

A Brief Introduction to Advanced Programming with R

Mikhail Y. Popov
popovm@csu.fullerton.edu
<http://bearloga.net>

February 13, 2011

1 Introduction

There are three sections to this basic tutorial: **Data Manipulation**, **Advanced Custom Functions**, and **Recursion**.

2 Data Manipulation

In this section, I demonstrate how to work with vectors, lists, matrices, and data frames without using a forloop.

2.1 Data

Let us create some data to work with:

```
x = list(a=1:3,b=4:6,c=7:9)
M = matrix(unlist(x),byrow=T,nrow=3)
attach(iris)
data = iris[,c(1,2,5)]
colnames(data) = c("Height","Weight","Gender")
data[,3] = c(rep("Male",75),rep("Female",75))
data[,2] = data[,2]*70
data[,1] = data[,1]
```

Here is what that data looks like:

```
> x
$a
[1] 1 2 3
$b
[1] 4 5 6
$c
[1] 7 8 9

> M
  [,1] [,2] [,3]
[1,]  1   2   3
[2,]  4   5   6
[3,]  7   8   9

> head(data)[1:4,]
  Height Weight Gender
1    5.1   245   Male
```

```

2  4.9  210  Male
3  4.7  224  Male
4  4.6  217  Male

```

We used a built-in dataset `iris` within R to create fake data. Don't do this in actual research.

2.2 FOR-getting the FOR loop

So let's say we wanted to obtain the square root of each element of `x` or `M`. The instinct is to use a `for` loop:

```

# VECTORS WITHIN LISTS
for ( i in 1:length(x) ) {
  for ( j in 1:length(x[[i]]) ) {
    y[[i]][j] = sqrt(x[[i]][j])
  }
}
# MATRICES
for ( i in 1:nrow(M) ) {
  for ( j in 1:ncol(M) ) {
    M[i,j] = sqrt(M[i,j])
  }
}

```

Too many lines and `for`s pretty inefficient. I mean, it gets the job done for the moment but so does Chinese fast food.

2.3 Introducing Awesome Functions

There are a bunch of functions in R that can help you go through each row, each column, each cell, each list member, each vector element in your data. Let me showcase some of them:

2.3.1 `apply`

SUM OF EACH ROW OF M:

```

> apply(M,1,sum) # 1 is for rows
[1] 6 15 24

```

SUM OF EACH COLUMN OF M:

```

> apply(M,2,sum) # 2 is for columns
[1] 12 15 18

```

SQUARE ROOT OF EACH ELEMENT OF M:

```

> apply(M,1:2,sqrt) # MARGIN=1:2 is for rows and columns (a.k.a. each cell)
      [,1] [,2] [,3]
[1,] 1.000000 1.414214 1.732051
[2,] 2.000000 2.236068 2.449490
[3,] 2.645751 2.828427 3.000000

```

```

# SIMILARLY WE CAN USE INLINE FUNCTIONS
# INSTEAD OF BUILT-IN FUNCTIONS
# FOR EXAMPLE IF WE WANT TO SQUARE EACH CELL:

```

```
> apply(M, 1:2, function(x) { return( x^2 ); } )
      [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
```

```
## USING ARGUMENTS ##
```

```
foo = function(x,y) { return(x^y); }
```

```
> apply(M,1:2,foo,y=3) # CUBES EACH ELEMENT OF M
      [,1] [,2] [,3]
[1,]    1    8   27
[2,]   64  125  216
[3,]  343  512  729
```

Suppose we wanted to obtain the cumulative probabilities up to each element of **M** for Poisson($\lambda = 7$):

```
> apply(M,1:2,ppois,lambda=7) -> M1
> M1
      [,1]      [,2]      [,3]
[1,] 0.007295056 0.02963616 0.08176542
[2,] 0.172991608 0.30070828 0.44971106
[3,] 0.598713836 0.72909127 0.83049594
```

