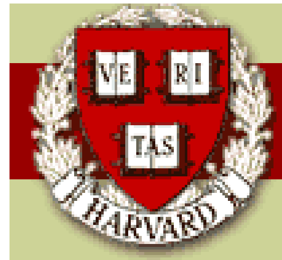
Programming in S - Part II

Statistics 135

Autumn 2005



Copyright ©2005 by Mark E. Irwin

Error Checking

When writing functions, it is usually a good idea to make sure the input arguments are valid. For example, with the factorial functions shown before, all are based on a single, integer being input.

```
facte <- function(x) {
  if (length(x) > 1)
    warning("x should be of length 1, only first component used.")

  if (x[1] <= 0)
    stop("x must be positive.")

  return(gamma(x[1] + 1))
}
```

```
> facte(1:4)
[1] 1

Warning message: x should be of length 1, only first component
used. in: facte(1:4)

> facte(-2)
Error in facte(-2) : x must be positive.
```

The function `warning` will allow the function to continue with the appropriate warning message printed and output returned.

However the function `stop` will terminate the function, yielding no output.

Printing Output with Functions

Sometimes it is useful to print intermediate calculations from within a function, such as during debugging.

For example

```
testprint1 <- function(x) {
  for( i in 1:x)
    c(i, facte(1))
}

> testprint1(4)
>
```

In this case nothing is printed, even though the default action for the c function is to print the value if the output is not assigned to a variable. However while in a function, this will not happen.

To print information from within a function, you must explicitly do it. For example

```
testprint2 <- function(x) {
  for( i in 1:x)
    print(c(i, facte(i)))
}

> testprint2(4)
[1] 1 1
[1] 2 2
[1] 3 6
[1]  4 24
>
```

Note that neither of these functions actually return anything. Usually not a good thing to do, but not always. Doing something like this might be useful for formatted output. For example, doing something like `summary` does with a `lm` object. (Note: that `summary.lm` doesn't work this way, though it could.)

If you wish to format the output, the function `cat` is useful. For example

```
testprint3 <- function(x) {
  for (i in 1:x)
    cat("x = ", i, ", ", i, "! = ", facte(i), "\n", sep="")
}

> testprint3(4)
x = 1, 1! = 1
x = 2, 2! = 2
x = 3, 3! = 6
x = 4, 4! = 24
>
```

Recursive Functions

Another approach that can be useful is that of recursive functions. For example, the factorial function can be written as

$$n! = n \times (n-1)!$$

This can be implemented by

```
factr <- function(n) {
  if (n != trunc(n))
    stop("n is not an integer")

  if (n == 1) factr <- 1
  else factr <- factr(n-1) * n

  factr
}
```

```
> factr(4)
[1] 24

> factr(82)
[1] 4.753643e+122
```

Note that this approach is usually slow and memory-intensive. Each time the function is called, a copy of the important information is made and passed onto the new call. In some cases, the recursive approach may not return an answer. Instead something like the following may happen.

```
> factr(83)
Error in factr(n - 1): evaluation nested too deeply: infinite
recursion / options(expressions=)?
```

This sort of problem is more severe in **S-Plus** than **R**. In problem occurs in **S-Plus** with $n = 83$, but doesn't occur in **R** until $n = 1000$ (at least on my laptop).

The recursive approach does have its uses with intrinsically recursive problems. For example how to list all possible subsets of size r from n objects.

```
subsets <- function(n, r, v = 1:n) {
  if (r <= 0) NULL
  else if (r >= n) v[1:n] else
    rbind(cbind(v[1], subsets(n - 1, r - 1, v[-1])),
                         subsets(n - 1, r, v[-1]))
}
```

```
> subsets(4,2)
     [,1] [,2]
[1,]    1    2
[2,]    1    3
[3,]    1    4
[4,]    2    3
[5,]    2    4
[6,]    3    4
```

The idea behind this function is that $n = r$, there is only one possible subset, the whole vector. Otherwise pick one element from from the set. Then you need to look at all subsets with that element combined with subsets of size $r - 1$ from the remaining $n - 1$ elements plus the subsets of size r taken from the other $n - 1$ elements.

Note that this isn't the best way to write a recursive function as if you change the name, the function with break (subsets won't exist anymore). See pages 49-50 is **S** Programming by Venables and Ripley. A better approach uses the `Recall` function.

