1.11 Functions

Functions are defined and customized within **R** according to the syntax

```
> fname = function(x,y,z, w=w0) {
```

. . . body of function (series of commands, usually assignments)

```
  lastexpr }
```

where 'lastexpr' is an **R** expression — the resulting object created by the function — which can involve variables x,y,z,w as well as 'local' variables defined within the function body and *not* saved in the **R** work-area afterwards. The function arguments can be any **R** objects, including functions and lists, and may be named in the original function specification (which allows you to enter the function-arguments by name in possibly the wrong order). In addition, the use of named arguments allows you to designate a default value for the named argument (i.e., a value which will be assumed if you do not specify it when calling the function). EXAMPLES:

```
> f1 = function(x, c=3) sqrt(1+c*x*x)
> f2 = function(x,g=f1) g(x)
> f1(12)
[1] 20.80865
> f2 = function(x,g=f1) g(x)
> f2(12)
[1] 20.80865
> f2(12,sin)
[1] -0.5365729
> f2(12,function(x) x)
[1] 12
```

Several **very** useful **R** commands do exactly this, with functions as arguments, including **apply**, **sapply**, and **uniroot**. EXAMPLES:

```
> bmat = matrix(runif(40), ncol=8)
> apply(bmat,1, function(z) sqrt(var(z)) )
```

```
> rtlst = uniroot(function(x) x^2-2, c(0,2)  )
> rtlst$root
[1] 1.414213
> rtlst$f.root
[1] -6.855473e-07
```

You can define functions *either* in-line following **R** prompts as above (including copying in of text from separate text-editor windows) *or* by

```
> fn1 = ed(fn1,editor="emacs")    ### or
> fix(fn1)
```

1.11.1 Vectorizing Function Operations

Functions can always be applied to vectors or matrices. For example, one way to calculate sample variances of the rows of a matrix **bmat** is

```
> n = ncol(bmat)
> v1 = rep(1,n)
> avec = ( bmat^2 %*% v1 - (bmat %*% v1)^2/n )/(n-1)
```

Another way to do it is using **apply**, which also allows complete flexibility in the choice of function, which we exploit here by directing **R** to omit missing values before calculating variances:

```
> avec = apply(bmat,1,var, na.rm=T)
```

But functions written for scalars may not always allow vectors to pass through, or may not work as you expect if you are not careful. For example

```
> yfcn1 = function(v, w, Acond, fnam) {
# Take  Acond  to be booleanl; v,w  vectors of same length
        fnam(if(Acond) v else w) }
> yfcn1(pi,pi/2,T,sin)
[1] 1.224606 e-16
```

```
> yfcn1(pi,pi/2,F,sin)
[1] 1

> yfcn1((1:10)*pi/10, rep(c(pi,pi/2),5), cos(1.2^(1:10)*pi) > 0, sin)
[1] 1.224606e-16 1.000000e+00 1.224606e-16 1.000000e+00 1.224606e-16
[6] 1.000000e+00 1.224606e-16 1.000000e+00 1.224606e-16 1.000000e+00
Warning messages:
  Condition has 10 elements: only the first used in: if(Acond) v else w
```

This function performs correctly if **v, w, Acond** are of length 1 but not otherwise ! If we want to allow them all to be vectors, change the function to

```
> yfcn2 = function(v, w, Acond, fnam)  fnam(ifelse(Acond,v,w))

> yfcn2((1:10)*pi/10, rep(c(pi,pi/2),5), [cos(1.2^(1:10)*pi > 0], sin)
[1] 1.224606e-16 1.000000e+00 8.090170e-01 9.510565e-01 1.000000e+00
[6] 1.000000e+00 8.090170e-01 5.877853e-01 1.224606e-16 1.224606e-16
```

1.12 Working with Tables & Arrays

Whether because you are working with categorical data, or because you want to summarize cross-tabulated characteristics of key explanatory variables, it is helpful to summarize quickly cross-tabulations arising from conditions. The primary command for this is **table**:

```
> table(LTdata$Last10./LTdata$Cellct>0.10,ifelse(LTdata$Cellct>100000,
+   "Big",ifelse(LTdata$Cellct>30000,"Med","Sm")),  LTdata$Locale)
, , R
      Big Med Sm
FALSE   4   1  1
 TRUE   5   2  3

, , U
      Big Med Sm
FALSE   8   0  0
 TRUE   7   1  0
> dim(.Last.value)
[1] 2 3 2
```

The output from **table** is an array; numeric or character variables are treated as factors, with 'levels' arising from distinct occurrences appearing as *dimnames*. Note that the same command can be used to provide dimnames in transforming a data-frame like **LTdata** into an array, which can also be useful:

```
> LTarray = array(c(LTdata[,2], LTdata[,2]/LTdata[,3]), dim=c(rep(2,6)),
    dimnames=dimnames(table( rep(c("Cell","LTfrac"), 16), LTdata[[4]],
    LTdata[[5]], LTdata[[6]], LTdata[[7]], LTdata[[8]])))
> unlist(dimnames(LTarray))
 [1] "Cell"   "LTfrac" "O"      "R"      "NW"     "WH"     "MP"     "SP"
 [9] "HI"     "NH"     "R"      "U"
```

Another operation which can be very handy is to collapse a data-frame by combining certain table-values (such as cell-totals). The command is **aggregate** or **aggregate.data.frame**:

```
> LTcomb = aggregate.data.frame(LTdata[,2:3], by=LTdata[,4:7], sum)
> dim(LTcomb)
[1] 16  6
> names(LTcomb)
[1] "Tenure"  "Race"    "NumPer"  "Ethnic"  "Last10." "Cellct"
> LTcomb[1,]
  Tenure Race NumPer Ethnic Last10. Cellct
1      O   NW     MP     HI   60687 897530
> LTdata[1:2,]
  Stratum Last10. Cellct Tenure Race NumPer Ethnic Locale
1       1    7267  94069      O   NW     MP     HI      R
2       2   53420 803461      O   NW     MP     HI      U
```

The output is another data-frame with columns 'Cellct' and 'Last10.', obtained by summing these column entries over all records corresponding to each distinct combined value of the columns in the list 'by'. There is an important option (drop=T) in the argument list which can be used to delete any occurrence combination for the 'by' list which does not actually occur.