We can then reverse the process:

```
> apply(M1,1:2,qpois,lambda=7) -> M2
> M2
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

2.3.2 by

That's pretty cool in theory but in real life we have data frames of observations and categories/groups. Suppose we wanted to find the mean height and mean weight of males and females in our dataset. Check this out:

```
# by(data,INDICES,FUN) applies a function to the data separated by indices
```

```
# MEANS OF HEIGHT AND WEIGHT BY GENDER:
```

```
# by(data[,c(1,2)],data[,3],mean)
```

```
# OR SIMILARLY:
```

```
> by(data[,1:2],data$Gender,mean)
```

```
data$Gender: Female
```

```
  Height    Weight
6.345333 203.280000
```

```
-----
data$Gender: Male
```

```
  Height    Weight
5.341333 224.746667
```

```

# LET'S AUGMENT OUR DATA:

> Orientation = sample(c("Straight","Gay"),150,repl=T,prob=c(.8,.2))
> Age = sample(24:36,150,repl=T)
> data = cbind(data,Age,Orientation)

# LET'S GET MEAN HEIGHT, WEIGHT, AND AGE BY GENDER AND ORIENTATION:

> attach(data)
> by(cbind(Height,Weight,Age),list(Gender,Orientation),mean)
: Female
: Gay
  Height      Weight      Age
6.352632 207.789474 29.368421
-----
: Male
: Gay
  Height      Weight      Age
5.028571 237.000000 32.285714
-----
: Female
: Straight
  Height      Weight      Age
6.342857 201.750000 30.250000
-----
: Male
: Straight
  Height      Weight      Age
5.413115 221.934426 29.508197

```

As you can see, we can specify different columns and different indices (categories).

2.3.3 lapply and sapply

But we don't have to restrict ourselves to matrices and data frames. Sometimes we just have lists we need to work with efficiently. Here's how to do that:

```

# LET'S REMIND OURSELVES WHAT x IS:
> x
$a
[1] 1 2 3
$b
[1] 4 5 6
$c
[1] 7 8 9

# OK, LET'S GET A SQUARE ROOT VERSION OF THAT LIST:
> lapply(x,sqrt)
$a
[1] 1.000000 1.414214 1.732051
$b
[1] 2.000000 2.236068 2.449490
$c
[1] 2.645751 2.828427 3.000000

```

```

# OK, BUT SUPPOSE I DON'T WANT TO OBTAIN A LIST:

> sapply(x,sqrt) # GIVES US A MATRIX
      a      b      c
[1,] 1.000000 2.000000 2.645751
[2,] 1.414214 2.236068 2.828427
[3,] 1.732051 2.449490 3.000000

# LET'S SEE WHAT HAPPENS WHEN OUR FUNCTION IS THE MEAN FUNCTION
# (NOT TO BE CONFUSED WITH A MEAN FUNCTION WHICH HATES PUPPIES)

> lapply(x,mean)
$a
[1] 2
$b
[1] 5
$c
[1] 8

> sapply(x,mean) # GIVES US A VECTOR
a b c
2 5 8

```

Be careful when using these functions. The easy way to remember the difference between `lapply` and `sapply` is that with `lapply`: list goes in, list comes out; with `sapply`: list goes in, matrix OR vector comes out. It depends on what you're doing to the data.

2.4 Replicate

I suppose you could run a loop that goes through 100 iterations of generating 10 random numbers from a Normal distribution with mean $\mu = 2$ and variance $\sigma^2 = 9$. But why would you when you can do it with 1 line:

```
Y = replicate(100,rnorm(100,2,3))
```

This results in a matrix consisting of 10 rows and 100 columns. Thus each column is a single iteration. Can we get 100 means (1 from each "iteration")? Sure! Using our now-familiar friend `apply`:

```
Means = apply(Y,2,mean)
```

And what's the mean of means?

