row (contained in double-quotes).

When working with matrices, it is necessary to be able to add, subtract, multiple, take the transpose, and find the inverse, of the matrix. Matrix addition and subtraction are accomplished with the standard plus (+) and minus (-) signs. The other operations are accomplished in the R and S languages in the following manner.

- (a) Standard matrix multiplication: Let A denote an $n \times m$ matrix, and B denote an $n \times k$ matrix. Then the product AB is the $m \times k$ matrix $C = AB$ with elements $c_{i,j}$ defined by

$$c_{i,j} = \sum_{l=1}^m a_{i,l}b_{l,j}, \quad \text{for } i = 1, \dots, n \text{ and } j = 1, \dots, k. \quad (1)$$

In R and S , to multiply two matrices the operator `%*` is used. The next two examples are sequences of commands. You should take time to go through each and examine the result.

1. **Example 1.** In this example, you will multiply two matrices correctly.

```
A <- matrix(1:10,5,2)
B <- matrix(15:20,2,3)
A                                     # See the 5x2 matrix A
B                                     # See the 2x3 matrix B
A%*%B                                 # Get the product of AB
```

The result of `A%*%B` is the 5×3 matrix that (if you take time to check) was computed using the formula in (1). The other thing you probably noticed is that anything following a `#` did not affect the result. That's because the pound-sign (`#`) in R and S is the comment character. Anything following the (`#`) is taken to be a comment and not part of any computation.

2. **Example 2.** In this example, you will multiply the same two matrices, but since the number of columns in B is not the same as the number of rows in A , you will get an error message. Read it carefully to see what to expect when you (accidentally) make this mistake – note the R and S is telling you exactly what the problem is!

```
B%*%A
```

- (b) Element-by-element multiplication: Instead of standard matrix multiplication, it is often necessary to multiply the elements of one $n \times m$ matrix by the corresponding elements of a second $n \times m$ matrix (note the dimensions of the two matrices are the same). Specifically, if D and E denote two $n \times m$, matrices, then the element-by-element product F of D and E is the $n \times m$ matrix whose element $f_{i,j}$ are defined by

$$f_{i,j} = d_{i,j}e_{i,j} \quad \text{for } i = 1, \dots, n \text{ and } j = 1, \dots, m.$$

The element-by-element product of two matrices is found using the `*` operator. Try the following two examples:

1. **Example 1.** In this example, you will get the element-by-element product of two matrices in the correct way.

```
D <- matrix(1:10,5,2)
D
# See the value of the 5x2 matrix D
E <- matrix(11:20,5,2)
E
# See the value of the 5x2 matrix E
D*E
# Element-by-element produce of D and E
```

2. **Example 2.** In this example, you will attempt to get the element-by-element product of two matrices whose dimensions are not the same. Note the error message. *R* is telling you exactly what the problem is.

```
B
# Recall that B is a 2x3 matrix
B*E
# Element-by-element product of B and E.
```

- (c) Transpose: It is easy in *R* and *S* to get the transpose of a matrix. Simply use the `t` function. Try the following examples.

1. `t(B)`
2. `t(D*E)`
3. `t(A%*%B)`

- (d) Solving systems and inverting matrices: The `solve` function is used in two ways: (1) to find the solution to a set of n equations and n unknowns, or (2) to find the inverse of a square matrix.

1. *Solving a system of equations:* Consider the following system of 3 equations and 3 unknowns x, y , and z .

$$\begin{aligned}x + 2y - z &= 3 \\x - y - z &= -3 \\2x + y + z &= 0\end{aligned}\tag{2}$$

What is the single point (x, y, z) that all three planes have in common? Another way of phrasing this question is, “What is the value of (x, y, z) that satisfies all three questions?” Or, in other words, what is the solution to the system of equations in (2)? We can use the `solve` function to find the answer to this question. Issue the following set of commands:

```
H <- matrix(c(1,1,2,2,-1,1,-1,-1,1),3,3)
H
# Elements of H correspond to coefficients in system
g <- c(3,-3,0)
g
# Elements of g correspond to right-side of system
solve(H,g)
```

Now, try substituting in the values returned by `solve` into each of the three equations; that is, let $x = -1, y = 2$ and $z = 0$ - you’ll see that when you do this, the left side of all three equations in (2) will equal their right side. If you

were to graph the three planes represented in the system of equations, their point of intersection would be the three-tuple with coordinates $(-1, 2, 0)$.

2. *Inverse of a matrix:* Let S denote an $n \times n$ (square) matrix. Then the inverse of S , if the inverse exists, is the $n \times n$ matrix denoted S^{-1} satisfying

$$SS^{-1} = S^{-1}S = I_n,$$

where I_n is an $n \times n$ matrix with diagonal elements all equaling 1 and off-diagonal element all equaling 0. The matrix I_n is called the *identity matrix* since multiplying any $n \times n$ matrix by I_n will return the original matrix; that is

$$SI_n = I_nS = S.$$

Issue the following sequence of commands, which uses the `solve` function to find the inverse of a matrix

$$H = \begin{bmatrix} 1 & 2 & -1 \\ 1 & -1 & -1 \\ 2 & 1 & 1 \end{bmatrix}$$

As you issue the commands, be sure you read the numbers correctly. They'll be given in exponential notation. The last two matrices that are printed should be the 3×3 identity matrix (subject to some round-off error).

```
H <- matrix(c(1,1,2,2,-1,1,-1,-1,1),3,3)
H
Hi <- solve(H)
Hi
H%%Hi
Hi%%H
```

4. **Arrays:** An array is a set of numbers arranged in a hypercube with more than two dimensions. The `array` function looks very much like the `matrix` function:

```
array(data, dim, dimnames = NULL)
```

Just as in the `matrix` function, the value of `data` is replaced by a vector containing the data that is to fill the array. The argument `dim` is a vector giving the extent for each dimension. The argument `dimnames` works very much like the argument of the same name in the `matrix` function, except now the list much contain a vector of names for each dimension (or the word `NULL` if no name is desired for a dimension). Try these examples to see how the `array` function works.

- (a) `array(0,c(2,3,2,2))`
(b) `array(1:24,c(2,3,2,2))`

(c) `array(1:30,c(5,2,3))`

Before we move on to lists, one more note is necessary concerning vectors, matrices, and arrays. There are occasions when it is desired to access only a set of elements of the vector, matrix, or array. Square brackets are used for doing this. For example, to access the element of the matrix `A` in the second row and first column, you would use `A[2,1]`. If you wanted to access all of the second row, you could type `A[2,]`. On the other hand, to access all of the first column, you would type `A[,1]`. Try the following to see what happens:

```
A
A[3,2]
A[,2]
A[3,]
A[c(1,3,5),2]
A[1:5,1]
g
g[1]
g[3]
g[1:2]
solve(H)[1:2,]
```

5. **Lists:** The final data structure we will encounter in *R* and *S* is a list. We already encountered this term on page 10, when we saw how to assign names to dimensions of matrices and arrays. Simply speaking, a *list* in —it *R* and *S* is (literally) a list of data objects. Each element in the list can be assigned its own name. To access an element of a list, a `$` is necessary. The best way to see what a list is is to go through some examples.