Vectorized Functions

Standard **S** functions, such as `sin`, `log`, `dnorm`, etc have the property, that if the first argument is a vector, the result is a vector. Note that a similar result will also happen with matrices and higher level arrays.

For example, in **R**, try `iris3`. You'll see that the result is a 3 dimensional array, the same as `iris3`.

When designing your own functions, you should strive to do the same thing. Often it is easy to do, as where possible, you should base your own functions on the built in vectorized functions.

For example, probably the best version of the factorial function you could write is

```
fact <- function(x) gamma(x+1)
```

In fact, **R**s built in function `factorial` does exactly this. Since it uses the built in function `gamma`, all of its built in error checking will be there.

Also it is automatically vectorized. However, here is an example where the structure of the output is the same as the input.

```
fact.vec <- function(x) {
  size <- dim(x)
  fact <- NULL
  for (i in x)
    fact <- c(fact, prod(1:i))
  array(fact, dim=size)
}

> fact.vec(mat)
     [,1] [,2]   [,3]      [,4]
[1,]    1   24   5040   3628800
[2,]    2  120  40320  39916800
[3,]    6  720 362880 479001600
```

```
> factorial(mat)
     [,1] [,2]   [,3]      [,4]
[1,]    1   24   5040   3628800
[2,]    2  120  40320  39916800
[3,]    6  720 362880 479001600
```

Loops vs Vectorized Calculations

Where possible, you generally want to avoid using loops, particularly in **S-Plus**. The situation isn't quite as bad in **R**. The reason for this is similar to why you don't want to write recursive functions.

For example, lets look at the Fisher Iris data, getting summary statistics for the different species and measurements

```
> meanmat <- matrix(0, ncol=3, nrow=4,
+  dimnames = list(c("Sepal L", "Sepal W", "Petal L", "Petal W"),
+      c("Setosa", "Versicolor", "Virginica")))

> for (i in 1:4)
+   for (j in 1:3)
+     meanmat[i,j] <- mean(iris3[,i,j])
```

```
> meanmat
        Setosa Versicolor Virginica
Sepal L  5.006      5.936     6.588
Sepal W  3.428      2.770     2.974
Petal L  1.462      4.260     5.552
Petal W  0.246      1.326     2.026
```

However this can be done much easier with `apply`

```
> apply(iris3, c(2,3), mean)
         Setosa Versicolor Virginica
Sepal L.  5.006      5.936     6.588
Sepal W.  3.428      2.770     2.974
Petal L.  1.462      4.260     5.552
Petal W.  0.246      1.326     2.026
```

The general form of `apply` is

```
apply(X, MARGIN, FUN, ...)
```

```
Arguments:

       X: the array to be used.

  MARGIN: a vector giving the subscripts which the function
          will be applied over. '1' indicates rows, '2' indicates
          columns, 'c(1,2)' indicates rows and columns.

     FUN: the function to be applied. In the case of functions
          like '+', '%*%', etc., the function name must be quoted.

     ...: optional arguments to 'FUN'.
```

If you just want to average for each variable (over observations and species) use the following

```
> apply(iris3, 2, mean)
Sepal L. Sepal W. Petal L. Petal W.
5.843333 3.057333 3.758000 1.199333
```

An example where additional arguments are passed onto the function is

```
> apply(iris3, 2, mean, trim=0.1)
Sepal L. Sepal W. Petal L. Petal W.
5.808333 3.043333 3.760000 1.184167
```

where the 10% trimmed mean of each variable is calculated.

An advantage of vectorized calculations is that they are usually much faster. If there is any looping to be done, it tends to occur in compiled c code, not in interpreted **S** code. While I'm not sure where there are any implementations of **S** that do this, but some processors allow for calculations to be done at the vector level, not the item level, which can be much more efficient. Vectorized calculation are also useful for parallel processing, as various pieces of the calculations can be pass to different processors and reconstructed later. This is harder to do when loops are involved.

Note that for linear computations, such as the mean, using matrix multiplication can be even more efficient. For example, instead of `apply(iris3, c(2,3), mean)`, the following could be used

```
> matrix(rep(1/50,50) %*% matrix(iris3, nrow=50),
+   nrow=4, dimnames = dimnames(iris3)[-1])
         Setosa Versicolor Virginica
Sepal L.  5.006      5.936     6.588
Sepal W.  3.428      2.770     2.974
Petal L.  1.462      4.260     5.552
Petal W.  0.246      1.326     2.026
```

While more efficient, I'll often use `apply`, since it is more readable. Also it may take longer to figure out how to do it more efficiently than what you get in improvement in calculation time.

