Statistical Computing with **R**

Eric Slud, Math. Dept., UMCP

August 30, 2009

Overview of Course

This course was originally developed jointly with Benjamin Kedem and Paul Smith. It consists of modules as indicated on the Course Syllabus. These fall roughly into three main headings:

(A). **R** (& **SAS**) language elements and functionality, including computer-science ideas;

(B). Numerical analysis ideas and implementation of statistical algorithms, primarily in **R**; and

(C). Data analysis and statistical applications of (A)-(B).

The object of the course is to reach a point where students have some facility in generating statistically meaningful models and outputs. Wherever possible, the use of **R** and numerical-analysis concepts is illustrated in the context of analysis of real or simulated data. The assigned homework problems will have the same flavor.

The course formerly introduced **Splus**, where now we emphasize the use of **R**. The syntax is very much the same for the two packages, but **R** costs nothing and by now has much greater capabilities. Also, in past terms **SAS** has been introduced primarily in the context of linear and generalized-linear models, to contrast its treatment of those models with the treatment in **R**. Students in this course have often had a separate and more detailed introduction to **SAS** in some other course, so in the present term we will not present details about **SAS**, in order to leave time for interesting data-analytic topics such as Markov Chain Monte Carlo (`MCMC`) and multi-level modeling in **R**.

Various public datasets will be made available for illustration, homework problems and data analysis projects, as indicated on the course web-page.

The contents of these notes, not all of which are posted currently, and which will be augmented as the term progresses, are:

1. `Introduction to R`
Unix and R preliminaries, R language basics, inputting data, lists and data-frames, factors, functions.

2. `Random Number Generation & Simulation`
Pseudo-random number generators, shuffling, goodness of fit testing.

3. `Graphics`

4. `Simulation Speedup Methods`

5. `Numerical Maximization & Root-finding`
(respectively for log-likelihoods and estimating equations)

5. `Commands for Subsetting`
Manipulating Arrays and Data Frames

6. `Spline Smoothing Methods`

7. `EM Algorithm`

8. `The Bootstrap Idea`

9. `Markov Chain Monte Carlo`
Metropolis and Gibbs Sampling Algorithms
Connvergence Diagnostics for MCMC
Bayesian Data Analysis applications using `WinBugs`

10. `Multi-level Model Data Analysis`
Linear and Generalized Linear Model Fitting and Interpretation

A few Exercises are contained in these notes, but all formal Homework assignments are posted separately in the course web-page Homework directory.

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); secondly, it includes 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 (within `MATLAB` documentation) at:
`www.mathworks.com/access/helpdesk/help/techdoc/math/brrztpq.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 $\bmod\ m$. 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 Bulletin-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

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

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. Most of 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 — well 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. But by now, there are several random number generators implemented in **R** with well-tested good properties. See the help-page for `.Random.seed` for lots of information about the choices, but in all recent versions of **R** the default is the `"Mersenne Twister"` type with `Inversion`.

2.3 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
    "7200112"    "numeric"
> xseed = 65351
  unix.time( for (i in 1:900000)
      { xseed = Pseudo(xseed,69069,17,4294967296)
        longrand[i] = xseed/4294967296 } )
   user  system elapsed
  38.65    0.05   39.92    ### Roughly 40 CPU seconds !
```

By comparison, a timing-run to obtain 900000 uniform random numbers via **runif** in **R** took 0.33 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 } )
   user  system elapsed
   0.13    0.00    0.64   ## Great speedup !
```

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 approximately 1 second required by **R** to generate $9.e5$ binary digits via: `rbinom(9e5, 1, 0.5)`.

```
> GASRrngA =
function(inblk,nnum) {
   outvec = c(inblk,rep(0,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)})
   user  system elapsed
   5.84    0.00    5.94  ## about  6 CPU sec for 1e5

> GASRrngB =
function(inblk,nnum) {
#  now we generate 32 at a time
   numblk = (nnum+1) %/% 32
   outvec = c(inblk,rep(0,32*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 = rbinom(521,1,0.5)
  unix.time(GASRrngB(inblk,32*(1+(9.e5%/% 32))))
   user  system elapsed
   3.78    0.07    4.02  ### about 4 CPU sec for 9e5
```

In **R** – as in **Splus** – the speedup due to blocking the random-number generation in this way was considerable (around a 10-fold improvement). 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 4 seconds, while `runif` generates $9 * 10^5$ in .7 second !

