An Introduction to `mrandom 3.0`

Clark Thomborson*
Computer Science Department
U Minnesota, Duluth MN 55812 USA
cthombor@ua.d.umn.edu

Abstract

We offer a set of public-domain tools to aid the development and test of pseudorandom C-language computer programs running under 4.3bsd Unix. Our tools make it easier to reproduce experimental results, to report them in a standardized format, and to check their validity by using a different random number generator (RNG). Furthermore, the tools should help users avoid some of the common pitfalls in randomized experimentation. These pitfalls include the initial seeding of the RNG, the generation of random integers in a restricted range, and the use of RNGs with known weaknesses.

Our tool suite also includes several of the standard tests on the "randomness" of an RNG. All the tests we implement are based on examining a Pearson's chi-square statistic for a process that, under the null hypothesis, independently places N items into k equiprobable bins. We provide a routine to calculate an approximate p-value for such test statistics, applying novel corrections for "small N."

*Research supported by the National Science Foundation, through its Design, Tools and Test Program under grant number MIP 9023238.

Contents

1 Introduction

Many experimental and analytic techniques are based on the assumption that pseudorandomness is "as good as randomness." That is, we assume that a computer program can generate a sequence of pseudorandom numbers whose properties are, at least in our application, indistinguishable from those of a truly random sequence. This assumption is a potential source of experimental bias [8]. According to Knuth, "The most prudent policy for a person to follow is to run each Monte Carlo program at least twice using quite different sources of random numbers, before taking the answers of the program seriously" [16, p. 173].

It is surprisingly difficult to follow Knuth's advice. Every random number generator (RNG) seems to have a distinctive set of calling conventions: different seeding parameters, different arguments, different argument orders, etc. Some floating-point RNGs return values in the range 0.0 to 1.0 inclusive, and some omit one or both of the endpoints of this range. Fixed-point RNGs have different ranges. For all these reasons, it is difficult to write source code that calls an RNG without embedding RNG-specific assumptions in one's application code. Once these assumptions are embedded, it can be difficult to switch to a different RNG algorithm without introducing a "bug."

My research program relies heavily on RNGs, and I got tired of modifying and revalidating my application codes whenever I wanted to switch generators. I had also become disgusted with the generally-poor RNGs in 4.3bsd Unix. Suspecting that others had faced, or would soon face, similar issues, I made a public-domain release in December 1991 of the first version of my `mrandom` package of C-language routines. In September 1992, I released version 2.3 of `mrandom` into the public domain.

The major advance in version 3.0 of `mrandom` is a thorough modularization of the RNG interface routines, making it much easier to install new RNG codes than in the preceding versions. The modularization was designed and realized by Robert Plotkin, a talented MIT undergraduate, as part of his Bachelor's thesis project.

This report outlines the features of the Version 3.0 release of `mrandom`, now (June 1993) in alpha-test. Both Version 3.0 and the fully-tested Version 2.3 are available by anonymous ftp from `theory.lcs.mit.edu`, in directory `/pub/cthombor/Mrandom`.

Herein, I use the first person singular "I" for statements of my belief about RNG usage, and for analytical results I have not discussed with Robert Plotkin. I use the authorial "we" when describing the package as a whole, to reflect our joint authorship of its code.

I intend to release `mrandom` into the public domain, but I am not sure that I have successfully done so, for the following reasons.

1. I am, by no means, an expert in software law.
2. The development of `mrandom` was partially funded by the National Science Foundation of the United States, under my research grant MIP-9023238. I have not yet asked the NSF to waive any financial interest they may have in this code.
3. The University of Minnesota and/or Massachusetts Institute of Technology might, someday, claim a financial interest in this code. Both institutions paid a portion of my salary, and provided equipment, during `mrandom`'s development. Perhaps I should seek a waiver from these institutions?
4. The `mrandom` code is built on top of 4.3bsd Unix. This suggests that a commercial `Unix` license would be needed for some uses of `mrandom`.

5. Marsaglia's C-language `Ultra` code is included as part of `mrandom` version 3.0. Marsaglia's code bears the warning(?) "To obtain permission to incorporate this program into any commercial product, please contact the authors at the e-mail address given above [`afir.stat.fsu.edu` or `geo@stat.fsu.edu`] or at Department of Statistics and Supercomputer Computations Research Institute, Florida State University, Tallahassee, FL 32306."

2 Overview of the `mrandom` package

The principal features of the `mrandom` package are

- A standardized interface to many simultaneously-active RNGs,
- An unbiased, and apparently novel, method for generating random integers in the range $0..m-1$,
- The standard (fast but biased) method for generating random integers in the range $0..m-1$,
- A standardized method for generating floating point numbers uniformly distributed in $[0.0, 1.0)$,
- Two standardized methods for generating streams of "random bits,"
- Rapid, vectorized calls returning an arbitrary number of uniform variates of any desired type,
- Buffered and unbuffered calls, for efficient generation of pseudorandom generates in both large and small quantities,
- The ability to "split" an RNG output stream into several nonoverlapping output streams,
- A shorthand notation for completely specifying the algorithm and current state of an RNG, in an 80-character human-readable ASCII string,
- A method for reconstructing a complete RNG state from its shorthand notation,
- A standardized method for adding new RNGs to the package, and
- A file-I/O interface to allow fast saves and restarts of complete RNG state vectors.

The `mrandom` package also includes a set of routines to test the accuracy of a compilation on a new system. If you type the Unix command `make test` in a directory containing the `mrandom` source code, the results of running several statistical tests on several different RNG algorithms on your workstation will be compared to the results on a SparcStation 1+. All differences will be printed; any difference, except in the runtimes, should be reported to the author as a bug.

The test routines are packaged in an executable called `mrtest`, documented in a `man` page and in Section 3 of this report. You may use these routines to study the properties of the various RNG algorithms. The tests include equidistribution, pairwise (both short- and long-range) correlation, and 3-tuple correlation [21]. I chose to implement these tests because they were simple to code, because they would exhibit the known defects of the 4.3bsd generators `rand()` and `nrand48()`, and because their test statistics could all be analyzed with the same code.

A final feature of the `mrandom` package is the statistical analysis code `xsq.c`. Originally, I thought it would be easy to write this code, using the easily-computable, and asymptotically

correct, χ^2 approximation to the actual distribution, X^2, of the Pearson's chi square test statistic arising in our application under the hypothesis that each of N independent observations of our RNG's output is equally likely to fall into any of k categories. I soon learned that the χ^2 approximation is valid only for rather large N; and even for large N, the χ^2 approximation is valid only near the center of the X^2 distribution [16, p. 43]. This prodded me to develop, and code, a novel approximation to the tail probability of the X^2 statistic on a symmetric multinomial variate. My approximation is described and justified in Section 3.

2.1 Naming Conventions

Our access routines are named mnemonically. The first letter of the access routine is `d`, `f`, `l`, `m`, or `b`, depending on whether it is returning a `d`ouble precision float, a single-precision `f`loat, a `l`ong integer in 0..`range_rng()`-1, a restricted-range integer in 0..`m`-1, or a single-bit integer whose value is either 0 or 1.

If the second letter of the access routine is an `x`, the routine is unbuffered. Otherwise it is buffered. See Section 2.4.

The suffix characters on an access routine indicate whether the `rng` or the `v`ectorization parameters must be supplied explicitly by the user. Thus `drandom()` takes no parameters, `drandomr(rng)` requires a pointer to an active RNG, and `drandomv(n,v)` must be given a vector length and a vector pointer.

Note: I use an italic font for mathematical quantities, and a typewriter font for analogously-named variables in the C programming language. For example, the value of `(double) m*z` will in general be only an approximation, and not necessarily a good approximation, to the algebraic value mz. Typical sources of computational imprecision are floating-point roundoff, underflow, and overflow.

2.2 Generating Integers Uniformly Distributed in a Restricted Range

It is common practice to use one of two naive methods to generate a "random" integer in 0..$m-1$ from a pseudorandom generate z in the range 0..`range_rng() - 1`: either

1. `z % m`, a C-language expression computing the value $z \bmod m$; or
2. `(int) m * ((double)z) / range_rng())`, a double-precision floating point approximation to the quantity mz/r where $r =$ `range_rng()`.

The second method is vastly preferable to the first, for two reasons. The modulo-`m` arithmetic of the first method can expose the "nonrandom" behavior of the least-significant bits of many commonly-used RNGs, most notably those based on a multiplicative congruential method [5, Section 6.7.1], [16, Section 3.4.1]. In contrast, we gain confidence in the "random" properties of the second method's outputs whenever the RNG passes a test based on its floating-point outputs `((double)z) / range_rng())`.

Another reason to prefer the second naive method over the first is that present-day workstations are more efficient at performing floating-point divisions "`/`" than modulo-`m` calculations "`% m`". A floating point division is, nowadays, accomplished with a single CPU instruction. The modulo-m operation might be a single instruction of comparable speed to a floating-point divide (about 20 clock cycles on an RS-6000), but more often it is a slower subtract-then-test loop (on a SparcStation or an HP-PA) or a series of floating-point instructions (on a DECstation).

```
double dm, r, ifreq;
long zmax, z;
dm = (double) m; /* the desired range */
r = rng_range(); /* the ``raw'' range of the RNG */
ifreq = floor( r/dm );  /* desired probability density */
zmax = r - (int) ( r - ifreq*dm ); /* max acceptable RNG output */
do ( z = lrandom(); z > zmax;  ) { /* Is z acceptable? */
  z = lrandom();  /* No, cycle the RNG again */
};
return( (long) ( (double) z / ifreq ) ); /* an unbiased integer */
```

Figure 1: An algorithm for generating unbiased integers in 0..`m-1`.

For these reasons, we do not support the first naive method in `mrandom`. We do offer support for the second naive method because it is faster than our unbiased method, and because it is essentially unbiased when `m` is small. When `m` is almost as large as $r =$ `range_rng()`, however, the naive methods put twice as much probability density on some integers in `0..m-1` than on others [16, Problem 3.4.1(2)]. Easily detectable bias will in fact occur for any `m` $> r/1000$, unless `m` divides r exactly.

An unbiased method for generating integers. We recommend using the acceptance-rejection method of Figure 1 when generating restricted-range integers. Each of the `m` values in the desired range will occur with probability exactly $1/m$, if each of the r values in the range of the underlying RNG occur with probability $1/r$. Note that `ifreq` is computed exactly by an IEEE-standard double-precision floating point division operation, because it is exactly representable in the 54-bit precision of a `double` mantissa.

The code of Figure 1 avoids the complicated and slow integer calculations that were necessary to generate unbiased integers before floating-point operations were standardized [5, Figure 6.7.1]. Since our integer-mapping procedure is also relatively insensitive to any "least-significant bit nonrandomness" in the underlying RNG, we believe it superior to the otherwise-similar acceptance-rejection scheme of the public-domain package `ranlib`. (Note: `ranlib` may be obtained by anonymous ftp from `odin.mda.uth.tmc.edu` in directory `/pub/unix`, or by sending the message "send ranlib.c.shar from general" to `statlib@lib.stat.cmu.edu`.)

