Getting Started With R

Installation.

The R software package can be obtained free from www.r-project.org. To install R on a Windows machine go to this web address; in the left margin under Download, select CRAN; select the CRAN Mirror site nearest you; select Windows; select base; and then finally, select the .exe setup program, which corresponds to the latest release of R. After downloading is finished (seconds or minutes depending on the speed of your connection), the R icon should appear on your desktop. Select it to start R. When R starts you will see the Console window with a prompt > at which you can type an instruction.

Using R as a calculator.

You should follow through this tutorial seated at a computer running R. To get the feel of how the Console works, try typing a few simple computations, pressing the Enter key after each. Use * for multiplication and ^ for exponentiation. Here are some simple examples.

```r
> 2 + 2
[1] 4
> (2 + 2)*5
[1] 20
> 3^2 + 4^2
[1] 25
> (4/2)^(1/2)
[1] 1.414214
> sqrt(2)
[1] 1.414214
```

After pressing Enter, if you discover you have made a mistake, it may be easier to use the up-arrow key to retrieve the previous instruction and edit it than to retype everything afresh.

```r
> 1 + 2 + 3 + 4^2
[1] 22
> (1 + 2 + 3 + 4)^2  (Previous instruction edited by inserting parentheses.)
[1] 100
```

If you want to save one result before going on to the next, you can use the "<-" sign to store the preliminary result as an "object."

```r
> a <- 1 + 2 + 3 + 4
> a
[1] 10
```

Names of objects in R are case sensitive. For example n and N are different objects. If you have not previously defined A, then typing A at the > prompt in the Console window will give the following.

```r
> A
Error: Object "A" not found
```
Vectors.

R is an object-oriented language. For us, the most fundamental objects are vectors. To begin, you can think of a vector as an ordered list of numbers called elements. The index of an element specifies its position in the list: index 1 for the first element, index 2 for the second, and so on.

Defining a vector.

The construct \( \texttt{c}(\ldots) \) can be used to make a vector with several elements. You can think of \( \texttt{c} \) as standing for "combine." But R ordinarily treats vectors as columns of elements (even though vectors are sometimes printed out in rows), so you may prefer to think as \( \texttt{c} \) as standing for "column vector." The name of a vector can be several characters long, but the first character must be a (capital or small) letter.

\[ > \texttt{v1} \leftarrow \texttt{c}(7, 2, 3, 5) \]

\[ > \texttt{v1} \]

\[ [1] \ 7 \ 2 \ 3 \ 5 \]

Here are a few convenient shorthand notations for making vectors. They are often used to make vectors that are too long to make conveniently using the \( \texttt{c} \)-notation and that have elements following a particular pattern.

\[ > \texttt{v2} \leftarrow \texttt{numeric}(4) \]

\[ > \texttt{v2} \]

\[ [1] \ 0 \ 0 \ 0 \ 0 \]

\[ > \texttt{numeric}(10) \]

\[ \]

\[ [1] \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \]

What do you suppose \( \texttt{numeric}(100) \) does? Try it. Can you figure out the meaning of the bracketed numbers at the beginning of each printed line?

This method is often used to "initialize" a very long vector of 0s that is to be used in a loop. Inside the loop, each element may be changed, one at a time, from 0 to some other value. More about loops later.

The next notation allows you to make a vector where all elements are equal to a specified value (not necessarily 0).

\[ > \texttt{v3} \leftarrow \texttt{rep}(3, 4) \]

\[ > \texttt{v3} \]

\[ [1] \ 3 \ 3 \ 3 \ 3 \]

\[ > \texttt{rep}(4, 3) \]

\[ [1] \ 4 \ 4 \ 4 \]

Verify that there is no difference between the effect of \( \texttt{numeric}(4) \) and of \( \texttt{rep}(0, 4) \).

The colon notation is used for consecutive integers.

\[ > \texttt{v4} \leftarrow 1:4 \]

\[ > \texttt{v4} \]
Predict the output of \(4:\text{-}1\), and verify. Now try to predict the output of \(3.5:10\), and verify.