- (a) **Example 1.** In this first example, we'll create a list called `list1` containing three elements. The first element will have the name `one` and will contain the integer sequence 1, 2, 3, 4, 5. The second element will have the name `two` and will contain the scalar value 2. The third element of the list `list1` will have the name `three` and will contain a 5×2 matrix of integers 1 through 10. To accomplish this, issue the following commands:

```
list1 <- list(one=1:5,two=2,three=matrix(1:10,5,2))
list1
```

The output from you typed `list1`, should have looked like this:

```
$one:
[1] 1 2 3 4 5
```

```
$two:
```

```
[1] 2
```

```
$three:  
      [,1] [,2]  
[1,]    1    6  
[2,]    2    7  
[3,]    3    8  
[4,]    4    9  
[5,]    5   10
```

Notice that the names `one`, `two` and `three` of the elements in the list were preceded by a `$`. Also note: `$one` is the vector sequence 1, 2, 3, 4, 5; `$two` is the scalar 2; and `$three` is the 5×2 matrix containing the integers 1 through 10.

To access a specific element of a list, we use the `$`. Issue the following command.

```
list1$three
```

You should get back only the 5×2 matrix of integers 1 through 10. The `list` data construct is used in functions written in the *R* and *S* languages to return multiple outputs. We will see an example of this when we discuss the details of the function `weibmle` found in Appendix A.

4 Functions of Random Variables

A multitude of reasons exists to generate a set of random numbers, computing the cumulative probability distribution function $F(x) = P(X \leq x)$, computing the probability density function $f(x) = F'(x)$ of a continuous random variable, or probability mass function $f(x) = P(X = x)$, of a discrete random variable or the quantile function $Q(u) = F^{-1}(u) \inf\{x : F(x) = u\}$. *R* and *S* contain a set of functions, all following a naming convention, for doing these computations for list of distributions in Table 3. * The syntax of the function calls in Table 3 should be interpreted in this way

- If the value of the probability density function (or probability mass function) is required then the `X` should be replaced by a `d` and the first argument (`Y`) is the variable name of the scalar, vector, matrix, or array for which the value of the pdf or pmf is required.
- If the value of the cumulative probability function is required then the `X` should be replaced by a `p` and the first argument (`Y`) is the variable name of the scalar, vector, matrix, or array for which the value of the cdf is required.
- If the value of the quantile function is required then the `X` should be replaced by a `q` and the first argument (`Y`) is the variable name of the scalar, vector, matrix, or array for which the value of the quantile function is required.

Table 3: Distributions in S and R

Distribution	Function call*
$\text{beta}(\alpha, \beta)$	<code>Xbeta(Y, shape1, shape2)</code>
$\text{binomial}(n, p)$	<code>Xbinom(Y, size, prob)</code>
$\text{Cauchy}(\alpha, \beta)$	<code>Xcauchy(Y, location=0, scale=1)</code>
χ^2_ν	<code>Xchisq(Y, df)</code>
$\text{exponential}(\lambda)$	<code>Xexp(Y, rate=1, scale)</code>
F_{ν_1, ν_2}	<code>Xf(Y, df1, df2)</code>
$\text{gamma}(\alpha, \beta)$	<code>Xgamma(Y, shape, rate=1, scale)</code>
$\text{geometric}(p)$	<code>Xgeom(Y, prob)</code>
$\text{hypergeometric}(N, M, K)$	<code>Xhyper(Y, m, n, k)</code>
$\text{lognormal}(\mu, \sigma^2)$	<code>Xlnorm(Y, meanlog=0, sdlog=1)</code>
$\text{logistic}(a, b)$	<code>Xlogis(Y, location=0, scale=1)</code>
$\text{negative binomial}(r, p)$	<code>Xnbinom(Y, size, prob)</code>
$\text{normal}(\mu, \sigma^2)$	<code>Xnorm(Y, mean=0, sd=1)</code>
$\text{Poisson}(\lambda)$	<code>Xpois(Y, lambda)</code>
t_ν	<code>Xt(Y, df)</code>
$\text{uniform}(a, b)$	<code>Xunif(Y, min=0, max=1)</code>
$\text{Weibull}(\alpha, \gamma)$	<code>Xweibull(Y, shape, scale=1)</code>

- If a set of random variates from the specified function is desired, then the X should be replaced by an \mathbf{r} and the first argument (Y) is an integer containing the length of the desired sample.

4.1 The Cumulative Distribution Function

Let X denote a random variable, and let the function $F_X(x)$ be the function defined as

$$F_X(x) = P(X \leq x), \quad \text{for all } x \in \mathcal{X},$$

where \mathcal{X} is the sample space of X . The function $F_X(x)$ is the cumulative distribution function (cdf) of X . The cdf is used in many calculations in statistical analysis of data. Let's look at some examples of how to get the value of the cdf.

Table 3 lists the distributions for which R and S will compute the cdf. For our purposes, let's assume that X has a normal distribution with mean 3 and standard deviation 2. Suppose we wanted to know the probability that $X \leq 5.5$; then, at the prompt, type the command `pnorm(5.5, 3, 2)`. The value returned $0.8943502 = P(X \leq 5.5)$.

What if we wanted an idea of the shape of the cdf? How could we do that? Issue the following sequence of commands to see how!

```
x <- seq((3-3.5*2), (3+3.5*2), length=101)
Fx <- pnorm(x, 3, 2)
```

```
plot(x,Fx,ylim=c(0,1),xlab="x",ylab="F(x)",type="l")
title("CDF of N(3,stdev=2)")
```

The resulting plot should look like what you see in Figure 2. To get the cdf of any other