What **R** does with the columns in the 'by' list is to treat them as factors and then to create unique combinations by making the factor which is the **interaction** of those columns.

```
> interaction(factor(rep(c(2,5),3)),factor(rep(c("A","B","C"),2)))
[1] 2.A 5.B 2.C 5.A 2.B 5.C
> levels(.Last.value)
[1] "2.A" "5.A" "2.B" "5.B" "2.C" "5.C"
```

In case a specified 'by' list is made up of a few columns (say 3), each of which has a large number of levels (say n_1, n_2, n_3), but for which relatively few of the $n_1 \cdot n_2 \cdot n_3$ factorial combinations actually occur, **R** still creates a factorial array of length $n_1 \cdot n_2 \cdot n_3$ before deleting anything: this wastes time and space and occasionally causes **R** to get stuck. There is an old computer-science trick, called **hash-coding**, which can be used to circumvent this problem. Instead of creating the by-list from separate columns, which we may think of as the first 3 columns of a data-frame **sampfram**, one could create a single factor (in an ordering which has very low probability of being

any different from the lexicographical ordering of the 3 factors) with exactly the same number of distinct levels occurring, as follows:

```
by = list( hashcod = cbind(as.numeric(sampfram[[1]]),
     as.numeric(sampfram[[2]]), as.numeric(sampfram[[3]]) ) %*%
       (runif(ncol(sampfram))*10^(5*(1:ncol(sampfram)))) )
```

The idea here is that the pseudo-random numbers generated via **runif** are given to double precision so that there is virtually no chance of a tie when the numerically indexed factor-levels are linearly combined with pseudo-random coefficients. Moreover, if the **sampfram** columns are of moderate size then there is a very small chance that the lexicographical factor-ordering is modified by this method of re-coding into a single factor.

The weights and the random numbers play different roles here: (i) the weights accomplish a (near-) lexicographical ordering in case the columns being manipulated are numeric with roughly the same range of orders of magnitude; (ii) in case there is no particular desire to maintain the lexicographical ordering, and there are possibly many columns with possibly very different dynamic ranges, the hash-coding trick with or without weights recodes the multiple columns into a single one in such a way that with high probability the number of distinct values in the re-coded column is equal to the number of distinct combinations of values in the columns actually occurring in the columns being combined.

The same trick (with weights) can be used (and this was its original application) in sorting based on multiple categories: the syntax is *order(vec1, vec2, vec3)* e.g. to find the index-permutation based on sorting **vec3** within **vec2** within **vec1**.

1.13 Functions in an Illustrative Simulation

Several aspects of vectorization and organizing **R** computations in terms of functions can be illustrated effectively in terms of a statistically meaningful example, a simulation to show the effect on logistic-regression coefficients of a random intercept coefficient.

1.13.1 *Description of Simulation*

The model which we propose to simulate and analyze repeatedly is the *Logistic regression* model

$$Y_j \sim Binom(30, \frac{e^{\beta_1+\beta_2 X_j}}{1+e^{\beta_1+\beta_2 X_j}}) \quad , \quad j = 1, \ldots, 50$$

The idea is that each Y_j represents a number of positive responses among 30 independent individuals who share a common value of the explanatory variable X, which we take to have *iid* unit-exponential components, and that all 50 groups of 30 share the same coefficients β, which we take to be $(-0.8, 0.3)$. A generalization of this model to include a *random intercept* would allow β_1 to be replaced by a (normally distributed) random variable, consisting of the previous constant value common to all groups of 30 and a random 'error' which is *iid* with one value for each group of 30. We do a 500-iteration simulation for each of two settings: first, with $\beta_1 = -0.8$, and second with $\beta_{1j} \sim \mathcal{N}(-0.8, 0.16)$.

```
> X1 = rexp(50)
> ymat1 = matrix(rbinom(25000,30,plogis(-0.8 + 0.3*X1)),
+      ncol=50, byrow=T)
> ymat2 = matrix(rbinom(25000,30,plogis(-0.8 + 0.3*X1+
+      0.4*rnorm(25000))), ncol=50, byrow=T)
> outmat1 = apply(ymat1,1, function(yrow,xvec)
+       c(glm(cbind(yrow,30-yrow) ~ xvec, family=binomial)$coef,
+       sum(yrow)/1500), xvec=X1)
# Note: could have omitted xvec argument & substituted X1 directly
> dim(outmat1)
[1]   3 500
> outmat1 = t(outmat1)
```

```
> dimnames(outmat1) = list(NULL,c("beta1","beta2","ptest"))
> outmat2 = outmat1
> for(i in 1:500) outmat2[i, 1:2] =
+   glm(cbind(ymat2[i,],30-ymat2[i,]) ~ X1, family=binomial)$coef
> outmat2[,3] = ymat2 %*% rep(1/1500,50)
```

We can compare the estimated coefficients in various ways:

```
> apply(outmat1[,1:2],2,mean)
      beta1     beta2
 -0.7972393 0.2996552
> sqrt(apply(outmat1[,1:2],2,var))
      beta1      beta2
 0.07332403 0.05828725
> c(apply(outmat2[,1:2],2,mean),sqrt(apply(outmat2[,1:2],2,var)))
      beta1    beta2     beta1      beta2
 -0.7692242 0.289693 0.1085505 0.09157609
```

