R Programming

Control Flow and Functions

Expressions and Compound Expressions

R programs are made up of expressions. These can either be simple expressions (of the type we've seen already) or compound expressions consisting of simple expressions separated by semicolons or newlines and grouped within braces.

{ $expr_1$; $expr_2$; ... ; $expr_n$ }

Every expression in R has a value and the value of the compound expression above is the value of $expr_n$. E.g.

```
> x = { 10 ; 20 }
> x
[1] 20
```

Assignments within Compound Expressions

It is possible to have assignments within compound expressions and the values of the variables that this produces can be used in later expressions.

```
> z = { x = 10 ; y = x^2; x + y }

> x
[1] 10

> y
[1] 100

> z
[1] 110
```

If-Then-Else Statements

If-then-else statements make it possible to choose between two (possibly compound) expressions depending on the value of a (logical) condition.

`if (`*condition*`)` $expr_1$ `else` $expr_2$

If *condition* is true then $expr_1$ is evaluated otherwise $expr_2$ is executed.

Notes

1. Only the first element in *condition* is checked.
2. The value of the whole expression is the value of whichever expression was executed.

If-Then-Else Examples

The expression

```
if (x > 0) y = sqrt(x) else y = -sqrt(-x)
```

provides an example of an if-then-else statement which will look familiar to Pascal, Java, C, or C++ programmers.

The statement can however be written more succinctly in R as

```
y = if (x > 0) sqrt(x) else -sqrt(-x)
```

which will look familiar to Lisp or Algol programmers.

If-then Statements

There is a simplified form of if-then-else statement which is available when there is no $expression_2$ to evaluate. This statement has the general form

`if (`*condition*`)` *expression*

and is completely equivalent to the statement

`if (`*condition*`)` *expression* `else NULL`

For Loops

As part of a computing task we often want to repeatedly carry out some computation for each element of a vector or list. In R this is done with a `for` loop.

A `for` loop has the form:

`for(`*variable* `in` *vector*`)` *expression*

The effect of such a loop is to set the value of *variable* equal to each element of the *vector* in turn, each time evaluating the given *expression*.

For Loop Example 1

Suppose we have a vector `x` that contains a set of numerical values, and we want to compute the sum of those values. One way to carry out the calculation is to initialise a variable to zero and to add each element in turn to that variable.

```
s = 0
for(i in 1:length(x))
    s = s + x[i]
```

The effect of this calculation is to successively set the variable `i` equal to each of the values `1`, `2`, ..., `length(x)`, and for each of the successive values to evaluate the expression `s = s + x[i]`.

For Loop Example 2

The previous example is typical of loops in many computer programming languages, but R does not need to use an integer *loop variable*.

The loop could instead be written

```
s = 0
for(elt in x)
    s = s + elt
```

This is both simpler and more efficient.

The "Next" Statement

Sometimes, when given condition are met, it is useful to be able to skip to the end of a loop, without carrying out the intervening statements. This can be done by executing a `next` statement when the conditions are met.

```
for(variable in vector) {
  expression1
  expression2
  if (condition)
    next
  expression3
  expression4
}
```

When *condition* is true, $expression_3$ and $expression_4$ are skipped.

While Loops

For-loops evaluate an expression a fixed number of times. It can also be useful to repeat a calculation until a particular condition is false. A *while-loop* provides this form of control flow.

```
while (condition) expression
```

Again, *condition* is an expression which must evaluate to a simple logical value, and *expression* is a simple or compound expression.

While Loop Example

As a simple example, consider the problem of summing the integers until the sum exceeds a particular threshold. For a threshold of 100, we can do this as follows.

```
> threshold = 100
> n = 0
> s = 0
> while (s <= threshold) {
      n = n + 1
      s = s + n
  }
> c(n, s)
[1]  14 105
```

Repeat Loops

The last form of looping in R is the *repeat-loop*.

This kind of loop has the form

```
repeat expression
```

The effect of this kind of loop is to execute the given expression for ever. To exit from this kind of loop it is necessary to use the `break` statement.