```
> mean(Means) # OBVIOUSLY THIS NUMBER WILL BE DIFFERENT FOR YOU
[1] 2.221885
```

And that wraps up this section! There are still some functions left that I didn't go over because I'm not yet comfortable using them and I don't see myself using them until I truly have to. For the time being, I've got these to play around with.

3 Advanced Custom Functions

In this section, I will walk you through the function `random` I wrote that lets you create sets of data from multiple distributions using only a single line of code like such:

```
> x = random(10,"Normal(0,1);N(10,2);Binom(20,.5)")
> x
[[1]]
 [1]  1.48675007 -2.27607713  1.13517593  0.21180946 -0.97170598  0.48645169
 [7] -0.51838533 -0.02351667 -0.11951048  0.02492427
[[2]]
 [1]  8.922229  7.982272 11.978645 12.253068 11.683184 11.342811  8.418807
 [8] 10.608766  9.476554 10.709307
[[3]]
 [1]  9 10 10 12 14 13 11 11 11  8
```

Distributions with their parameters are entered as a string, separated by semi-colons, and the quantity of random numbers to be generated is given by the first number. Furthermore, we can specify how many numbers we want from the distributions as such:

```
> x = random(c(10,100,20),"Bernoulli(.7);Poisson(8);Dirichlet(.2,.3,.4,.1)")
> x
[[1]]
 [1] 1 1 1 1 1 1 0 0 1 1
[[2]]
 [1]  6 12  7  6 12  9 10  9 11  8  8  2  5  8  3  7 12  7  6  5 10  8  6  9 10
[26]  8  4  5  6  8 10  9 15  7  7 10  9  5  9  4 10  6  9  9  6  5 10  9  6 11
[51]  5  9  7  8 10 10  7  5  9  5  8  6  7 11  9  7  8  6  7 12 11  4  9  5  7
[76]  7 11  8 10  3 14 12  9 10  6  5 10 10  7 12  7  4 11  7  8  9  6  5  9 13
[[3]]
           a1           a2           a3           a4
[1,] 8.198897e-01 1.249000e-01 5.521028e-02 2.648070e-16
[2,] 3.679049e-06 8.569510e-01 1.430186e-01 2.672432e-05
[3,] 5.800351e-03 7.854333e-01 2.087658e-01 6.071883e-07
[4,] 1.019565e-01 8.901994e-01 7.843927e-03 1.724093e-07
[5,] 1.341129e-02 2.728833e-01 7.016185e-01 1.208696e-02
[6,] 2.983929e-01 4.508184e-02 6.345720e-01 2.195332e-02
[7,] 7.855984e-01 4.522924e-02 1.691724e-01 3.058516e-08
[8,] 8.283949e-09 9.844613e-01 1.526385e-02 2.748382e-04
[9,] 1.880763e-01 3.809027e-01 2.818641e-05 4.309927e-01
[10,] 1.181605e-01 1.861500e-02 4.027483e-01 4.604762e-01
[11,] 5.635252e-01 2.033987e-01 2.198002e-01 1.327584e-02
[12,] 8.577659e-02 6.358846e-01 1.413768e-01 1.369621e-01
[13,] 9.903787e-02 6.346955e-01 1.660398e-01 1.002268e-01
[14,] 2.046179e-01 7.776395e-01 1.774215e-02 4.582586e-07
[15,] 9.691443e-01 8.786704e-06 3.084694e-02 1.439377e-10
[16,] 1.588442e-01 5.746762e-01 2.530209e-01 1.345871e-02
[... ] ...           ...           ...           ...
[20,] 3.050480e-06 9.148964e-01 8.510044e-02 6.701921e-08
```

There are quite a few advanced concepts involved and the creation of this function was truly a journey for me. As I guide you through its code step-by-step, keep in mind that the code did not just magically write itself. I had to think hard about what I wanted to accomplish and then research how I can accomplish it, learning a lot of interesting concepts along the way. Also the next few subsections are totally going to be named after movies. Why? Why not!