The third components of **outmat1** and **outmat2** were included so that we can check in a coarse general way that the simulated data behaves as expected. For example, we know for the first simulation that the average response rate is given by

$$\int_0^\infty \frac{e^{-0.8+0.3x}}{1+e^{-0.8+0.3x}}\, e^{-x}\, dx$$

so we can check

```
> integrate(function(x) exp(-x)/(1+exp(0.8-0.3*x)), 0, Inf)$integral
[1] 0.3791168                                      ### error < 4.e-07
> c(summary(outmat1[,3]),sqrt(var(outmat1[,3])))
   Min. 1st Qu. Median   Mean 3rd Qu.  Max.
 0.3427   0.364 0.3727 0.3729  0.3807 0.408 0.01202048
> summary(outmat2[,3])                     ### Note more spread out !
   Min. 1st Qu. Median   Mean 3rd Qu. Max.
 0.3313  0.3667 0.3773 0.3773  0.3887 0.43
```

and we could do a similar (but more laborious, because double-) integration to check **outmat2**. We will discuss next time a vectorization-trick to speed up the function-evaluations needed to make double or multiple integrations feasible. One could also provide a simple graphical output to indicate that the group-wise means fall within appropriate confidence-bands:

```
> motif()
> pvec = 30*plogis(-0.8+0.3*X1)
> inds = order(pvec)           ### needed for plotting lines
> plot(pvec, apply(ymat1,2,mean)-pvec,
+ xlab="Theoretical Mean Response for Group",
+ ylab="Observed Groupwise Responses")
> abline(0,0)
> title("Plot of Response Rates vs Theoretical Means & Confidence Bands",
+    cex=1)
> lines(pvec, -qnorm(0.95)*sqrt(pvec*(1-pvec)/15000), lty=2)
> lines(pvec,  qnorm(0.95)*sqrt(pvec*(1-pvec)/15000), lty=5)
```

1.13.2 *Passing Arguments to Functions within Functions*

So far, our simulation was done by looping within the primary **R** *work-frame*. More complicated operations, or simulations which would be re-done for each of several different parameter-settings, might be organized as the result of a custom-defined function, which itself would call other **R** functions. It turns out there are subtleties concerning passing of arguments to functions within functions. Although this material is very hard to understand on a first or second reading of Venables & Ripley (or any other **R** book !), the issue is that within a k-level nested functions all arguments must be identified either from the ordinary (**frame 1**) permanent-frame search-list *or* from the temporary frame (**frame 0**, where **.Options** and **.Devices** reside), *or* from the function-level frame $k + 1$.

```
> Fexamp
function(nstrat=50, nrep=500, np = 30, beta=c(-0.8,0.3), sig=0)
{
 Xv = rexp(nstrat)
 ym = matrix(rbinom(nstrat * nrep, np, plogis(beta[1] +
```

```
        beta[2] * Xv + if(sig > 0) sig * rnorm(nstrat * nrep)
        else 0)), ncol = nstrat, byrow = T)
 outm = t(apply(ym, 1, function(yrow, xvec, nstr0, np0)
        c(glm(cbind(yrow,np0-yrow) ~ xvec, family=binomial)$coef,
        sum(yrow)/(nstr0*np0)), xvec=Xv, nstr0=nstrat, np0=np)
# Note: without this use of 'apply', direct use of Xv,
#      nstrat etc would not have been recognized !!
 dimnames(outm) = list(NULL,c("beta1","beta2","ptest"))
 outm
}
```

When the arguments **xvec=Xv**, etc. were dropped in this function within **apply**, and the arguments **np**, **Xv**, and **nstrat** were placed directly into the body of the function-argument of **apply**, the result was

```
> Fexamp(3,100,30)
Error: Object "np" not found
```

There is another way, even harder to digest from the **R** books and help-files but even more generally applicable than **apply**, to pass arguments to functions within functions. If, within the **Fexamp** function, we had defined **outm** as a *nrep* × *nstrat* matrix and then wanted to fill its rows with a *for*-loop, then we could have written

```
  for(j in 1:nrep)
     outm[j, 1:2] = eval(glm(cbind(ym[j,  ], np - ym[j]) ~ Xv,
               family = binomial)$coef, local = sys.parent(1))
```

This command tells **R** to evaluate the *glm* expression by finding variable-names within the *frame* of variables defined within the function-level from which *glm* was called.

1.14 Managing Longer Runs: BATCH and nohup

There are a few **R** and Unix commands which make the submission of longer simulation-runs more manageable, especially if you are logging on remotely and want to run your **R** in background after you have logged off.

These are: **source** and **BATCH** in **R**, and *nohup* in unix.

We begin by illustrating `source`: suppose that you have little text file called "scomm" containing the lines:

```
> !more scomm
xx = c(1:37, list(labs=c("AB","BC"), dmat=matrix(1:9, ncol=3)))
cat(7 + 53*exp(-4))
```

Then the **R** command `source` can be used to execute these commands within **R** , all together.

```
> source("scomm")
7.97072886110291 >
```

Next, for longer 'batch'-style runs, one can use the **R CMD BATCH** command following a Unix prompt. The source-file plays the same role as a file called by the **source()** command. But a file-name to contain the echoed commands and printed output (if any) must also be given — the 'output' file. We illustrate the command together with the use of the Unix **nohup** command.

```
% nohup R CMD BATCH Sstuff/infil.src Sstuff/outfil &
```

1.15 *Why Vectorize ?*

The simplest way to understand the need to vectorize is to time and compare operations on a large matrix which can be done either by linear algebra or looping:

```
> ymat = matrix(runif(500000), ncol=50)
> unix.time(ysum = apply(ymat,1,sum))
[1] 11.90  2.91 65.00  0.00  0.00
> unix.time(ysum = apply(ymat,2,sum))
[1]  6.04  2.25 39.00  0.00  0.00
> unix.time(ysum = ymat %*% rep(1,50))
[1] 1.1499996 0.8699999 8.0000000 0.0000000 0.0000000
> unix.time(ysum = rep(1,10000) %*% ymat)
[1] 1.1099987 0.8299999 5.0000000 0.0000000 0.0000000
```

The times given are user, system, and elapsed time in seconds to perform the requested **R** expression-evaluation.