2.4 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 10000
  summary(xtmp)
      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 0.0001683  0.2473 0.4985 0.5023  0.7576 0.9999
> rbind(runif=unix.time(runif(9.e5)),
        GASRrngB=unix.time(GASRrngB(inblk,9.e5)),
        Shuffler=unix.time(Shuffler(9.e5, inblk=inblk)))[,1:3]
         user.self sys.self elapsed
runif         0.13     0.01    0.15
GASRrngB      3.87     0.06    4.19
Shuffler     73.00     0.19   76.75
```

2.5 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.

2.6 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 `/nfs/projects/statData/SplusCrs/Rstf.RData` 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.

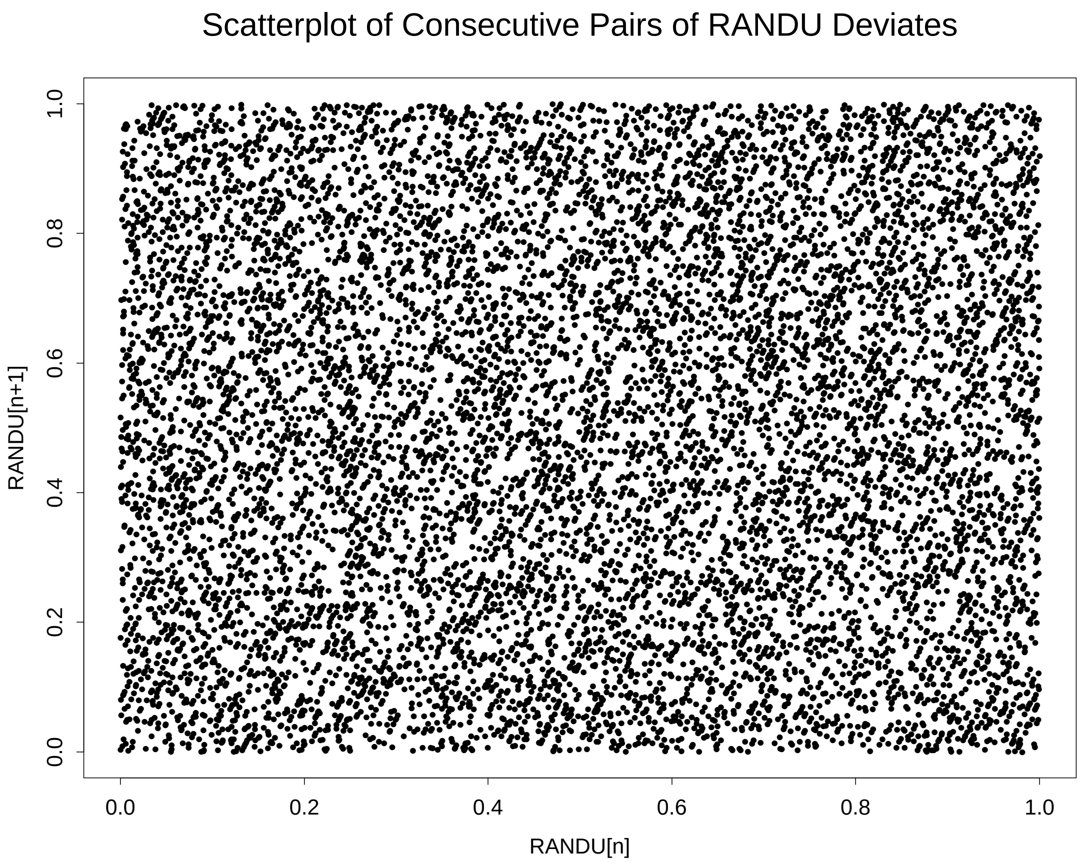

Figure 1: Scatterplot showing joint empirical distribution with 'holes' for consecutive pairs of points produced by the RANDU Linear-Conruential RNG.