NAT'L INST. OF STAND & TECH R.I.C.

A11105 865146

NIST PUBLICATIONS

NISTIR 6569

Fair Selection of Jury Panels From Jury Pools

James L. Blue
U.S. DEPARTMENT OF COMMERCE
Technology Administration
Mathematical and Computational Sciences Division
National Institute of Standards and Technology
Gaithersburg, MD 20899

October 25, 2000

QC
100
.U56
NO.6569
2000
c.2

NIST
National Institute of Standards and Technology
Technology Administration
U.S. Department of Commerce

NISTIR 6569

Fair Selection of Jury Panels From Jury Pools

James L. Blue
U.S. DEPARTMENT OF COMMERCE
Technology Administration
Mathematical and Computational
Sciences Division
National Institute of Standards
and Technology
Gaithersburg, MD 20899

October 25, 2000

U.S. DEPARTMENT OF COMMERCE
Norman Y. Mineta, Secretary

TECHNOLOGY ADMINISTRATION
Dr. Cheryl L. Shavers, Under Secretary
of Commerce for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

Fair Selection of Jury Panels from Jury Pools

James L. Blue
Mathematical and Computational Sciences Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

1. The Problem

A standard problem of courts at all levels is the selection of a panel of potential jurors from a jury pool, which is a list of people who are eligible to serve as jurors.

By some means beyond the scope of this report, we suppose that the pool of eligible jurors has been identified in a court jurisdiction. Let M be the number of people in the pool. In a large jurisdiction, M could be as large as a few million. Frequently, some of the pool members must be chosen as potential jurors, to form a jury panel. Let N be the number to be chosen, where N is less (and usually much less) than M.

The problem is to select N people out of the M possible people in a fair way. One definition (F1) of "fair" is that each of the M people has an equal chance to be selected for any jury pool. A much more rigorous definition (F2) is that each of the possible combinations of N people has an equal chance to be selected as the entire jury pool.

2. History

In 1994, the author received a request from Mr. David Schenken of the Administrative Office of the United States Courts. The request was to produce a specification for a fair algorithm for the problem stated above. The specification was to go into a Request for Proposal (RFP) for a turnkey system for jury selection.

As producing a working computer program incorporating a fair algorithm seemed to be both easier and less ambiguous than producing a specification, the author did the former. The program and a brief testing program, both in the "C" language, are presented in Appendix 1. Two other versions of the program were also produced, but are not included here. One was in a then-common, but substandard, dialect of the C language, and one was in Fortran.

In this report we will describe the properties of this algorithm and present an improved version.

3. What Makes a Fair Algorithm?

We start with an artificially small value of M in order to illustrate. Suppose that the jury pool consists of four members: Chris, Dale, Lee, and Pat.

If N is one, what is a fair algorithm of choosing one out of the four? Clearly each should be equally likely to be chosen, as in fairness definition F1. (This can be done in a fair way with three coin flips. One flip chooses between Chris and Dale, the second chooses between Lee and Pat, and the third chooses between the two winners. If M is not one of the numbers 2, 4, 8, 16, 32, etc., it's not as easy to do this using coins.)

If N is two, life gets more complicated. At the very least, a fair algorithm will still choose each of the four people with equal likelihood (F1). However, this might be done by flipping a coin and choosing Chris and Dale if the coin comes up heads, or Lee and Pat if the coin comes up tails. This procedure satisfies fairness definition F1, but not definition F2.

There are six ways of choosing two out of four: Chris and Dale, Chris and Lee, Chris and Pat, Dale and Lee, Dale and Pat, Lee and Pat. Fairness definition F2 requires that each of the six possibilities be chosen with equal likelihood. One way is to pick randomly one of the integers from one to six, and choose the pair of people corresponding to that integer.

Unfortunately for the purposes of exposition, realistic values of M and N are much larger, and the number of ways to choose N out of M people grows rapidly as M and N get larger. There may be hundreds, thousands, or even millions, of people eligible to be jurors. Of these a much smaller number, perhaps hundreds, are to be chosen. With realistic values of M and N, enumerating the possibilities is not realistic. For example, if M is merely 50 and N is 25, there are 126,410,606,437,752 ways to choose N out of M. This number is much larger than the largest integer that can be stored in a 32-bit computer word.

The number of ways to choose N out of M is equal to the binomial coefficient $\binom{M}{N}$. (See reference 1, pages 828-830, for a table of binomial coefficients for M up to 50 and N up to 25.) The binomial coefficient is defined by $\frac{M!}{(M-N)!N!}$ where the "factorial function," $M!$, is defined as the product of all the integers from M down to 1, $M \times (M-1) \times (M-2) \times \ldots \times 2 \times 1$.

In summary, a fair procedure according to definition F2 will randomly select one of the $\binom{M}{N}$ possible sets of N people, and each possibility will have an equal chance of being selected.

4. Is There a Fair Algorithm?

Fortunately, the equivalent mathematical problem of choosing a random selection of N integers out of the integers from 1 through M has been solved. (See reference 2 and reference 3, pages 136-137, and references therein). An advantage of this algorithm ("Algorithm S") is that it works even if M is so large that the integers from 1 through M cannot be stored in the computer being used. This was useful in the days when even large, expensive computers had little memory. A disadvantage is that the algorithm cannot be shown to be correct except by using mathematics beyond the reach of the intended readers of this report.

Though not easy to prove correct, Algorithm S is easy to state. We look at each of the integers from 1 through M in turn and decide if each should be chosen. At any point in the algorithm, let t be the total number of integers looked at so far and c be the number chosen so far.

S1. Start with $t = 0$ and $c = 0$.
S2. Choose a random number x, uniformly distributed between 0 and 1.
S3. If $(M - t)\, x < (N - c)$, increase t by 1, choose t, and increase c by 1.
S4. Otherwise, increase t by 1, but do not choose t.
S5. If c equals N, we are done.
S6. Otherwise, go to step S2.

The key part of the algorithm is in step S3. If we have already looked at t integers and have chosen c of them, we need to choose $(N - c)$ of the remaining $(M - t)$ integers. Step S3 chooses the next integer with probability $(N\text{-}c)/(M\text{-}t)$.

Thus the procedure for jury panel selection is to assign each of the members of the pool one of the integers from 1 through M, with no duplications. Then use Algorithm S to choose a set of N integers out of the integers from 1 through M. Assigning an integer to a person can be done in any way as long as there are no duplicates. The method used for assigning integers to people will not affect the fairness of the procedure.

There is a simpler algorithm for choosing N of M integers, although it requires that M be small enough that an array of M integers can be held in computer memory. They are then re-arranged in random order and the first N are chosen. (See reference 3, page 139, Algorithm P).

P1. Put the integers 1 through M in a list in positions 1 through M.
P2. Set j to M.
P3. Choose a random number x, uniformly distributed between 0 and 1.
P4. Set k to $1 + xj$, dropping any fractional part.
P5. Exchange the integers in the jth and k^{th} positions.
P6. Decrease j by one. If $j = 1$, go to P7. Otherwise go to P3.
P7. Choose the integers in positions 1 through N.