1.15.1 *A Trick to Vectorize Function Evaluations*

We already encountered a use for the numerical-integration routine 'integrate'. We were calculating the probability of positive response in the logistic-regression simulation, over all (randomly generated) X values. We can take this further by designing a function to provide, for the random-intercept case of the simulation, the probability of response within a cell as a function of the observed explanatory variable X for that cell. What we want to calculate is

$$\int_{-\infty}^{\infty} plogis(\beta_1 + \beta_2 X + \sigma z)\, e^{-z^2/2}\, \frac{dz}{\sqrt{2\pi}}$$

and we want its value for a possibly long vector of X values, for fixed (β_1, β_2, σ). Here is a function to calculate it:

```
> lgstne
function(a, b , acc = 1e-06)
{
```

```
   bv = if(length(b) < length(a)) rep(b, length(a)) else b
# b  must be either scalar or of same length as  a
   avec = ifelse(a > 12 + 5 * bv, 1 - exp( - a + bv^2/2) +
          exp(-2 * (a - bv^2)), ifelse(a < -12 - 5 * bv,
          exp(a + bv^2/2) - exp(2 * (a + bv^2)), -1))
   ni = (1:length(a))[avec < 0]
   for(i in ni)
          avec[i] = integrate(function(xp,c1,c2) dnorm(xp) *
          plogis(c1 + c2 * xp)  , -5, 5, rel.tol = acc, c1 = a[i],
          c2 = bv[i])$value
   avec  }
```

You can see that in the case where the input argument **a** is a vector, the output of **lgstne** is also a vector, which has been created by a (necessarily slow) for-loop. But the function is very smooth, so we can vectorize the evaluations by using a function defined by **R** as a *smooth.spline object*:

```
> npts = 50
> sig = 0.4
> xv = c( ((-4):(-1)) * 5, -5 + ((1:(npts - 1)) * 10)/npts, (1:4) * 5)
> unix.time(lgsplin = smooth.spline( xv ,
      lgstne(xv, sig), spar = 1e-06, all.knots = T))
[1] 11.83  0.36 15.00  0.00  0.00
> U1 = -0.8 + rexp(500) + 0.3
> unix.time(cat(summary(lgstne(U1,0.4))))
0.3823 0.4514 0.5546 0.5943 0.7023 0.9905  ### This is summary
[1] 110.3600006   0.6099999 112.0000000   0.0000000   0.0000000
> unix.time(cat(summary(predict(lgsplin,U1)$y)))
0.3832 0.4518 0.5542 0.5941 0.7011 0.9907
[1] 0.08999634 0.00000000 0.00000000 0.00000000 0.00000000
```

Smoothing splines are actually piecewise polynomial functions closely approximating (a smoothed version of) the designed input set of points (here, the 50 evaluations of **lgstne**). That is why they are so quick to evaluate, and the **smooth.spline** object allows **R** to *vectorize* evaluation at many different new points.

2 Random-Number Generation & Simulation

We already saw a preliminary example of a small simulation, as an illustration for looping, functions, and the need for vectorization. In the next segment of the course, we discuss at greater length the strategy and implementation of simulations of statistical experiments using **pseudo-random number generators**. This topic includes first of all the algorithms used to generate random numbers in **R** (deterministically !?); second, some of the goodness-of-fit cross-checks which one would make in checking the quality of a new random-number generator and which (in modified and simpler form) it is also good practice to use in checking for the correctness of a simulation; and third, some of the variance-reduction and speedup algorithms which have become part of standard practice in simulating random experiments with a view to calculating probabilities (like type-1 and type-2 errors in hypothesis tests) which are not large.

2.1 Pseudo-Random-Number Generation

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. — John von Neumann (1951)
>
> Anyone who has not seen the above quotation in at least 100 places is probably not very old. — D. V. Pryor (1993)
>
> Random number generators should not be chosen at random. — Donald Knuth (1986)

You can read about pseudo-random number generators in many places. The **R** documentation suggests consulting

> Kennedy, W. J. and Gentle, J. E. (1980). Statistical Computing. Marcel Dekker, New York.
>
> Marsaglia, G. et al. (1973). Random Number Package: "Super-Duper". School of Computer Science, McGill University.

the latter of which is the stated source of the **R** **runif** random-number generator. The old standard reference (which has recently been reincarnated as a paperback) is:

Knuth, D. (1981) Seminumerical Algorithms, 2nd. ed. vol. 2 of The Art of Computer Programming .

and a generally useful book on computational algorithms, including those related to simulations, is

Press, W. et al. (1986) Numerical Recipes: The Art of Scientific Computing. Cambridge Univ. Press.

There are also useful survey articles, such as a 1983 International Statistical Review article by B. Ripley. For fancier theoretical properties, see a 1992 book (with number-theoretic flavor, and some of the most interesting rigorously proved results) by H. Niederreiter. There are many books, bibliographies, and software, both in the campus libraries and online. A "Selected Bibliography of Random Number Generation" can be found on WWWeb at:

http://rainbow.rmii.com/ comscire/QWBIB.html

(A) The simplest and most common random number generators: *Linear-Congruential Generators* (**LCG**'s):

$$x_{n+1} \;=\; a \cdot x_n + b \qquad \bmod m$$

(a is called the *multiplier*, b the *addend*, m the *modulus*, usually close to a word-size, e.g. 2^{32} or $2^{31}-1$). With carefully chosen a, the period will be m if m is a power of 2, $m-1$ if m is prime (and the latter is recommended).

> Multiplicative, congruential generators [i.e. LCG's with addend 0] are adequate to good for many applications. They are not acceptable ... for high-dimensional work. They can be very good if speed is a major consideration. Prime moduli are best. However, moduli of the form 2^m are faster on binary computers. — Anderson (1990)

S. L. Anderson. "Random Number Generators on Vector Supercomputers and Other Advanced Architectures," **SIAM Review**, 32 (2), pp. 221-251, 1990.