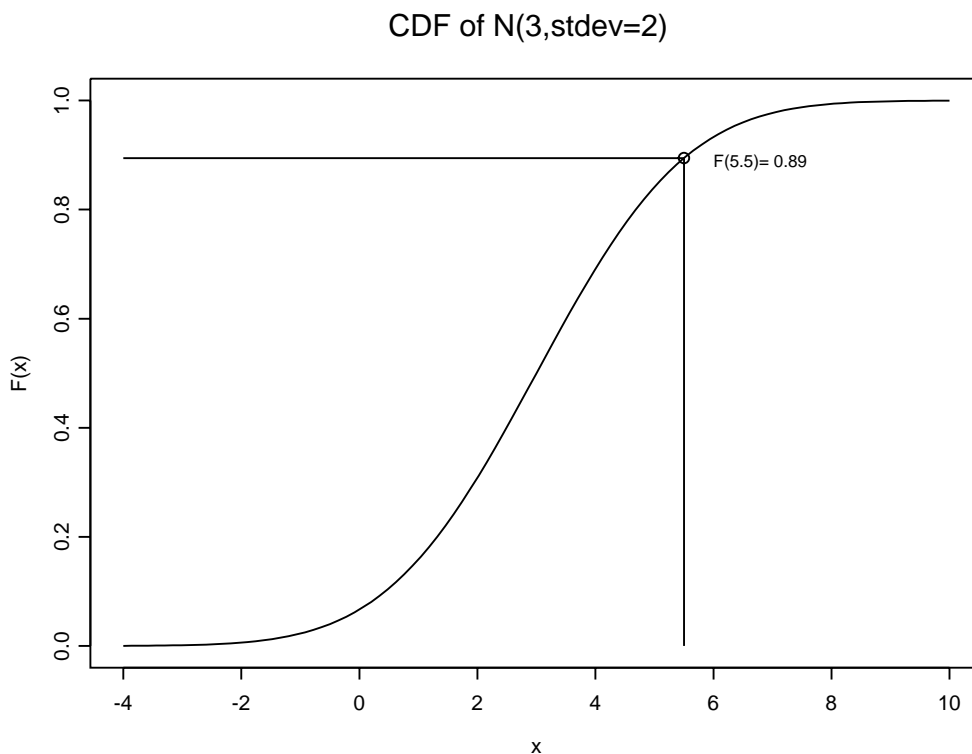


Figure 2: S plot of a $N(3, \sigma = 2)$ cumulative distribution function.

distribution in S or R , use the function calls in Table 3, replacing the X with a p and the argument Y with the value or values for which the cdf is desired.

This example gives us an opportunity to discuss the `plot` function. The first argument of the plot function is a vector of length n containing the set of x coordinates (abscissas) and the second argument is a vector (also of length n) containing the y coordinates (ordinates). If only one argument is given (e.g., `plot(y)`), the argument is assumed to be a set of ordinates. After these first two arguments are numerous arguments which allow the user to specify the limits of the x - and y -axes (`xlim` and `ylim`), to label the x (horizontal) and y - (vertical) axes (`xlab` and `ylab`), to determine the plotting symbol (the `type`) argument. There are many other arguments, but for the time being we will restrict our attention to these few.

1. `xlim` and `ylim`: These two arguments of the plot function allow you to specify the lower- and upper-limits of the x - and y -axes, respectively. The values assigned to each should be a vector of length two. The first element of the vector specifies the lower limit, and the second element of the vector specifies the upper limit. If the argument is

omitted from the plot function, the lower and upper limits of the axes are determined automatically by R (or S) using the data values.

2. **xlab** and **ylab**: These two arguments allow you to customize the labels on the x - and y - axes. The values assigned to **xlab** and **ylab** should be character strings.
3. **type**: The **type** argument specifies the plotting symbol to be used. The default value for **type** is the character **p** (for points). Other values are a lower-case letter **l** (for lines), or **b** for (both) points and lines.

The **title** function puts a title on the plot. For more help on customizing the title, use **help(title)**.

4.2 The Probability Density Function

The probability density function (pdf) of a continuous random variable X is the function $f_X(\cdot)$ satisfying the following:

$$P(X \leq x) = F_X(x) = \int_{-\infty}^x f_X(t) dt, \quad \text{for all } x \in \mathcal{X}.$$

Again, let $X \sim N(3, \sigma = 2)$. To compute the pdf at 5.5 of the $N(3,2)$ distribution, the function call is **dnorm(5.5,3,2)**; type this at the prompt. That is $f_X(5.5) = 0.09132454$. To get a picture of the $N(3,2)$ pdf, issue the following sequence of commands.

```
fx <- dnorm(x,3,2)
plot(x,fx,xlab="x",ylab="f(x)",type="l")
points(5.5,dnorm(5.5,3,2))
lines(c(min(x),5.5),rep(dnorm(5.5,3,2),2))
lines(rep(5.5,2),c(0,dnorm(5.5,3,2)))
text(6,dnorm(5.5,3,2),"f(5.5)=0.09",cex=0.8,adj=0)
title("PDF of N(3,stdev=2)")
```

This sequence of commands will result in the plot seen in Figure 3. This also gives the opportunity to introduce the three functions described below for superimposing information on a plot.

1. **lines**: The **lines** function superimposes lines onto the current plot. The syntax of the **lines** function is **lines(x, y, type="l")**. The **x** and **y** arguments are of the same length contain the coordinates of the points. The **type** argument is an optional argument. For the **lines** function, the default of **type** is **l** (for line). Other values of **type** are "p", "l", "b", "o", "n", "s" and "h" which produce, respectively, points, lines, both, both (overlaid), nothing, stairsteps and high-density lines.
2. **points**: The **points** function superimposes points onto the current plot. The syntax of the **points** function is the same as that of the **lines** function, except the default of the argument **type** is **p**.

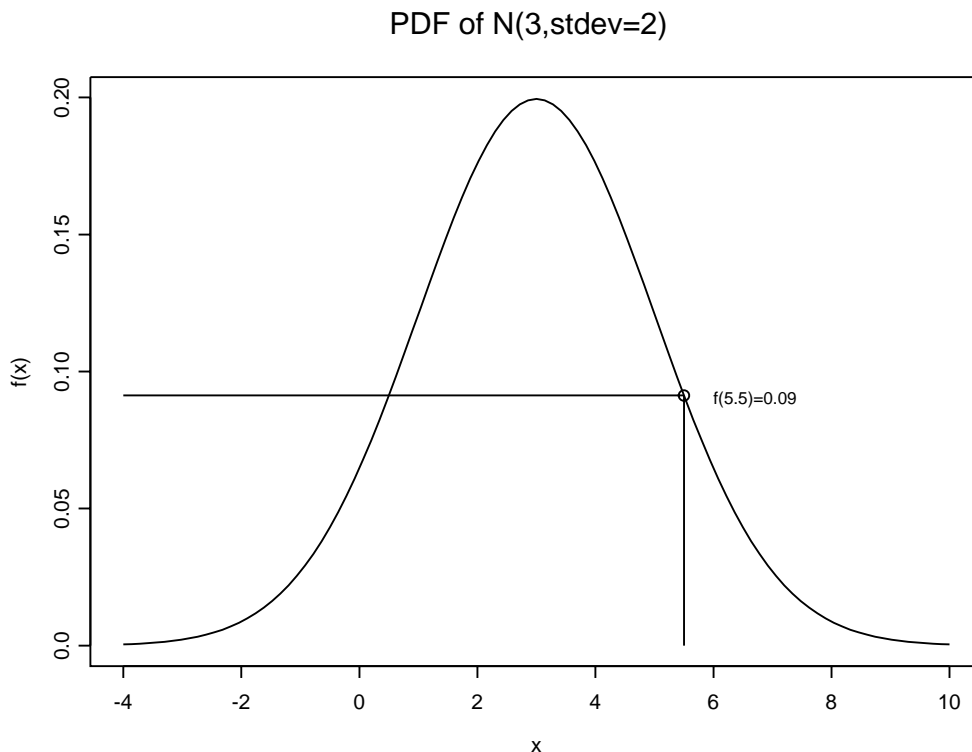


Figure 3: S plot of a $N(3, \sigma = 2)$ probability density function.