5. What About Choosing a "Random Number?"

Each of these algorithms requires a method for choosing a random real number between zero and one. True random number generators (RNGs) require a physical random event, such as the decay of an atomic nucleus. Their nature is that their results are not repeatable and that they are difficult to test for randomness; they are rarely used in practice. Instead, we use a computer algorithm that is both repeatable and whose outputs satisfy various properties of randomness. Sometimes such algorithms are called Pseudo-Random Number Generators (PRNGs) to emphasize the difference. These PRNGs exist in profusion. They have been and continue to be the subject of numerous studies. Some examples are references 4, 5, and pages 1-177 of reference 3. PRNGs require a starting value, usually called a "seed," to begin the procedure. They produce a repeatable sequence of random numbers (RNs), different for each seed.

Eventually the sequence starts repeating itself; the distance between repeats is called the "period."

The PRNG used for the program is *uni*, an example of a lagged-Fibonacci generator, which is denoted F(17, 5, –) in reference 4. Programs implementing F(17, 5, -) have been tested extensively and testing has shown that F(17, 5, –) is an excellent algorithm for a PRNG. *Uni* was written in 1981 by the current author and David Kahaner, then of the National Institute of Standards and Technology, and George Marsaglia, then of Washington State University. *Uni* produces 31-bit random numbers between 0 and 1, so it can produce about 2^{31}, or more than 2,000,000,000 different numbers. Its theoretical period can be much longer, as much as approximately $2^{32} * 2^{17}$, or more than 560,000,000,000,000, which is ample for our purposes, as choosing N of M integers requires at most a small multiple of M random numbers. Each of the 2^{31} possible RNs is produced many times in the sequence.

F(17, 5, –) requires an initial set of 17 random numbers. Strictly speaking, PRNGs based on F(17, 5, –) have the above period only "with probability one." An unlucky choice of the initial 17 numbers could result in a bad sequence of RNs, but this is extremely rare. *Uni* uses another PRNG to generate its initial set of 17 random numbers; it is not known under what conditions *uni*'s period is as long as the theoretical maximum, but for the current purpose it is unimportant.

6. Testing the Random Number Generator

Testing a PRNG completely would take a very long time, as all possible aspects of a PRNG would have to be tested for all possible seeds. In fact, it is inherent from the algorithmic nature of PRNGs that every PRNG is guaranteed to fail some tests of randomness. A few tests clearly are always important, and others may be important depending upon the application where the PRNG is to be used. The few tests presented here do not constitute a complete test of *uni*. Rather, they demonstrate that F(17, 5, –), the algorithm behind *uni,* has been programmed correctly and that the RN sequences produced by *uni* have some of the most important properties of truly random sequences. Additional sources of tests for randomness include the Computer Security Division at NIST (see *http://csrc.nist.gov/rng*) and NETLIB (see the software library RANDOM at *http://www.netlib.org*).

Uniformity

The random numbers should be uniform. In the long run, each number between 0 and 1 should be chosen the same number of times. They shouldn't be too uniform, though, as that wouldn't be truly random. Suppose we divide the interval from 0 to 1 into K bins, each of length $1/K$, and also choose an integer n. We choose a seed for the PRNG and generate nK random numbers. Perfect uniformity would mean n numbers would fall into each bin. That is unlikely to happen, however. Let y_k be the number that actually fall into bin k. One useful test of the deviation from uniformity is the chi-square test (see Reference 6, pages 164ff, or any standard textbook on statistics). For this, we use the quantity $V = \sum_{k=1}^{K} (y_k - n)^2 / n$ as the measure of the deviation.

What is a reasonable value for V? This is a well-known problem (see Reference 3, pages 39-45). The answer is that V should have the "chi-square distribution with K-1 degrees of freedom," denoted by $\chi^2(K-1)$. For K larger than 4 or 5, the mean value that V should take on is approximately $K - 5/3$.

If V is too large, the numbers aren't uniform enough. If V is too small, the numbers are too uniform. One such test isn't sufficient, though. Any single test might by chance give an extreme V-value In fact, if this test is done many times with different starting seeds, most of the V's should cluster near $K - 5/3$; occasionally a V should be rather small or rather large, even with a "perfect" series of RNs.

As an example, take K to be 101; some of the values for the chi-square distribution for 100 degrees of freedom are published in reference 1, pages 984-985. Take n to be 100; generate nK = 10,100 random numbers, put them into K bins, and calculate V. For one particular seed, this gave $V = 87.26$, rather below the "expected" value of 99.33. How unusual is this? Rather than answering this question, we calculate many V's and see if they have the right distribution.

Do the above 100 times, starting with a different seed each time, and sort the V's according to size. This gives a reasonable idea of the distribution of the V's. Figure 6.1 shows the result. The line connects the 100 V points and the open diamonds show some theoretical values for the chi-square distribution with 100 degrees of freedom. The fit is approximately right.

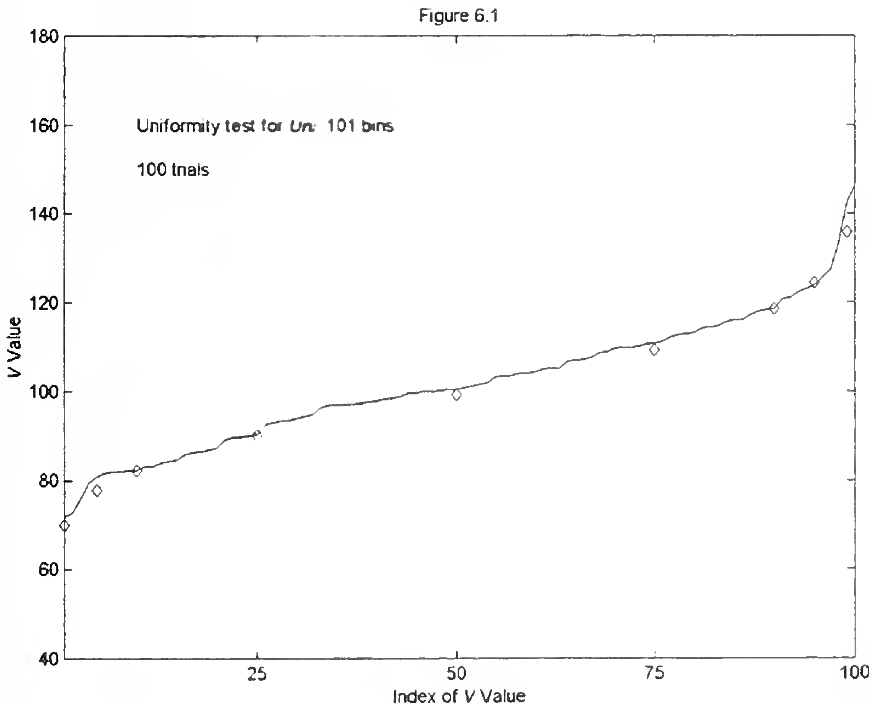


Figure 6.2 is the same, except that 1000 V's were used. The fit is excellent except near the ends.