The actual integer-making code in `mrandom` is a little more complicated than shown in Figure 1. Our code is "vectorized" (*i.e.* coded as a series of pipelinable loops with large iteration counts), allowing efficient execution on the highly-pipelined CPU of a workstation. Also, we do not write indefinite `do` loops: our code will not get "stuck" in a loop such as the one in Figure 1, even if the underlying RNG stream converges on a constant. Instead, our code will abort your program run and print a message asking you to make a bug report, if any loop runs for an improbable ($p < 10^{-30}$) length of time. Note that a zero seed, or a state table that becomes all zeroes, could cause an RNG to return the same value forever.

Using `mrandom` to generate integers. You must select either the unbiased method or the second naive method for computing restricted-range integers, whenever you restart or initialize an RNG (see Section 2.9). After an RNG is activated, you can obtain a single restricted-range integer by evaluating `mrandom(m)`.

The most general call is `mrandomrv(rng, m, n, v)`, which fills the vector `v` with `n` pseudo-random integers, uniformly distributed in `0..m-1`, using `rng` as the underlying generator. Here, `rng` is a variable of type `RNGdata*`; this type was introduced in `mrandom` so that an expert user can conveniently access more than one RNG at a time. See Section 2.6.

In future research, I hope to develop a code that uses just `n` comparisons, $O(1)$ floating point divides and `n` $+O(1)$ floating-point multiplications to transform an `n`-vector of pseudorandom floats in into an `n`-vector of 32-bit signed integers uniformly distributed in the range 0..`m-1`, for any `m` $\leq 2^{31}$. The following method comes tantalizingly close to my objective. Calculate `a = 1.0 / ifreq` in IEEE-standard double-precision format, where `ifreq` is defined as in Figure 1. The desired integers are "close" to `a*v[i]`, for $1 \leq$ `i` $\leq$ `n`, where `v` is an element of the n-vector of pseudorandom floats produced by a call to the `drandom(n,v)` procedure described below. Some additional cleverness is necessary, either in coding or in correctness proof: I believe that neither truncation nor rounding of `a*v` will give an unbiased result for all `m`.

2.3 Uniform Variates in [0..1.0)

The `drandom()` function of `mrandom` uses a naive method to generate 64-bit floats:

```
((double)lrandom() / range_rng())
```

where `lrandom()` is a "raw" output from the underlying integer-valued RNG. Rarely, one encounters an RNG code whose raw output is a floating-point number. Such generators can be installed in the current version of `mrandom`, if their outputs x are restricted to the range $0 \leq x < 1$.

Some authors [9] believe that a floating-point RNG should have range (0.0, 1.0), and some [16] believe that its range should include both 0.0 and 1.0. I find the range $[0.0, 1.0)$ to be most convenient for my work. Perhaps a future release of `mrandom` will provide efficient interfaces for all the competing definitions. For now, I suggest that any users dissatisfied with $[0.0, 1.0)$ should write their own macro, for example using `( 1.0 + (double)lrandom() ) / ( 1.0 + range_rng() )` to transform the outputs of the current integer-valued RNG into the open interval (0.0, 1.0). Please note that the sums must be calculated with more than 32 bits of precision, because `range_rng()` $= 2^{32}$ for some generators in the `mrandom` package. Also note that, unless one uses an acceptance-rejection method or unless the underlying RNG has an integral multiple of 2^{24} values in its range, the 24-bit mantissas of a single-precision pseudorandom variate must be biased.

A possible future enhancement to `mrandom` would minimize discretization error by providing clever floating-point transformations [24]. Note in particular that rounding is not appropriate when converting a double-precision variate in $[0, 1)$ to a single-precision variate: the range of the single-precision variate obtained by this method will include 1.

I am currently investigating RNGs with greater integer range, to see if a future release can recommend the use of "all bits" in a double-precision floating-point RNG output. At the moment, I'd say that no floating-point RNG output, in any software package, should be assumed to have more than four decimal digits of accuracy. Since a double-precision float can represent a real number with more than fifteen decimal digits of precision, the problem is not one of representability. Instead, it is a lack of credibility, due to a dearth of RNG algorithms whose outputs have been extensively tested at high accuracies. Most of the RNG tests in common use are sensitive only to defects in the first few digits after the decimal point in a floating-point RNG output. All RNG algorithms in common use are known to be "cryptographically insecure" if all their output bits are used. Finally, many popular RNG algorithms have noticeable defects in

their least-significant bits.

2.4 Buffering

The preceding sections have introduced the idea of vectorized calls to a random number generator, as a method of decreasing the CPU time required per RNG output. The ideal application code will use only long (100+) vector lengths, although this is sometimes difficult to arrange in practice. Accordingly, we provide internal buffers in `mrandom`, so that some of the benefit of vectorization can be obtained in codes using just one RNG output at a time. Our optimization is similar to that provided in the C-language macro `getchar()`, which is expanded in-line into a conditional branch instruction testing whether the file buffer is empty or not. The buffer is replenished with a single procedure call to the operating system, in the case of `getchar()`. In the `mrandom` package, the buffer is replenished with a single vectorized call to the underlying RNG. Thus, even unvectorized codes can use a highly-optimized version of the underlying RNG.

2.5 Random Bits

Some applications are most naturally expressed as a transformation on a pseudorandom bitstream. In such cases, it is tempting to extract 32 "random" bits from every 32-bit RNG output. We do not recommend this practice, for two reasons. This coding trick can only be employed on RNGs whose range is equal to 2^{32}, so your application will be restricted to a small subset of the available RNGs. Secondly, your experimental results will be heavily dependent on the cross-correlation properties among the various bitfields in your RNG's outputs. These properties have been tested lightly, if at all, for most RNG codes in existence. Note that the process of seeding an RNG may introduce bitwise cross-correlation defects even in RNG algorithms whose bitfields are "obviously" independent, such as the shift-register method $X_n = X_{n-103}$ `.XOR.` X_{n-250}.

The imperatives of runtime efficiency, however, are such that we grudgingly provide `fast` interface routines (*e.g.* `brandom_f()`) for convenient, efficient, and sequential access to each of the 32 bits in a 32-bit RNG output. In contrast, the recommended `brandom()` call extracts just one "random bit" from each RNG output word.

2.6 Splitting an RNG's Output Stream

In a multithreaded application code, *i.e.* one that runs simultaneously on several workstations, we usually want a different RNG stream in each thread [19]. A popular method of accomplishing this objective is to use a distinct seed for a separately-initialized RNG in each thread. The resulting streams might be statistically independent. However, if the seeding process is poorly designed and/or "unlucky," the tail of one fairly-short RNG stream will be identical to the head of another RNG stream.

A safer method of writing a multithreaded application, also supported by `mrandom`, is to "split" a single RNG's stream. For example, if one has three threads, the i-th thread should use the $(i + 3k)$-th elements, for all $k \geq 0$, from a single RNG stream. To the extent that an RNG has good serial correlation properties, the three streams will be statistically independent.

If you call the `mrandom` procedure `split_rng(rng,x)`, the immediate efect is to store the integral "split value" `x` in the internal data structure pointed to by `rng`. Note that `rng` must be an active RNG; see Section 2.9. The desired side-effect of `split_rng()` is that subsequent

```
RNGdata rng1,rng2;
long z;
restart_rng( rng1, RNGstatefile ); /* activate rng1 from a file copy */
restart_rng( rng2, RNGstatefile ); /* rng2 is identical to rng1 */
z = lrandomr( rng2 ); /* discard the first ``raw'' output of rng2 */
split_rng( rng1, 1 ); /* split 1: skip over alternate elements */
split_rng( rng2, 1 ); /* rng2 also gets a split value of 1 */
```

Figure 2: Splitting an RNG in two.

accesses to this RNG will "skip over" or "leapfrog" elements in its pseudorandom sequence. For example, if the first 16 outputs of an (apparently defective) RNG were

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...$$

and if this RNG were given a split value of 1 immediately after initialization, it would return the sequence

$$0, 2, 4, 6, 8, 10, 12, 14, ...$$

The other branches of a split-sequence RNG should be obtained by activating "duplicate" RNG data areas, using exactly the same initialization sequence (*e.g.* seed values and RNG algorithm specifier) as on the first branch. Each RNG area should be "split" after it has been cycled a different number of times. See Figure 2 for an example of a 2-way split; if the underlying RNG (specified by the `RNGstatefile`, see Section 2.9) and `rng1` are as described in the previous paragraph, then the next eight outputs of `rng2` would be

$$1, 3, 5, 7, 9, 11, 13, 15, ...$$

As indicated above, the splitting operation is useful in multiprocessed codes. Please be warned, however, that our present implementation is inefficient for large skip values `x`, because `x` raw RNG outputs are discarded for every one that is returned to the user. Such inefficiency is not necessary in some RNG algorithms. For example, the constants in any multiplicative congruential RNG can be adjusted, as a function of `x`, so that it generates every `(x+1)`-th element of its original output stream [19]. Perhaps a future release of `mrandom` will allow an efficient generation method to be installed for those RNGS for which fast splitting is possible.

2.7 Installing New RNGs

To install a new RNG in version 3.0, you must supply a seeding routine and a generation routine. Your seeding routine will be called when a user wants to initialize an `RNGdata` area referencing the new RNG: see Section 2.9. Your generation routine will be called whenever the data buffer of this `RNGdata` area is empty. The generation routine may be declared as returning either a 32-bit `long` pseudorandom integer or a 64-bit `double`-precision pseudorandom float: `mrandom` will perform the appropriate type conversion, as necessary, in response to the user's request for any supported type of pseudorandom data.

We recommend that each installed RNG also have a nontrivial implementation for a third functional interface to `mrandom`. A "state vector check" function will be called whenever an RNG

statefile is read from disk, for example during the `restart_rng` operation described in Section 2.9. Note that a descriptor of the RNG algorithm is stored with each statefile, so that the appropriate state-check function can be called. A trivial implementation of your state-check function would always return the value `1`, implying that the file appears to be intact. A more carefully-coded check function will examine the state vector in some RNG-specific fashion, testing if its values are consistent with your RNG algorithm, and perhaps taking into account the initializing seeds and the number of times your RNG has been cycled since seeding. Note: seeding and cycle-counting information is also contained in the RNG statefile, to allow error-checking as well as complete experimental documentation.

The RNG check function need be neither excessively complicated nor time-consuming to have some utility. For example, in many RNG codes, some elements of a valid state vector are restricted-range integers. If an out-of-range value is detected, the RNG-specific check code should return the value `0`; this will cause `mrandom` to print an error message and to abort the user's job.

The installer of a new RNG must provide a unique ASCII string identifying the RNG algorithm, for printing in RNG statefiles. Finally, new entries must be made in several arrays of compile-time constants. See Section 6.3 of the *mrandom 3.0 User's Manual*, provided with the source-code distribution, for a complete description of the RNG installation process.

A future version of `mrandom` may allow the RNG installer to provide a vectorized interface to the generation routine. This should greatly reduce the "overhead" associated with entering and exiting the RNG generation code. The heavily pipelined CPUs of modern workstations and supercomputers are most efficient when executing inner loops with a small number of operations per iteration, and with a large number of iterations per loop. Other sources of inefficiency are frequent procedure calls and certain forms of data-conditional branches. On most workstations, the generation routines of `mrandom` version 3.0 are compiled into short sequences of in-line code to retrieve data from a buffer, with rare (and well-predicted) conditional branches to a buffer-filling routine. The buffer-filling routine, however, typically contains one procedure call per RNG value inserted into the buffer. A vectorized interface to the RNG generation routine would allow the buffer to be filled in a tightly-coded inner loop containing no procedure calls. The average number of procedure calls, per RNG output, can thus be as low as $1/n$ in a completely vectorized code with vector length n.