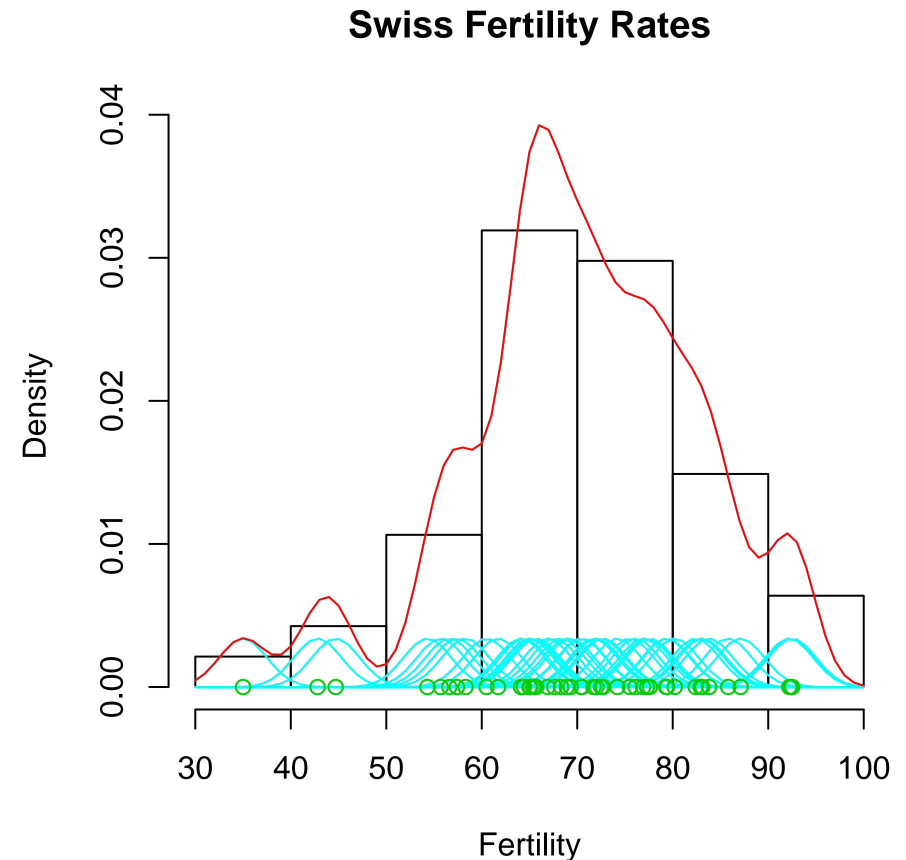
Example: Kernel Density Estimation

One approach to estimating densities of continuous distributions. Suppose you have a data set taking values $x_1, x_2, \ldots, x_n$.