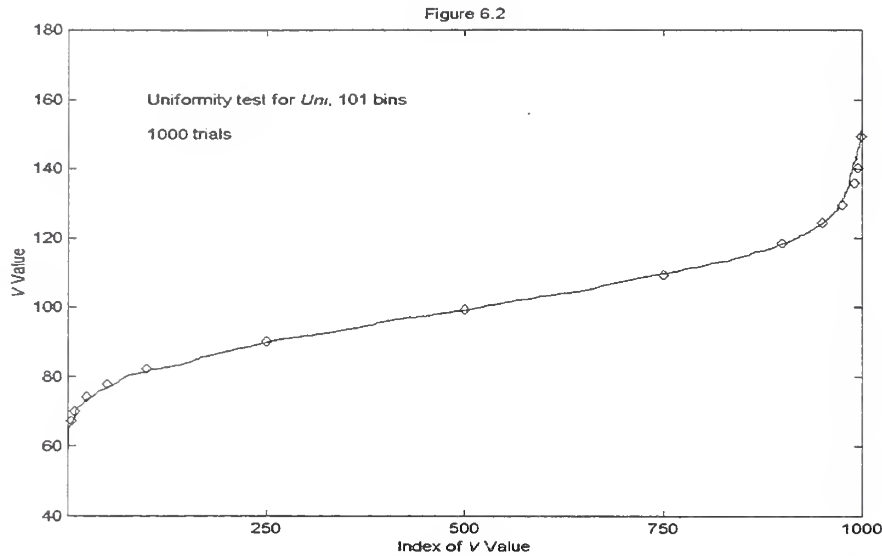
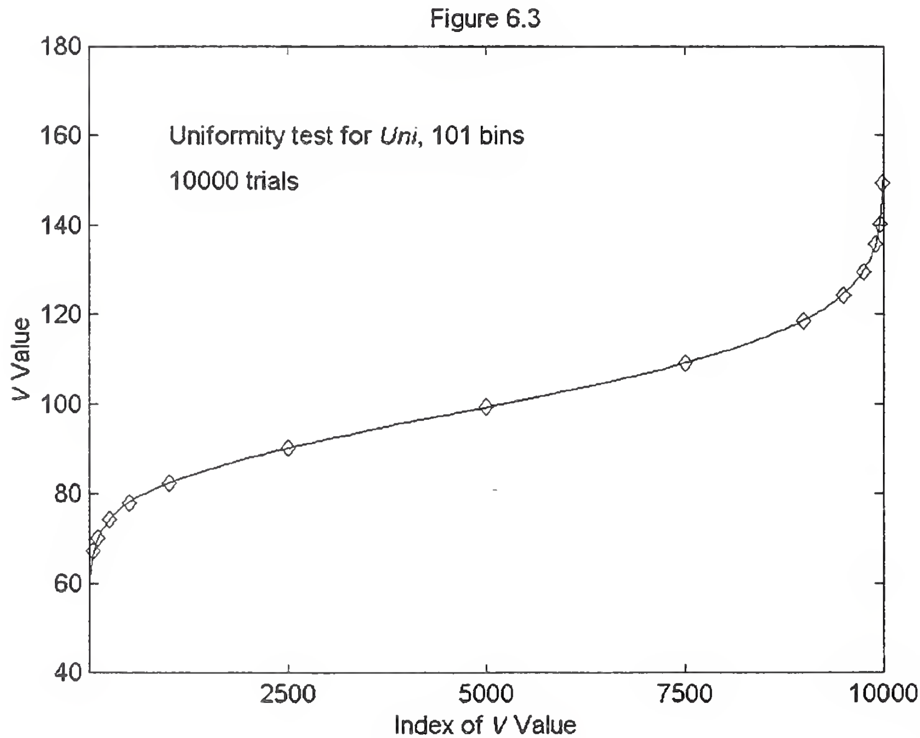


Figure 6.3 is the same, except that 10,000 V's were used. Now the fit is excellent even near the ends.

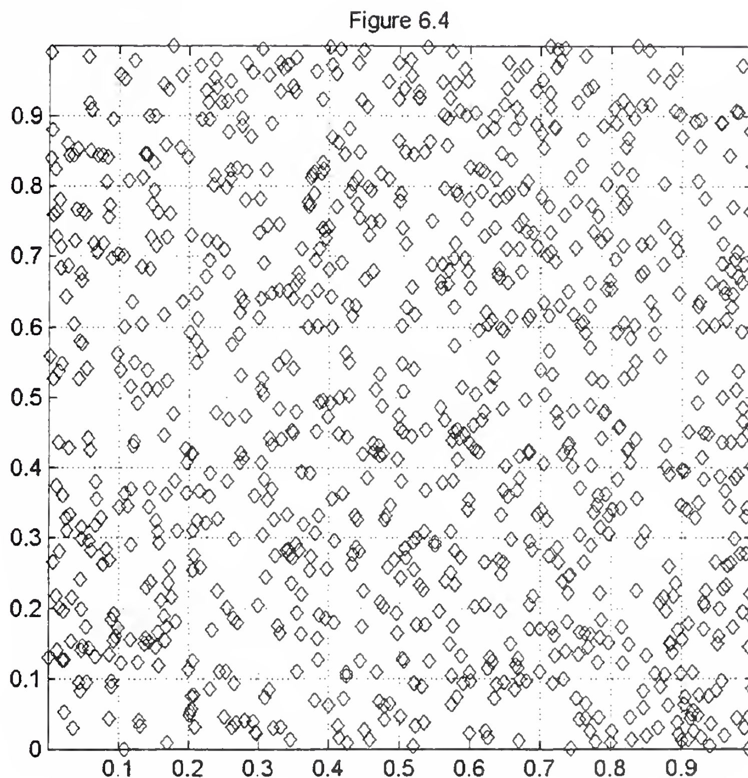


This demonstrates that *uni* behaves properly for this test, for this particular choice of n and K. As there is nothing special about these values, we may conjecture that *uni* behaves properly for this test in general.

Serial Correlation

Even if a PRNG satisfies the chi-square test, it could be poor in other ways. For example, consider successive pairs of random number produced by the PRNG. The second number ought to be independent of the first number, but might not be. For example, a smallish number might tend to be followed by a largish number, and vice versa.

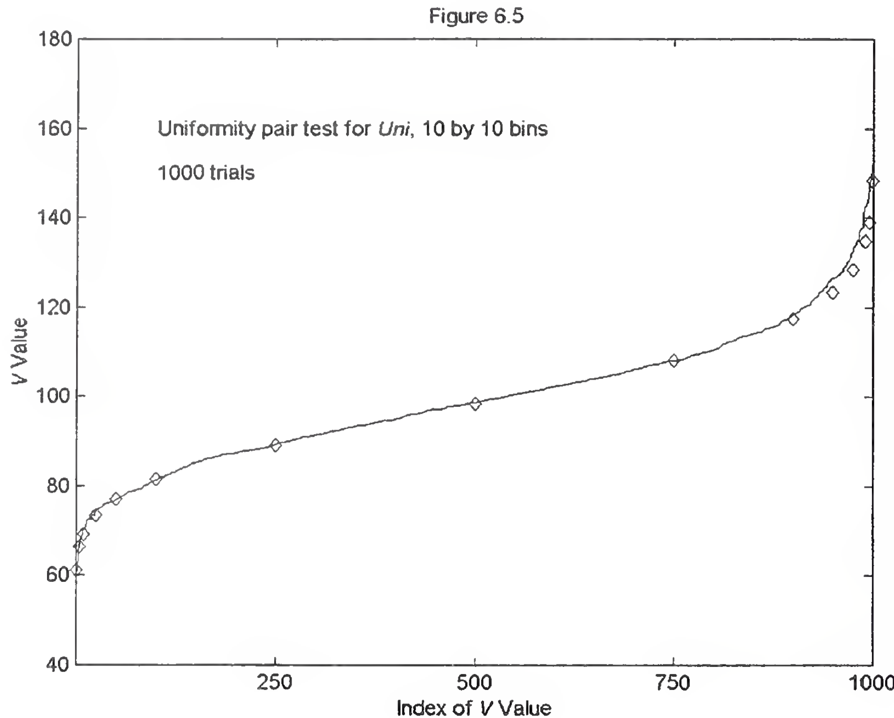
Figure 6.4 shows 1000 consecutive pairs of numbers produced by *uni,* plotted as points in a square. (Point 1 has RN 1 as the horizontal value and RN 2 as the vertical value. Point 2 uses RN 3 and RN 4, etc., through RN 1999 and RN 2000.) The 1000 points should be randomly distributed throughout the square.