An example of a good small multiplier/modulus pair, according to Knuth (1981) and my own many-years' experience, is due to G. Marsaglia:

$$\textit{Modulus} \quad = \quad 2^{32} \,, \qquad \textit{Multiplier} \quad = \quad 69069$$

Authors Park and Miller (S. K. Park and K. W. Miller, "Random Number Generators: Good Ones are Hard to Find," Transactions of the ACM, Nov. 1988) recommend :

$$m \;=\; p \;=\; 2^{31} - 1 \;=\; 2,147,483,647\,, \quad a \;=\; 16,807 \;=\; 7^5\,, \qquad b = 0$$

(Period $= p - 1$). Source code (e.g. in Pascal) for this generator is available as **random.f** on the Net.

A famously bad LCG example is the combination of multiplier 7^5 with $m = 2^{32}$ and $a = 0$: if I recollect correctly, this is the one used in the notorious routine RANDU of an IBM package of subroutines).

(B) What can go badly wrong with linear-congruential RNG's ? The main issue is a number-theoretic property spotted by G. Marsaglia in a famous article ("Random numbers fall mainly in planes", 1968 Proc. Nat. Acad. Sci.): the LCG rule results in sequences which fall along hyperplanes in some number of dimensions at most (but sometimes much less than) $\sqrt{m}$. There are several tests of randomness (mentioned e.g. by Knuth) which test how finely spaced these hyperplanes are (the "lattice test" of Marsaglia, the "spectral test" of Coveyou - Macpherson). More generically, test:

- via chi-square, equidistribution in cells of k-tuples
- empirical d.f.'s of statistics arising in simulations
- serial correlation
- relative frequency properties of various permutations

(C) One important source of problems with dynamical RNG's is too-small periods. Several constructions have been proposed for 'shuffling' RNG's. The idea of shuffling, briefly, is to take two or more RNG's and use them together to 'increase randomness' or at least destroy known periodicity (usually the period of a LCG will be the modulus m or $m - 1$) without introducing

systematic behavior. One idea (the most common *shuffle*) is to use one RNG to indirect-address another', i.e., if x_n and y_n are both at least moderately good RNG's, one can initally fill a buffer of size D with successive x_n values and use successive values $y_n \bmod D$ to choose one; then the one chosen is replaced with the next newly generated element of the x_n sequence. A specific algorithm (coded in Fortran or C, along with comments about it) is given in the *Numerical Resipes* book, and we implement a related shuffle in **R** below.

(D). There are many other sorts of pseudorandom uniform-deviate generators, which we now survey briefly. One generic difficulty with these new methods is that, while the tests performed on them become more and more numerous and more and more sophisticated, they are mostly too complicated to prove anything about mathematically. One author who has done a lot of work on the number theoretic aspects of precise proofs concerning LCG's and some nonlinear congruential generators is H. Niederreiter, whose bibliography can be viewed from the web-page mentioned above.

Marsaglia (1985) studied the class of *Lagged Fibonacci Generators*:

$$x_n \;=\; x_{n-L} \,+\, x_{n-k} \qquad \bmod m \qquad (L > k > 0)$$

Based on extensive tests of their randomness properties, Marsaglia rates this type of generator highly (finding deficiency only in his 'Birthday Spacings test', for small L, k.) See

> G. Marsaglia, A Current View of Random Number Generators, Computer Science and Statistics, The Interface, Elsevier Science Publishers B. V. (North Holland) L. Billard (ed.), 1985

Another variant class is that of *Inversive RNG's*

$$y_{n+1} \;=\; a \cdot (1/y_n) \,+\, b \quad \bmod m$$

where m is again either a prime or a power of 2, and the reciprocal is taken mod m. There are number-theoretic proof-techniques for this which give maximal period (e.g. $m/2$ when m is a power of 2 and $a \equiv 1 \bmod 4,\ b \equiv 2 \bmod 4$) and which bound the maximum discrepancy from uniform (over the whole period of the RNG) of the distribution of k-tuples, e.g. by terms of order of magnitude $(\log m)^k/\sqrt{m}$ when m is prime and a and b are chosen so that the generator has maximal period m.

NB: Analogous, but not quite as positive, results and bounds on discrepancies exist for LCG's (many due to H. Niederreiter surveyed in a 1992 book or 1978 Bulltein-of-AMS article.)

Still another new class of generators is that of *Multiply With Carry RNG's*, illustrated from an email posted by Marsaglia in '94 concerning 'the mother of all RNGs'. The idea is to separate out low and high order digits or bits, e.g. starting with $n_0 = 123456,\ x_0 = 456,\ y_0 = 123$ (called the *carry*). Then calculate $n_1 = 672 * 456 + 123 = 306555$ and return $x_1 = 555,\ y_1 = 306$. The general step of this generator would be

$$n_{k+1} \ = \ 672 * x_k \ + \ y_k$$

where the three low-order digits of the answer n_{k+1} would be defined as the *output* x_{k+1}, and the three high-order digits as the *carry* y_{k+1}. Marsaglia recommends this kind of generator using the low- and high- 16 bits in a 32-bit word, with the 'carefully chosen' multiplier **30903**. But he has many complicated variants of this.

Finally, a very handy class of really long period methods is that of *Generalized Additive Shift Register* or **GASR** RNG's. These methods generate binary digits x_n by recursions

$$x_n \ = \ c_1\, x_{n-1} \ + \ c_2\, x_{n-2} \ + \ \cdots \ + \ c_L\, x_{n-L} \qquad \mod 2$$

where the (binary) coefficients c_k are fixed once and for all and will mostly be 0's. One particular choice studied by Fushimi (1988 *Jour. for Assn. of Computing Machinery*), which also falls in the *Lagged Fibonacci* class described above, is

$$L \ = \ 521\ , \qquad c_{32} \ = \ c_{521} \ = \ 1\ , \qquad c_k = 0\ , k \ \neq \ 32,\ 521$$

This generator has a huge period (2^{251}), has some theory behind it, and seems empirically to pass randomness tests. Therefore we recommend its use as a shuffler of other RNG's.