```

if ( length(grep(pattern="Unif",x=d)) > 0 ) { return(runif); }
else if ( length(grep(pattern="Mult",x=d)) > 0 ) { return(rmultinom); }
else if ( d == "Cauchy" ) { return(rcauchy); }
else if ( length(grep(pattern="Chi",x=d)) > 0 ) { return(rchisq); }
else if ( d == "t" ) { return(rt); }
else if ( length(grep(pattern="Weib",x=d)) > 0 ) { return(rweibull); }
else if ( d == "F" ) { return(rf); }
else if ( d == "U" ) { return(runif); }
else if ( length(grep(pattern="Exp",x=d)) > 0 ) { return(rexp); }
else if ( length(grep(pattern="Pois",x=d)) > 0 ) { return(rpois); }
else if ( d == "Gamma" ) { return(rgamma); }
else if ( d == "Beta" ) { return(rbeta); }
else if ( length(grep(pattern="Norm",x=d)) > 0 ) { return(rnorm); }
else if ( d == "N" ) { return(rnorm); }

# BECAUSE GENERATING n BERNOUILLIS IS EQUIVALENT TO GENERATING
# n BINOMIALS WITH SIZE 1, WE MAKE A BERNOUILLI FUNCTION

else if ( length(grep(pattern="Berno",x=d)) > 0 ) {
  rbern = function(n,prob) {
    return(rbinom(n,1,prob))
  }
  return(rbern)
}

```

Thus, `foo("Normal")(10)` is equivalent to `rnorm(10)`. For another example of a function returning a function, let us write:

```

f = function() {
  return( function(x) { return(x^2); } )
}

square = f() # SINCE f RETURNS A SQUARING FUNCTION
square(5) # GIVES US 25

# WE CAN ALSO JUST USE f
f()(5) # GIVES US 25

```

Anyway, let us continue with the `foo` function:

```

else if ( length(grep(pattern="Binom",x=d)) > 0 ) {
  if ( length(grep(pattern="Neg",x=d)) > 0 ) { return(rnbinom); }
  else { return(rbinom); }
} else if ( length(grep(pattern="Log",x=d)) > 0 ) { # Lognormal
  rlognorm = function(n,mu,sigma) return(exp(rnorm(n,mu,sigma)));
  return(rlognorm)
} else if ( length(grep(pattern="Dirich",x=d)) > 0 ) { # Dirichlet
  dirichlet = function(a) { # a single dirichlet vector
    Y = numeric(length(a))
    for ( g in 1:length(Y) ) {
      Y[g] = rgamma(1,a[g],1)
    }
    return(Y/sum(Y))
  }
}

```

```

    rdirichlet = function(n,...) { # a is a vector of probabilities that sum to 1
      a = as.numeric(list(...))
      df = t(rbind(replicate(n,dirichlet(a))))
      colnames(df) = paste(rep("a",length(a)),1:length(a),sep="")
      return(df)
    }
  }
  return(rdirichlet)
}

```

There's really nothing spectacular until we get to the dirichlet condition. Let's talk about that one for a bit. First, `dirichlet` accepts the vector \vec{a} of some length k (for example) so that $\vec{a} = (a_1, \dots, a_k)$ such that $\sum_{i=1}^k a_i = 1$. It uses an algorithm for generating a single observation vector of length k . We then define a `rdirichlet` function that accepts a specific argument `n` and an unspecified number of anonymous arguments. That's what `...` means. We can then treat `...` a variable and `as.numeric(list(...))` turns a bunch of arguments p_1, \dots, p_k into a numeric vector of size k which we can use as the vector of probabilities to give to `dirichlet` function.

Since each time we run the `dirichlet` function it outputs a vector of size k and we're replicating it (see Section 2.4) `n` many times, we wish to create a matrix of observations by row-binding each observation vector. For some reason this will give us a $k \times n$ matrix instead of the other way around, so we just transpose it. We want to make it a bit more user friendly and so we give each i -th column the name a_i by using the code I wrote a long time ago for sequential column naming. We then return the matrix/dataframe from the `rdirichlet` function. Just like with `rnorm` and `rbinom` and others, we return the freshly written `rdirichlet`.

3.1.3 Pulp Fiction

Now we are ready to complete the puzzle by putting all the pieces together to make a single awesome whole.