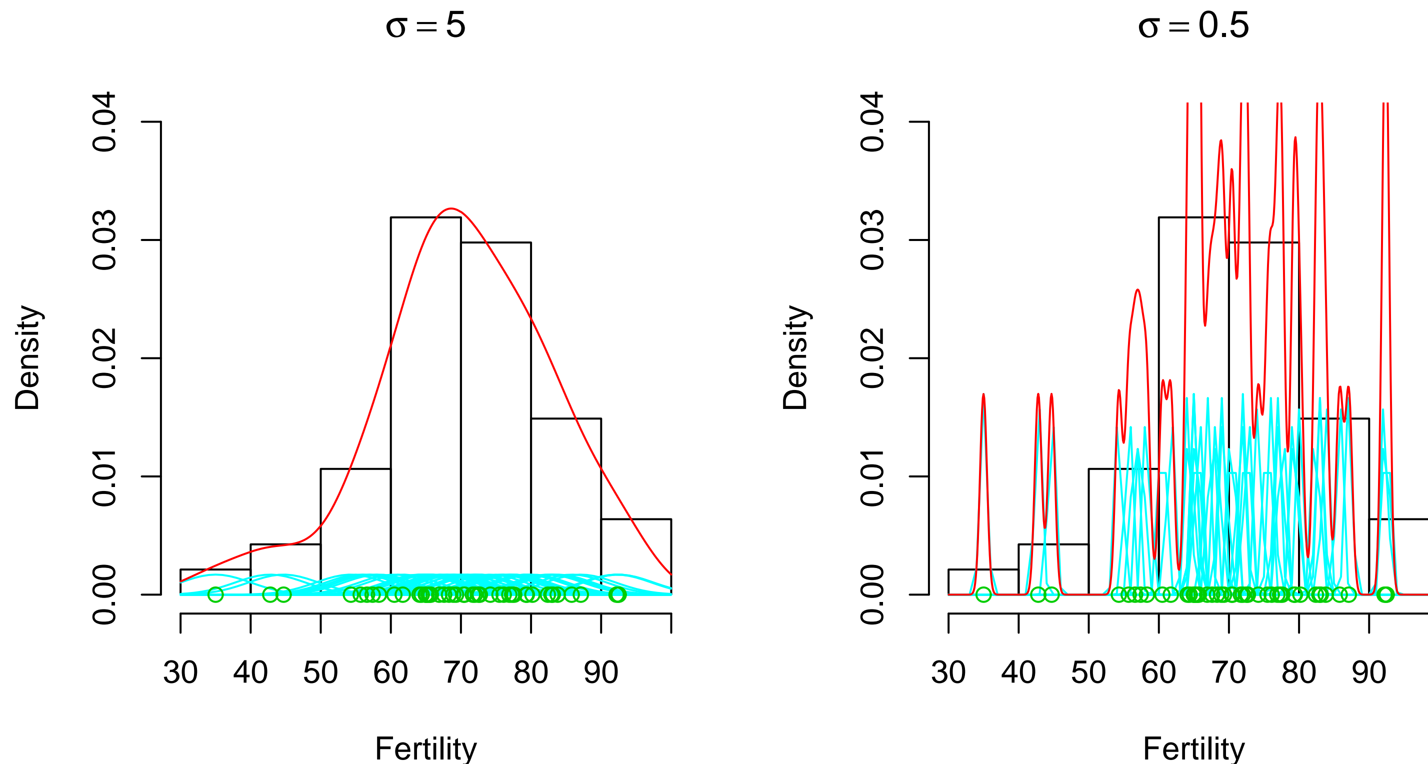
The estimate is of the form

$$\hat{f}_\sigma(y) = \frac{1}{n} \sum_{i=1}^{n} g(y|x_i, \sigma)$$

where g is a density centered at x with standard deviation σ. Usually g is chosen to be Gaussian, though any unimodel symmetric density could be used.



The choice of σ influences the smoothness of the estimates. Small σs will give bumpy estimates and larger ones will give smoother estimates.



There are schemes for automatically picking σ. The built in function `density` has a number of these schemes available.

Lets create functions to implement this using a normal kernel, one with loops and one vectorized. Note that **S** has a built in function `density` that you would normally want to use. For comparing histograms with kdes, in the trellis plotting function histogram, the panel function, `panel.densityplot` will add the kernel density estimate to each panel. As an aside, `panel.mathdensity` can be used to add the density from a parametric distribution, such as the normal.

A version that involves looping is

```
kdeloop <- function(x, data, sigma=1) {
  kde <- rep(0, length(x))
  for (i in 1:length(x))
    kde[i] <- mean(dnorm(x[i], data, sigma))
  kde }
```

This function will calculate the density at each point in `x`, given data in the vector `data` and smoothing parameter `sigma`.

(Note: the originally posted version had an error in the

```
for (i in 1:length(x))
```

line. It was missing the `i:`, which would lead to the estimated density only being calculated at the last component of `x`.)

A similar vectorized function is

```
kdevec <- function(x, data, sigma) {
  xmat <- matrix(rep(x, length(data)), ncol=length(data))
  dmat <- matrix(rep(data, length(x)), nrow=length(x), byrow=T)
  den <- dnorm(xmat,dmat,sigma)
  kde <- apply(den, 1, mean)
  kde
}
```

This version of function creates matrices which allows every entry in `x` to be matched with every entry in `data` (the `xmat` and `dmat` lines). The `den` line evaluates the density for each `x[i]` with the mean set to `data[j]`. Finally the `kde` line, averages the density values for each `x[i]`.

There is an easier way of implementing this idea with the `outer` function. It allows for every combination of `x[i]` and `y[j]` to be evaluated in a function `f(x,y)` and to be stored in `z[i,j]`.

```
outer(X, Y, FUN="*", ...)

Arguments:

       X: A vector or array.

       Y: A vector or array.

     FUN: a function to use on the outer products, it
          may be a quoted string.

     ...: optional arguments to be passed to 'FUN'.
```

The output is an array with dimensions `c(dim(X),dim(Y))`.