2.2 Uses of RNG's & Recommended Choices

The most stringent requirements on RNG's arise in Monte-Carlo applications requiring huge simulations, including (i) Statistical Physics, (ii) Mathematical Finance (pricing of exotic financial instruments), (iii) Telecommunications

Queueing Networks, and (iv) Bayesian / Bootstrap / Markov Chain Monte Carlo applications in statistics. Basically all the recent fuss about this topic is because of these applications. For us, the uses in large *statistical* simulations would be most important, together with items (iv). In probability modeling (examples of which are (i)-(iii) above), the main issue is to evaluate an analytically intractable expectation or probability. Other algorithmic uses of RNG's which we will talk about in the course are:

- random re-starts for iterative numerical optimization methods whose quality is sensitive to starting values;
- optimization of incomplete-data likelihoods based on EM or 'imputation' algorithms which 'fill in' or 'impute' missing data repeatedly between successive stages of likelihood maximization.

In each of these latter settings, one would pay a price in rapidity of convergence, but not in wrong answers, if the RNG were not good. There are many other uses of RNG's where one simulates jitter or noise which are not senstitive to moderate failures of randomness.

As to the choice of RNG, the best recommendation — very nicely argued in the *Numerical Recipes* book — is to stick with relatively simple, well-tested algorithms (such as the better LCG's) and shuffle them by a long-period (perhaps less well tested) generator such as the Fushimi GASR. For uses in specifically *statistical* simulations and algorithms, the speed of the RNG is much greater than that of the other steps in the simulation-iterations, so reliable equidistribution and independence properties, including very long period in some applications, are much more important than the highest possible speed.

3 Coding RNG's, Shuffles, & Tests in R

Suppose we want to code a RNG ourselves in **R**, initially an LCG. Let the multiplier, addend, and modulus respectively be (for illustration): $a = 69069$, $b = 17$, $m = 2^{32} = 4294967296$. A fairly slow **R** routine for generating (individual) pseudorandom deviates is

```
> Pseudo                          ### divide by  mm  for  Unif[0,1]
function(xseed, aa, bb, mm)
  (aa * xseed + bb) %% mm
```

It is slow only because it has to be called with for-loops as in the following calling sequence.

```
> longrand = array(data=0,c(900000))
> c(size = object.size(longrand), storage.mode=mode(longrand))
         size storage.mode
    "7200120"    "numeric"
> xseed = trunc(runif(1)*1.e5)
> xseed
[1] 65351
>  unix.time( for (i in 1:900000)
      { xseed = Pseudo(xseed,69069,17,4294967296)
        longrand[i] = xseed/4294967296 } )
[1] 10.53  0.03 10.62  0.00  0.00     ### Just 11 CPU seconds !
```

By comparison, a timing-run to obtain 900000 uniform random numbers via **runif** in **R** took 0.12 second.

How could we parallelize this ? Realizing that the **R** generator is very quick, we could use it to generate a long block of seeds for us, which we will run in parallel.

```
> longrand = array(data=0,c(10000,90))
  xseed = trunc(runif(1000)*1.e7)    ### Now want more seeds
> unix.time(for (i in 1:90)
     { xseed = Pseudo(xseed,69069,17,4294967296)
     longrand[,i] = xseed/4294967296 } )
[1] 0.05 0.00 0.06 0.00 0.00
> summary(c(longrand))      ### c()  to get single vector
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
1.216e-05 2.485e-01 4.986e-01 4.988e-01 7.495e-01 1.000e+00
```

As a further exercise in coding and parallelization, we discuss the implementation in **R** of Fushimi's (1988) GASR RNG. To begin, we contrast the

simplest possible implementation, in **R** function **GASRrngA** below, and then a speeded-up version (generating identical output) which makes limited use of **R**'s vectorization capacities. Contrast the speeds below with the 1/4 second required by **R** to generate $9.e5$ binary digits via: **rbinom(9e5, 1, 0.5)**.

```
> GASRrngA
function(inblk,nnum) {
   outvec = c(as.logical(inblk),rep(F,nnum))
   for (i in 1:nnum) outvec[521+i] = xor(outvec[i],outvec[i+489])
# generates binary string of length nnum from length-521
# input binary string  inblk  of length 521
   outvec[521+(1:nnum)]
}
> unix.time({xrng = GASRrngA(rep(T,521),1.e5)})
1] 1.60  0.01 1.65  0.00  0.00   ## about  1.6 CPU sec for 9e5

> GASRrngB
function(inblk,nnum) {
#  now we generate 32 at a time
   numblk = (nnum+1) %/% 32
   outvec = c(as.logical(inblk),rep(F,32*numblk))
   for (i in 1:numblk) {
        irang = (i-1)*32+(1:32)
        outvec[521+irang] = xor(outvec[irang],
                outvec[489+irang])
   }
   outvec[521+(1:nnum)]
}
> inblk = as.logical(rbinom(521,1,0.5))
> unix.time(GASRrngB(inblk,32*(1+(9.e5%/% 32))))
[1] 1.42 0.05 1.50 0.00 0.00         ### about 1.5 CPU sec
```

In **R** – as opposed to Splus – the speedup due to blocking the random-number generation in this way was postive but small. (It was nearly 10-fold improvement in **Splus**.) The difference is purely due to vectorization and shorter loops (by a factor of 32). But still, assuming that we wanted to use these binary digits to construct Uniform deviates, say to 6-figure decimal

(=20-figure binary) accuracy, we would be generating $9 * 10^5/20 = 450,000$ random numbers in 1.4 seconds, while **runif** generates $9 * 10^5$ in .12 second !

Optional Exercise. (*Extra-Credit, due in 2 weeks.*) Can you find a still better **R** implementation of the same GASR generator ? In the **Splus** versions, by exploiting matrix structures, using an iterated form of the recursion defining the GASR, I managed a 10-fold speedup over **GASRrngB**.