```

  for ( i in 1:k ) {

    # THIS IS WHEN WE REMEMBER THAT WE CAN HAVE A SINGLE REQUEST
    # FOR THE NUMBER OF OBSERVATIONS TO BE GENERATED OR WE CAN
    # HAVE A VECTOR THAT SPECIFIES HOW MANY OBSERVATIONS WE WANT
    # PER EACH DISTRIBUTION LISTED. IN EITHER CASE, THERE WILL
    # BE A LIST parms WHERE THE FIRST ELEMENT IS HOW MANY OBSERVATIONS
    # WE WISH TO CREATE (ALWAYS THE FIRST ARGUMENT IN rnorm, ETC.)
    # AND THEN FOLLOWED BY THE PARAMETERS WHERE EACH PARAMETER IS GOING
    # TO BE AN ELEMENT IN THIS LIST. WHEN OUR DISTRIBUTION IS THE DIRICHLET
    # OUR PARMS IS GOING TO CONSIST OF n, a1, a2, a3, ..., ak

    if ( length(n) > 1 ) parms = c(list(n[i]),parameters[[i]])
    else parms = c(list(n),parameters[[i]])

    # do.call TAKES TWO ARGUMENTS/PARAMETERS: function AND arguments/parameters (as list)
    # TO USE A CONCRETE EXAMPLE, LET US DEFINE x = list(n=100,mean=2,sd=3)
    # WE CAN THEN SAY do.call(rnorm,x) WHICH IS EQUIVALENT TO SAYING
    # rnorm(n=100,mean=2,sd=3). WE'VE SIMPLY SEPARATED THE ARGUMENTS/PARAMETERS
    # FROM THE FUNCTION. ANYWAY, WE POPULATE THE i-th ELEMENT OF THE randoms LIST
    # WITH "n" NUMBER OF OBSERVATIONS GENERATED FROM i-th DISTRIBUTION.

    RANDOMS[[i]] = do.call(foo(d=distributions[i]),parms)
  }
  return(RANDOMS)
}

```

3.2 The Rundown

And that wraps up the function! There are many improvements to be made to make it even smarter. A lot of it has to do with catching any mistakes in the input and notifying the user about those mistakes. Like, what happens when the user forgets to separate distributions with a semi-colon? Errors. What happens when we want to specify a per-distribution number of observations to be generated by the length of `n` doesn't match the number of distributions? Errors. What happens when a distribution is misspelled or isn't recognized by our smart function? Clearly we've programmed it to be pretty darn smart but not SUPER smart. Error catching, however, is beyond the scope of this paper as of the moment (otherwise there would be a section on user interface design and user input, but that's really something I don't know much about and don't want to go into yet). In conclusion, I am a total geeky nerd dorkmaster. For numerous reasons. But this is definitely one of them.

P.S. The full code for the `random` function is on Page 13.

4 Recursion

Who watches the Watchmen?

Recursion is the method of using a function inside of its own definition. The most popular example is the factorial function. We can define it as thus:

$$f(n) = \begin{cases} n \cdot f(n-1) & \text{if } n \in \mathbb{N} \\ 1 & \text{if } n = 0 \end{cases} \quad (1)$$

One may think of recursion as a process of distillation that continues to distill until it can distill no more. It is when this base condition is met that the loop self-terminates. We can use this methodology to resample data in R to form permutations of observations. The full code for the `permute` and `permuter` functions is on Page 15.

4.1 Algorithm

Suppose we have k groups of observations. Each i -th group has size n_i and the total number of observations is $\sum_{i=1}^k n_i = N$. Let us define each i -th group as $\vec{y}_i = (y_{i,1}, \dots, y_{i,n_i})$. Thus we define our data $\vec{y} = (\vec{y}_1, \dots, \vec{y}_k)$.

We first pool all the observations into a single vector of size N and call it Y .

$$Y = (y_{1,1}, y_{1,2}, \dots, y_{1,n_1}, y_{2,1}, \dots, y_{k,n_k}) \quad (2)$$

Then we randomly sample Y without replacement to obtain k new groups where each i -th group has size n_i . Thus we define our new data as $\vec{y}^* = (\vec{y}_1^*, \dots, \vec{y}_k^*)$.

Furthermore, the first iteration of this algorithm yields $\vec{y}^{*,1}$ and the process is repeated B times until we have $\vec{y}^{*,j}$ for all $j \in \{1, \dots, B\}$.

4.2 Implementation

First, we assume that the data we are working with is a list. We begin our recursive function as thus:

```
permute <- function(data) {  
  if ( length(data) > 1 ) {
```

The reason for the condition check will be explained soon. For now, we collect the group sizes into `n.vec` and pool all the observations into a single vector `bucket`:

```

n.vec = as.numeric()
bucket = as.numeric()
for ( i in 1:length(data) ) {
  n.vec[i] = length(data[[i]])
  for ( j in 1:length(data[[i]]) ) {
    bucket = c(bucket,data[[i]][j])
  }
}
N = sum(n.vec)

```

We pick n_1 indices for our first group:

```
chosen = sample(1:N,n.vec[1])
```

And then we gather the remaining observations into a new vector of size $N - n_1$.

```
leftovers = bucket[is.na(match(1:length(bucket),chosen))]
```

Because this is a recursive function, we need to transform the remaining observations into a list that will be accepted by the function. We are merely creating a new dataset consisting of $k - 1$ groups:

```

n.vec[1] = 0
  new.data = list()
  for ( i in 2:length(n.vec) ) {
    start.at = 1 + sum(n.vec[1:i-1])
    end.at = sum(n.vec[1:i])
    new.data[[i-1]] = leftovers[start.at:end.at]
  }

```

This is why we originally conditioned the function on the length of the list of data. While there is more than one group of observations, we are going to keep calling all of the code above and this line:

```
return(c(list(bucket[chosen]),permute(new.data)))
```

And when we reach an irreducible dataset of remaining observations, we simply return itself and in doing so we terminate the loop:

```

  } else {
    return(data)
  }
}

```

To repeat this process B times, we write a function to output a list of those permuted datasets:

```

permuter <- function(data,B=1000) {
  temp = list()
  for ( i in 1:B ) {
    temp[[i]] <- permute(data)
  }
  return(temp)
}

```

4.3 Example

Suppose we have 4 groups of sizes 5, 3, 11, and 3 respectively:

```
data <- list(c(12,3,1,6,19),c(31,14,10),c(1:10,2),100:102)
```

```
> data
```

```
[[1]]
```

```
[1] 12 3 1 6 19
```

```
[[2]]
```

```
[1] 31 14 10
```

```
[[3]]
```

```
[1] 1 2 3 4 5 6 7 8 9 10 2
```

```
[[4]]
```

```
[1] 100 101 102
```

Running the function once we obtain the following dataset:

```
> permute(data)
```

```
[[1]]
```

```
[1] 3 2 3 31 19
```

```
[[2]]
```

```
[1] 7 8 9
```

```
[[3]]
```

```
[1] 2 1 4 101 100 102 10 1 6 12 5
```

```
[[4]]
```

```
[1] 6 14 10
```

Did you know you can also run it on itself?