3. **text**: The `text(x,y,labels)` function superimposes the text contained in the object `labels` at the coordinates `x` and `y`. If `x` and `y` are vectors, they must be of the same length, n , say. The argument `labels` can also be a vector containing n labels. Other optional arguments of `text` control justification (`adj`), color (`col`), character expansion (`cex`), etc. For more on these parameters, see `help(par)`.

Other functions which superimpose objects onto the current plot include the functions `segments`, `arrows`, and `symbols`. The `par` function also controls many objects in the `plot` function. It is worth the time to familiarize yourself with the function `par` through `S` help.

4.3 The Quantile Function

The quantile function $Q_X(u)$, $0 < u < 1$ is the function satisfying the following,

$$Q_X(u) = F_X^{-1}(u) = \inf\{x : F_X(x) = u\}.$$

For continuous random variables, the `inf` can be removed from the definition. However, for discrete random variables, it must be included. For our example, where $X \sim N(3, \sigma = 2)$, we have been considering $x = 5.5$. For this value $F_X(5.5) = 0.8943502$. Therefore

$Q_X(0.8943502) = F_X^{-1}(0.8943502) = 5.5$. This is found by calling the function `qnorm(0.8943502,3,2)`. A picture of the quantile function for our random variable is given in Figure 4. This figure was created using the following sequence of function calls.

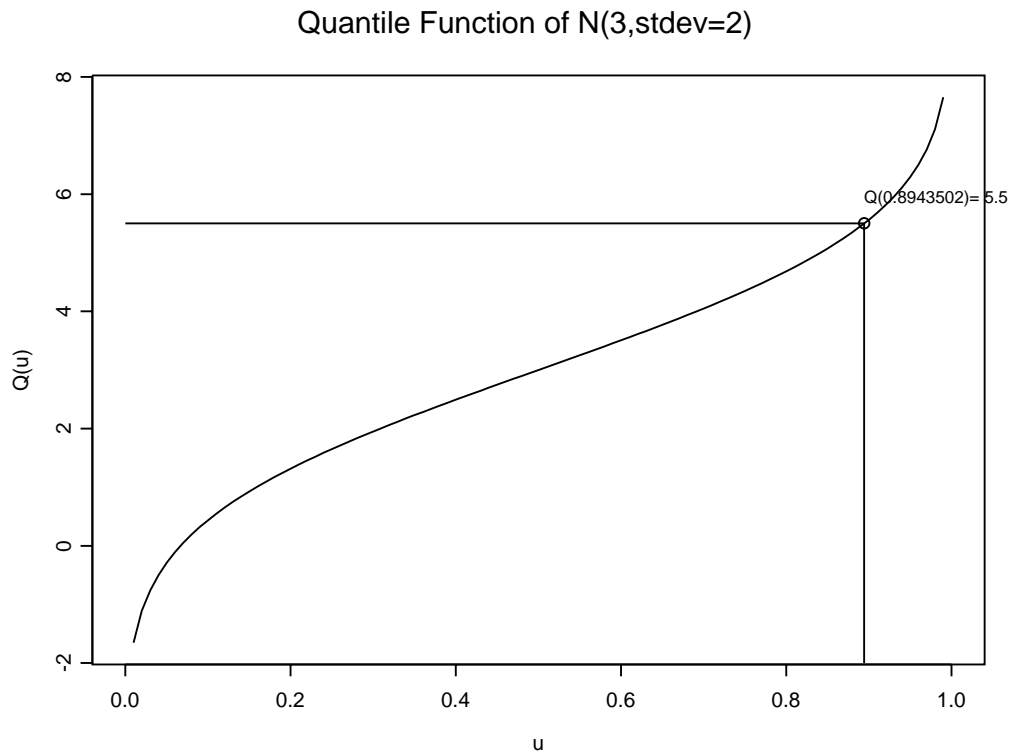


Figure 4: S plot of a $N(3, \sigma = 2)$ quantile function.

```
u <- seq(0,1,length=101)
Qu <- qnorm(u,3,2)
plot(u,Qu,xlab="u",ylab="Q(u)",xlim=c(0,1),type="l")
title("Quantile Function of N(3,stdev=2)")
text(0.8943502,6,"Q(0.8943502)= 5.5",cex=0.8,adj=0)
points(0.8943502,qnorm(0.8943502,3,2))
lines(c(0,0.8943502),rep(qnorm(0.8943502,3,2),2))
lines(rep(0.8943502,2),c(-2,qnorm(0.8943502,3,2)))
```

4.4 Generating Random Variates

Simulation provides a convenient numerical way for statisticians to examine the behavior of statistics when that may not be possible analytically. We will learn more about specific simulation methods in another handout. For right now, we'll just concentrate on how to

generate random numbers using S or R . We'll also take a little time to learn about two other functions in S and R – the *hist* function and the *density* function.

The functions used for generating a set of n random variates all begin with the letter **r**. To generate $n=1000$ $N(3, \sigma = 2)$ random variates, the function call is `rnorm(1000,3,2)`. Do this, and assign these normal variates to the variable name **x**; that is, type `x <- rnorm(1000,3,2)`.

4.4.1 Estimating a Density with a Histogram

To get a histogram of these random variates, at the command prompt, type `hist(x)`. The resulting histogram is a frequency histogram (note the values of the vertical axis). The full call of the function `hist` is

```
hist(x, nclass=<<see below>>, breaks=<<see below>>, plot=T,  
      probability=F, include.lowest=T, ...,  
      xlab=deparse(substitute(x)))
```

The arguments are described by the S help in the following manner.

Required arguments: **x** – numeric or factor vector of data for histogram. If **x** is a factor (or category) `hist` will call `hist.factor` with all but the `nclass` and `breaks` arguments. If **x** is integer-valued `hist.factor(x)` may give you better results than `hist(x)`. Missing values (NA) are allowed.

Optional arguments:

nclass: recommendation for the number of classes (i.e., bars) the histogram should have. This may be an integer, a function to apply to **x** which returns an integer, or a character string specifying which built-in method to use. Available methods for calculating the number of classes are Sturges (`sturges`), Freedman-Diaconis (`fd`), and Scott (`scott`).