3.1 Shuffling in R

Here is an **R** routine using GASR randomly generated bits computed via **GASRrngB** to shuffle **runif**. Recall that the GASR functions give 0, 1 output, so we combine using binary expansion in order to get random integers uniformly distributed on $1 \ldots 2^{18}$. The idea of maintaining a big block of **runif** deviates to select from is in part to shuffle well but also to allow selection of $2^7 = 128$ at a time with only a very small chance of ever choosing the same one twice before re-filling the array.

```
> Shuffler
function(nnum, shufbits = 7, blkbits = shufbits + 11,
       inblk = rbinom(521, 1, 0.5))
{
## idea of shuffling is to indirect-address the usual
## runif sequence in blocks. For parallel
## implementation, address  nshuf  uniform deviates before
## replacing them.
## ASSUME: blkbits >=5 and  shufunit*blkbits > 521
   blksiz = 2^blkbits
   shufunit = 2^shufbits
   nout = (nnum + shufunit - 1) %/% shufunit
   uniblk = runif(nout * shufunit + blksiz - shufunit)
## Will ultimately waste  blksiz-shufunit  of these deviates.
   pwrs = 2^(0:(blkbits - 1))
   tmpunit = blkbits * shufunit
   outdev = array(0, dim = c(shufunit, nout))
   newblk = uniblk[1:blksiz]
```

```
   ctr = blksiz   ### counts uniform deviates already used
   for(j in 1:nout) {
## Single step consists in assigning & replacing shufunit
## deviates in newblk addressed by row of gasrblk entries
      gasrtmp = GASRrngB(inblk, tmpunit)
      inds = 1 + c(matrix(gasrtmp, ncol = blkbits,
          byrow = T) %*% pwrs)
      inblk = gasrtmp[(tmpunit - 520):tmpunit]
      outdev[, j] = newblk[inds]
      newblk[inds] = uniblk[ctr + (1:shufunit)]
      ctr = ctr+shufunit
   }
   c(outdev)[1:nnum]
}

> xtmp = Shuffler(1.e4,inblk=inblk)
> length(xtmp)
[1] 10000
> summary(xtmp)
      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 0.0001683  0.2473 0.4985 0.5023  0.7576 0.9999
> unix.time(runif(9.e5))
[1] 0.12 0.01 0.13 0.00 0.00
> unix.time(GASRrngB(inblk,9.e5))
[1] 1.28 0.05 1.33 0.00 0.00
> unix.time(Shuffler(9.e5, inblk=inblk))
[1] 25.44  0.10 25.55  0.00  0.00

> summary(xtmp)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
4.191e-08 2.503e-01 5.002e-01 5.002e-01 7.500e-01 1.000e+00
```

3.2 Goodness-of-fit Tests of Randomness

We have mentioned above the important activity of testing randomness of the outputted pseudo-random sequences generated by the many RNG algorithms. A quick version of a goodness of fit test is given in the following **R** function. The chi-squared test of fit to a specified multinomial distribution is used to assess the equidistribution of the non-overlapping K-tuples $(x_n, \ldots, x_{n+K-1})$, $n = 0, K, 2K, \ldots$ by tabulating the counts of these K-tuples falling in sets $A \subset [0,1]^K$ of the form $\prod_{i=1}^{K} [a_i/L, (a_i+1)/L)$ for $a_i \in \{0, 1, \ldots, L-1\}$. Of the arguments used in the following **R** function, *ncoord* corresponds to K and *nquant* corresponds to L.

```
> FitNtupl
function(ncoord, nquant, indata)
{
## assumes block of pseudo-random uniform[0,1)
## numbers in  indata; to be tested for fit based on
## empirically generated contingency-table of nquant
## equal-length intervals in each of ncoord
## consecutive coordinates
   ntup = length(indata) %/% ncoord
   idata = c(matrix(trunc(nquant * indata - 1e-11), ncol =
           ncoord) %*% nquant^(0:(ncoord - 1))) + 1
   cellexp = ntup/(nquant^ncoord)
   cells = table(idata)
   diagind = 1 + (0:(nquant - 1)) * sum(nquant^(0:(ncoord - 1)))
   chistat = sum((cells - cellexp)^2)/cellexp
   diagstat = (sum(cells[diagind]) - nquant * cellexp)^2/(nquant *
           cellexp * (1 - ((cellexp * nquant)/ntup)))
   list(chisq = chistat, pval = 1 - pchisq(chistat,
           nquant^ncoord - 1), diagstat = diagstat, diagPval =
           1-pchisq(diagstat, 1), CountTbl = cells)
}

> FitNtupl(2,4,runif(1.e4))
$chisq:
[1] 14.6432
```

```
$pval:
[1] 0.4774102

$diagstat:
[1] 0.1536

$diagPval:
[1] 0.6951185

$CountTbl:
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
 327 297 288 304 347 288 306 321 312 310 334 316 335 311 315 289
```

A quick glance at the output of **FitNtupl** shows what we want to see in these tests of (K-th order) equidistribution. First, the 5000 nonoverlapping pairs are very evenly distributed in the 16 cells A partitioning the unit square by small squares of side-length $1/4$. The chi-square statistics *chisq* and *diagstat*, respectively to test overall balance and the relative fraction of observations falling along the 4 diagonal cells of the unit square, fall well within the middle range of values for the respective chi-squared distributions with 15 and 1 degrees of freedom.