This can be checked by using the chi-square test and distribution, as in the previous section. Divide the square into 100 equal-sized boxes; there is an average count per box of 10. For this figure, $V = 102.6$, a bit above the mean of 97.33 expected for a chi-square distribution with 99 degrees of freedom.

Doing the same exercise for 1000 different seeds, sorting the V's, and plotting the results along with some theoretical points for the chi-square distribution with 99 degrees of freedom is shown in Figure 6.5. As the fit is excellent, we may say that *uni* behaves properly for this test. (The theoretical chi-square points were calculated according to the formula on page 41 of reference 3;

this formula is asymptotically correct for large K, and is certainly as accurate as can be seen on the plot for K of 50 or more.)



A full test would divide the square into more and more boxes, and would also look at triples of random numbers, quadruples, quintuples, etc.

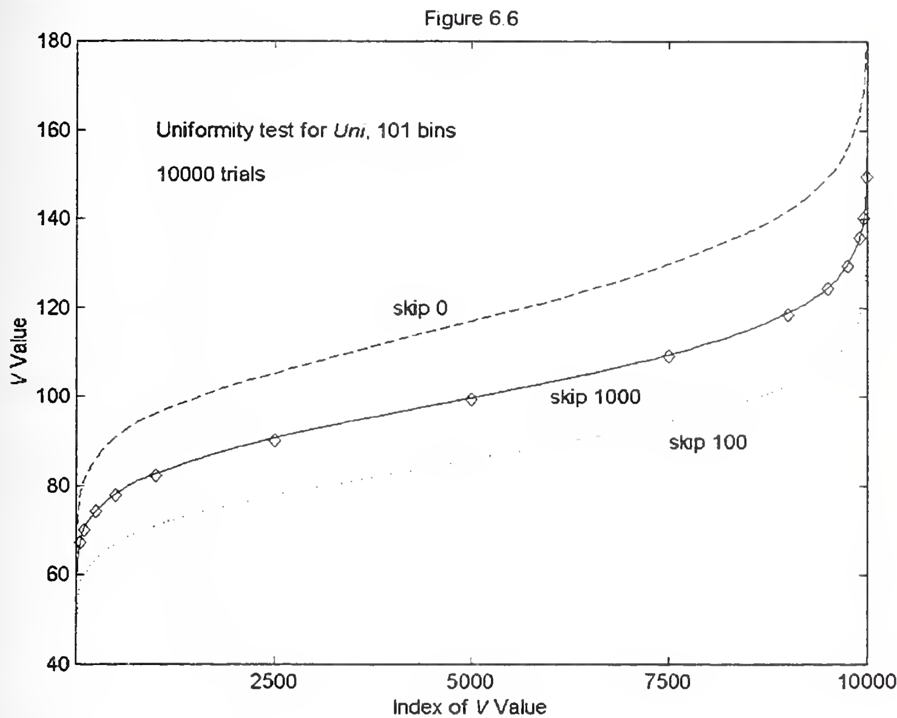
A full test would also look at pairs of RNs that are not consecutive. As *uni* contains the parameters 17 and 5, those separations (as well as 12 = 17 – 5) are reasonable to test. These tests have been performed and did not produce results that look different from the tests just done. Accordingly, they are not shown.

Short-Sequence Testing

Most testing of PRNGs focuses on long runs of RNs, as many applications use billions of RNs; the long-term behavior of PRNGs is crucial to many simulations and Monte-Carlo studies. For the jury panel selection problem, however, no more than a small multiple of M numbers are needed per application.

How well-behaved is *uni* for short runs of RNs? Not very, if the short runs are taken from the beginning of the RN sequence after starting with a new seed. Consider the uniformity test above, where 10,100 RNs were divided into 101 bins and the chi-square statistic was calculated. Suppose the first 50 RNs are the first 50 from a seed of 1, the next 50 are the first 50 from a seed of 2, and so on. The calculated V-value is 60.84, well below the expected value of 99.33. The RNs are too uniform! (They may have other imperfections, too.)

Lagged-Fibonacci PRNGs, when just started out, do not produce high-quality sequences of RNs. For long sequences, as seen in the earlier tests, the initial transient is outweighed by the excellent long-term behavior. If only a relatively small number of RNs is wanted, then the initial transient should be avoided; the initial RNs should be skipped. This effect is seen in Figure 6.6. This figure corresponds to Figure 6.3, with 10,000 *V*-values calculated for each curve, except that only 50 RNs are taken for each seed. The five curves shown have, from top to bottom, the first 0, 1000, and 100 RNs from each seed skipped before using the next 50. The diamonds show the theoretical values. Skipping the first 10,000 RNs is indistinguishable on this plot to skipping the first 1000, but is slightly closer to the theoretical curve.



Thus for best results, the initial transient from *uni* should be skipped. Since calling *uni* takes so little time, skipping 1000 or 10,000 is recommended. This adds little to the computer time. For example, skipping 10,000,000 RNs from *uni* takes only a few seconds on any reasonable computer (about 3 seconds on a 400 MHz PC).

7. Using the Algorithm: Choosing the Seed

All PRNGs require a starting value, the "seed," to begin the procedure. Different seeds produce different choices for the *N* integers. With the original program, for example, with seed 1, choosing 5 out of 100 produces (21, 45, 76, 79, 89); with seed 2, the results are (1, 36, 40, 82, 98). If seed 1 is used a second time to choose 5 out of 100, the same 5 integers will be produced, so the same seed should not be used repeatedly.

To produce fair results, either a new seed should be used each time the procedure is used, or the PRNG should be resumed at the same point. The latter is not as feasible. A perfectly reasonable method is to start with a seed of 1 and then keep increasing the seed by 1 at each use. Other methods, such as the date (for example, 20001031 on Halloween in the year 2000), and yesterday's lottery number, are also fine.