A sequence of equally spaced elements that do not differ by integer values can be specified as follows.

\[
\text{\texttt{seq(1, 2.2, by=.1)}}
\]

\[
\begin{array}{l}
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 \\
[13] 2.2
\end{array}
\]

In the \texttt{seq} notation above there are three arguments (in order of appearance): the first argument is the \textit{beginning} number, the second argument is the \textit{ending} number, and the additional argument (which must be labeled using \texttt{by=} ) is the \textit{increment} between successive elements of the vector. This vector has 13 elements. How many elements print on each line depends on the width of the Console window. You can change the width of the Console window by selecting an edge and dragging it to the desired width. With a wider window the entire result above would fit on one line. With a narrower window your output might look like this:

\[
\text{\texttt{seq(1, 2.2, by=.1)}}
\]

\[
\begin{array}{l}
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 \\
[8] 1.7 1.8 1.9 2.0 2.1 2.2
\end{array}
\]

Alternatively, an additional argument (which must be labeled using \texttt{length=} ) can be used specify the length of the vector of equally spaced values. The following code gives a vector with 11 elements.

\[
\text{\texttt{seq(1, 2.2, length=11)}}
\]

\[
\begin{array}{l}
[1] 1.00 1.12 1.24 1.36 1.48 1.60 1.72 1.84 1.96 2.08 \\
[11] 2.20
\end{array}
\]

The notation \texttt{seq(1, 10)}, with no additional argument, gives the same result as \texttt{1:10}. Try it. What vector results from \texttt{seq(1, 10.5)}? From \texttt{(10:22)/10}?

The \texttt{c}-symbol can be used to combine vectors as well as single numbers.

\[
\text{\texttt{v5 <- c(v3, v4, 7)}}
\]

\[
\text{\texttt{v5}}
\]

\[
[1] 3 3 3 3 1 2 3 4 7
\]
Simple arithmetic.

Operations on vectors are performed element by element. In particular, when an operation involves a vector and a number, the number is used to modify each component of the vector as specified by the operation. Here are some examples.

```r
> w1 <- 3*v1
> v1; w1
 [1]  7  2  3  5
[1] 21  6  9 15
> w2 <- v1/2
> w2
 [1] 3.5 1.0 1.5 2.5
> w3 <- 5 - v1
> w3
 [1] -2  3  2  0
> w4 <- w3^2
> w4
 [1]  4  9  4  0
```

Notice that two (preferably brief and closely related) statements can be put on the same line if separated by a semicolon (;). You should experiment with minor variations of the statements above. Also, show that `(0:10)/10` and `seq(0, 10, by=.1)` give the same results.

When arithmetic involves two vectors of the same length, then the operation applies to elements with the same index.

```r
> w5 <- w3 + w4
> w5
 [1]  2 12  6  0
> (5:1)*(1:5)
 [1] 5 8 9 8 5
> (5:0)^(0:5)
 [1] 1 4 9 8 1 0
```

Predict and verify the results of \( w3 - w4 \), \( w4 - w3 \), \( w4 \times w3 \), \( w4^w3 \), and \( w4/w3 \). Here are a few more examples.

```r
> (1:4)^2
 [1]  1  4  9 16
> (5:1)/(1:5)
 [1] 5.0 2.0 1.0 0.5 0.2
```

What do you suppose is the result of \( (1:4)^{(0:3)} \)? Verify your answer.

Note: R can also compute vector products and do matrix multiplication, but we do not discuss these topics here.
Indexes and Assignments.

Sometimes we want to use only one element of a vector. The index notation \[ \] helps to do this. The simplest version of referencing by index is just to specify the index (position number within the vector) you want.

\[ w1 \]
\[
\begin{bmatrix}
  21 & 6 & 9 & 15 \\
\end{bmatrix}
\]
\[ w1[3] \]
\[
\begin{bmatrix}
  9 \\
\end{bmatrix}
\]

\[ v5 \]
\[
\begin{bmatrix}
  3 & 3 & 3 & 3 & 1 & 2 & 3 & 4 & 7 \\
\end{bmatrix}
\]
\[ v5[9] \]
\[
\begin{bmatrix}
  7 \\
\end{bmatrix}
\]

The bracket notation can be used along with the assignment operator = to change the value of an element of an existing vector.

\[ v2 \]
\[
\begin{bmatrix}
  0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
\[ v2[1] = 6 \]
\[ v2 \]
\[
\begin{bmatrix}
  6 & 0 & 0 & 0 \\
\end{bmatrix}
\]

The bracket notation can also be used to specify or change more than one element of a vector.

\[ v3 = \text{numeric}(10) \]
\[ v3 \]
\[
\begin{bmatrix}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
\[ v3[1:3] = 4:6 \]
\[ v3 \]
\[
\begin{bmatrix}
  4 & 5 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
\[ v3[2:4] \]
\[
\begin{bmatrix}
  5 & 6 & 0 \\
\end{bmatrix}
\]

We will see still other uses for the bracket notation later.