```
> permute(permute(permute(data)))
```

5 Full Random Code

```
random = function(n,dist) { # dist="Normal(0,1);Exp(5);Binom(6,.9);Dirichlet(.3,.5,.2)"
  x = strsplit(dist,",")
  x = lapply(x[[1]],strsplit,split="[:(,):]")
  k = length(x)
  distributions = character(k)
  parameters = list()
  for ( i in 1:k ) {
    distributions[i] = x[[i]][[1]][1]
    parameters[[i]] = numeric(length(x[[i]][[1]])-1)
    for ( j in 2:length(x[[i]][[1]]) ) { # cycle through parameters
      parameters[[i]][j-1] = as.numeric(x[[i]][[1]][j])
    }
  }
  RANDOMS = list()
  foo = function(d) {
    if ( length(grep(pattern="Unif",x=d)) > 0 ) { return(runif); }
    else if ( length(grep(pattern="Mult",x=d)) > 0 ) { return(rmultinom); }
    else if ( d == "Cauchy" ) { return(rcauchy); }
    else if ( length(grep(pattern="Chi",x=d)) > 0 ) { return(rchisq); }
    else if ( d == "t" ) { return(rt); }
    else if ( length(grep(pattern="Weib",x=d)) > 0 ) { return(rweibull); }
    else if ( d == "F" ) { return(rf); }
    else if ( d == "U" ) { return(runif); }
    else if ( length(grep(pattern="Exp",x=d)) > 0 ) { return(rexp); }
    else if ( length(grep(pattern="Pois",x=d)) > 0 ) { return(rpois); }
    else if ( d == "Gamma" ) { return(rgamma); }
    else if ( d == "Beta" ) { return(rbeta); }
    else if ( length(grep(pattern="Norm",x=d)) > 0 ) { return(rnorm); }
    else if ( d == "N" ) { return(rnorm); }
    else if ( length(grep(pattern="Berno",x=d)) > 0 ) {
      rbern = function(n,prob) {
        return(rbinom(n,1,prob))
      }
      return(rbern)
    } else if ( length(grep(pattern="Binom",x=d)) > 0 ) {
      if ( length(grep(pattern="Neg",x=d)) > 0 ) { return(rnbinom); }
      else { return(rbinom); }
    } else if ( length(grep(pattern="Dirich",x=d)) > 0 ) { # Dirichlet
      dirichlet = function(a) { # a single dirichlet vector
        Y = numeric(length(a))
        for ( g in 1:length(Y) ) {
          Y[g] = rgamma(1,a[g],1)
        }
        return(Y/sum(Y))
      }
      rdirichlet = function(n,...) { # a is a vector of probabilities that sum to 1
        a = as.numeric(list(...))
        df = t(rbind(replicate(n,dirichlet(a))))
        colnames(df) = paste(rep("a",length(a)),1:length(a),sep="")
        return(df)
      }
    }
  }
  return(rdirichlet)
```

```

    } else if ( length(grep(pattern="Log",x=d)) > 0 ) { # Lognormal
      rlognorm = function(n,mu,sigma) return(exp(rnorm(n,mu,sigma)));
      return(rlognorm)
    }
  }
for ( i in 1:k ) {
  if ( length(n) > 1 ) {
    parms = c(list(n[i]),parameters[[i]])
  } else {
    parms = c(list(n),parameters[[i]])
  }
  RANDOMS[[i]] = do.call(foo(d=distributions[i]),parms)
}
return(RANDOMS)
}

```

6 Full Permutation Code

```
permute <- function(data) {
  if ( length(data) > 1 ) {

    n.vec = as.numeric()
    bucket = as.numeric()
    for ( i in 1:length(data) ) {
      n.vec[i] = length(data[[i]])
      for ( j in 1:length(data[[i]]) ) {
        bucket = c(bucket,data[[i]][j])
      }
    }

    N = sum(n.vec)
    chosen = sample(1:N,n.vec[1])
    leftovers = bucket[is.na(match(1:length(bucket),chosen))]

    n.vec[1] = 0
    new.data = list()
    for ( i in 2:length(n.vec) ) {
      start.at = 1 + sum(n.vec[1:i-1])
      end.at = sum(n.vec[1:i])
      new.data[[i-1]] = leftovers[start.at:end.at]
    }

    return(c(list(bucket[chosen]),permute(new.data)))

  }else {
    return(data)
  }
}

permuter <- function(data,B=1000) {
  temp = list()
  for ( i in 1:B ) {
    temp[[i]] <- permute(data)
  }
  return(temp)
}
```