2.8 RNG Algorithms Supported

When using `init_rng()` to initialize an `RNGdata` area, or when using `mrtest` to initialize an RNG data file, you must specify an RNG algorithm with an integer in the range 1 through 9. The association of integers with algorithms is defined in the list below. Note that you may use RNG algorithm 0 to see how your application would work with the highly "nonrandom" RNG at the tail of this list.

1. 4.3bsd `random()`, a non-linear additive feedback RNG, with a range of 2^{31}. This code is not re-entrant: some of the RNG state is saved in a single "owned" variable, even if multiple RNG state tables have been defined. We have partially repaired this bug in the `mrandom` interface, so that a single instance of this RNG can be run, saved, and restarted from a file without error. We do not recommend running multiple instances of this generator in a single job, however, because the various RNG states can not be saved and restarted accurately and reliably.

2. The Knuth/Bentley `prand`, a C-language code for an additive generator, developed by Bentley[3] from an assembly-language code in Knuth [16, p. 27]. Knuth attributes this RNG to unpublished work by Mitchell and Moore. The recurrence is $X_n = (X_{n-24} + X_{n-55}) \bmod r$, where $r = 10^6$ in the Bentley code imported into `mrandom`. Knuth requires merely that r must be even, and that $X_0, ..., X_{54}$ are "arbitrary integers not all even." This RNG seems to be acceptable for most uses on a workstation. However, recent unpublished work by Profs. Bill Knight and Greg Shannon of Indiana U. indicates that the least-significant bits of this RNG are flawed, as revealed by the following experiment. Take 10^4 blocks of 100 generates. In each block of 100 generates, extract the least-significant (parity) bits. If the generates were uniformly distributed in $0..r-1$, the number of zero bits would be binomially distributed with parameters $n = 100$ and $p = 0.5$. The observed distribution (*i.e.* our 10^4 observations of the number of zero LSBs in a block of 100 generates) has the correct mean (50), but an incorrect "shape": it fails a traditional chi-square test of fit. The closely-related `random()` algorithm of 4.3bsd Unix also fails "Bill's test" on its LSBs. In both cases, the failure could, conceivably, be attributed to the seeding algorithm rather than to the generation algorithm; these codes use very poor multiplicative RNGs to convert a 32-bit seed into a multiple-word state table.

3. L'Ecuyer's "portable combined" 32-bit multiplicative congruential generator [18] with a range of $2147483561 = 2^{31}-87$. This generator seems to be acceptable for most workstation uses.

4. 4.3bsd `nrand48()`, an 48-bit multiplicative congruential RNG with a range of 2^{31}. I do not recommend the use of this generator. I know of no "champion" willing to argue in its favor. Furthermore, its defective long-range correlation properties are exhibited by the test script supplied with `mrandom`.

5. 4.3bsd `rand()`, a 32-bit multiplicative congruential RNG with a range of 2^{31}. I do not recommend the use of this generator: it has a number of poor properties, including the one documented in its 4.3bsd Unix `man` page.

6. Press and Teukolsky's `ran0` [26], identical to the "minimal standard generator" of Park and Miller [25]. It is a multiplicative congruential generator with a range of $r = 2^{31} - 1$ and a recurrence of $X_{j+1} = (16807 X_j) \bmod r$. This generator has been used successfully in a wide variety of applications since its proposal by Lewis *et al* in 1969. Despite an "obvious" defect [26], it was the only RNG, of five tested, to give good results in a recent Monte Carlo simulation [8].

7. Press and Teukolsky's `ran1` [26], a generator with a range of $2^{31} - 1$. This code uses the minimal standard generator `ran0` to fill a table of size 32. Elements are selected for output and replacement in the table with the "Bays-Durham shuffle." Since Press and Teukolsky's column is widely read, this new RNG might be used in a wide variety of applications. As soon as I hear favorable reports from some knowledgable users, I will consider this RNG to be a reliable source of pseudorandom variates. For the time being, however, I recommend that it (and any other novel RNG) be used only to verify experimental results obtained with other RNGs.

8. Press and Teukolsky's `ran2` [26], a Bays-Durham shuffle applied to the output of L'Ecuyer's portable combined generator (`mrandom` algorithm 3, above). Its range is $2^{31} - 1$. Press and Teukolsky offer a reward of $1000 to the first reader who finds "a statistical test that this generator fails in a nontrivial way, excluding the ordinary limitations of a machine's floating-point representation." [26]. This reward should accelerate the testing/validation

process: perhaps I will recommend it for general workstation use by 1995. However, I doubt this code will ever obtain widespread acceptance, because it requires so many machine operations per RNG output. Furthermore, the Bays-Durham shuffle is, I believe, inherently inefficient on the heavily-pipelined CPUs used in workstations and supercomputers.

9. Marsaglia's `Ultra`, which we obtained by anonymous ftp from `nic.funit.fi`, directory `/pub/msdos/science/math/fsultra`, file fsultra.zip. This is a "subtract-with-borrow" generator with a range of 2^{32} [22]. Marsaglia offers a C-language code as well as various assembly-language codes, for those users interested in runtime efficiency at the expense of code portability. We use only his C-language code, for portability and to maintain a little pressure on compiler developers and language designers: Marsaglia's basic subtract-with-borrow operation might someday be recognized as an idiom, and then translated into an efficient sequence of machine instructions, by a future C-language compiler. In the near future, I expect to recommend the `Ultra` generator for general use. It certainly has a good pedigree: it was developed by a highly respected researcher in the field of random number generation. However, I lack feedback from "satisfied users." A notable advantage of `Ultra` is that, unlike Press and Teukolsky's `ran2`, it can be rewritten into a form that is efficiently executed on a pipelined workstation or a vector supercomputer. James Russell, an MIT undergraduate, recently wrote such a vectorizable code. We have not included Russell's code in the present distribution due to a lack of time to make a thorough test of its correctness, and due to our uncertainty over the patent status of the algorithm underlying `Ultra`.

0. a linear additive generator (`long state=seed1; state += seed2`) capable of generating any 32-bit constant or any arithmetic sequence. This generator is included only for testing purposes: short sequences of its output will fail almost any test of pseudorandomness.

2.9 Describing an RNG

To support reporting and reproducing randomized experiments, we provide a string-valued function `describe_rng( rng, rngid )`. This function writes a one-line description of the RNG algorithm and its current state into the character string `rngid`. The value of `rngid` is returned, so that `printf(describe_rng(rng, rngid))` is a convenient way to print the `rng` state to the standard output.

Here is a sample output from `describe_rng()`:

```
RNG state identifier is (1, 0: 2, 0; 1047, 1; 1024, 0)
```

This string indicates that we are using RNG algorithm number `1` of Section 2.8, the 4.3bsd Unix `random()`. The second digit indicates that the recommended (unbiased but slower) algorithm is employed by `mrandom` to calculate restricted-range integers. See Section 2.2. The initial seeds supplied to our RNG were `2` and `0`; in this case, the second 32-bit seed was ignored, because `random()` uses only a single seed. The RNG code has been called `1047` + `1` $\times 10^9$ times since initialization. The buffer size is `1024`, and the final `0` indicates that this RNG's output is not being split.

Note that the state identifier is both concise and complete (for RNGs with at most two 32-bit seeds), freeing most users of `mrandom` from the need to keep copies of state tables in their experimental records.

Initializing, Saving, and Restarting RNGs In general, I believe that users should extend the sequence of an existing RNG, whenever possible, instead of seeding a new one. I suggest this methodology because it is so difficult to properly seed an RNG when performing multiple program runs during a single experiment. Where, after all, can you get truly random seeds? To use the time of day, or a process ID, is to invite disaster in the form of subtle experimental correlations or, catastrophically, cyclic experimental repetitions due to the inadvertent reuse of a seed.

The `mrandom` package makes it easy to save and restart RNGs. The call `save_rng(rng, filename)` writes a complete state table to a file. The call `restart_rng(rng, filename)` reads a RNG state file into the RNGdata area `rng`. Before calling `restart_rng`, the user must supply sufficient space for an RNG state table, either with a storage declaration of the form `RNGdata rng`, or with a memory allocation statement such as `rng = malloc(sizeof (RNGdata))`.

For portability and ease of use, the RNG statefiles are written in ASCII. The statefile contains enough information to completely specify the RNG algorithm, its current state, its buffer size and split. It also specifies the initial seeds, the number of times it has been cycled, and the next output that should be produced by this RNG algorithm on this state table. The "next output" value provides a modicum of protection against a corrupted statefile. It also gives a convenient mechanism for spot-checking the portability of the `mrandom` package across operating systems, compilers, and computers.

Sometimes it is necessary either to reconstruct an RNG state from a state identifier, or to seed a new RNG. The function `init_rng(rng, alg, mralg, seed, count1, count2, bufsize)` provides a convenient way to do this. The `alg` parameter should have a value between 0 and 9, to indicate which RNG algorithm is desired; see the list in Section 2.8. The `seed` parameter is a pointer to an array of seeds, of an appropriate size for the RNG algorithm selected: see the User's Manual for details. The `count` parameters are used for initialization and "cycling" of the generator before control is returned to the calling program. Note that if `count2` is non-zero, the underlying RNG will be called billions of times!

Perhaps the easiest way to initialize a new RNG is to invoke the `mrtest` program described in Section 3.2. Using `mrtest`'s command-line arguments, you can "fire up" any RNG you like, with any seed. You may also efficiently "cycle" this generator any number of times, if you like, with another command-line argument. Before exit, `mrtest` writes its final RNG state to a state file named `RNGstatefile`.

By the way, I believe that great care should be taken when initializing any large-table RNGs. At a bare minimum, such generators should be cycled a large number of times before use in an application. For example, Knuth's RNG initialization code (`mrandom` algorithm #2) cycles his generator 165 times after using a fairly simple method of "stretching" the single-word seed into initial values for a 55-word state table. Cycling a generator in this way is a (possibly fruitless) attempt to get to a more-or-less "random" point in the RNG output sequence, rather than one of the relatively few ($\leq 2^{32}$) points reachable by the initialization routine. Since the period of an RNG with a 55-word large state table is, typically, much larger than 2^{55}, we would have to "cycle" our RNG more than $2^{55-32} \approx 8$ million times on average, to reach a random point even if our 2^{32} initialization points were uniformly distributed in an RNG period of 2^{55}. Since large-table RNG periods are typically much longer than 2^{55}, and since there is no reason to *assume* that our initialization routine has the necessary "uniformity" properties, the cycling process should be viewed as one that "can't hurt, and it just might help."

Another possible method of initializing a large-table RNG, not supported by `mrandom`, would use a previously-initialized RNG as a source of values to be used for initializing the new table.

Perhaps a later version of `mrandom` will support such a generalized "bootstrap initialization" algorithm for any RNG code designed to accept such user-randomized state tables. Note that few RNG codes will operate correctly on arbitrary state tables. Most impose constraints. For example, Knuth's code (`mrandom` algorithm #2) requires a state table whose entries are not all even.