Exiting from Loops with "Break"

The `break` statement provides quick way of exiting from any type of control loop. E.g.

```
> threshold = 100
> n = 0
> s = 0
> repeat {
      if (s > threshold)
          break
      n = n + 1
      s = s + n
  }
> c(n, s)
[1]  14 105
```

(This loop is equivalent to the `while` loop we saw earlier.)

The `switch` Function

The switch function makes it possible to choose between a number of alternatives. It is rather like a generalised if-then-else statement.

`switch(`*expr*`,`
tag$_1$ `=` *rcode-block*$_1$`,`
tag$_2$ `=` *rcode-block*$_2$`,`
$\vdots$
`)`

The function selects one of the code blocks, depending on the value of *expr*.

Switch by Index

If *expr* has a numeric value equal to *n*, then the *n*th of the code blocks is executed. In this case the code blocks do not need to be tagged.

```
> k = 5
> switch(k, "I", "II", "III", "IV", "V",
          "VI", "VII", "VIII", "IX", "X")
[1] "V"
```

If *n* falls outside the range of possible indices then `NULL` is returned.

Switch by Tag

If *expr* has a character string value then the tags are searched for one that is an exact match. If there is one the corresponding code block is evaluated and its value returned.

```
> x = rnorm(10)
> loc.est = "mean"

> switch(loc.est,
         median = median(x),
         mean = mean(x))
[1] 0.2258422

> mean(x)
[1] 0.2258422
```

Variations

If there is no matching tag then:

1. if there is an untagged code block it is executed and its value returned,
2. if there is no untagged code block then `NULL` is returned.

```
> switch("c", a = 1, 3, b = 2)
[1] 3

> switch("c", a = 1, b = 2)
```

Using Empty Code Blocks

Empty code blocks can be used to make several tags match the same code block.

```
> switch("a",
         a =,
         b = "gotcha",
         c = "missed")
[1] "gotcha"
```

In this case a value of either `"a"` or `"b"` will cause `"gotcha"` to be returned.

Functions

- R differs from many other statistical software systems because it is designed to be extensible. Users can add new functionality to the system in way which makes it impossible to distinguish that functionality from the capabilities shipped with the system.
- Additional functionality is added to R by defining new *functions*.
- Functions take a number of values as *arguments* and return a single *value*.

Defining Functions

Here is a function which squares its argument.

```
> square = function(x) x * x
```

It works just like any other R function.

```
> square(10)
[1] 100
```

Because the `*` operator acts element-wise on vectors, the new square function will act that way too.

```
> square(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Functions Defined in Terms of Other Functions

Once defined, the `square` function can be used in other function definitions.

```
> sumsq = function(x) sum(square(x))

> sumsq(1:10)
[1] 385

> sumsq = function(x) sum(square(x - mean(x)))

> sumsq(1:10)
[1] 82.5
```

Functions in General

In general, an R function definition has the form:

```
function (arglist) body
```

where

arglist is a (comma separated) list of variable names known as the *formal arguments* of the function,

body is a simple or compound expression known as the *body* of the function.

Functions are usually, but not always, assigned a name so that they can be used in later expressions.

A Simple Example

Here is a simple function which, given the values a and b, computes the value of $\sqrt{a^2+b^2}$.

```
> hypot = function(a, b) sqrt(a^2 + b^2)
```

- The formal arguments to the function are `a` and `b`.
- The body of the function consists of the simple expression `sqrt(a^2 + b^2)`.
- The function has been assigned the name "`hypot`."

Evaluation of Functions

Function evaluation takes place as follows:

1. Temporarily create a set of variables by associating the arguments passed to the function with the variable names in *arglist*.
2. Use these variable definitions to evaluate the function body.
3. Remove the temporary variable definitions.
4. Return the computed values.

Evaluation Example

Evaluating the function call

```
hypot(3, 4)
```

takes place as follows:

1. Temporarily create variables `a` and `b`, which have the values `3` and `4`.
2. Use these values to compute the value (5) of `sqrt(a^2 + b^2)`.
3. Remove the temporary variable definitions.
4. Return the value 5.

Example: Computing Square Roots

- There is a simple iterative procedure for computing square roots of positive numbers.
- To compute the square root of x:
  1. Start with an initial guess r_0 (using $r_0 = 1$ works).
  2. Update the guess using:

$$r_{i+1} = \frac{(r_i + x/r_i)}{2}.$$

  3. Repeat step 2 until convergence.

How the Procedure Works

- The values r and x/r bracket $\sqrt{x}$.
- The value halfway between r and x/r should be closer to $\sqrt{x}$ than either of the individual values.
- Repeating the process should produce a sequence of approximations that approach $\sqrt{x}$.

The Procedure in R