3.3 Illustration with RANDU

We illustrate by means of a scatter-plot and the Goodness-of-fit function just presented, the terrible behavior of the RANDU LCG random number generator with multiplier 7^5, addend 0, and modulus 2^{32}. First, we do some preprocessing so that we can produce blocks of 32 variates at a time from this generator.

```
> coef32 = numeric(32)
> two32 = 2^32
> sev5 = 7^5
> fac = 1
> for (j in 1:32) {
+     fac = (fac*sev5) %% two32
+     coef32[j] = fac }
```

```
> rm(fac,sev5)

> randublk = numeric(32*320)
> xseed = trunc(runif(1)*two32)
> for (j in 1:320) {
+      xtmp = (coef32 * xseed) %% two32
+      randublk[(j-1)*32+(1:32)] = xtmp/two32
+      xseed = xtmp[32] }
> summary(randublk)
      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 0.0001993   0.255 0.4998 0.5007  0.7488 0.9999
      ### So far, this random-number output looks OK
```

With the 10240 generated deviates, we can get a visual indication of something wrong by means of a scatterplot:

```
> plot(randublk[1:10239],randublk[2:10240],
+  xlab="RANDU[n]", ylab="RANDU[n+1]",  main=
+  "Scatterplot of Consecutive Pairs of RANDU Deviates")
```

The scatterplot in the Figure is an indication of nonuniformity only because it seems to have 'holes', although those holes do seem to be widely (and even randomly) dispersed across the unit square. To confirm this failure of equidistribution more formally, we apply FitNtupl:

```
> unlist(FitNtupl(2,10,randublk))[1:2]
    chisq      pval
 132.5781 0.0136907
```

The chi-quared statistic here had degrees of freedom $10^2 - 1 = 99$. In order to confirm that the significant result found here was not a fluke, we do a larger calculation:

```
> randublk = numeric(32*3200)
> xseed = trunc(runif(1)*two32)
> for (j in 1:3200) {
+      xtmp = (coef32 * xseed) %% two32
```

```
+      randublk[(j-1)*32+(1:32)] = xtmp/two32
+      xseed = xtmp[32] }
> unlist(FitNtupl(2,10,randublk))[1:4]
    chisq pval diagstat    diagPval
 29699.43    0       18 2.20905e-05
```

Thus, not only was the failure of equidistribution in the unit square not a fluke according to the 99 *df* chi-square statistic for multinomial fit, but the statistic for fit to the binomials with probability 0.01 of falling within the diagonal cells is also dramatically larger than can be accounted for by chance fluctuations. However, the fraction of the 51200 nonoverlapping observation-pairs falling along the diagonal was 0.1002148, which is hardly different from the theoretically expected value 0.10, since the standard deviation for a *Binomial*(51200, 0.1) variable divided by 51200 is $\sqrt{(0.1)(0.9)/51200} = 0.001326$.

Exercise. Try shuffling the RANDU generator just described using the **Shuffler R** function, modified so that the initial block **uniblk** of *runif*-generated deviates is instead generated by RANDU. (Note: the functions **Shuffler** and **FitNtupl**, along with the preprocessed vector **coef32** of coefficients defined above to facilitate blockwise generation, are available in the MathNet directory **/usr/local/StatData/SplusCrs/.Data or I can email their text versions to you). You should find that the shuffled random-number generator passes all of the randomness tests you can think of. Try it, using blocks of at least 102400 to make your tests.**

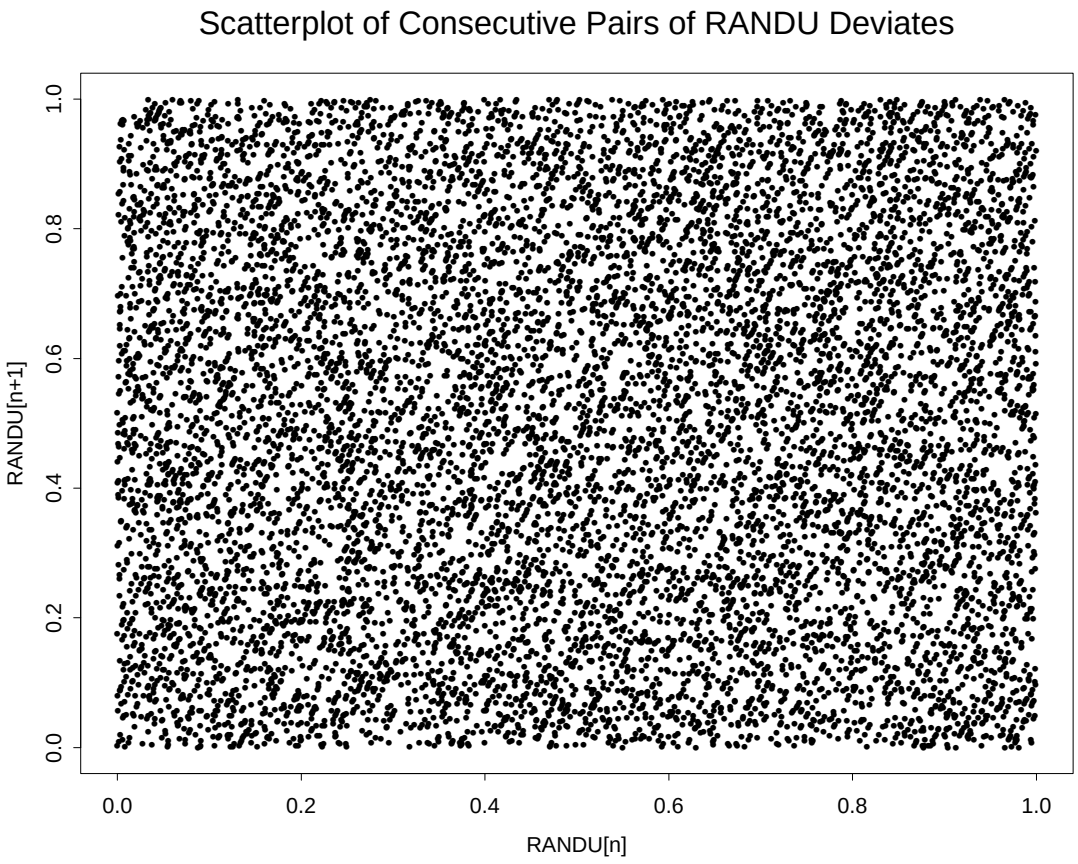

Figure 1: Scatterplot showing joint empirical distribution with 'holes' for consecutive pairs of points produced by the RANDU Linear-Conruential RNG.