2.10 Killing an RNG

Because `mrandom` 3.0 allocates internal buffers for each RNG being used, we provide a method for de-allocating such buffers. This procedure, `kill_rng`, takes a single parameter: a pointer to an initialized RNG. By the way, if you wish to re-initialize an active RNG, you should first kill that RNG, then call `init_rng`. See Section 5.4.1 of the User's manual for more details.

3 Testing Randomness with `mrtest`

The final component of the `mrandom` package is a set of routines for testing the "randomness" of a pseudorandom sequence. These routines are not in particularly elegant form. Originally, I had merely planned on writing a short code to illustrate how to call `mrandom`, and to test the correctness of the compilation. I called this routine `mrtest.c`.

As so often happens in software development, the `mrtest` code "took on a life of its own." I found myself getting more and more interested in the topic of testing RNGs. I discovered a dearth of public-domain code for such tests. Even after restricting my attention to the class of tests based on Pearson's chi-square statistic, I found it difficult to develop a code to compute an accurate confidence interval for this test statistic, in all cases of interest. Finally, I found that mathematical statisticians, for the most part, had little respect for a test based solely on a confidence interval for one's test statistic under the null hypothesis. In response to this last finding, I have begun developing a robust, feasibly-computable, decision-theoretic approach to RNG testing. This project is far from fruition, however, and is nowhere reflected in the current release of `mrandom`, so I will not discuss it further in this report.

3.1 Notation

The analyses in this report are focussed on the equiprobable-category form of the Pearson chi-square test statistic, used in several of the tests suggested by Marsaglia [21]. My notation is Knuthian, *i.e.* as concise and mnemonic as possible. Accordingly, I use $x^2()$ to denote the Pearson function; only where necessary will I specify the value of k and N in the subscript forms $x^2_k()$ or $x^2_{k,N}()$:

$$x^2(s) \stackrel{\text{def}}{=} \sum_i \frac{(n_i - N/k)^2}{N/k} \tag{1}$$

where

$$s \stackrel{\text{def}}{=} (n_1, n_2, ..., n_k), \qquad n_i > 0 \tag{2}$$

is a k-vector of non-negative integers, and

$$N \stackrel{\text{def}}{=} \sum_i n_i \tag{3}$$

I use an s, rather than something like $\vec{n}$, as the argument of the Pearson function $x^2()$, because I will refer to different possible values $(s_1, s_2, t, ...)$ of the experimental "state." Also, since the Knuthian summation operator $\sum_i$ might be unfamiliar to some, please note that it is rarely necessary to write explicit bounds such as $\sum_{1 \le i \le k}$. Sometimes, however, it is very convenient to exploit the commutative property of addition, writing set-theoretic specifiers for a summand range, for example using $\sum_{s:n_1=0}$ for a sum over all states s with $n_1 = 0$.

In general, I use upper case to denote a random variate, or, with some abuse of notation, to denote a distribution. The sole exception to this convention is N, introduced both in conformance with a standard reference [13] and to avoid confusion with the components n_i of a state s. I give my variates mnemonic names wherever possible. Hence S is a random variate distributed as a symmetric multinomial, with density function $d_S()$:

$$d_S(s) \stackrel{\text{def}}{=} Pr[S = s] \stackrel{\text{def}}{=} \frac{N!}{k^N \prod_i n_i!} \tag{4}$$

Occasionally I find it necessary to specify k, or both k and N, as in S_k or $S_{k,N}$.

A general multinomial variate is denoted with M, or more rarely, with an explicit list of defining parameters $M_{k,N,p_1,p_2,...,p_k}$:

$$d_M(s) \stackrel{\text{def}}{=} Pr[M_{k,N,p_1,p_2,...,p_k} = s] \stackrel{\text{def}}{=} N! \prod_i \left(\frac{p_i^{n_i}}{n_i!} \right) \tag{5}$$

A unit normal variate is, as is customary, U:

$$d_U(u) \stackrel{\text{def}}{=} Pr[U = u] \stackrel{\text{def}}{=} \frac{1}{\sqrt{2\pi}} e^{-u^2/2}, \qquad -\infty < u < +\infty \tag{6}$$

The general form of the Pearson statistic can be written $x^2_M()$, as a shorthand for the more specific form $x^2_{k,N,p_1,p_2,...,p_k}()$. Note that the parameters $(p_1, p_2, ..., p_k)$ of M must be specified, or understood from context, if one is to evaluate the Pearson statistic:

$$x^2_M(s) \stackrel{\text{def}}{=} \sum_i \frac{(n_i - Np_i)^2}{Np_i} \tag{7}$$

Analogously, I write X^2_M for the distribution of the Pearson statistic under the (multinomial) hypothesis M. However, I will not use the constant m as a possible value for a random variate M, for fear of confusion with the parameter `m` appearing a call to `mrandom(m)`. Instead, I use s in such contexts, for example when defining the density function for an X^2 variate:

$$d_{X^2_M}(a) \stackrel{\text{def}}{=} Pr[X^2_M = a] \stackrel{\text{def}}{=} \sum_{s:x^2_M(s)=a} d_M(s), \qquad a \ge 0 \tag{8}$$

The continuous chi-square distribution χ^2 is sometimes confused with the discrete X^2 distribution, perhaps because a χ^2_{k-1} variate can be obtained by applying a version of the Pearson statistic $x^2_M()$ to the results of k independent samplings from a unit normal distribution. A glance at the density function for χ^2_ν should be sufficient to dispel anyone's belief that $X^2_{k,N} \approx \chi^2_{k-1}$ for any finite N:

$$f_{\chi^2_\nu}(a) \stackrel{\text{def}}{=} \frac{a^{(\nu-2)/2}}{2^{\nu/2}\Gamma\left(\frac{\nu}{2}\right)} e^{-a/2}, \qquad a > 0, \qquad \nu > 0 \tag{9}$$

The statistical tests in `mrtest.c` are based on the "frequentist" concept of computing tail probabilities such as $Pr[X^2 \leq z]$. This notation suggests that we are testing whether a random variate X^2 is no larger than a fixed value z; by the conventions outlined above, we may assume that the test statistic is the equiprobable Pearson's $x^2()$, but we must be more specific. Accordingly, let

$$Pr[X^2 \leq z] \stackrel{\text{def}}{=} Pr[x^2(S) \leq z] \stackrel{\text{def}}{=} \sum_{s:x^2(s)\leq z} d_S(s) \tag{10}$$

denote the probability that an $x^2_{k,N}()$ test statistic is no larger than some real-valued constant z, for an experiment whose result vector s is distributed as the symmetric multinomial $S_{k,N}$, in a context where the values of k and N are fixed. If a non-symmetric hypothesis H is under active consideration, I write the tail probability as $Pr[x^2(H) \leq z]$. (Note that the density function $d_H()$ for any specific alternative H to the symmetric null hypothesis S is not necessarily a multinomial $d_M()$, especially in a test sensitive to sequential correlations in an RNG's output. Also note that I have somewhat arbitrarily chosen to define a cumulative on a discrete distribution D in terms of $Pr[D \leq y]$ rather than $Pr[D < y]$.)

Considerable care must be taken on a computer, when calculating and manipulating tail probabilities, to avoid underflow and roundoff errors. In particular, when z is very large, a tail probability $Pr[X^2 \leq z] \approx 1$. In such cases, we will not have much accuracy in a code that computes an approximation to $Pr[X^2 > z]$ by calculating $1 - Pr[X^2 \leq z]$. The problem is that a floating-point number on a computer is stored as a (mantissa, exponent) pair, implying that $1 - x$ will not be represented accurately when x is very small in absolute magnitude.

Fortunately, it is easy to find a much better number representation for tail probabilities. The idea is to compute with variables whose values have the units of a "normalized standard deviation" for the distribution being analyzed; this idea is computationally and analytically very attractive when the distribution is more-or-less normal. A large positive value is interpreted as a tail probability very near to 1. A large negative value is a tail probability very near to 0. To state this formally, I transform a cumulative probability p into a normalized standard deviation a by evaluating the quantile-normal function `a = qnorm(p)` in the statistical computation language `S`. Johnson and Kotz [13] use the algebraic notation U_p, but I prefer to avoid subscripted arguments, thus

$$q_U \stackrel{\text{def}}{=} \Phi^{-1} \tag{11}$$

Here, I have defined the quantile-normal $q_U(p)$ in terms of its functional inverse, the cumulative standard normal distribution Φ:

$$\Phi(a) \stackrel{\text{def}}{=} Pr[U \leq a] \stackrel{\text{def}}{=} \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{-u^2/2}\, du \tag{12}$$

Much of the analysis in this report is devoted to finding a good approximation to the u-cumulative of the X^2 statistic, *i.e.* its tail probability expressed in units of normalized standard deviations:

$$u_{X^2}(y) \stackrel{\text{def}}{=} q_U(Pr[X^2 \leq y]) \stackrel{\text{def}}{=} q_U\left(\sum_{s:x^2(s)\leq y} d_S(s)\right) \tag{13}$$

The u-cumulative of χ^2 is useful in my analysis:

$$u_{\chi^2}(y) \stackrel{\text{def}}{=} q_U\left(\int_0^y f_{\chi^2_\nu}(a)\, da\right), \qquad y \geq 0 \tag{14}$$

The α-point of a distribution D is the value y such that $Pr[D \leq y] = \alpha$. In our transformed representation, we can obtain an α-point, for any fixed α, by solving for y in

$$\alpha = \Phi(u_D(y)) \tag{15}$$

In coding `mrtest`, I found little need to compute α-points. Instead, I avoid functional inversion (and minimize computational error) by interpreting a chi-squared test result $y = x^2(s)$ in the following way. If $u_{X^2}(y) \geq q_U(1 - \alpha/2)$, the test fails on the upper tail of a two-tailed test of size α. If, instead, $u_{X^2}(y) < q_U(\alpha/2)$, the test fails on the lower tail. I will justify this test methodology in Section 3.4.

3.2 Tests Implemented in `mrtest`

The current version of `mrtest` allows the user to specify, with the `-S` command-line option, any RNG algorithm supported by `mrandom`, its initial seed, and the number of times the RNG algorithm should be cycled before its output is tested. When the testing is complete, the RNG state is written into a file named `RNGstatefile`. If the `-S` option is not specified, `mrtest` will restart the RNG specified in `RNGstatefile`. Complete details of the `-S` option are described in the `man` page for `mrtest`, provided with the `mrandom` software distribution.

The user may specify the number of RNG outputs to be tested in the command line, for example, `mrtest 1000` will test the next 1000 outputs of the current RNG in various ways, as outlined below.

It is possible to "skip over" elements in the RNG output sequence, using the `-d` command line option. For example, `mrtest -d2` will test every third element in the output sequence. The letter `d` was chosen for this option, to suggest the mnemonic "discard 2."

The RNG outputs are, by default, tested in their "raw" state. To test their properties after conversion to a restricted-range integer, we provide the command-line options `-m`, `-M`, `-f`, and `-p`. The `-m` and `-M` options specify the range of the RNG, in decimal or binary-exponential form. Only one of these should be used in a single command line. For example, `mrandom -m100 1000` will test a thousand pseudorandom integers in the range 0 to 99, and `mrandom -M16` will test integers in the range 0 to $2^{16} - 1$.