```
> root =
      function(x) {
          rold = 0
          rnew = 1
          while(rnew != rold) {
              rold = rnew
              rnew = 0.5 * (rnew + x/rnew)
          }
          rnew
      }
> root(2)
[1] 1.414214
> root(3)
[1] 1.732051
```

Convergence Issues

- While the procedure for computing square roots "should" converge, if it fails to do so there is a problem.
- If the procedure fails to converge the function enters an infinite loop and does not return.
- This problem can be eliminated by limiting the number of iterations which can take place.

An Improved Procedure

```
> root =
      function(x) {
          rold = 0
          rnew = 1
          for(i in 1:10) {
              if (rnew == rold)
                  break
              rold = rnew
              rnew = 0.5 * (rnew + x/rnew)
          }
          rnew
      }
> root(2)
[1] 1.414214
```

Optional Arguments

- The last version of `root` fixes the number of iterations at 10.
- It would be useful if this value could be overridden by the users.
- This can be done by adding an additional optional argument which takes the default value 10.

Making Use of Optional Arguments

```
> root =
      function(x, maxiter = 10) {
          rold = 0
          rnew = 1
          for(i in 1:maxiter) {
              if (rnew == rold)
                  break
              rold = rnew
              rnew = 0.5 * (rnew + x/rnew)
          }
          rnew
      }
> root(2)
[1] 1.414214
```

Vectorisation

- The test `if (rnew == rold)` uses only the first element of `rnew == rold` this can create problems if `root` is applied to a vector argument.

```
> root(1:3)
[1] 1.0 1.5 2.0
Warning message:
In if (rnew == rold) break :
  the condition has length > 1 and
  only the first element will be used
```

- The wrong answer is returned and a warning message is printed.
- Fortunately the problem is very easy to fix.

A Vectorised Square Root Function

```
> root =
      function(x, maxiter = 10) {
          rold = 0
          rnew = 1
          for(i in 1:maxiter) {
              if (all(rnew == rold))
                  break
              rold = rnew
              rnew = 0.5 * (rnew + x/rnew)
          }
          rnew
      }
> root(1:3)
[1] 1.000000 1.414214 1.732051
```

Specifying a Starting Point

```
> root =
      function(x, maxiter = 10, start = 1) {
          rold = 0
          rnew = start
          for(i in 1:maxiter) {
              if (all(rnew == rold))
                  break
              rold = rnew
              rnew = 0.5 * (rnew + x/rnew)
          }
          rnew
      }
```

Argument Matching

- Because it is not necessary to specify all the arguments to R functions, it is important to be clear about which argument corresponds to which formal parameter of the function.
- This can be done by providing names for the arguments in a function call.
- When names are provided for arguments, they are used in preference to position which matching up formal and actual arguments.

Argument Matching Examples

```
> root(1:10, maxiter = 100)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278

> root(1:10, start = 3)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278

> root(1:10, maxiter = 100, start = 3)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278
```

Argument Matching Rules

The general rule for matching formal and actual arguments is as follows:

1. Use any names provided with the actual arguments to determine the formal arguments associated with the named arguments. Partial matches are acceptable, unless there is an ambiguity.

2. Match the unused actual arguments, in the order given, to any unmatched formal arguments, in the order they appear in the function declaration.

Argument Matching Examples

Using the argument matching rules, it is easy to see that all the following calls to `root` are equivalent.

```
> root(1:3, maxiter = 20)

> root(x = 1:3, maxiter = 20)

> root(maxiter = 20, 1:3)

> root(max = 20, x = 1:3)

> root(20, x = 1:3)
```

Computational Methods

To show that there are choices to be made when creating functions, lets look at methods for computing the factorial function.

$$n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$$

This is the building block of many applications in statistics. For example, the binomial coefficients are defined by:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Computing Factorials by Iteration

A direct implementation of the factorial function is as follows.