For factor data `nclass` is `length(levels(x))` and cannot be overridden.

breaks: vector of the break points for the bars of the histogram. The count in the i -th bar is `sum(breaks[i] < x & x <= breaks[i+1])` except that if `include.lowest` is TRUE (the default), the first bar also includes points equal to `breaks[1]`. If omitted, evenly-spaced break points are determined from `nclass` and the extremes of the data. For factor data breakpoints are at half integers and cannot be overridden.

plot: logical flag: if TRUE, the histogram will be plotted; if FALSE, a list giving breakpoints and counts will be returned.

probability: logical flag: if TRUE, the histogram will be scaled as a probability density; the sum of the bar heights times bar widths will equal 1. If FALSE, the heights of the bars will be counts.

`include.lowest`: logical flag: if TRUE (the default), the lowest bar will include data points equal to the lowest break, otherwise it will act like the other bars (see the description of the `—tt breaks` argument).

`...`: additional arguments to `barplot`. The `hist` function uses the function `barplot` to do the actual plotting; consequently, arguments to the `barplot` function that control shading, etc., can also be given to `hist`. See the `barplot` documentation for arguments `angle`, `density`, `col`, and `inside`. Do not use the `space` or `histo` arguments.

`xlab`: label for the plot x-axis. By default, this will be `x`.

Graphical parameters may also be supplied as arguments to this function (see `par`). In addition, the high-level graphics arguments described under `plot.default` and the arguments to `title` may be supplied to this function.

Value:

if `plot` is TRUE, a vector containing the coordinate of the center of each box is returned, and a plot is created on the current graphics device.

if `plot` is FALSE, `hist` returns a list with components:

`counts` – count or density in each bar of the histogram.

`breaks` – break points between histogram classes.

Although it may seem counter-intuitive to want to call the histogram function without producing a plot, it's not. Issue the following set of commands:

```
hist(x,probability=T)
fx <- dnorm(sort(x),3,2)
lines(sort(x),fx)
```

It is possible that you got a number of errors in the command window warning you that some point coordinates were out of bounds. One way to avoid this is to find out what vertical ranges will be required from both the histogram and from the density, and then to create the plot using those limits. Try this sequence of function calls:

```
fx <- dnorm(sort(x),3,2)
hist.info <- hist(x,probability=T,plot=F)
hist.info
hist(x,probability=T,ylim=c(0,max(hist.info$counts,fx)))
lines(sort(x),fx)
title("Histogram of N(3,2) Variates with \n Density Superimposed")
```

The argument `ylim=c(0,max(hist.info$counts,fx))` sets the lower bound for the vertical axis to the value 0 and the upper bound for the vertical axis to the maximum value of the heights of the histogram bars and the points `fx` (all heights are contained in `hist.info$counts`). One other note: the `\n` creates a new line in the title. The resulting plot is seen in Figure 5.

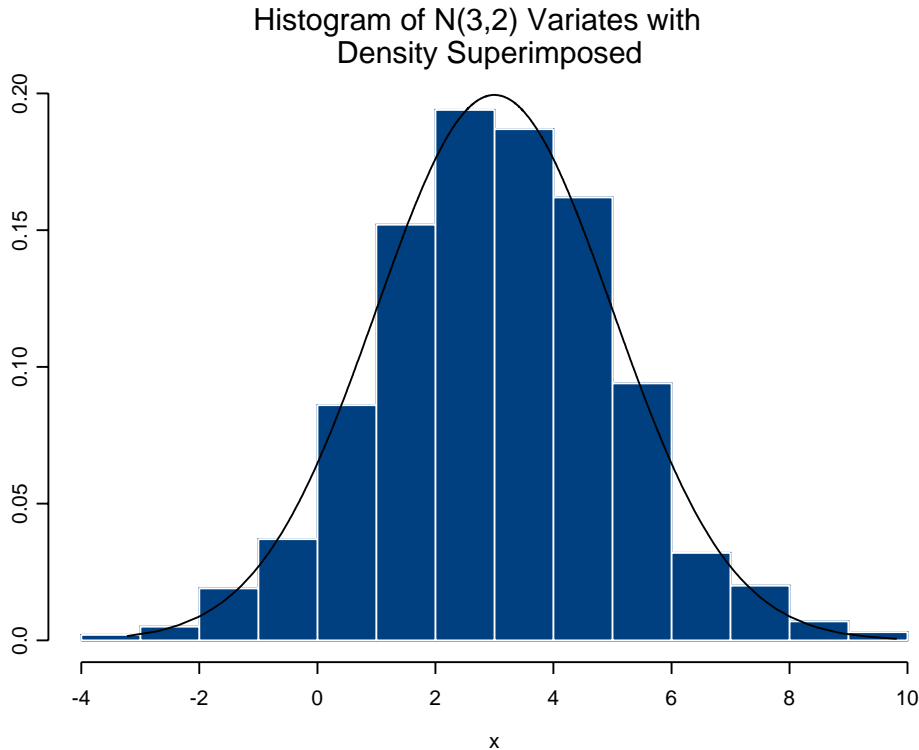


Figure 5: S histogram 1000 $N(3, \sigma = 2)$ variates with density.

4.4.2 Estimating a Density with a Kernel Estimator

While histograms are the simplest, and most popular method of estimating a density, statistically speaking, they're also one of the worst ways to estimate a density. Kernel density estimators (KDE) are superior methods based on weighted moving averages of the data. There are volumes of literature on KDEs. Here, we only begin to explain the principal behind them. Please recognize that the ideas presented here are very general. The interested reader should consult more advanced texts on this subject, for example, Hart (1997).

Let X_1, X_2, \dots, X_n represent the data for which a density estimate is desired; in other words X_1, X_2, \dots, X_n is an i.i.d. sample from a population having *unknown* density $f_X(x)$, and we wish to get a numerical estimate of the density f_X across a subset of the domain of the density. Let $u_1 < u_2 < \dots < u_m, m \leq n$, represent an evenly-spaced grid of points across the domain of f , and suppose we want to estimate f at each point $u_i, i = 1, \dots, m$. One way to accomplish this is by letting

$$\hat{f}_X(u) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{u - X_i}{h}\right),$$

where the function K is called a *kernel* and acts as a filter. Examples of kernel functions

include, but are certainly not limited to the following

$$K(t) = \begin{cases} 1 & \text{if } |t| < 1, \\ 0 & \text{otherwise} \end{cases}, \quad (\text{Rectangular}) \quad (3)$$

$$K(t) = \begin{cases} (1 - |t|) & \text{if } |t| < 1, \\ 0 & \text{otherwise} \end{cases}, \quad (\text{Triangular}) \quad (4)$$

$$K(t) = 0.54 + 0.46 \cos \pi t, \quad (\text{Cosine}) \quad (5)$$

$$K(t) = (\sqrt{2\pi})^{-1} \exp\{-t^2/2\}, \quad (\text{Gaussian}). \quad (6)$$

The kernel functions act as a weight function applied to each data value X_i , so that the closer X_i is to u , the more weight it gets in the weighted average used as the density estimate. Note that the histogram is a kernel density estimate that uses the rectangular window.