The `-f` and `-p` options can be used to select alternative methods of producing restricted-range integers. The default, if neither `-f` nor `-p` is specified, is the unbiased method. The "fast" method `(int (dxrandom() * m)` is selected with `-f`, and the "poor" method `random() % m` is selected with `-p`. Neither of these options is available if you have compiled `mrtest` with the `-DVECTORIZED` compile-time flag. The `makefile` provided with the `mrandom` package names its vectorized executable `mrtestv`. This executable runs significantly faster than the unvectorized one, because it takes advantage of the vectorized RNG calls in `mrandom`.

The remaining command line options are `-q`, `-t`, and `-e`. The `-q` option, for "quiet," doesn't print out the RNG outputs as they are produced. I imagine that most RNG testing will be done with the `-q` flag enabled. The `-t` flag suppresses most RNG testing; this option is provided to allow more accurate timing of the RNG generation process. Finally, the `-e` flag causes `mrtest` to echo its command line arguments.

Summarizing the discussion above, the command line arguments of `mrtest` specify a pseudorandom sequence of N integers in the range 0 to $m - 1$. These integers are subjected to the following filters and tests.

1. The maximum value is printed.

2. The number n_{i+1} of integers equal to i is counted. The statistic $X_1 = x^2_{m,N}(n_1, n_2, ...n_m)$ is then printed and analyzed for significance. This is an equidistribution test.

3. A count is made of the number $n_{i,j}$ of times an integer equal to i is immediately followed by an integer equal to j. The RNG sequence is assumed to be "circular," so that its last element is "followed" by its first, giving us N (non-independent) observations to classify. The test statistic is $X_2 = x^2_{m^2,N}() - X_1$, where X_1 is the equidistribution test statistic defined above. Under the null hypothesis, *i.e.* if the RNG outputs are independent and uniformly-distributed in `0..m-1`, X_2 is distributed asymptotically as χ^2 with $m^2 - m$ degrees of freedom [21]. My routine analyzes the statistic as though it were distributed as $X^2_{m^2-m+1,N}$. This is a test of pairwise correlation.

4. If $m \bmod 8$ is equal to 0, an "overlapping 3-tuple correlation test" is made on the $8 \times 8 \times 8$ possible patterns that can occur in the least-significant three bits of three successive integers. The test statistic is $X_3 = x^2_{512,N}(s_3) - x^2_{64,N}(s_2)$, where s_3 are the counts in our $8 \times 8 \times 8$ contingency table and s_2 are the counts in an 8×8 contingency table for overlapping 2-tuples. The statistic X_3 is asymptotically χ^2 with $8^3 - 8^2 = 448$ degrees of freedom under the null hypothesis [21]. My routine analyzes X_3 as though it were distributed as $X^2_{449,N}$.

5. A count is made of the number of integers less than $m/2$; the resulting $x^2_{2,N,p_1,p_2}()$ statistic (binomially distributed) is printed and analyzed for significance as an $X^2_{2,N}$ variate, where $p_1 = \lceil \frac{m}{2} \rceil / m$ and $p_2 = 1 - p_1 = \lfloor \frac{m}{2} \rfloor / m$. I call this the "most-significant bit" test with some misgivings, because this name is somewhat misleading when m is not an integral power of 2.

6. Unless $m = 2$, 4, or 5, a count is made of the number of integers whose mod-3 residue is equal to 0, 1, and 2; the resulting $x^2_{3,N,p_1,p_2,p_3}()$ statistic is printed an analyzed for significance as an $X^2_{3,N}$ variate, where $p_1 = \lceil \frac{m}{3} \rceil / m$, $p_3 = \lfloor \frac{m}{3} \rfloor / m$, and $p_2 = 1 - p_1 - p_3$. I call this the "mod-3" test.

These tests are more general than may appear at first. Using a command line option to change the range m of the pseudorandom integers, it is possible to perform 2-tuple and 3-tuple correlations on almost any subrange of bits in the "raw" pseudorandom output. Also, by using the skip option `-d`, one can test for long-range correlations.

In tests 3 through 6, the distribution of the test statistic under the null hypothesis is not precisely equal to the distribution of an equiprobable Pearson's chi-square statistic on a symmetric multinomial. In the 3-tuple correlation test, for example, if $m \bmod 3$ is not equal to zero, the frequency counts (n_1, n_2, n_3) do not have expectations of exactly $(N/3, N/3, N/3)$. The asymmetry becomes larger as m decreases. For all $m \geq 6$, however, the magnitude of Hoel's corrections [13, p. 286] are small enough to suggest that reasonably accurate results will be obtained with an analysis based on a symmetric multinomial. This suggestion is borne out by a few small-m experiments with my mod-3 test on RNGs: "good" RNGs pass the test, but "bad" ones fail it, implying that my code's analysis of $x^2()$ variates is sufficiently accurate in these instances. (Despite these assurances, I must admit to nervousness in making a public release of my mod-3 test code.) I have made similar "small-N" experiments with the other tests, satisfying myself that my X^2 approximations lead to informative tests.

Tests 4 and 6 will only be conducted for m in a restricted range, because we do not know how to compute a good approximation to the test statistic for all m. Another type of infeasibility is due to memory limitations. For example, our coding of the 2-tuple correlation test implements

the frequency counters with an $m \times m$ array of integers. The maximum size of this array is fixed by a compile-time constant. For this reason, a warning message will be printed, and the 2-tuple correlation test will not run, if m is greater than a thousand (in the default compilation). A possible future enhancement to `mrtest` would implement Marsaglia's overlapping pairs sparse occupancy (OPSO) test [21] in $O(N \log N)$ time and $O(N)$ space, using an offline sorting algorithm rather than our current, on-line, algorithm requiring $O(m^2)$ space. Such an offline sort is, in fact, used in our equidistribution test whenever m is larger than a million or so.

A third type of infeasibility can occur in any test: N may be "too small" for my statistical analysis routine to give valid results. The criteria for printing a "too small" message are developed in the following sections.

3.3 The Minimal-Power Criterion for N

My x^2-interpretation code `xsq.c` enforces the following, minimalistic, requirement on the power of an RNG test. It exits with an error message if N is so small that *no* hypothesis can be rejected on the lower tail of a two-tailed chi-square test of size $\alpha = 0.10$, *i.e.* if no value in the range of $x^2(S)$ lies below the 5% point of the X^2 distribution under the null hypothesis (that the RNG is "random"). The complementary constraint, requiring at least one point in $x^2(S)$ on the upper $(1 - \alpha/2)$ tail, is always satisfied if the lower-tail constraint is met.

Furthermore, my code issues a warning message if N is so small that no hypothesis can be rejected on the lower tail of a test of size 10^{-6}. As will be shown in Section 3.6, if this warning is not printed, this implies that N is large enough that my `uxsq()` approximation to the "exact" $u_{X^2}()$ interpretation of the test statistic will be reasonably accurate. Roughly speaking, the normalized standard deviations $z =$ `uxsq()` printed by my routines have an absolute error of at most 0.3 when $|z| \leq 1$ is small, and a relative error of at most 25% if $|z| > 1$. See Section 3.7 for a more thorough discussion.

If either a warning message or an error message is printed, my routine will suggest a value of N that is large enough to avoid the warning or error on this test. This value is never more than 10% larger than necessary to avoid the message: I compute it by repeatedly increasing a variable N', originally equal to N, by 10% until the violated "too small" condition becomes satisfied. This multiplication process does not take much CPU time, and it was extremely simple to code and debug. Clearly, a more carefully-written code could efficiently find the minimum N that isn't "too small." Perhaps someone, someday, will write this improvement into my `xsq.c` routine.

My minimal requirements on test power are contrary to most, if not all, statistical practice. I confine my discussion here to the classical Neyman-Pearson framework, deferring Bayesian and decision-theoretic considerations to some later date.

A Neyman-Pearson analysis of a statistical test of a hypothesis H_0 is concerned with two types of error [15, p. 164]:

I. We may wrongly reject H_0, when it is true; and

II. We may wrongly accept H_0, when it is false.

The probability of a type-I error is equal to the size of the critical region used, α. The probability β of a type-II error can only be evaluated as a function of the "alternative hypothesis" H_1 that is assumed to be true whenever H_0 is false. If we can construct such an alternative, then we define $1 - \beta$ to be the *power* of the test of the hypothesis H_0 against the alternative

hypothesis H_1. Note that a good test of H_0 against H_1 would have a power approaching 1 (few type-II errors) and a size approaching 0 (few type-I errors).

Cochran, in an admirable and frequently-cited survey article on chi-squared testing, says [6, p. 323]:

> The literature does not contain much discussion of the power function of the X^2 test. There has been little demand for this from applications, because the test is most commonly used when we do not have a clear-cut alternative in mind, and are not in a position to make computations of the power.

An oft-cited analysis of the asymptotic power of an X^2 test was made by Mann and Wald [20]; this analysis is summarized both by Cochran [6] and by Kendall and Stuart [15]. Mann and Wald show that the limiting power of the χ^2 test (against a class of H_1 approaching H_0 as N increases) is maximized when $k = O(n^{2/5})$, with the optimal N/k equal to about 6 and 8 respectively when $N = 200$, $\alpha = 0.05$ and 0.01.

The Mann-Wald analysis is beautiful, and useful in any situation in which k can be varied at will. In my application, however, k is fixed whenever one selects an RNG test. For example, if one is studying the cross-correlation of consecutive pairs of pseudorandom positive integers less than one thousand, then $k = 10^6$.

Also, even if I were willing and able to write a code that evaluates the power of an RNG test against a specific H_1, I don't know how to induce my users to formalize, and reveal, their current H_1.

Lacking the ability to modify k, and lacking a precise definition of H_1, of what use is a power calculation? One answer is supplied by Haberman [11]:

> When applied to frequency tables with small expected cell counts, Pearson chi-squared test statistics may be asymptotically inconsistent even in cases in which a satisfactory chi-squared approximation exists for the distribution under the null hypothesis. ...unless all cell probabilities are equal, it is possible to select a significance level and cell probabilities under the alternative hypothesis such that the power is less than the size of the test.

Zelterman [31] raises a similar warning of possible bias in a chi-squared test, suggesting a modification to the x^2 statistic that removes "the basic cause of large bias" [11]. Both Zelterman's correction and Haberman's analysis focus on asymmetry in the expected frequencies Np_i. Quoting Haberman [11]:

> A reasonable index to potential difficulties is provided by $h = g/\sqrt{32(k-1)}$, where $g = \sum_i (1/(Np_i) - k/N)$. Larger values of h are associated with increasing bias problems. As will become evident, values of h greater than .1 are disturbing, and values of h exceeding 1 indicate serious problems of bias.

Since $p_i = 1/k$ in most cases of interest in RNG testing, $h = 0$. Even in the slightly-asymmetric nulls that arise in both my mod-3 and MSB tests, $h < 0.01$, suggesting that the bias analyzed by Haberman will not be a serious issue.

Koehler and Larntz, comparing the Pearson statistic x_k^2 to the log-likelihood ratio statistic G^2 (see equation 30), characterize the power functions as follows [17, p. 340]:

> Monte Carlo power comparisons showed that, for the null hypothesis of symmetry, $x_k^2()$ is slightly more powerful for near alternatives. The Pearson test is decidedly

> dominant as the alternative moves toward a boundary of T_k [the simplex of all multinomial alternatives M_k] that contains a high proportion of zeros and a few relatively large probabilities. The likelihood ratio test is dominant at alternatives that lie near boundaries of T_k that contain a small proportion of near-zero probabilities and have nearly-equal probabilities in the remaining cells.