Operations with vectors of unequal length.

When the vectors operated upon are of unequal lengths, then the shorter vector is "recycled" as often as necessary to match the length of the longer vector. If the vectors are of different lengths because of a programming error, this can lead to unexpected results. But sometimes recycling of short vectors is the basis of clever programming. Also, all of the examples we have shown so far with operations involving a vector and a single number recycle the single number (which R regards as a one-element vector).
We illustrate with three examples of operations with vectors of unequal lengths. You should verify each of them by hand computation.

```
> (1:10)/(1:2)
[1] 1 1 3 2 5 3 7 4 9 5
> (1:10)/(1:5)
[1] 1.000000 1.000000 1.000000 1.000000 1.000000 6.000000
[7] 3.500000 2.666667 2.250000 2.000000
```

Notice that R gives a warning message if the recycling comes out "uneven"; this situation often arises because of a programming error.

```
> (1:10)/(1:3)
[1]  1.0  1.0  1.0  4.0  2.5  2.0  7.0  4.0  3.0 10.0
Warning message: longer object length is not a multiple of shorter object length in: (1:10)/(1:3)
```

**Output formats.**

In case you haven't figured it out by now, the bracketed number at the beginning of each line of printed output gives the index of the first element printed on the line. In the output for `(1:10)/(1:5)`, the 9th element is $9/4 = 2.25$.

Depending on the width you have chosen for your Console window and the number of digits in each result, R may put various numbers of results on a line. With a narrower window, the display for `(1:10)/(1:5)` might have been printed as shown below.

```
> (1:10)/(1:5)
[1] 1.000000 1.000000 1.000000 1.000000
[9] 2.250000 2.000000
```

There are other more complex ways to control the appearance of printed output, but this is enough for now.

**Vector functions.**

We begin with vector functions that return a single number. The meanings of the functions in the following demonstration should be self-explanatory.

```
> w2
[1] 3.5 1.0 1.5 2.5
> max(w2)
[1] 3.5
> w3
[1] -2 3 2 0
> mean(w3)
[1] 0.75
```
> v1
[1] 7 2 3 5
> sum(v1)
[1] 17

> v4
[1] 1 2 3 4
> prod(v4)
[1] 24

> v5
[1] 3 3 3 3 1 2 3 4 7
> length(v5)
[1] 9

By using parentheses for grouping, one can combine several expressions that involve functions.

> (sum(w3^2) - (sum(w3)^2)/length(w3)) / (length(w3) - 1)
[1] 4.916667

A simpler way to get the same result would be to use the var function.

> var(w3)
[1] 4.916667

Verify that the standard deviation of the numbers -2, 3, 2, 0 is computed by using the expression sqrt(var(w3)).

Some vector functions return vectors. For example, sqrt takes the square root of each element of a vector, and cumsum forms cumulative sums of the elements of a vector.

> sqrt(c(1, 4, 9, 16, 25))
[1] 1 2 3 4 5

> cumsum(1:5)
[1]  1  3  6 10 15

> cumsum(5:1)
[1]  5  9 12 14 15

> v5
[1] 3 3 3 3 1 2 3 4 7
> cumsum(v5)
[1] 3 6 9 12 15 18 22 29

The vector function unique, eliminates "redundant" elements in a vector and returns a vector of elements with no repeated values.

> unique(v5)
[1] 3 1 2 4 7
\begin{verbatim}
> c(rep(3,5), rep(4,10))
 [1] 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4
> length(c(rep(3,5), rep(4,10)))
[1] 15

> unique(c(rep(3,5), rep(4,10)))
[1] 3 4
> length(unique(c(rep(3,5), rep(4,10))))
[1] 2
\end{verbatim}

The function \texttt{round} with no second argument rounds values to integers. With a second argument, it rounds to the requested number of digits.

\begin{verbatim}
> round(2.5)
[1] 2
> round(3.5)
[1] 4
> round(5/(1:3))
[1] 5 2 2
> round(5/(1:3), 3)
[1] 5.000 2.500 1.667
\end{verbatim}

The functions \texttt{floor} (to round down) and \texttt{ceiling} (to round up) work similarly. You should explore them, using the vectors just shown to illustrate \texttt{round}.