The choice of kernel function is not nearly as important as the selection of the value of h , called the *bandwidth*. If the bandwidth is too large, the estimate of the density is “over-smooth” and will be biased. On the other hand, if the value of h is too small, the estimate will be too “wiggly” and will have too much variation to be useful. Theory has shown that, in general, good choices of the bandwidth h will be proportional to $n^{1/5}$.²

One S and R function used to estimate a probability density function using KDEs is the `density` function. The call to the `density` function is

```
density(x, n=50, window="g", na.rm=F, width=<<see below>>,
        from=<<see below>>, to=<<see below>>, cut=<<see below>>)
```

The argument `x` is the data for which an estimate of the density is desired. The argument `n` is the number of points at which to estimate the density of `x`. The argument `n` is **not** the size of the sample `x`; it **is** the number of points at which the density is to be estimated. The value of the argument `window` can be one of `g` for “Gaussian”, `c` for “cosine”, `r` for “rectangular” or `t` for “triangular.” The argument for `width` can be a numerical value containing the bandwidth, or the function can choose that value for you. The arguments `from` and `to` determine the upper and lower values that the `n` estimates are obtained at. These arguments are also automatically computed if not specified. The argument `cut` is the fraction of the window width that the `x` values are to be extended by. It is highly suggested that, unless you learn more about these techniques, you rely on the defaults supplied by `S`. The function `density` returns a list with two components called `x` and `y`. The object `density$x` contains the vector of `n` points at which the density is estimated. The object `density$y` is a vector of density estimates at each `x` point.

To use this function on the data you have generated to estimate the normal density, you would issue the following sequence of function calls.

```
dens.est <- density(x,n=101)
plot(dens.est$x,dens.est$y,type="l",xlab="x",ylab="y")
title("Kernel Density Estimate \n (1000 observations)")
```

²The choice of bandwidth selection is much more complicated than this simple rule. Again, the interested reader should consult more authoritative texts on this subject (Hart, 1997).

The graph in Figure 6 is a picture to illustrate the closeness of the kernel density estimate to the true value of the $N(3, \sigma = 2)$ density.

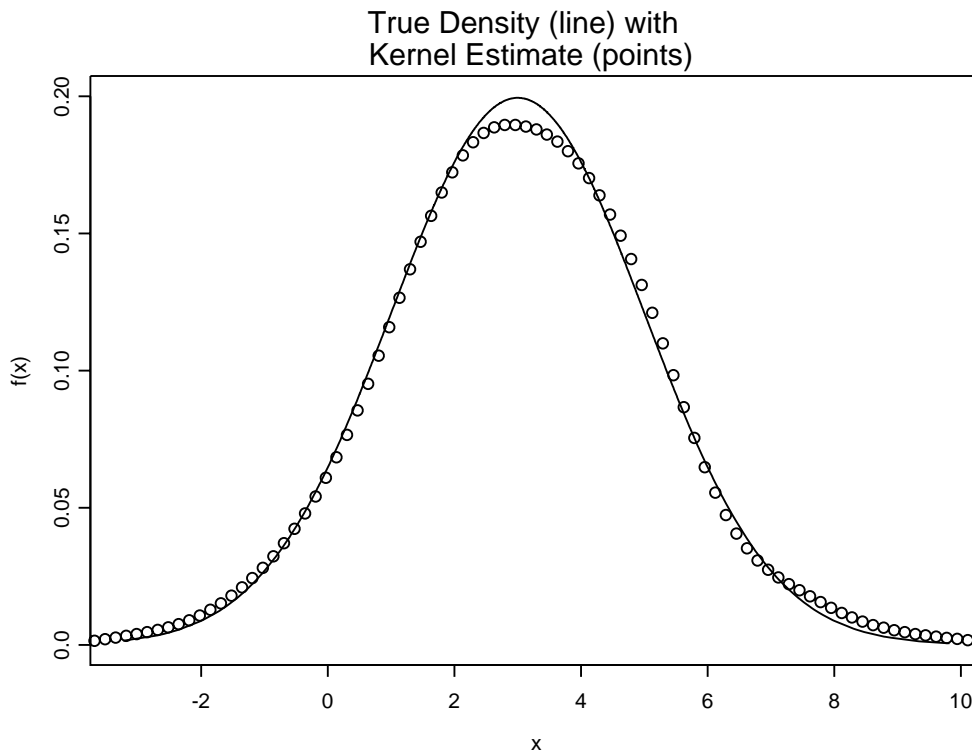


Figure 6: S Kernel Density Estimate (points) with True Density (line).

5 The weibmle Function

This last section is an attempt to pull all of the above together and to teach you how to use the S or R language to write your own functions for your own analysis. While S and R are powerful as they stand, their true power comes from the flexibility of being able to use the language to write your own function to perform your own analysis. For an explanation of how to find the maximum likelihood estimates of the parameters of a Weibull(α, β) distribution, the reader is referred to the handout *Topics in Computing*. The function found in Appendix A is written to perform the computations necessary for returning estimates of the MLE given a set of data.

5.1 Constructing a Function in S and R

All user-written functions begin in the same way: a function name that is assigned to the object type `function` with a set of *required* and *optional arguments*. To see how a function

is constructed, we will rely heavily on the example in Appendix A. The beginning line of this function is where the function is given a name (`weibmle`), and where the name `weibmle` is assigned to the object type `function` having arguments `x`, `maxit`, and `epsilon`. The arguments `maxit` and `epsilon` are both followed by `=VALUE`. This is how defaults are assigned to function values in *S* and *R*. If the user does not specify a value for `maxit`, then `maxit` will take the value 20. Likewise, if the user does not specify a value for `epsilon`, it will take the value 0.0001. Therefore `maxit` and `epsilon` are called *optional arguments* since it is only optional that the user specify a value for the argument. On the other hand, the argument `x` is not followed by an `=VALUE`. So the user is required to specify a value for `x`. The argument `x` is called a *required argument*. The first line of the function ends with an open curly brace (`{`), and the last line of the function is the close-curly brace (`}`) that matches it. **There must be a carriage return after the last close-curly brace.**

Following the function assignment is a large number of lines that begin with a `#`. These are comment lines included for two purposes (1) to instruct a user in the purpose and syntax of the function, and (2) for my (the author's) memory! Omitting these lines will not affect the computations completed by the function, but it will make it harder for someone using the function to know what is necessary.

The computations performed after the comments are described mathematically in the document *Topics in Computing*. For your convenience, a part of that discussion is included in Section 5.1.1. Within that discussion are additional comments on which lines in `weibmle` are used to accomplish the numerical work. The section of `weibmle` in this discussion begins with the line `n <- length(x)` and ends with `theta <- theta - del`. More explanation of specific lines in this function is given beginning in Section 5.2.