Thus the earlier function can be written much more compactly as

```
kdeouter <- function(x, data,sigma) {
  den <- outer(x, data, FUN="dnorm", sd=sigma)
  kde <- apply(den,1,mean)
  list(x=x, y=den, bw=sigma, n=length(data))
}
```

The return of this version is closer to what `density` returns. It is also an example of how to return multiple objects from a function.

Note that all three versions return the same estimated density function, such is a slightly different manner.

Back to `apply` type functions

In addition to `apply`, there are three similar functions for different data structures

- `tapply`: This is useful for data that you want to summarize is in one vector and label information is in one or more additional vectors. For example, to get the sample variances for Sepal Width in the Fisher Iris data for each species

```
> sepalw <- iris[,2]
> species <- iris[,5]
> tapply(sepalw,species,var)
    Setosa Versicolor  Virginica
0.14368980 0.09846939 0.10400408
```

The general form for `tapply` is

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)

Arguments:

       X: an atomic object, typically a vector.

   INDEX: list of factors, each of same length as 'X'.

     FUN: the function to be applied.  In the case of functions
          like '+', '%*%', etc., the function name must be
          quoted.  If 'FUN' is 'NULL', tapply returns a vector
          which can be used to subscript the multi-way array
          'tapply' normally produces.

     ...: optional arguments to 'FUN'.
```

```
simplify: If 'FALSE', 'tapply' always returns an array of mode
          '"list"'.  If 'TRUE' (the default), then if 'FUN'
          always returns a scalar, 'tapply' returns an array
          with the mode of the scalar.
```

It is possible to use your own functions with `tapply` (and the other `apply` like functions). They can be functions you have already created, or they created in the function call.

```
> logrw <- log(crabs$RW)
> sp <- crabs$sp
> sex <- crabs$sex
>
> tapply(logrw, list(sex,sp),
         function(x) {sqrt(var(x)/length(x))})
           B          O
F 0.03001332 0.02374343
M 0.02729349 0.02647192
```

- `lapply` and `sapply`

  These two functions are used with lists (& dataframes). The function `lapply` will return its output as a list, whereas `sapply` will try to return, if possible, a vector. The form of the functions are

```
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)

Arguments:

       X: list or (atomic) vector to be used.

     FUN: the function to be applied to each element of 'X'.
          In the case of functions like '+', '%*%', etc., the
          function name must be quoted.

     ...: optional arguments to 'FUN'.
```

```
simplify: logical; should the result be simplified to a
          vector or matrix if possible?

USE.NAMES: logical; if 'TRUE' and if 'X' is character, use
          'X' as 'names' for the result unless it had names
          already.
```

For example, the we can determine the type of variables in a dataframe with the command

```
> lapply(cars93, class)
$Manu
[1] "factor"

$Model
[1] "factor"

$Type
[1] "factor"
```

```
$MinPrice
[1] "numeric"

$MidPrice
[1] "numeric"

$MaxPrice
[1] "numeric"

$CityMPG
[1] "integer"

and so on
```

An example where both approaches can be used

```
> sapply(cars93, mean)
      Manu      Model       Type   MinPrice   MidPrice
        NA         NA         NA 17.1258065 19.5096774
  MaxPrice    CityMPG    HighMPG    AirBags   DriveTra
21.8989247 22.3655914 29.0860215  0.8064516  0.9354839
```

```
   Cylinder      EngSize        Horse          RPM     EngRevMi
         NA    2.6677419  143.8279570 5280.6451613 2332.2043011
     Manual     FuelTank      Passeng       Length     Wheelbas
  0.6559140   16.6645161    5.0860215  183.2043011  103.9462366
      Width        Uturn     RearSeat      Luggage       Weight
 69.3763441   38.9569892           NA           NA 3072.9032258
   Domestic     HighFuel     CityFuel    Cylinder0     cylinder
         NA    3.5414987    4.6992472           NA           NA
   domestic
  0.5161290
Warning messages: (Deleted)

> lapply(cars93, mean)
$Manu
[1] NA

$Model
[1] NA

$Type
[1] NA
```

```
$MinPrice
[1] 17.12581

$MidPrice
[1] 19.50968

$MaxPrice
[1] 21.89892

$CityMPG
[1] 22.36559

and so on again
```