\textit{Comparisons and logical values.}

If two vectors are compared element by element, the result is a \textit{logical} vector, which has elements \texttt{TRUE} and \texttt{FALSE}. Common comparison operators are \texttt{==} (equal), < (less than), \texttt{<=} (less than or equal to), \texttt{!=} (not equal) and so on. Here are some examples.

\begin{verbatim}
> 1:5 < 5:1
[1]  TRUE  TRUE FALSE FALSE FALSE
> 1:5 <= 5:1
[1]  TRUE  TRUE  TRUE FALSE FALSE
> 1:5 == 5:1
[1] FALSE FALSE  TRUE FALSE FALSE

> 1:4 > 4:1
[1] FALSE FALSE  TRUE  TRUE
> 1:5 < 4
[1]  TRUE  TRUE  TRUE FALSE FALSE

> w4
[1] 4 9 4 0
> x <- (w4 == 4)
> x
[1]  TRUE FALSE  TRUE FALSE
\end{verbatim}

If R is "coerced" (forced) to do \textit{arithmetic} on logical values, then it takes \texttt{TRUE} to be 1 and \texttt{FALSE} to be 0. The mean of a set of 0s and 1s is the proportion of 1s. So the first result below
shows that half of the elements of the numerical vector \( w4 \) are equal to 4; equivalently, half of the elements in the logical vector \( x \) are TRUE.

\[
\begin{align*}
> \text{mean}(x) \\
[1] & 0.5 \\
> \text{sum}(\text{c(T, T, F, F, T, T)}) \\
[1] & 4 \\
> \text{mean}(\text{c(T, T, F, F, T, T)}) \\
[1] & 0.6666667
\end{align*}
\]

Comparisons can be used inside brackets to specify particular elements of a vector. In such instances it is convenient to read [ ] as "such that."

\[
\begin{align*}
> v5 \\
[1] & 3 3 3 3 1 2 3 4 7 \\
> v5[v5 < 3] \\
[1] & 1 2 \\
> \text{length}(v5[v5 < 3]) \\
[1] & 2 \\
> \text{sum}(v5 < 3) \\
[1] & 2
\end{align*}
\]

The last two results above are logically equivalent. From all three of these statements we see that exactly two elements of \( v5 \) are smaller than 3.

To get more information about a function, type ? immediately followed by the function name. For example ?length. But some of the explanations may be incomprehensibly technical.

**Simple loops.**

Because R handles vectors as objects, many computations that would require loops in other software do not require loops in R. However, loops are convenient or necessary for some computations.

**A loop to find the mean of a vector.** We have already seen that the mean function can be used to find the mean of the numbers \(-2, 3, 2, 0\) is 3/4 or 0.75.

\[
\begin{align*}
> w3 \\
[1] & -2 3 2 0 \\
> \text{mean}(w3) \\
[1] & 0.75
\end{align*}
\]

Now, as a "toy" example, we show how we could find the mean using a loop. We initialize the object \( \text{avg} \) to have the value 0. Then on each passage through the loop we add (cumulatively) one more element of \( w3 \). And finally we divide by the accumulated total by the number of elements of \( w3 \) to find the mean of \( w3 \). We write this "program" without prompt characters (>) so you can cut and paste it from this document into the R Console. (When you do this, do not cut/paste the output, last two lines below.)

\[
\begin{align*}
w3 & \leftarrow \text{c}(-2, 3, 2, 0) \quad \# \text{omit if } w3 \text{ defined earlier in your R session} \\
\text{avg} & \leftarrow 0 \\
n & = \text{length}(w3) \\
\text{for} & \ (i \ \text{in} \ 1:n)
\end{align*}
\]
{  
  avg <- avg + w3[i]  
}

avg <- avg/n

avg

> avg
[1] 0.75

Because R is a statistical language, the mean function is already defined in R, and it was not necessary to write this program. But follow through the steps so you will know how the loop structure works. After each of the four passes through the loop, what is the current value of avg?

**Graphical functions.**

**Plotting a continuous function.** R has extensive graphical capabilities. We have already seen how the `plot` function can be used to show sequences. It is often used give the appearance of a continuous plot of a function. The following code produces the graph of the parabola \( d = 6t(1 - t) \). The parameter `type = "l"` connects 200 dots to give the appearance of a continuous curve. (The symbol inside quotes is the letter ell, for line—not the number one.) Use `?plot` to see other choices for the `type` parameter. Here we also choose to use additional parameters to make the line double width and blue. Ordinarily, it is understood that the first parameter is for the x-axis and the second for the y-axis; additional parameters should be designated by `name` (as with `type`), rather than by `position` (as for the first and second parameters).