```
> factorial =
      function(n) {
          f = 1
          for(i in 1:n)
              f = f * i
          f
      }

> factorial(10)
[1] 3628800
```

Computing Factorials by Recursion

A basic property of factorials is that $n! = n \times (n-1)!$. This can be used as the basis of a computational algorithm.

```
> factorial =
      function(n)
      if (n == 1) 1 else n * factorial(n - 1)

> factorial(10)
[1] 3628800
```

There are limitations however. The call `factorial(1000)` recurses too deeply and the computation fails.

Computing Factorials by Vector Arithmetic

The summary function `prod` can be used to compute factorials.

```
> factorial = function(n) prod(1:n)

> factorial(10)
[1] 3628800
```

The Gamma Function

The gamma function is a special function of mathematics defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt.$$

The gamma function has the following properties:

$$\Gamma(x+1) = x\Gamma(x),$$

$$\Gamma(1) = 1.$$

This means that

$$n! = \Gamma(n+1).$$

Using $\Gamma(x)$ to Compute Factorials

There are many published algorithms for the gamma function. One of these is built into R and can be invoked through the function `gamma`. This makes it possible to compute factorials as follows.

```
> factorial =
      function(n) gamma(n+1)

> factorial(1:10)
 [1]       1       2       6      24     120
 [6]     720    5040   40320  362880 3628800
```

Of all the algorithms present for computing factorials, this is the best one, because it is the most efficient and it is vectorised.

Factorials and Binomial Coefficients

The factorial function $n!$ is defined in R by

```
factorial = function(n) gamma(n + 1)
```

The binomial coefficients

$$ {}^{n}\mathrm{C}_{k} = \binom{n}{k} = \frac{n!}{k!(n-k)!} $$

can be defined in terms of the `gamma` function.

```
choose =
    function(n, k)
    factorial(n)/
        (factorial(k) * factorial(n - k))
```

Binomial Coefficient Example

Question: This class has 19 (official) members. There are 11 females and 8 males. If the class decides to send a delegation of 2 female and 2 male students to the mental health authorities in order to have the instructor committed to a mental institution, how many ways are there of selecting the delegation?

Answer: There are ${}^{11}C_2$ ways of choosing the females and ${}^{8}C_2$ ways of choosing the males. These choices do not interfere with each other so the total number of choices is ${}^{11}C_2 \times {}^{8}C_2$. Using R we can compute this as follows:

```
> choose(11,2) * choose(8,2)
[1] 1540
```

The Binomial Distribution

Suppose we have an experiment which has a chance of *success* equal to p. If we repeat the experiment n times, the chance of obtaining k successes is

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

The distribution of the number of successes is said to have a *binomial distribution* with parameters n and p.

Statistics text books usually have tables of the binomial distribution giving individual and cumulative probabilities for selected values of n and p. With a computational tool like R, these tables are now obsolete.

Individual Binomial Probabilities

It easy to define an R function which produces individual binomial probabilities.

```
# The chance of k successes in n trials
# with chance of success p.

binp =
  function(k, n, p)
  choose(n, k) * p^k * (1 - p)^(n - k)
```

Having this function makes many statistics homework questions trivial.

Binomial Probability Example

Suppose that a coin is tossed 100 times. We would expect, on average, to get about 50 heads. But how close to 50 is the value likely to be?

Using our binomial probability function we can find out. Let's work out the probability of getting between 45 and 55.

```
> binp(45:55, 100, 1/2)
 [1] 0.04847430 0.05795840 0.06659050 0.07352701
 [5] 0.07802866 0.07958924 0.07802866 0.07352701
 [9] 0.06659050 0.05795840 0.04847430

> sum(binp(45:55, 100, 1/2))
[1] 0.728747
```

There's about a 73% chance.

Binomial Probability Example Continued

What is the chance of getting between 40 and 60 heads in 100 tosses?

```
> sum(binp(40:60, 100, 1/2))
[1] 0.9647998
```

What is the chance of getting between 35 and 65 heads in 100 tosses?