5.1.1 The Estimators

We consider the two dimensional Weibull distribution which has probability density, cumulative distribution, and quantile functions

$$\begin{aligned}
 f_X(x) &= \begin{cases} \alpha\beta x^{\beta-1}e^{-\alpha x^\beta}, & x > 0, \alpha > 0, \beta > 0 \\ 0 & \text{otherwise,} \end{cases} \\
 F_X(x) &= 1 - e^{-\alpha x^\beta}, \quad x > 0, \\
 Q_X(u) &= \left[\frac{-\log(1-u)}{\alpha} \right]^{1/\beta}, \quad 0 \leq u \leq 1.
 \end{aligned}$$

The MLE's $\hat{\alpha}$ and $\hat{\beta}$ are the values of α and β maximizing the log likelihood

$$l(\alpha, \beta; X) = n \log \alpha + n \log \beta + (\beta - 1) \sum_{i=1}^n \log X_i - \alpha \sum_{i=1}^n X_i^\beta.$$

The gradient and Hessian corresponding to minimizing $L(\boldsymbol{\theta}) = -l(\boldsymbol{\theta})$ are given by

$$g_1 = -\frac{\partial l}{\partial \alpha} = \sum_{i=1}^n X_i^\beta - \frac{n}{\alpha}, \tag{7}$$

$$g_2 = -\frac{\partial l}{\partial \beta} = \alpha \sum_{i=1}^n X_i^\beta \log X_i - \sum_{i=1}^n \log X_i - \frac{n}{\beta}, \quad (8)$$

$$H_{11} = \frac{\partial^2(-l)}{\partial \alpha^2} = \frac{n}{\alpha^2}, \quad (9)$$

$$H_{12} = H_{21} = \frac{\partial^2(-l)}{\partial \alpha \partial \beta} = \sum_{i=1}^n X_i^\beta \log X_i, \quad (10)$$

$$H_{22} = \frac{\partial^2(-l)}{\partial \beta^2} = \alpha \sum_{i=1}^n X_i^\beta (\log X_i)^2 + \frac{n}{\beta^2}. \quad (11)$$

In `weibmle`, the variable names that will contain the gradient and Hessian matrix are `g` and `H`. Because both of these objects have dimension (one is a vector and the other is a matrix), they must be initialized. Initialization allows us to refer to specific elements using the `[]` syntax. The lines `g <- c(0,0)` and `H <- matrix(0,2,2)` initialize `g` and `H` to be a vector of length 2 and a 2×2 matrix respectively, of zeros.

From $Q(u) = (-\log(1-u)/\alpha)^{1/\beta}$, we have $y = \beta_0 + \beta_1 x$, where $y = \log(-\log(1-u))$, $\beta_0 = \log \alpha$, $\beta_1 = \beta$, and $x = \log Q(u)$. Thus if we let

$$u_i = \frac{i - \frac{1}{2}}{n}, \quad i = 1, 2, \dots, n, \quad (12)$$

$$y_i = \log(-\log(1-u_i)), \quad i = 1, 2, \dots, n, \quad (13)$$

$$x_i = \log \hat{Q}(u_i) = \log X_{(i)}, \quad i = 1, 2, \dots, n, \quad (14)$$

where $X_{(i)}$ is the i th order statistic of the random sample, we can regress the y_i 's on the x_i 's to get estimators $\hat{\beta}_0$ and $\hat{\beta}_1$, from which we get starting values for the Newton-Raphson procedure as

$$\boldsymbol{\theta} = (e^{\hat{\beta}_0}, \hat{\beta}_1)'. \quad (15)$$

These initial values are obtained in just two lines – the lines

```
theta <- as.vector(lsfrit(log(sort(x)),log(-log(1-((1:n-.5)/n))))$coef)
theta[1] <- exp(theta[1])
```

In the first of these lines, the variable name `theta` is assigned the object type `vector` using the `as.vector` function. The argument of the `as.vector` function is

```
lsfrit(log(sort(x)),log(-log(1-((1:n-.5)/n))))$coef
```

This is a call to the R (or S) function `lsfrit`. A full call to the function `lsfrit` has the following syntax

```
lsfrit(x, y, wt=<<see below>>, intercept=T, tolerance=1.e-07, yname=NULL)
```

There are two required arguments in `lsfrit` – and these required arguments are the only arguments necessary for obtaining the MLEs of Weibull distribution parameters.

1. The first required argument is a vector or matrix containing the predictor variable(s) observations. The expression for the predictor variables for the problem at hand is given in equation (14), and is simply the natural logarithm of the sorted values of the sample. Hence the expression `log(sort(x))` yields the response values to pass to `lsfit`.
2. The second required argument is a vector or matrix containing the response variable(s) observations. The expression for the response values is a little more complicated and is given in equation (13), where the u_i are given in equation (12). Let's start with the u_i . Recall that the syntax `1:n` will give the sequence of integers $1, 2, \dots, n$. So the expression `(1:n-.5)/n` is equivalent to $(i - 0.5)/n$, for $i = 1, 2, \dots, n$; in other words, the u_i . That the remainder of `log(-log(1-((1:n-.5)/n)))` is equivalent to the expression in (13) should be evident.

The function `lsfit` returns a list representing the result of the regression, with the following components:

coef: vector or matrix of coefficients. This is a matrix only if `y` has more than one column, in which case **coef** contains one column for each regression with optional constant terms in the first row. Its dimnames are taken from `x`, `y` and `yname` if applicable.

residuals: object like `y` containing residuals.

wt: if `wt` was given as an argument, it is also returned as part of the result.

intercept: logical value: records whether an intercept was used in this regression.

qr: object representing the numerical decomposition of the `x` matrix (plus a column of 1s, if an intercept was included). If `wt` was specified, the **qr** object will represent the decomposition of the weighted `x` matrix. See function `qr` for the details of this object. It is used primarily with functions like `qr.qty`, that compute auxiliary results for the regression from the decomposition.

Since `lsfit` returns this list, and the only object we need from this list is the regression coefficients, we access the **coef** object only using the `$coef` at the end of the call to `lsfit`. The end result is that the object `theta` is a vector of regression coefficients. The next line `theta[1] <- exp(theta[1])` replaces the first element of the vector `theta` with `exp(theta[1])`, in accordance with equation (15).

Moving down to the lines beginning with `alpha <- theta[1]` through `H[2,2] <- alpha*sum(xb*lx*lx + (n/beta^2))` perform the necessary computations to obtain the data-based values given for the expressions in equations (7) through (11). The `sum` function does what you would expect; it returns the sum of the values; and the `log` function is the natural logarithm. The remainder of these operations and functions have been explained in the previous pages of this document. You should spend time to convince yourself these lines perform the necessary operations.