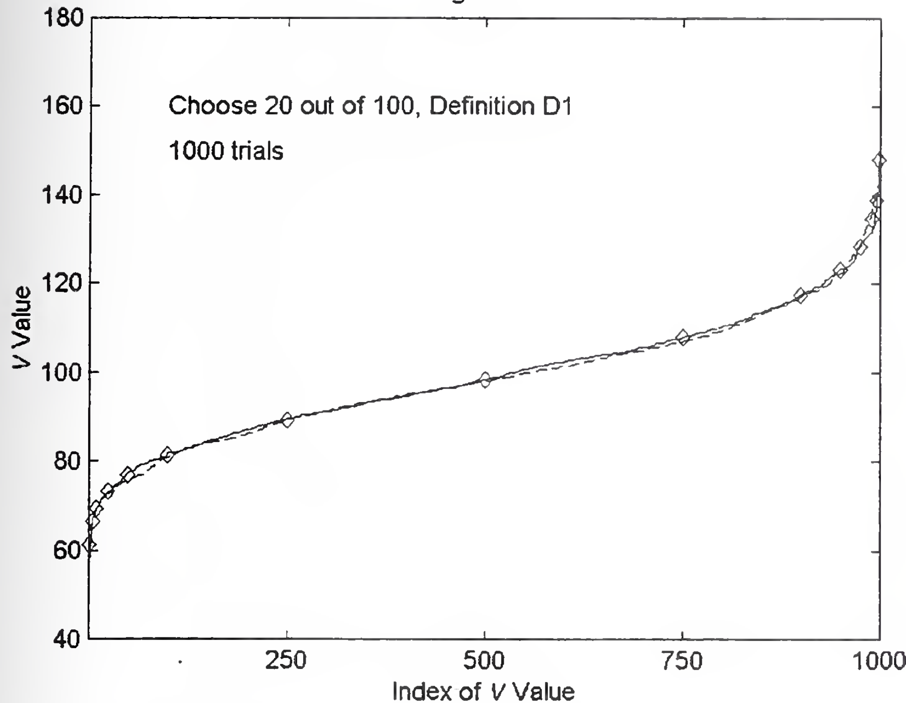
8. Testing the Implementation of the Algorithm: Definition F1

Testing our implementation of Algorithm S completely would take a very long time, as all possible aspects could be tested for all values of N, all values of M, and all possible seeds. The tests presented here do not constitute a complete test of Algorithm S. Rather, they demonstrate that it has been programmed correctly and that the sets of N integers chosen out of M have some of the most important properties of truly randomly chosen sets.

We first consider fairness by definition F1, testing to see if each of the M integers is chosen with equal likelihood when choosing N of M integers. One way to do this would be just the way we did in earlier sections. Make M bins. Many times choose N of M, each time adding 1 to its bin for each of the N integers chosen, and calculate the V value. Do all of this m times, and compare the resulting distribution of the V values to the theoretical distribution. The difficulty is that the theoretical distribution is not known except when N is 1, when it is the chi-square distribution with $M-1$ degrees of freedom. (Experiments show that, for $N > 1$, the distribution is similar to, but flatter than, the chi-square distribution with $M-N$ degrees of freedom, $\chi^2(M-N)$. We note that the asymptotic distribution of $V=\sum_{k=1}^{M}(y_k - Nm/M)^2/(Nm/M)$ is $(M-N)/(M-1)\chi^2(M-1)$, which is indeed similar to $\chi^2(M-N)$, the latter having the same expected value. However, the variance of $\chi^2(M-N)$ is greater than the variance of $(M-N)/(M-1)\chi^2(M-1)$, so that $\chi^2(M-N)$ is more "spread out." Thanks to Dr. Andrew Rukhin for pointing this out.)

To get around this problem, instead of adding 1 to the bins of each of the N integers chosen, select one of the N integers at random and add 1 only to its bin. Then the theoretical distribution is just the chi-square distribution with $M-1$ degrees of freedom. (Thanks to Dr. Mark Vangel for this observation.)

We present a single example from the tests that were done. We choose 20 of 100 integers 1000 times with different seeds, each time adding 1 to the bin of a random one of the 20 integers chosen, so that the average count per bin is 10, and calculate a V value. We do this 1000 times and plot the distribution against the chi-square distribution with 99 degrees of freedom. The result is shown in Figure 8.1. The dashed curve is for Algorithm S, and the solid curve for Algorithm P. In each case *uni* is used as the PRNG with the first 1000 RNs skipped. The fits are excellent.



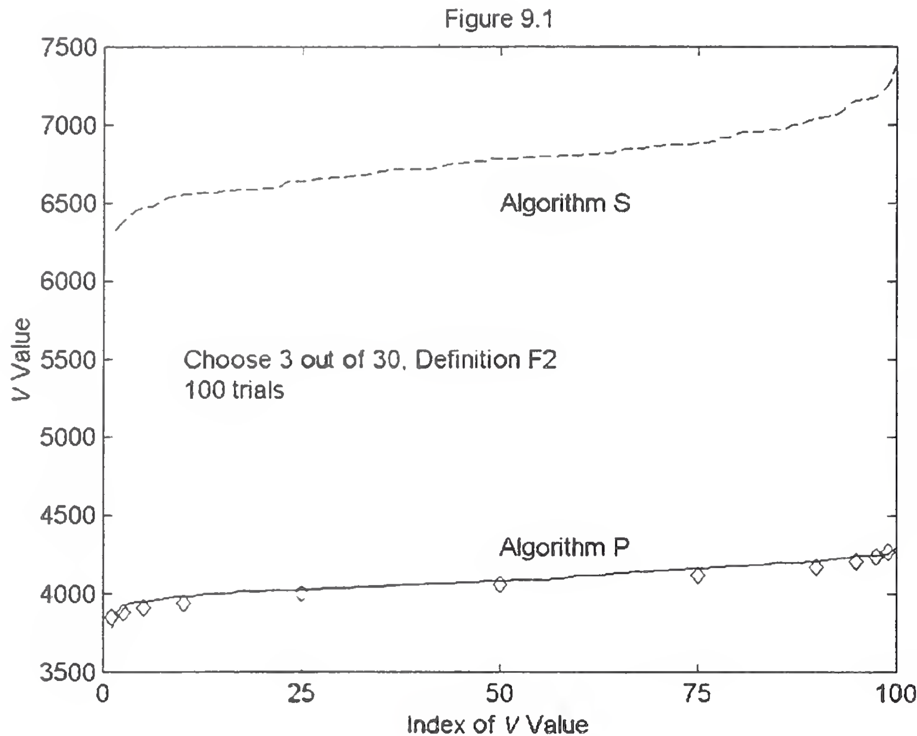
This test and the similar ones not reported here demonstrate that the program gives a fair method for selection jurors from a jury pool according to definition F1.

9. Testing the Implementation of the Algorithm: Definition F2

We now consider the much more rigorous definition of fairness, definition F2, that all of the $\binom{M}{N}$ possible sets of N integers be chosen with equal likelihood. Such testing is unfortunately impracticable for all but the smallest N and M.

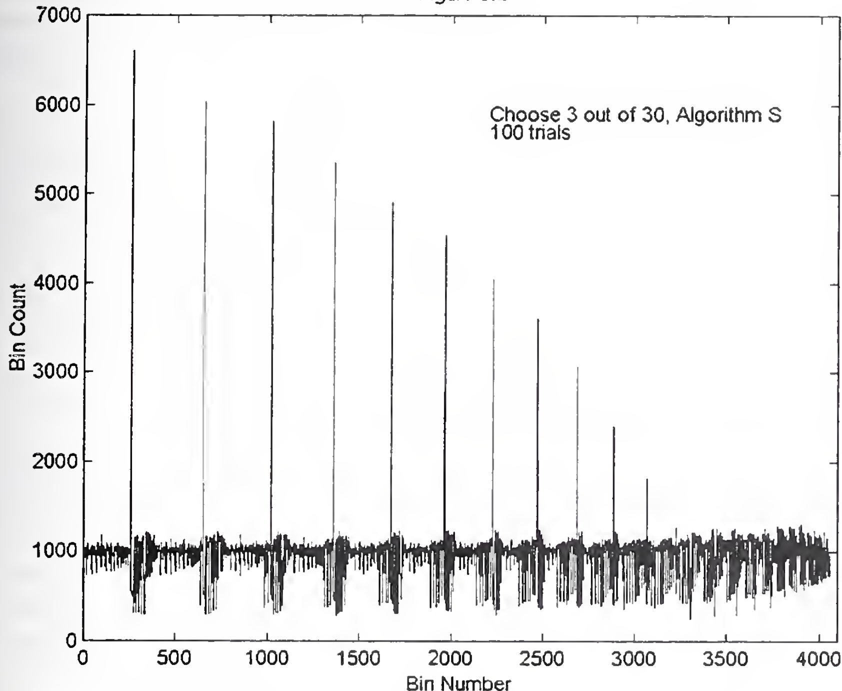
Make the $\binom{M}{N}$ bins, and identify each of the possible sets of N integers with one of the bins. Many times, choose N of M, each time adding 1 to the bin corresponding to the set chosen, and calculate the V value. Do all of this many times, and compare the distribution of the V values to the theoretical distribution, the chi-square distribution with $\binom{M}{N} - 1$ degrees of freedom.