```
> sum(binp(35:65, 100, 1/2))
[1] 0.9982101
```

A Coin Tossing Experiment

Suppose that we choose a value randomly between 0 and 1. There is a 50% chance that such a number will be less than .5. The R function `runif` generates random numbers between 0 and 1. We can used this as a way of carrying out coin tossing.

```
> runif(10) < .5
 [1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
 [8]  TRUE  TRUE FALSE
```

We can take `TRUE` to indicate a head and `FALSE` to indicate tails. We can count the number of heads in 100 tosses as follows:

```
> sum(runif(100) < .5)
[1] 43
```

Coin Tossing Continued

Now that we can easily get the number of heads in 100 tosses, let's repeat the process 10000 times and see what proportion of values fall between 45 and 55, etc.

```
> nheads = numeric(10000)
> for(i in 1:10000)
      nheads[i] = sum(runif(100) < .5)
> sum(45 <= nheads & nheads <= 55)/10000
[1] 0.7298
> sum(40 <= nheads & nheads <= 60)/10000
[1] 0.9624
> sum(35 <= nheads & nheads <= 65)/10000
[1] 0.998
```

These are close to the probabilities 0.728747, 0.9647998 and 0.99821.

Packaging the Experiment

If we we really interested in studying this kind of experiment in detail, it would be useful to package up the R statements in a function.

```
cointoss =
    function(ntosses, pheads, nreps = 1) {
        nheads = numeric(nreps)
        for(i in 1:nreps)
            nheads[i] = sum(runif(100) < pheads)
        nheads
    }
```

Running the Packaged Experiment

With the whole experiment packaged in this function, we can forget the details of what is happening in the function and just use it.

```
> x = cointoss(100, .5, 10000)

> sum(45 <= x & x <= 55)/length(x)
[1] 0.7341
```

What we've done here is to package the whole coin tossing experiment as a *black box*, whose internal details we don't really need to know about.

This process of hiding detail within black boxes is know in software engineering as *black-box abstraction*.

Statistical Computations

The formula for the standard two sample t statistic is

$$t = \frac{\bar{y}_1 - \bar{y}_2}{s_p \sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}},$$

where s_p is the pooled estimate of the standard deviation defined by

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}.$$

Here n_1, $\bar{y}_1$ and s_1 are the sample size, mean and standard deviation of the first sample and n_2, $\bar{y}_2$ and s_2 are the sample size, mean and standard deviation for the second sample.

Key Quantities in the Computation

The two-sample t-test can be thought of comparing the difference of the two sample means

$$\bar{y}_1 - \bar{y}_2$$

to its standard error

$$se = s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}},$$

where

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}.$$

The computation of these quantities can be thought of sub-tasks which must be carried out so that the t-statistic can be computed.

Implementing the *t* Test in R

We'll assume that we are given the two samples in vectors `y1` and `y2`. The sub-tasks can then be implemented as follows.

```
diff.means =
    function(y1, y2) mean(y1) - mean(y2)

pooled.se =
    function(y1, y2) {
        n1 = length(y1)
        n2 = length(y2)
        sqrt((((n1 - 1) * var(y1) +
               (n2 - 1) * var(y2)) /
               (n1 + n2 - 2)) * (1/n1 + 1/n2))
    }
```

Computing the t-Statistic

Because we have code which computes the required components of the t-statistic it is now very easy to compute the statistic itself.

```
tval = diff.means(y1, y2) /
       pooled.se(y1, y2)
```

By itself, the t-statistic is of little interest. We also need to compute the p-value.

Computing p Values

The p-value for the t statistic is the probability of observing a value as extreme as `tval` or more extreme for a $t_{n_1+n_2-2}$ distribution. This can be computed using one of the built-in distribution functions in R. For a two-sided test the p-value is

$$2P(T_{n_1+n_2-2} < -|\texttt{tval}|).$$

This can be computed with the R statement

```
pval = 2 * pt(- abs(tval), n1 + n2 - 2)
```

The Complete R Function

Here is the complete R function for the two-sided *t* test.