\[
\text{t} \leftarrow \text{seq}(0, 1, \text{length}=200) \\
\text{d} \leftarrow 6\times(t - t^2) \\
\text{plot(t, d, type="l", lwd=2, col="blue", main="Parabola")}
\]
Without the argument type="l" each plotted point would be shown as a small open circle, and there would be no connecting lines. Try it. If the first argument is omitted, the single vector argument is plotted on the y-axis against the index on the x-axis. With \( d \) defined as above, what is the result of \( \text{plot}(d) \)? The function \( f(x) = 6x(1-x) \), for \( 0 \leq x \leq 1 \), and 0 otherwise encloses unit area and is never negative. Thus it qualifies as a density function of a continuous random variable, to be studied later in this course.

If you want to use a graph in a report, you can adjust the R-Graphics window so that its size and its vertical-to-horizontal ratio is about what you want use, and then cut and paste it from the R-Graphics window directly into a word processor such as MS Word. (In our experience with MS Word, it is usually easiest to paste a graph into a blank paragraph that lies between two existing paragraphs.) Graphs can also be saved in a variety of graphic formats.

**Making a histogram of data.** Another important graphical function is the histogram function \( \text{hist} \). As an illustration of it, suppose the IQ scores of 20 elementary school students are listed in the vector \( \text{IQ} \) below. (There are cleverer ways to import data into R, but this way is simple to understand and will suffice for now.) The second statement below makes a histogram of these data.

\[
\text{IQ} \leftarrow c(84, 108, 98, 110, 86, 123, 101, 114, 121, 131,
90, 108, 105, 93, 95, 102, 119, 98, 94, 73)
\]
\[
\text{hist(IQ)}
\]
You should look at the entries in the data vector and determine for yourself that there is indeed one observation in the interval (70, 80], there are three in the interval (80, 90], and so on, to verify the heights of each of the histogram bars. (Looking at the result of `sort(IQ)` may help.) A histogram should appear to "balance" at the sample mean (considering the bars to have "weights" in proportion to their heights). Use `mean(IQ)` to find the sample mean of these data. (Answer: 102.65.) Also use `sd(IQ)` to find the standard deviation, and use the formula for the sample standard deviation to get the same result with a hand calculator. The so-called empirical rule says that for many samples all or almost all of the observations lie within an interval that extends three sample standard deviations on either side of the sample mean. Is that true here?

An algorithm in R determines how many intervals will be used and what cut-points (or breaks) will separate the intervals. For most kinds of data, as above, this algorithm gives reasonable results. But the `hist` function can take additional arguments that allow you to control how the histogram is drawn. If we would like to use bins of length 15, we can program it as below:

```r
> brk<-c(70, 85, 100,115,130,145)
> hist(IQ,breaks=brk)
```

**Missing data.**

Note that missing values are represented in R by the NA symbol. Now replace the first element by NA in IQ and calculate the mean again.

```r
> IQ[1]<-NA
```
Sampling from a finite population.

The `sample` function selects a random sample of a specified size from a given population. Its first argument is a vector whose elements are the population. Its second argument is the sample size. If sampling is to be done with replacement, a third argument `repl=T` is required, otherwise sampling is without replacement. (When sampling without replacement, the sample size can't exceed the population size.) The `sample` function uses a pseudorandom number generator to do the sampling in a fashion that is not obviously distinguishable from results of actual chance draws from a population, for example by cards from a well shuffled deck. We illustrate its use with two examples.

**Cards.** Suppose the population is the 52 cards of a standard deck, and we want to select 5 cards at random (without replacement, of course). The following code does the sampling. Because this is a random process, your answer will (almost surely) not be the same as the one shown below.

```r
> h <- sample(1:52, 5)
> h
[1] 36 10 14 39 26
```

In order to view the output as actual playing cards, you would have to make a numbered list of the cards in a standard deck. If, for example, the numbers 1, 2, 3, 4 correspond to the four Aces (A♣, A♦, A♥, A♠), then there are no Aces in the 5-card hand shown above. This could be determined in R by the function `sum(h < 5)`. In the printout above, what are the elements of the logical vector `(h < 5)`? What is returned by `sum(h < 5)` in this instance? If you repeat this random experiment several times in R, you will find that you do sometimes get aces in your randomly dealt hand. What are the possible values that might be returned by `sum(h < 5)`? Using combinatorial methods, can you figure out the probability of getting no aces in a 5-card hand? What would happen if you tried to sample 60 cards instead of five? Try it in R and see.