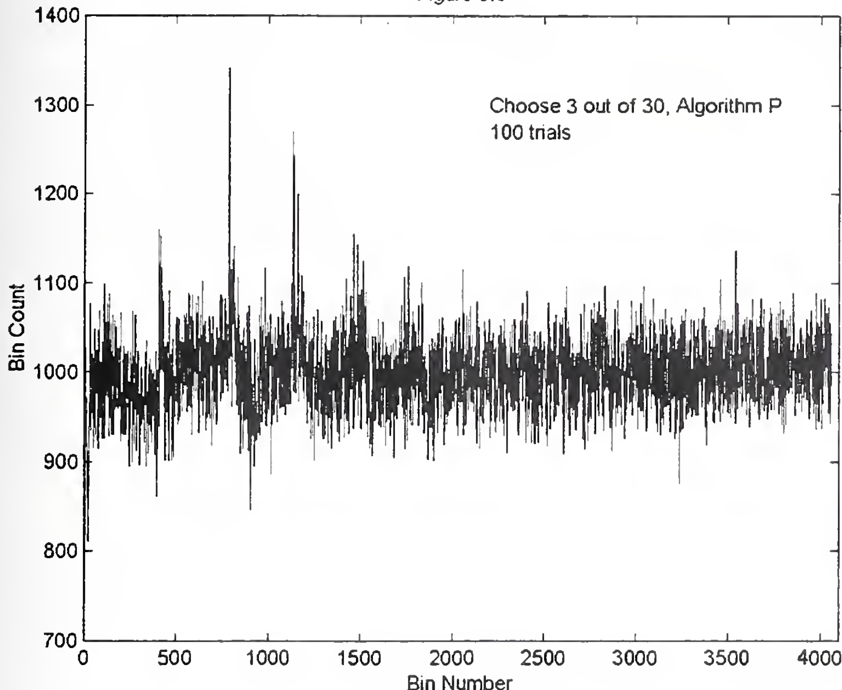
We now look at a small enough problem to test in some detail, choosing 3 of 30 integers. There are $30 \times 29 \times 28/(3 \times 2 \times 1) = 4060$ different sets of integers possible: $\{1, 2, 3\}$, $\{1, 2, 4\}$, ..., $\{28, 29, 30\}$. We choose 3 of 30 integers 40,600 times with seeds 1 through 40,600, each time adding 1 to the bin of the set chosen, so that the average count per bin is 10, and calculate a V value. We do this 100 times, increasing the seed by 1 each time; the result is shown in Figure 9.1. The upper curve is for Algorithm S, and the lower curve for Algorithm P, and the diamonds are the chi-square distribution with 4059 degrees of freedom. . In each case *uni* is used as the PRNG with the first 1000 RNs skipped. The fit is terrible for Algorithm S, good for Algorithm P.



What is wrong with Algorithm S? Nothing – Algorithm S is correct if the PRNG used is perfect, but no PRNG is. The problem is, presumably, that Algorithm S depends critically on the PRNG used, and the small inherent correlations within *uni* somehow interact with Algorithm S to give poor results. We can see some of what has happened by adding up the bin counts for all 100 trials, for a total of 4,060,000 runs. The over-all *V*-value is 275,246, far above the expected value of 4058. Figure 9.2 has a plot of the bin counts. Clearly there is something odd happening here.

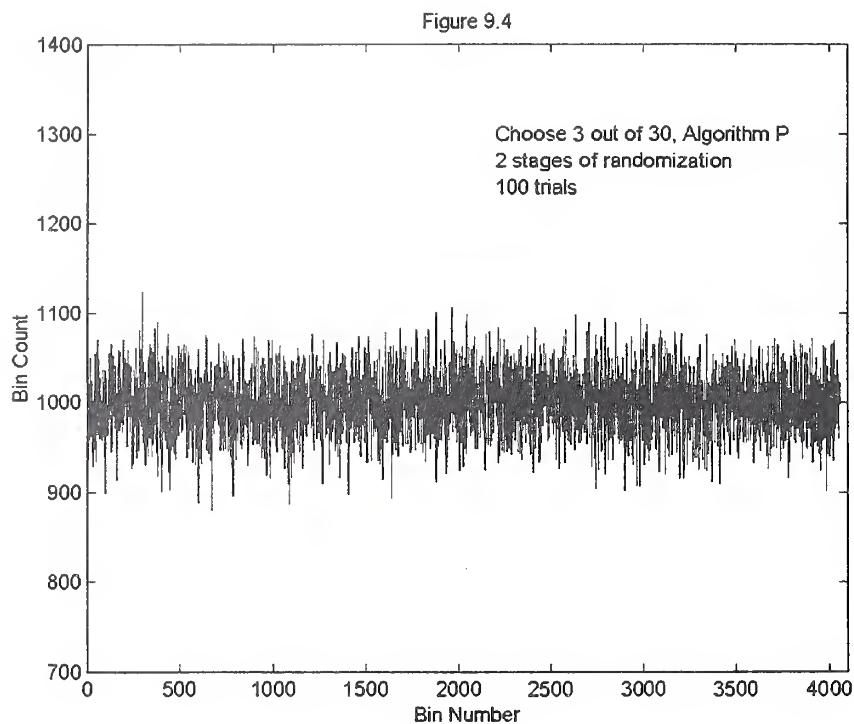
Either Algorithm P depends less critically than Algorithm S on the properties of the PRNG used, or the small inherent correlations within *uni* do not interact as much with Algorithm P as with Algorithm S. Algorithm P as stated in Section 4 also has problems, though. Its over-all *V*-value is 6181, and the plot of its bin counts in Figure 9.3 shows some difficulties, though of a lesser magnitude than for Algorithm S.