```
ttest =
    function(y1, y2)
    {
        tval = diff.means(y1, y2) /
               pooled.se(y1, y2)
        pval = 2 * pt(- abs(tval), n1 + n2 - 2)
        list(t = tval, df = n1 + n2 - 2,
             pval = pval)
    }
```

Using The *t*-Test Function

Here is a very simple application of the *t*-test.

```
> y1 = rnorm(10)
> y2 = rnorm(10) + 2

> ttest(y1, y2)
$t
[1] -4.98341

$df
[1] 18

$pval
[1] 9.62368e-05
```

Rational Approximations

Most people are familiar with the approximation

$$\pi \approx 22/7.$$

In fact the approximation is not very good, and gives only 3 correct digits of accuracy.

It is very easy to derive much better approximations than this.

We'll do this via *continued fractions*.

Continued Fractions

- A construction of the form

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}$$

  is called a *continued fraction*.

- Continued fractions have an important role in statistical computation, particularly for computing a number of important distribution functions.

- We will use a very simple class of continued fraction to derive rational approximations.

Creating a Continued Fraction

Split the number into integer and fractional parts.

$$3 + 0.14159\ldots$$

Next, take the reciprocal of the decimal part.

$$3 + \frac{1}{7.06251\ldots}$$

Repeat these steps.

$$3 + \frac{1}{7 + 0.06251\ldots}$$

$$3 + \frac{1}{7 + \dfrac{1}{15.99659\ldots}}$$

Continued Fractions (Cont...)

With decimal parts greater than 0.5, you can go down from above rather than up from below:

$$3+\cfrac{1}{7+\cfrac{1}{16+\cfrac{1}{-293.63459\ldots}}}$$

The process can obviously be continued indefinitely.

Continued Fractions (Cont...)

To produce a rational approximation, truncate the continued fraction and simplify, starting from the bottom and working upwards. We truncate the above fraction and rewrite in one line as follows:

$$3 + \frac{1}{7 + \frac{1}{16}} = 3 + \frac{1}{\frac{113}{16}}$$

$$= 3 + \frac{16}{113}$$

$$= \frac{355}{113}$$

This approximates π accurately to 7 digits.

Problem Analysis

We can divide this problem into two parts:

- deriving the continued fraction, and
- producing the rational approximation.

Dividing the problem in this way means that we can concentrate on the two parts separately. Writing the code for each sub-task is easy.

Deriving The Continued Fraction

```
> cf.expand =
      function(x, n = 5) {
          cf = numeric(n)
          for(i in 1:n) {
              cf[i] = round(x)
              delta = x - cf[i]
              x = 1/delta
          }
          cf
      }

> cf.expand(pi, 7)
[1]    3    7   16 -294    3   -4    5
```

Producing the Rational Approximation

Suppose that the truncated continued fraction has been reduced to the state

$$\cfrac{\cdots}{c_{i-1}+\cfrac{1}{c_i+\cfrac{1}{\cfrac{n}{d}}}}.$$

The next step is to reduce this to

$$\cfrac{\cdots}{c_{i-1}+\cfrac{1}{c_i+\cfrac{d}{n}}}.$$

Producing the Rational Approximation

The representation

$$\cfrac{\cdots}{c_{i-1}+\cfrac{1}{c_i+\cfrac{d}{n}}},$$

can then be reduced to

$$\cfrac{\cdots}{c_{i-1}+\cfrac{1}{\cfrac{nc_i+d}{n}}} = \cfrac{\cdots}{c_{i-1}+\cfrac{1}{\cfrac{n'}{d'}}}$$

with $n' = nc_i + d$ and $d' = n$.

Producing the Rational Approximation

This means that the continued fraction can be reduced to a simple ratio as follows.

Start with $N = c_n$ and $D = 1$. then for $i = n-1, \ldots, 1$ set

$$T = N$$
$$N = N * c_i + D$$
$$D = T$$

Producing the Rational Approximation

```
> rat.approx =
      function(x, n) {
          cf = cf.expand(x, n)
          num = cf[n]
          den = 1
          if (n > 1)
              for(j in (n - 1):1) {
                  tmp = num
                  num = cf[j] * tmp + den
                  den = tmp
              }
          if (den > 0) c(num, den)
          else c(-num, -den)
      }
```

Examples