The following program draws 100,000 five-card hands, counts the aces in each hand, and makes a histogram of the resulting 100,000 counts. The parameter `prob=T` puts "densities" (that is relative frequencies or proportions out of 100,000) on the vertical axis of the histogram. These relative frequencies simulate the probabilities of getting various numbers of Aces. The `breaks` parameter improves the break points along the horizontal axis. The last statement shows how many times each number of Aces was observed. (The command `as.factor` interprets the vector Aces is a vector of integers to be tallied. What do you get from just `summary(aces)`? Comments following the symbol `#` are annotations for humans, ignored by R.

You should run this program for yourself. Because random sampling is involved, you will get somewhat different counts, but your histogram should not look a lot different from the one shown below. (The exact probability, to four places, of getting no aces in a five-card hand is 0.6588. As any poker player already knows, the probability of getting four Aces in a five-card hand is very
small indeed; it's $1.847 \times 10^{-5}$. Perhaps somewhat more likely than that is amateur poker games
where the cards left over from a previous game are not often shuffled well enough to imitate randomness as well as the pseudorandom numbers in R.)

```r
m <- 100000
aces <- numeric(m) # define the initial vector of results

for (i in 1:m)
{
    h <- sample(1:52, 5)
    aces[i] <- sum(h < 5)
}

cut <- (0:5) - .5
hist(aces, breaks=cut, prob=T)
summary(as.factor(aces)) # observed counts
summary(as.factor(aces))/m # simulated probabilities

> summary(as.factor(aces)) # observed counts
 0     1     2     3     4
 66038 29736  4057   168     1

> summary(as.factor(aces))/m # simulated probabilities
 0       1       2       3       4
 0.66038 0.29736 0.04057 0.00168 0.00001
```

**Histogram of aces**

```
0    0.1    0.2    0.3    0.4    0.5    0.6
0 1 2 3 4
aces
```
**Dice.** The code below simulates rolling two dice. For each roll, the population is 1:6. Because it is possible that both dice show the same number, we are sampling with replacement.

\[
> d <- \text{sample}(1:6, 2, \text{repl}=\text{T})
\]
\[
> d
\]
\[
[1] 6 6
\]

In our run of this code in R, it happens that we got two 6s. If you repeat this experiment several times, you will find that you do not usually get the same number on both dice. What is the probability of getting the same number on both dice? You could determine whether this happens on a particular simulated roll of two dice with the code \texttt{length(unique(d))}. What would this code return in the instance shown above? What are the possible values that might be returned?

Here is a program that simulates the sums of the numbers in 100,000 two-dice experiments. The actual probability of getting a sum of 7 in rolling two fair dice is 1/6.

\[
m <- 100000
x <- \text{numeric}(m)
\]
\[
\text{for} \ (i \ \text{in} \ 1:m) \\
\quad \{ \\
\quad \text{x}[i] \ <- \ \text{sum(\text{sample}(1:6, 2, \text{repl}=\text{T}))} \\
\quad \}
\]
\[
cut <- (1:12) + .5
\]
\[
\text{hist}(x, \text{breaks=cut, prob=\text{T}}) \\
\text{summary(\text{as.factor}(x))}
\]

\[
> \text{summary(\text{as.factor}(x))}
\]
\[
2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \\
2793 \ 5440 \ 8428 \ 11048 \ 13794 \ 16797 \ 13902 \ 11070 \ 8337 \ 5595 \ 2796
\]
In this case, we could avoid using a loop. The program below produces essentially the same results for the two-die experiment as the one above. (Can you figure out why?) But the program below runs faster because R does the internally-programmed loops necessary for vector arithmetic faster than it executes loops you write yourself. Can you give the exact probabilities corresponding to each of the possible sums 2, 3, ..., 12? [Hint: Each result can be expressed as a fraction with denominator 36.]

```r
m = 100000
d1 = sample(1:6, m, repl=T)
d2 = sample(1:6, m, repl=T)
x = d1 + d2

cut <- (1:12) + .5
hist(x, breaks=cut, prob=T,
     main="Sums on Two Fair Dice", col="cornsilk2")
summary(as.factor(x))

> summary(as.factor(x))

2     3     4     5     6     7     8     9    10    11    12
2730  5513  8401 11061 13813 16621 13913 11156   84 50  5638  2704
```

Note: For a list of colors that can be used for histogram bars and other purposes, type `colors()` at the prompt in the R Console.