In other words, even if the chi-squared statistic is not biased, it may be either more or less powerful than other easily-computable statistics, depending upon the set of alternatives of interest. Cressie and Read, whose I^λ family includes both the chi-squared statistic x^2 and the log-likelihood ratio statistic G^2, write "Multinomial goodness-of-fit testing using statistic I^λ is best performed:

1. for any $\lambda \in [0, 1\frac{1}{2}]$, when no knowledge of the type of alternative is available;
2. for $\lambda = 0$ (i.e. G^2) if the alternative is thought to be dipped; but the approximate percentage point should be determined by matching moments;
3. for $\lambda = 1$ (i.e. x^2), if the alternative is thought to be peaked, where the approximate percentage point can be found from the chi-squared tables [7, p. 462]."

Thus, a future version of `mrandom` may offer one or more alternatives to the Pearson test statistic.

Finally, power calculations can be used in Neyman-Pearson testing to adjust the relative sizes of the two rejection areas in a two-tailed test. Since `mrtest` is oblivious to its user's H_1, I believe it most appropriate to use equal-area testing, accepting only x^2 scores such that $\alpha/2 < \Phi(u_{X^2}(x^2)) \leq 1 - \alpha/2$ in a test of size α.

3.4 Evidence Against Randomness

I believe we must resign ourselves to non-classical testing of RNGs, because we can gain only "one-sided evidence" about randomness. By one-sided evidence, I mean that we can hope to stumble upon a statistical test that convincingly demonstrates a source's nonrandomness, but we cannot hope to certify that a source is indistinguishable from a random source. This one-sided notion of evidence appears to be standard practice in RNG testing [16, 21]. Perhaps it has been given a firm theoretical foundation: if you know of one, please let me know.

It is possible, at least in principle, to pose the RNG testing problem in more classical terms. For example, it is computationally feasible to decide whether a source is well-described by any member of various classes of randomized algorithmic methods [4, 23]. The class need not include only a fixed number of alternative hypotheses. Indeed the number of alternatives can grow, albeit slowly, with the amount of data being analyzed. However, since I can only afford to run any RNG-testing process for a finite amount of time, a universal RNG-testing method is not necessarily useful in practice. In a bounded amount of time, a universal method will test only a fixed number of alternatives from a restricted class; furthermore, it is not at all clear that these alternatives are very interesting ones.

For these reasons, I believe that any universal test should be evaluated on the merits of a couple of its finite prefix tests, say the tests that are conducted in its first CPU-minute and its first CPU-year of runtime. I would ask for a demonstration that a "practically universal RNG test" would give one-sided evidence, in a reasonable amount of time, against all the forms of non-randomness detected by the other RNG tests commonly administered [16, 21]. Finally, I would not consider a finite run of any universal test as giving useful evidence "for randomness,"

without seeing a proof that my randomized application will give correct results if is provided a pseudorandom source not in the class of "nonrandom sources" detectable by that run of that universal test.

For the reasons outlined above, the RNG tests in the `mrandom` package are designed with a one-sided testing methodology. If an RNG fails a single test dramatically enough, it is unsuitable for use in any application depending on the "random" property under test. If, however, an RNG passes all the tests we have run on it, we can say only that we haven't discovered a way in which it is non-random.

Summarizing this train of thought, the problem of analyzing the significance of a given x^2 value can be reduced to the problem of analyzing the distribution of $x^2()$ under the hypothesis of a "random" source. Since all categories are (almost) equally likely in the RNG tests implemented by `mrandom`, I sought analytical results for tail probabilities of $x^2(S)$, for S a symmetric multinomial. Finally, I sought results for the extreme tails, in order to support inferences of the following form: if an RNG fails a test with $\alpha = 10^{-6}$, this can be considered evidence of non-randomness even if this RNG has already passed a similar test thousands of times.

This inference method will, no doubt, seem strange to any statistician who believes that my program "should" have access to all previous test results. Indeed, a later version of `mrandom` might maintain a database of test results, or at least provide some support for a sequential test procedure "with memory." For the present, however, I am not willing to write such a complicated code.

I thus offer rudimentary support only for the following, memory-free, sequential test for evidence of non-randomness. Unix-literate users can write a shell script that repeatedly calls `mrtest`, with command-line options specifying what RNG should be tested, how many RNG outputs should be tested, *etc.* (See the next section for details.) The shell loop should be exited whenever `mrtest` prints either

```
THIS RNG IS FAULTY!  Confidence level p > 1-1.0e-9
```

or

```
This rng is faulty!  Confidence level p > 1-1.0e-6
```

The validity of this "memoryless" inference depends on an assumption that no one will allow such a shell script to run for more than a day (= 86400 seconds), and that each repetition of the script will require at least a few seconds of elapsed time.

A more sophisticated user would introduce a little memory into the decision procedure. The idea is to write a loop that terminates "successfully" (*i.e.* with inconclusive results regarding the randomness of the RNG under test) after a dozen iterations, if `mrtest` does not produce any output containing the word `faulty` in either upper or lower case. To support such use, `mrtest` prints

```
This rng may be faulty.  Confidence level p > 1-1.0e-3
```

if any computed $x()$ statistic has a value that is below the cumulative 0.0005 point, or above the cumulative 0.9995 point, of the $x^2(S)$ distribution.

Any statistically-sophisticated user is encouraged to contact the author with alternative proposals for RNG testing methodologies of low memory complexity.

3.5 Limitations of the χ^2 Approximation to X^2

In my first coding attempt, I naively assumed that the distribution of my $x^2_{k,N}()$ values would be well-approximated by the χ^2_ν distribution with $\nu = k - 1$ degrees of freedom, in all cases of interest. Knuth's warning about this approximation seemed distant:

> A common rule of thumb is to take N large enough so that eadch of the expected values Np_i is five or more; preferably, however, take N much larger than this, to get a more powerful test [16, p. 42].

Since I was only interested in the symmetric case, this "rule of thumb" implies that, if $N \geq 5k$, the chi-squared approximation would be valid for my analysis.

Reading a little farther in Knuth, I found that the case $k = 2$ is a counterexample to the "rule of thumb." Indeed, it is not hard to show that $k = 3$ is also a counterexample: an X^2 statistic with $N = 5k$, for $k < 4$, is poorly approximated by χ^2_{k-1}. For example, $Pr[X^2_{3,15} = 0] > 0.04$, but $f_{\chi^2_2}(0) = 0$.

After preliminary experimentation with my RNG tests, I discovered that the $N \geq 5k$ rule of thumb suggests collecting "too much data" for very large k. I rarely found it necessary to examine $5k$ RNG outputs before it became obvious whether or not this RNG would fail a k-category test with equally-likely categories, for $k > 10^4$. I was particularly interested in equidistribution tests over the whole of an RNG's range, for which $k \approx 10^9$. It takes many minutes, even on a high-speed workstation, to gather and analyze $N = 5k$ samples of an RNG's output for such large k. Why couldn't I test the value of my x^2 statistic after collecting just a few hundred thousand samples? This reduces to a problem in mathematical statistics: what is the distribution of $x^2_{k,N}(S)$ for $N \ll k$, when k is large?

Further search through my bookshelves and my computer programs uncovered many repetitions and minor variants of the $N \geq 5k$ rule of thumb, with few *caveats* or explanations. For example, the documentation of the relevant Splus [28] routine, `prop.test`, states "At the very minimum, all (estimated) expected counts of successes or failures should be at least five." Sedgewick [27] suggests a slightly stronger rule, stating that $N \geq 10k$ will assure validity. However, Sedgewick's rule is contradicted by Knuth's example with $k = 2$, $N = 21$, $p_1 = 1/4$, $p_2 = 3/4$; and in any event is not helpful for my question about $N \ll k$ for very large k. Kendall and Stuart pronounce [15, p. 440]

> A rough rule which is commonly used is that no expected frequency (Np_{0i}) should be less than 5. There seems to be no general theoretical basis for this rule...

A thorough literature search revealed some tables of exact values for X^2 for various cases of small k and N [10, 30], and several proposed improvements on the X^2 statistic [29, 31]. Wise states that "the expected frequencies p_i can be quite small provided they are nearly equal" [29]. Haberman says "In the special case of all p_i equal, it is well known that very small values of N/k lead to satisfactory approximation" of X^2 by χ^2, "especially for large k" [11].

At long last, I found a paper by Koehler and Larntz stating [17, p. 343]

> Clearly, for the null hypothesis of symmetry, the chi-squared approximation for the Pearson statistic is quite adequate at the .05 and .01 nominal levels for expected frequencies as low as .25 when $k \geq 3$, $N \geq 10$, $N^2/k \geq 10$... Hence the Pearson goodness-of-fit test based on the traditional chi-squared approximation is preferred for the test of symmetry.

The analysis of the following section can be read as an extension of the Koehler and Larntz investigation, covering a few more cases: $N/k < 0.25$, $k = 2$, and/or $\alpha < .01$. Also, my analysis is for two-tailed tests of fit, as is appropriate for RNG testing.

3.6 The u-cumulative of $x^2(S)$

My analysis starts with Johnson and Kotz's equation 11.2.4(12), defining an approximation to the density of a multinomial, neglecting terms in $N^{1/2}$:

$$P(n_1, n_2, ..., n_k) \approx \frac{1}{\sqrt{2\pi N \prod_i p_i}} e^{-X^2/2} \tag{16}$$

where X^2 is given by equation (7).

To paraphrase Johnson and Kotz[13, p. 287]: Although the "accuracy of the X^2 approximation to the multinomial distribution increases as $\min\{Np_1, Np_2, ..., Np_k\}$ increases and decreases with increasing k, "it is not easy to give a simple summary" of the theoretical investigations in this area. Furthermore, "it is important to keep in mind that there are *two* approximations to be considered:

1. the accuracy of equation (16), and
2. the accuracy of the χ^2 distribution as an approximation to the distribution of X^2."

I have made only anecdotal investigation of the inaccuracies stemming from the first approximation. The issue here is that the $x^2()$ statistic is biased, even for a symmetric null, against some point alternatives. This can be stated formally as follows. When comparing two test results $z_1 = x^2(s_1)$ and $z_2 = x^2(s_2)$, the smaller value is not necessarily the less probable one: $z_1 < z_2$ does not imply $d_S(s_1) < d_S(s_2)$. My preliminary investigations tell me that this pointwise bias can be very large even for moderate N. Indeed, I see no reason to believe that $d_{X^2}(x^2(s))/d_S(s)$ is bounded, either from above or below by any positive constant, for any fixed k as $N \to \infty$.

Still, I believe the bias of the chi-squared approximation does not greatly affect the utility of the RNG tests in `mrandom`, since these are based on a cumulative probability function. I formalize this belief as follows:

Conjecture 1 *For any k, N, and null hypothesis M satisfying the test-feasibility conditions of* `mrtest` *(see Section 3.2 and Table 1), and for any test result s, let $z = u_{X^2}(x^2(s))$. If $|z| < 10$, then $|z - u_M(s)| < 1$, where $u_M(s)$ is the u-cumulative multinomial defined by*

$$u_M(s) \stackrel{\text{def}}{=} q_U \left(\sum_{t: d_M(t) \le d_M(s)} d_M(t) \right) \tag{17}$$

Turning to the second problem identified by Johnson and Kotz, see Figures 3 and 4. These illustrate the errors encountered for various k when $N = 10$ and $N = 30$, when using $u_{\chi^2_{k-1}}$ to approximate $u_{X^2_k}$.