```
> rat.approx(pi, 1)
[1] 3 1
> rat.approx(pi, 2)
[1] 22  7
> rat.approx(pi, 3)
[1] 355 113

> print(pi, digits = 15)
[1] 3.14159265358979
> print(22/7, digits = 15)
[1] 3.14285714285714
> print(355/113, digits = 15)
[1] 3.14159292035398
```

Sample Sizes from Percentages

An advertisement quoted a table showing the percent of IT managers considering the purchase of particular brands of computer. It showed the following table.

Rank	Vendor	Percent
1	A	14.6%
2	B	12.2%
3	C	12.2%
4	D	7.3%
5	E	7.3%

The granularity of this data and the fact that the largest value is exactly twice the smallest one suggests that the sample size was very small. Is it possible to infer what the sample size was from the values in the table?

Problem Analysis

The percentages in the table correspond to fractions f of the form i/n, where n is the sample size. We want to determine n, the denominator of the fractions and the vector of i values corresponding to the values in the table.

Let's assume that the fractions f have been rounded to d decimal places (in this case $d = 3$). In other words, the fractions are accurate to $\pm\varepsilon = 1/2 \times 10^{-d} = .0005$. In symbols

$$(f - \varepsilon) \leq \frac{i}{n} \leq (f + \varepsilon)$$

and it follows that

$$n(f - \varepsilon) \leq i \leq n(f + \varepsilon).$$

Algorithm

We can determine the sample size n by searching for the smallest value of n for which it is possible to find i values so that this expression is true for all the f values.

For $n = 1, 2, 3, \ldots$ compute i as the rounded value of $n * f$ and then test to see whether

$$n(f - \varepsilon) \leq i \leq n(f + \varepsilon).$$

Stop when the first such n is found.

The Algorithm as an R Function

```
> find.denom =
      function(f, places) {
          eps = .5 * 10^-places
          n = 1
          repeat {
              i = round(n * f)
              if (all(n * (f - eps) <= i &
                      i <= n * (f + eps)))
                  break
              n = n + 1
          }
          n
      }
```

Results

The fractions corresponding to the percentages 14.6%, 12.2% and 7.3% are: 0.146, 0.122 and 0.073. We can run the search with these fractions as follows.

```
> find.denom(c(.146, .122, .073), 3)
[1] 41
```

The actual i values are:

```
> round(41 * c(.146, .122, .073))
[1] 6 5 3
```

Determining the Number of Digits

The function `find.denom` requires that the number of significant digits be passed as an argument. This argument isn't actually required, as the value can be computed from the fractions themselves.

Carrying out the computations can done using the function `round` which rounds values to a given number of decimal places.

```
> 1/3
[1] 0.3333333
> round(1/3, 4)
[1] 0.3333
```

Determining the Number of Digits

We could test for a specified number of digits as follows:

```
> x = .146
> round(x, 2) == x
[1] FALSE
> round(x, 3) == x
[1] TRUE
```

There is a danger that this test of equality will fail because of roundoff errors (which we'll discuss more later). A better test uses a small numerical threshold.

```
> eps = 1e-7
> abs(x - round(x, 2)) < eps
[1] FALSE
> abs(x - round(x, 3)) < eps
[1] TRUE
```

An R Function

```
> places =
      function(f, eps = 1e-7) {
          places = 0
          repeat {
              if (all(abs(f - round(f, places))
                      < eps))
                  break
              places = places + 1
          }
          places
      }

> places(.146)
[1] 3
```

An Improved `find.denom` Function

```
> find.denom =
      function(f) {
          eps = .5 * 10^-places(f)
          n = 1
          repeat {
              i = round(n * f)
              if (all((f - eps) * n <= i &
                      (f + eps) * n >= i))
                  break
              n = n + 1
          }
          n
      }
```

Improving Performance

In most cases there is very little point in starting the denominator search with `n = 1`. It is possible to start the search at the value

```
n = floor(1/min(f) + eps)
```

which corresponds to `min(f)` of `1/n`. In the example, this would be

```
> floor(1/.073 + .0005)
[1] 13
```

which would correspond to a saving of 25% in the computation time. Even better savings are possible.