The lines between `del <- solve(H,g)` and `theta <- theta - del` will be discussed in Section 5.2. The last lines in this section to be discussed are the lines enumerated below.

1. `if(n < 3) stop("\n\nThe length of x must be greater than 2.\n")`

The `stop` function has only one argument, the error message to be produced when the function **terminates**. This line in `weibmle` prevents the user from trying to compute MLEs from a sample of size less than 3. If the user passes a vector for `x` that has `length(x) < 3`, the program will terminate and the error message

```
Problem in weibmle(as.vector(0)):  
The length of x must be greater than 2.
```

Use `traceback()` to see the call stack

will be printed. No further computations are executed. The `stop` function is a way to prevent user error.

Another function that checks user error is the `warning` function. It should be used to warn the user that they have given a value for an argument that may result in poor function performance. The `warning` function only passes a warning message, and **does not terminate** execution of the function. The syntax of the `warning` is the same as that of the `stop` function.

2. `return(list(theta=theta,V=solve(H),ier=0))` and
`return(list(theta=theta,V=solve(H),ier=1))`

The `return` function signals the end of the function and returns to the user the objects contained as the argument of the `return` function. If more than one object is to be returned (as in our example) the multiple objects should be returned in a list. To get help on the `return` function, the syntax is `help("return")` is because the `return` function is considered a special function that is a part of the *S* and *R* syntax (as opposed to an intrinsic computational function).

5.2 Logicals and Loops

The final section of this document will address the lines in the `weibmle` example that contain the words `for` and `if`; specifically, the lines

```
for(i in 1:maxit){... }  
if( sum(abs(del) < epsilon*abs(theta))==2) { ... }
```

Help for logical statements and construct statements (like `if` and `for`, respectively) must be found like for the `return` statement – by including them in double quotes (`help("for")` or `help("if")`).

5.2.1 Loop Statements

Loops can be executed in *R* and *S* using the `for`, `while`, or `repeat` statements, having the following syntax (along with variants)

```
for(NAME in VALUE) expr
while(test) expr
repeat expr
break
next
```

If the `expr` to be evaluated is more than one statement, all statements should be enclosed in curly braces (like in our example). The `NAME` is a syntactic name for the looping variable (in our example this is `i`). Valid syntactic names are combinations of letters, numbers, and periods not beginning with a number. The `VALUE` is replaced by an S-PLUS expression returning the elements to be looped over (in our example `1:maxit`).

If `while` is used instead, then `test` must be a logical expression of length 1. For more information, see `(help("which"))`.

5.2.2 Logical or “Test” Statements

Logical statements in *S* and *R* can be evaluated using combinations of the following

```
if(test) true.expr
if(test) true.expr else false.expr
e1 || e2    (or)
e1 && e2    (and)
```

If there is more than statement in `true.expr`, then all statements should be enclosed in curly braces. If `test` has more than two branches, the `switch` function may be used. The argument `test` must be a logical expression of length one.

This type of testing is done in the Newton-Raphson algorithm to see if the iteration has converged. In the `weibmle` function, the line of interest is

```
if( sum(abs(del) < epsilon*abs(theta))==2) { ... }
```

In this logical statement, the test is as follows (read from the inside of the parentheses out:

1. For the first element of `del`, determine if it is less than the product of the absolute value of `epsilon` and the absolute value of the first element of `theta`. If it is, a value of 1 is assigned. If it is not, a value of 0 is assigned.
2. Do likewise for the second elements of `del` and `theta`.
3. Sum the two values.

- If this sum is two, then the `test` is true, and the argument in the brackets will be executed. Note, in our example, if the argument is true, the algorithm has converged (because `del` is sufficiently small) and function returns the MLE, the estimated information matrix, and an error indicator value of 0.
- If the sum is not two, then the `test` is false, the function skips the argument in brackets and evaluates the following statements. In our example, if the test is false, the algorithm has not converged, and the iteration continues.

The final result of the `weibmle` function will be a list containing the following objects:

1. `theta`: A vector of length two containing the MLEs of α and β ,
2. `V`: A 2×2 matrix containing the estimated information matrix, and
3. `ier`: A scalar integer error indicator. If `ier` = 0, the Newton-Raphson algorithm converged in no more than `maxit` iterations. If `ier` = 1, the algorithm did not converge in `maxit` iterations.

6 References

Gentle, James E. (2002) *Elements of Computational Statistics*, Springer, New York.

Hart, Jeffrey D. (1997) *Nonparametric Smoothing and Lack-of-Fit Tests*, Springer, New York.

On-line *S-Plus* Help Files (2002) Insightful Corporation.

A Example of Function weibmle

```
weibmle <- function(x,maxit=20,epsilon=0.0001) {
#-----
# An R function for computing the maximum likelihood estimators
# for the parameters of a 2-parameter Weibull distribution.
#
# REQUIRED ARGUMENTS:
#   x - a vector containing the data
# OPTIONAL ARGUMENTS:
#   maxit - an integer containing the maximum number of
#           iterations for the Newton-Raphson algorithm.
#           The default value of maxit=20.
#   epsilon - a real number containing the convergence
#             criteria. The default value of epsilon=0.0001.
#
# VALUE: The function weibmle returns a list containing
#   theta - a vector of length two containing the maximum
#           likelihood estimates,
#   H - a 2x2 matrix containing the inverse of the Hessian
#       matrix for the last iteration, and
#   ier - an integer error indicator. If the algorithm
#         converged in <= maxit iterations, ier = 0.
#         Otherwise ier = 1.
#
# Created: 10/10/03 JLH
#-----

  n <- length(x)                # n = sample size

  if(n < 3) stop("\n\nThe length of x must be greater than 2.\n\n")

# Get starting values theta_0 for Newton-Raphson algorithm:

  theta <- as.vector(lsfrit(log(sort(x)),log(-log(1-((1:n-.5)/n))))$coef)
  theta[1] <- exp(theta[1])

# Initialize gradient vector and Hessian matrix:

  g <- c(0,0)
  H <- matrix(0,2,2)

# Newton-Raphson iteration:
```

```

for(i in 1:maxit) {

  alpha <- theta[1]
  beta  <- theta[2]

  xb <- x^beta
  lx <- log(x)
  ss <- sum(xb*lx)

  g[1] <- sum(xb)-(n/alpha)
  g[2] <- alpha*ss - sum(lx) - (n/beta)

  H[1,1] <- (n/alpha^2)
  H[1,2] <- H[2,1] <- ss
  H[2,2] <- alpha*sum(xb*lx*lx) + (n/beta^2)

  del <- solve(H,g)

  if( sum(abs(del) < epsilon*abs(theta))==2) {
    return(list(theta=theta,V=solve(H),ier=0))}

  theta <- theta - del

}

return(list(theta=theta,V=solve(H),ier=1))

}

```