The points on the upper curve in these plots are $u_{\chi^2}(x^2(s)+k/N)$ values: the necessity for this continuity correction of k/N is discussed below. The crosses on the lower curve are approximate $u_{X^2}(x^2(s))$ values obtained by computer evaluation of the `uxsq()` function developed later in this section. The abscissa is an exact calculation of the cumulative multinomial, however this is

defined with an ordering on states s differing from that of equation (17) above. The left-to-right ordering is by increasing values of $x^2(s)$, with ties broken in order of increasing $d_S(s)$. More precisely, the x-coordinate $x(s)$ of a point s is given by

$$x(s) = u_{X^2,S}(x^2(s)) \stackrel{\text{def}}{=} u_{X^2}(x^2(s)) \; - \sum_{\substack{t:x^2(t)=x^2(s) \\ d_S(t)<d_S(s)}} d_S(t) \tag{18}$$

Summarizing the explanation above, these plots contain points at $(x(s), u_{\chi^2}(x^2(s) + k/N))$ and crosses at $(x(s), \texttt{uxsq}(x^2(s)))$. The straight lines in the plot are defined by $y = x$; all the plotted points and crosses would lie on these lines if there were no approximation errors.

Due to the symmetrical null, all "degenerate" multinomial states s (*i.e.* those s whose $(n_1, n_2, ..., n_k)$ are permutations of each other) have the same u-value. Thus there are exactly as many points in a plot as there are partitions $(k_0, k_1, ..., k_N)$ of N elements, that is, solutions of the two equations $\sum_i ik_i = N$ and $\sum_i k_i = k$. I wrote my calculation and plotting routines in S [2], using a few of the enhancements in Splus [28].

If I had not applied a continuity correction to the u_{χ^2} values, the left-most point in most of my plots would be off-scale. The necessity for the continuity correction is clear: whenever N divides k, the value $z = 0$ has a finite probability, and therefore a finite (if sometimes small) value for its associated normalized normal standard deviate. The continuity correction of k/N used in my plots has approximately the right value, at least for these particular k and N. Cochran suggests using a piecewise linear function as a continuity correction, in situations where the "next possible" value of x^2 is known; he attributes this idea to F. Yates [6]. Good, Gover, and Mitchell suggest using a slightly different piecewise linear approximation; they also discuss the simple additive correction of k/N used in `mrtest` [10].

It is not hard to establish that k/N is a good "first term" in an asymptotically-accurate continuity correction, even for the $N \ll k$ cases of interest in this report. A good starting point is Wallace's definite bound [14] on $u_{\chi^2}(z)$,

$$w(z) \le u_{\chi^2}(z) \le w_2(z) + w_2(z)^{-1} \max\left\{0, \frac{1}{c_\nu e^{1/(9\nu)}} - 1\right\} \tag{19}$$

where

$$\nu = k - 1$$

$$w(z) = \sqrt{z - \nu - \nu \log\left(\frac{z}{\nu}\right)}$$

$$w_2(z) = w(z) + \frac{1}{3}\sqrt{\frac{2}{\nu}}$$

and

$$c_\nu = \frac{2\sqrt{\pi(\nu)^\nu}}{\Gamma(\frac{\nu}{2})\sqrt{\nu(2e)^\nu}}$$

Asymptotic analysis of $w(z)$ and $w_2(z)$ reveals that $u_{\chi^2_{k-1}}(z + k/N) = \sqrt{k/N} + o(\sqrt{k/N})$, for all $z \le k/N$. For larger z, a similar analysis shows that our continuity correction has little effect on the value of $u_{\chi^2}()$. The next paragraph establishes that $u_{X^2}(z) = \sqrt{k/N} + o(\sqrt{k/N})$ for any $z \le k/N$. Thus either of $w(z + k/N)$ or $w_2(z + k/N)$ is a good approximation to $u_{X^2}(z)$ for $z \le k/N$. Furthermore, we can (and will) use this continuity-corrected form of Wallace's bound as a basis for an approximation to $u_{X^2}(z)$ for larger z.

Figure 3: Approximated X^2 (dots on lower curve) and χ^2 (crosses on upper curve) distributions versus the multinomial distribution for $n = 10$, $k = 3, 5, 10, 20, 40, 100$.

Figure 4: Approximated X^2 (dots on lower curve) and χ^2 (crosses on upper curve) distributions versus the multinomial distribution for $n = 30$, $k = 3, 5, 10, 20, 40, 100$.

To establish $u_{X^2}(z) = \sqrt{k/N} + o(\sqrt{k/N})$ for any $z \leq k/N$, it suffices to consider only the minimum-possible value v in the range of $x^2(s)$. The minimizing s must have $n_i = \lfloor N/k \rfloor$ or $n_i = \lceil N/k \rceil$ for all i, implying that

$$v = \frac{(N \bmod k)(k - (N \bmod k))}{N} \tag{20}$$

A bit of combinatorics, along with Stirling's approximation, eastablishes the desired result: if $v \leq k/N$, the total density $d_S(s)$ of all x^2-minimizing states s is $\sqrt{k/N} + o(\sqrt{k/N})$.

As noted by Good, Gover and Mitchell, the range of $x^2_{k,N}()$ consists of values z such that $z = v + 2ck/N$, for integral $c \geq 0$ [10]. Furthermore, I observe relatively few "gaps": almost all values of c are possible on the left tail, except when $k < 5$. I conclude that the continuity correction is important only on the extreme left tail, and even then, only when N is approximately equal to a multiple of k. In all other cases, the additive correction of k/N amounts to a trapezoidal-rule approximation of the discrete X^2 distribution by the continuous χ^2 distribution.

Correcting the Right Tail Approximation. Judging from the upper curves of Figures 3 and 4, the u-cumulative of the X^2 distribution is not well-approximated by the u-cumulative of the χ^2 distribution. I develop a right-tail correction by finding an upper bound on $u_{X^2}()$, using combinatorial analysis. My `uxsq()` routine computes the minimum of this bound and Wallace's upper bound.

Throughout this analysis, I assume that all multinomial probability parameters p_i are equal: $p_i = 1/k$. This allows me to concentrate on the case of greatest interest in RNG testing. More importantly, it simplifies the mathematics considerably, since all configurations of N distinct items in k bins are equally likely. To transform a configuration count into a probability, I need merely divide by k^N.

First I consider configurations in which some of the bins are empty. Let k' be the number of non-empty bins. By a straightforward convexity argument, the chi-square statistic $x^2(s)$ for such states s is minimized for multinomial states such that all n_i are equal to one of three values: 0, $\lfloor N/k' \rfloor$, or $\lceil N/k' \rceil$. More precisely, $x^2(s) \geq kN/k' - N$, with equality possible when N divides k'. Solving for k' in the case of equality, I obtain $k' = k/(1 + x^2(s)/N)$. Note that the minimum possible value of $x^2(s)$, for s with k' non-empty bins, is a decreasing function of k'.

This line of reasoning leads to the following lower bound

$$Pr[X^2 \geq x] \geq \sum_{1 \leq j \leq k/(1+x/N)} \binom{k}{j} T(N,j)/k^N \tag{21}$$

where $T(N,j)$ is the number of ways of distributing j distinct items in N bins, with at least one item in each bin. By solving the recurrence $T(N,j) = j^N + \sum_{1<i<j} \binom{j}{i} T(N, j-i)$ with basis $T(N,1) = 1$, either directly or with an inclusion-exclusion argument, I find

$$T(N,j) = \sum_{0 \leq i \leq j} \binom{j}{i} (-1)^i (j-i)^N \tag{22}$$

Note that $T(N,j) = j^N - (j^2/2)(j-1)^N + O((j-2)^N)$, with $T(N,j) \approx j^N$ for $j > N$.

The lower bound (21) is tight at both extrema of the X^2 distribution. At the extreme right tail, *i.e.* when $k' = k/(1 + x/N) = 1$, there is just one relevant multinomial "state" (with degeneracy k). At the extreme left tail, we have counted all multinomial states exactly once.

My best approximation to date for the upper tail of $Pr[X^2 \geq x]$ is obtained by finding the largest term in the sum (21) by the following stochastic argument. I set

$$k_{\text{eff}} = \frac{k}{1 + x/N} \left(1 - e^{-(x+N)/k}\right) \tag{23}$$

so that k_{eff} is the expected number of non-empty bins when distributing N items into $k/(1+x/N)$ bins. When $j > k_{\text{eff}}$, by the stochastic estimation above, the value of $\binom{k}{j}T(N,j)$ is monotone decreasing in j. When $j < k_{\text{eff}}$, it is monotone increasing in j. Thus the sum in (25) can be roughly approximated by its largest term:

$$\sum_{1 \leq j \leq k_{\text{eff}}} \binom{k}{j} T(N,j)/k^N \approx \binom{k}{k_{\text{eff}}} T(N,k_{\text{eff}})/k^N \tag{24}$$

It is easy to show that $k_{\text{eff}} < N$ for $x \geq k$, implying that $T(N,k_{\text{eff}}) \approx (k_{\text{eff}})^N$, establishing

$$Pr[X^2 \geq x] \gtrsim \binom{k}{k_{\text{eff}}} \left(\frac{k_{\text{eff}}}{k}\right)^n \tag{25}$$

I transform my lower-bounding probability approximation (25) into an upper bound on the associated normal standard variate by applying the upper-tail normal quantile approximation

$$q_U(-2\log p) \approx \sqrt{(-2\log p) - \log\left((-2\log p) - \log(2\pi)\right) - \log(2\pi)} \tag{26}$$

to a Stirling approximation of (25):

$$l(x) \stackrel{\text{def}}{=} (2N - 2k_{\text{eff}} - 1)\log(k/k_{\text{eff}}) - 2k_{\text{eff}} + \log(2\pi) \tag{27}$$

Here I introduce the notation $l(x)$ as an approximation to $(-2\log Pr[X^2 \geq x])$. The right-hand side of (26) is monotone increasing in $(-2\log p)$ for $(-2\log p) > 2.83$, leading to my approximation

$$\texttt{uxsq}(x) \stackrel{\text{def}}{=} \begin{cases} \min\left\{w_2(x + k/N), q_U(l(x))\right\}, & \text{if } l(x) > 2.83 \\ w_2(x + k/N), & \text{otherwise} \end{cases} \tag{28}$$

Note that I use Wallace's $w_2()$ function (19) as an approximation to u_{χ^2}, because it is more accurate than his $w()$ function on the right tail [14]. I omit his term in w_2^{-1} because it is negligible on the right tail, and (in my experiments) this simpler form gives better results in the center of the distribution than either his upper or lower bound. If more accuracy near the distribution's center were required for some application, I would recommend using the Wilson-Hilferty approximation [14] in this region.

I am not optimistic about finding good bounds on the accuracy of (28) for $N < k$, even in the limiting case of large k. However, in the next section I report the results of a pseudorandom experiment demonstrating that (28) has reasonable accuracy near the center of the distribution when $N \geq 3\sqrt{k}$ for $k = 2^7$, 2^{19} and 2^{31}.