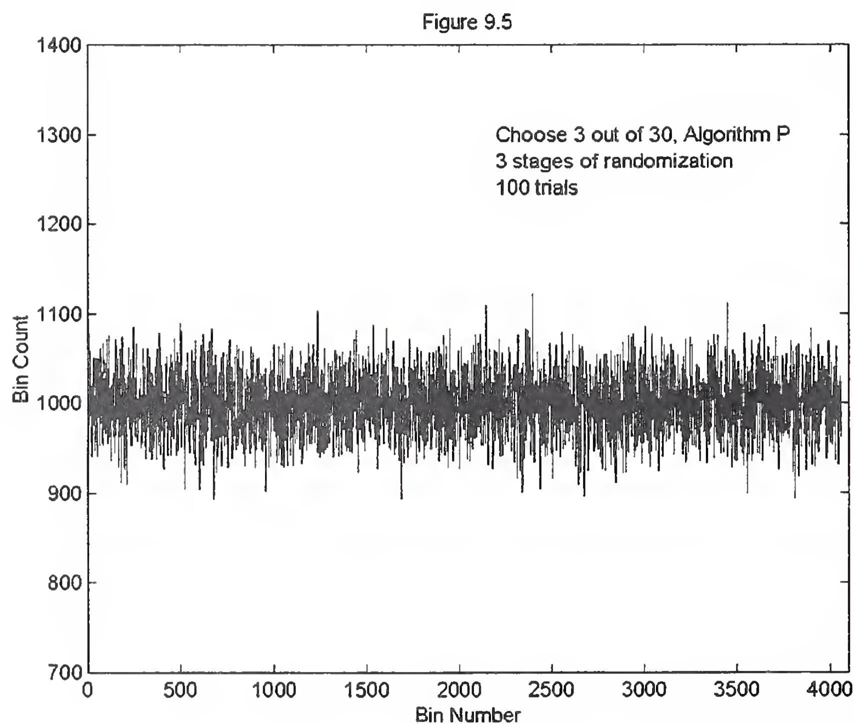




Algorithm P can be rescued, however, even with an imperfect PRNG. Further testing has shown that all that needs to be done is to repeat the randomization process (steps P2 – P6 in Section 4) a few times. (This might not suffice with a sufficiently bad PRNG.) With two stages of randomization, the bin count plot for Algorithm P, shown in Figure 9.4, is much improved, with only slight evidence of non-random regularity. With three steps of randomization, the bin count

plot, shown in Figure 9.5, shows no evidence of non-random regularity. The over-all V-values are 4148 and 4015 for two and three stages of randomization. Four stages do not provide further improvement.





Tests choosing 3 out of 35, 40, 45 or 50 show the same kind of behavior and so are not shown here. It is reassuring that matters do not seem to get worse as M grows. Unfortunately, as stated earlier, it is impractical to test realistic values of N and M.

Thus, if fairness definition F2 is required, using Algorithm P with three stages of randomization is recommended. Appendix 2 contains revised code for choosing N of M integers. If the computer can allocate enough space for M integers, Algorithm P with three stages of randomization is used. If not, an error message is printed and Algorithm S is used.

10. Conclusions

1. *Uni* is a stable implementation of F(17, 5, –), a theoretically sound PRNG.
2. *Uni* passes well-known tests of randomness, provided the initial segment of its sequence of RNs is discarded.
3. Both Algorithm S and Algorithm P are theoretically sound methods for choosing N out of M integers. If used with an ideal PRNG, they would produce results satisfying fairness criteria F1 and F2.
4. Algorithms S and P have been implemented and tested.
5. Testing for fairness definition F2 is impractical for practical sizes of N and M.
6. Algorithm S used with *uni* satisfies fairness definition F1, but not definition F2.
7. Algorithm P used with *uni* satisfies fairness definition F1, but not definition F2.
8. Algorithm P, modified to use a total of three stages of randomization, used with *uni* satisfies fairness definition F1 and definition F2, at least for small values of N and M.

11. Final Recommendations

- If fairness definition F1 is deemed adequate, the original program provided suffices, although skipping the first 1000 or 10,000 RNs is worth doing.
- If fairness definition F2 is deemed necessary, the original program provided in Appendix 1 will not suffice. The code provided in Appendix 2 should be used instead. This uses Algorithm P with three stages of randomization.

Acknowledgement

Thanks to Dr. Andrew Rukhin and Dr. Mark Vangel for an informative discussion on statistical testing.

References

1. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, M. Abramowitz and I. A. Stegun, editors, National Bureau of Standards, Applied Mathematics Series 55, Washington, D. C., 1964. Reprinted by Dover Publications, 1965.

2. *Combinatorial Algorithms*, Albert Nijenhuis and Herbert S. Wilf, Academic Press, New York, 1975.

3. *The Art of Computer Programming, Second Edition. Volume 2, Seminumerical Algorithms*, D. Knuth, Addison-Wesley, Reading, Massachusetts, 1981.

4. "A Current View of Random Number Generators," G. Marsaglia, *in Computer Science and Statistics: The Interface*, L. Billard, editor, Elsevier Science Publishers, 1985, pages 3-10.

5. "A Random Number Generator for PC's," G. Marsaglia, B. Narasimhan, and A. Zaman, *Computer Physics Communications*, volume 60, pages 345-349, 1990.

6. *Introduction to Mathematical Statistics, Second Edition*, Paul G. Hoel, Wiley, 1954.

Appendix 1. An Implementation in the C Language

Note: The following differs from the original distribution in that "float" has been replaced with "double." This makes no difference in the standard usage, but is needed for running millions of tests. It is appropriate for modern computers, especially those that support 64-bit floating-point computations.

```
/*

Random number package with simple test program.
James L. Blue
Mathematical Modeling Group
Mathematics and Computational Sciences Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

This package provides a portable method for choosing N integers from
the first M integers, with no duplications. There are four routines.

1. A simple test program
2. An initialization program, init. Init should be called with a different
argument for each call; the simplest method is to call it with argument 1 on
the first call, 2 on the second, and so on.
3. A program for choosing N of the first M integers.
4. The underlying uniform random number generator.

*/

#include <stdio.h>

#define mbig 2147483647

extern void init(int iseed);
extern void nofm(int n, int m, int *irnd);
extern double uni(int jd);

/* test program
   Correct results are

uni  0.3564443
uni  0.3584030
choosing 3 out of 20 with seed 12345
    12345      1      9      13
nofm requires n <= m; have n 21 and m 20
*/

main()
{
  int irnd[100];

  init(1);
```

```
  printf("uni %12.7f\n", uni(0));
  printf("uni %12.7f\n", uni(0));

  printf("choosing 3 out of 20 with seed 12345\n");
  init(12345);
  nofm(3, 20, irnd);
  printf("%12d %12d %12d %12d\n", 12345, irnd[0], irnd[1], irnd[2]);

  nofm(21, 20, irnd);

  exit(0);
}

/* init:
  Initialize the random number generator, uni. As a seed,
  uni requires an odd number in the range 1 to 2147483647, inclusive.
  Allow the caller to use the positive integers in order as initializers.

  Also, since very small seeds start off poorly, do a few random
  numbers to get over the transient, then use the last for a new seed.
*/
void
init(int iseed)
{
  int i, j;
  double x;

  i = iseed;
  if (i < 0) i = -i;
  if (i > mbig) i = mbig;
  if ((i % 2) == 0)
    i = mbig - i;
  x = uni(i);
  for (j = 0; j < 10; j++)
    x = uni(0);
  i = x * mbig;
  x = uni(i);
}

/*
 nofm:
  Select n items randomly from a total of m items.
  Return an array of the ones chosen.
  Method:
   Algorithm S, p. 137 of
   D.E. Knuth,
   "The Art of Computer Programming,
   Volume 2: Seminumerical Algorithms",
   Addison-Wesley, 1969.

 In the absence of roundoff, the algorithm always chooses n out of m.
 With roundoff, it might come up one short. This happens rarely, but
 if it does, just start over. (For example, in 10,000,000 different
 choosings of 19 out of 20, there were 3 startovers.)

 Sample values returned: If choosing 3 out of 20 after calling init(12345),
 the items chosen are 1, 9, and 13.
```

```
*/

void
nofm(int n, int m, int *irnd)
{
  int ihave, it, nn;
  double x;

  if (n > m)
  {  printf("nofm requires n <= m; have n %d and m %d\n", n, m);
    exit(4);
  }
  nn = m;
  ihave = 0;
  do
  {
    for (it = 0; it < nn; it++)
    {  x = uni(0);
      if ((nn - it) * x < (n - ihave))
      {  irnd[ihave++] = it + 1;
        if (ihave == n) break;
      }
    }
    if (ihave != n)
      printf(" start over: got %d, wanted %d\n", ihave, n);
  }
  while (ihave != n);
}


/* uni
***date written  810915
***revision date 940629 (JLB)
***authors
  Blue, James L., Applied and Computational Mathematics Division, NIST
  Kahaner, David K., Applied and Computational Mathematics Division, NIST
  Marsaglia, George, Florida State University

***purpose This routine generates quasi uniform real random numbers in the
    range 0 to 1, and can be used on any computer which allows
    integers at least as large as 2147483647 (in practice, any
    computer with 32 or more bits).

  use
    first time....
      z = uni(jd)
       here jd is any n o n - z e r o integer.
       this causes initialization of the program
       and the first random number to be returned as z.
    subsequent times...
      z = uni(0)
       causes the next random number to be returned as z.


  machine dependencies...
   mdig = 32; the machine must have at least 32 binary digits available
     for representing integers, including the sign bit.
```

```
   mbig = 2147483647; the machine must allow positive and negative
     integers of this size.

  remark
  This program gives repeatable results on different computers, within
  roundoff, since internal calculations are in integers.

***reference Marsaglia G., "a Current View of Random Number Generators, "
     Proceedings Computer Science and Statistics: 16th
     Symposium on the Interface, Elsevier, Amsterdam, 1985.
***routines called: none
***end prologue uni
*/

double
uni(int jd)
{
  int k;
  static int i, j, ihist[17], notyet = 1;

  if (jd != 0)
  {  int j0, j1, jseed, k0, k1, n, m;
  /*
   Fill up the history array, taking care not to overflow.

   mbig is the largest positive integer, 2**(mdig-1) - 1
   m = 2**(mdig/2)

   We use the simple congruential generator x(n) = [9069*x(n-1]mod(mm),
   with mm = (2**(mdig-1)), to fill up the history array with integers in
   the range 0 to mbig, inclusive.
   Note that mm = (m**2)/2 = mbig + 1.
   We use m to avoid overflow in the calculation, as follows:
   Any integer j, 0<=j<=mbig, can be expressed as j1*m + j0, with j1=j/m
   (discarding fractions) and j0=[j]mod m; then 0<=j0<m and 0<=j1<m.
   To multiply two numbers modulus mm, we do
   [j*k]mod(mm) = [j1*k1*m*m + (j0*k1+j1*k0)*m + j0*k0]mod(mm)
     The first term is 0 modulus mm, since m*m = 2*mm.
     We let j0*k0 = n1*m + n0, and combine n1*m with the 2nd term, giving
       [n1+j0*k1+j1*k0)*m + n0]mod(mm)
     The final term [n0]mod(mm) is just n0 = [j0*k0]mod(m)
     The remaining term is
       [n1+j0*k1+j1*k0)*m]mod(m**2/2)
     Write this as
       [a*m]mod(m**2/2)
     And expand a = (a/(m/2))*(m/2) + [a]mod(m/2). Then the first term is
       [a/(m/2))*(m/2)*m]mod(m**2/2) = 0
     and the second term is just
       [a]mod(m/2)*m
     since it is clearly already less than m**2/2.
  */
    jseed = jd;
    if (jseed < 0) jseed = -jseed;
    if (jseed > mbig) jseed = mbig;
    if ((jseed % 2) == 0) jseed--;
    m = 65536;
    k0 = 9069 % m;
```

```
    k1 = 9069/m;
    j0 = jseed % m;
    j1 = jseed/m;
    for (i = 0; i < 17; i++)
    {  n = j0*k0;
       j1 = (n/m+j0*k1+j1*k0) % (m/2);
       j0 = n % m;
       ihist[i] = j0+m*j1;
    }
    i = 4;
    j = 16;
    notyet = 0;
  }

  /* Begin main loop here. */

  if (notyet)
  {  printf("first call to uni must have nonzero argument");
    exit(2);
  }
  k = ihist[i] - ihist[j];
  if (k < 0) k += mbig;
  ihist[j] = k;

if (--i < 0) i = 16;
  if (--j < 0) j = 16;
  return (double)(k) / (double)(mbig);
}
/* end of program */
```

Appendix 2. Modifications to the Implementation in the C Language

Note: The following code should replace the *nofm()* module in the original if fairness definition F2 is required. This code uses Algorithm P with 3 stages of randomization if sufficient space can be allocated. Otherwise, it prints an error message and uses Algorithm S.

```
#include <stdio.h>

static void nofmP(int, int, int *, int*);
static void nofmS(int, int, int *);
static void randomize(int *, int);

extern double uni(int);

/*
 nofm:

 Do Algorithm P if possible, otherwise Algorithm S.
*/

void
nofm(int n, int m, int *irnd)
{
    int *list;

    if (n > m)
    {   printf("nofm requires n <= m; have n %d and m %d\n", n, m);
        exit(4);
    }
    list = (int *)malloc(m * sizeof(int));
    if (list == NULL)
    {
        printf("malloc failed to get %d integers; changing algorithms\n", m);
        nofmS(n, m, irnd);
    }
    else
    {
        nofmP(n, m, irnd, list);
        free(list);
    }
}

/* nofmP

      Algorithm P:

   Put integers 1 through m in a list.
   Randomize them.
   Pick the first n of them.

*/

#define RPT_MAX         3

void
```

```
nofmP(int n, int m, int *irnd, int *list)
{
    int j, rpt;

    for (j = 0; j < m; j++)
        list[j] = j + 1;
    for (rpt = 0; rpt < RPT_MAX; rpt++)
        randomize(list, m);
    for (j = 0; j < n; j++)
        irnd[j] = list[j];
}

/* randomize a list of m integers. Pick a random location before m and
   exchange its contents with the m-th location. Reduce m by 1 and
   continue. */

void
randomize(int *list, int m)
{
    while (m > 1)
    {
        int j, tmp;

        j = m-- * uni(0);          /* j is random [0:m-1] */
        tmp = list[j];             /* exchange j, m-1 */
        list[j] = list[m];
        list[m] = tmp;
    }
}

/* nofmS

   Algorithm S
*/

void
nofmS(int n, int m, int *irnd)
{
    int ihave, it, nn;
    double x;

    nn = m;
    ihave = 0;
    do
    {
        for (it = 0; it < nn; it++)
        {   x = uni(0);
            if ((nn - it) * x < (n - ihave))
            {   irnd[ihave++] = it + 1;
                if (ihave == n) break;
            }
        }
        if (ihave != n)
            printf(" start over: got %d, wanted %d\n", ihave, n);
    }
    while (ihave != n);
}
```

```
/* end of nofm module */
```