3.7 On the Sample Size Required for X^2 Testing

The analysis and experimental data of the previous section suggests the following classification of "computational" errors in X^2 statistical analysis:

1. Discretization error,
2. "Missing" left tail,
3. Error in the uxsq() approximation to the actual distribution on the right tail, and
4. Inaccurate "continuity correction" on the extreme left tail.

The magnitude of the discretization error can be easily bounded, at least near the center of the distribution, except in the case of very small k. As noted previously, the possible values for $x^2_{k,N}()$ are separated by $2k/N$, with larger "gaps" infrequent for $k > 5$ except on the extreme right tail. The standard deviation of X^2 is $\sqrt{2k-1}$, suggesting that almost all "gaps" in the range of $u_{X^2}(s)$ should be of size $2k/(N\sqrt{2k-1})$.

The plots of Figures 3 and 4 confirm this prediction of gap size, for small k. Also, the experimental Q-Q plots of Figure 5 confirm it near the center of the distribution, for several large k when $N = 3\sqrt{k}$ and $N = 7\sqrt{k}$. Note that the observed x^2 values in the top plots of Figure 4 are spaced at approximately 0.5 normalized standard deviations, in close conformance to $2k/(3\sqrt{k}\sqrt{2k-1}) \approx 0.47$. The bottom plots of Figure 4 shows $N = 7\sqrt{k}$ for the same values of k; here we observe a spacing of $2k/(7\sqrt{k}\sqrt{2k-1}) \approx 0.20$.

Thus, if one is satisfied (as I am) with rough estimates of the deviation from expectation, $N \geq 3\sqrt{k}$ is a sufficient sample size to overcome the inherent discretization error of the chi-squared statistic. However, to obtain `uxsq()` scores that are precise to within 0.1 standard deviation, a sample size of $N \geq 14\sqrt{k}$ is necessary to overcome the discretization error near the center of the distribution.

For small k, as will be shown below, N must be much larger than $14\sqrt{k}$ in order to avoid other "computational" errors when interpreting x^2 statistics.

Missing left tail. In many applications of x^2 testing, right-tail-only tests are appropriate. In tests of random number generators, however, one should be very interested to know if one's RNG outputs are "too equidistributed" to be congruent with a hypothesis of a uniform distribution.

It is not hard to see that the left tail "fills out" much more slowly than the right tail, on an X^2 distribution. Straightforward analysis reveals that the left tail contains the relatively high-probability states near the center of the multinomial distribution. Experimental plots, such as those in Figures 3 and 4, show the same thing: the left tail on such plots rarely goes below -3 normalized standard deviations, but there exist states s with very large u-cumulative values even when N is small.

As noted in equation (20), the smallest value v for an x^2 statistic is $v = (N \bmod k)(k - (N \bmod k))/N$. Note that v is *not* monotone decreasing in N. For analytic convenience, I use the following approximation to v', the monotone-decreasing envelope of v:

$$v' = \max\left\{k - N,\ \frac{k^2}{4\max(N,k)}\right\} \tag{29}$$

My code evaluates $z_{\min} = \texttt{uxsq}(v', k, N)$ to see if $z_{\min} > -1.645$, the 5% point of a normal deviate. I also test $z_{\min} > -4.892$, the 0.5×10^{-6} point of a normal deviate. If $z_{\min}$ is not rejected by these left-tail tests, my code issues an error message in the first case, and a warning message in the second case. My code also calculates and prints an $N_{\min}$ that would satisfy the violated constraint, finding an (approximate) minimum value for $N_{\min}$ by multiplying the given n by successive powers of 1.1 until the constraint is satisfied.

Figure 5: Quantile-quantile plots of uxsq() versus the normal standard deviate, obtained from 1000 experimental runs with $k = 2^7$, 2^{19}, 2^{31} and $N = 3\sqrt{k}$, $7\sqrt{k}$.

k	$N_{\min}$ for $\alpha = 0.05$	$N_{\min}$ for $\alpha = 0.5 \times 10^{-6}$
2	725	2.6×10^{13}
3	51	6.2×10^{6}
4	24	5.5×10^{4}
5	17	5.6×10^{3}
6	13	1600
10	11	182
20	$3\sqrt{k}$	60
30	$3\sqrt{k}$	48
39	$3\sqrt{k}$	49
$40 \le k \le 2^{15}$	$3\sqrt{k}$	$7\sqrt{k}$
$k > 2^{15}$	$3\sqrt{k}$	$7.2\sqrt{k}$

Table 1: Minimum n for a two-tailed X^2 test of the hypothesis of a symmetric multinomial distribution of n elements into k bins, for two confidence intervals.

It seems difficult to solve for $N_{\min}$ in $\texttt{uxsq}(v', k, N_{\min}) = c$, for $c = -1.645$ or -4.892, or indeed for any other value. Expanding the logarithm and square root in the definition of `uxsq()` with a Taylor series, however, it is easy to show that $N_{\min} = (-c\sqrt{2}+O(1))\sqrt{k}$. For $c = -1.645$ this gives, approximately, $N \ge 3\sqrt{k}$; judging from my code's calculations, this form is satisfied for all $k > 10$. For $c = -4.892$, we have $c\sqrt{2} = 6.92$, implying that $N \ge 7\sqrt{k}$ may be adequate for large k. This is indeed the case for $50 \le k \le 2^{15}$, but, according to my code's calculations, some "low order terms" are still appreciably large for some $k > 2^{15}$. I find that $N \ge 7.2\sqrt{k}$ for $k > 40$ is sufficient to guarantee the existence of an X-squared outcome below the $p < 0.5 \times 10^{-6}$ point of my `uxsq()` approximation.

For small k, the values of $N_{\min}$ rise rapidly, and my $N \ge 3\sqrt{k}$ or $7.2\sqrt{k}$ rules-of-thumb no longer apply. My recommendations are summarized in Table 1.

Error in `uxsq()`, on the extreme right tail. This analytic error is, in principle, avoidable: we should seek a better approximation to the X^2 distribution than that afforded by the `uxsq()` approximation. As shown in Figures 3 and 4, `uxsq()` is a significant improvement over the χ^2 approximation on the extreme upper tail when $n \ll k$. Nonetheless, the lower-rightmost plot of Figure 5 indicates that the `uxsq()` approximation has perhaps a 25% relative error for $k = 2^{31}$, $N = 7\sqrt{k}$, when analyzing points more than two normalized standard deviations above the mean. (An alternative explanation is that Figure 5 demonstrates some non-uniform behavior in `random()`'s full-scale 31-bit output. Still, I know from my small-N plots of Figures 3 and 4 that `uxsq()` has significant error in the mid-right tail.)

The middle and left plots on the bottom line of Figure 5, like the plots of Figures 3 and 4, indicate that the `uxsq()` approximation is accurate to within 0.2 standard deviations (i.e., to within the discretization error) for $k \le 2^{19}$, $N = 7\sqrt{k}$, for all experimental outcomes such that $-2 < \texttt{uxsq()} < 2$.

Inaccurate continuity correction on the extreme left tail. This type of analytic error may be difficult to measure experimentally, for large N, since extremal outcomes are (by definition) infrequent. For small N, however, note that all the lower-tail points in Figures 3 and 4 lie within a small fraction of a standard deviation from the "perfect-fit" line, except for the cases (top-left, bottom-middle and bottom-right plots of Figure 3) in which the lower extremum v of

Figure 6: The log-likelihood ratio statistic versus the multinomial distribution for $N = 30$, $k = 5, 20, 100$.

the X^2 curve is less than -1.5 standard deviations from the mean. In such cases, my code would print an error message, asking for larger N, because the v' value (see equation (29)) is too close to the mean.

I tentatively conclude that $N \geq 3\sqrt{k}$ is sufficient for two-tailed testing of X^2 variates with a 90% confidence interval, with results accurate to within 0.5 standard deviations, if $k > 10$. For smaller k, my program will print an appropriate lower bound on N, which may be much larger than this. If results accurate to within 0.2 standard deviations are desired for any $\alpha > 10^{-3}$, the large-k rule is $N \geq 7.2\sqrt{k}$. This rule also allows two-tailed testing for any $\alpha > 10^{-6}$, however upper-tail violations may be over-reported due to the `uxsq()` errors in this regime.

3.8 Other Test Statistics

The log-likelihood ratio statistic,

$$G^2() \stackrel{\text{def}}{=} 2 \sum_{1 \leq i \leq k} n_i \log\left(\frac{n_i}{Np_i}\right) \tag{30}$$

is sometimes suggested as an improvement to the x^2 statistic. Like $x^2_{k,N}$, under the null hypothesis $G^2()$ is distributed as χ^2_{k-1} in the limit of large N. I have briefly investigated this statistic, concluding that it deviates markedly from χ^2 almost everywhere when $N < k$. For example, see the plots of Figure 6: the log-likelihood ratio shows excellent conformity with χ^2 when $k = 5$ and $N = 30$, but huge errors when $k = 100$ and $N = 30$. I thus concur with Cressie and Read's statement that the percentage points of the G^2 statistic should not be computed from the χ^2 tables [7, p. 462]. In contrast, Figure 4 shows that the continuity-corrected $u_{\chi^2_{99}}(x^2(s)+k/N)$ is a good approximation to $u_{X^2_{100,30}}(x^2(s))$, everywhere except for those s lying outside 3 normalized standard deviations of the mean.

Despite the difficulty of interpreting a $G^2()$ test statistic, especially in the sparse case, a future version of `mrtest` might calculate the $G^2()$ statistic for every RNG test it runs. Perhaps it could interpret the statistic accurately, possibly using a transform method [1] to closely approximate

u_{G^2}, u_{X^2}, and even u_M for all the k and N of interest in RNG testing. Alternatively, we might develop a general Monte Carlo method for evaluating these u-cumulatives.

Many test statistics other than $x^2()$ and G^2 have been proposed. I am most intrigued by the E-test and the B-test [12]. The test statistics are, respectively, $\max_i n_i$ and $\min_i n_i$. Their distributions under the null hypothesis appear to be tractable, and I know of some randomized algorithms whose correctness proofs depend on an equidistribution hypothesis expressed in terms of $\max_i n_i$.

4 Summary

We offer a standardized C-language programming interface for all random number generators. This should greatly ease the programming nuisance, and possibility for the introduction of bugs, when switching RNGs to validate results, when generating integers in a large range, when running multifaceted experiments requiring multiple starts (or better, restarts) of an RNG, and when documenting the RNGs and RNG states used in an experimental run.

Our X^2 analysis routine is interesting in its own right. It differs markedly from the commonly employed chi-squared approximation in the sparse and symmetric null hypothesis (n items uniformly distributed among k bins) typical of RNG test procedures. The tests implemented in `mrtest.c` are able to distinguish the flawed RNGs still in common use from the better ones in our package, for n as small as $3\sqrt{k}$ for $k > 10$. Our X^2-based tests of randomness are thus both useful and computationally feasible in situations (*e.g.* $k = 10^9$, $N = 10^5$) far removed from those recommended by the traditional "rule of thumb," $N \geq 5k$, for X^2 testing.

Acknowledgements

I profited from several discussions with Profs. Barry James and Kang James of UMD, who pointed me to the standard references on statistical analysis, Johnson&Kotz and Kendall&Stuart. H. Chernoff patiently corrected fundamental errors in my statistical vocabulary and conceptual understanding; he suggested several references and a method for improving my X^2 approximation under the null; and he motivated me to start developing a decision-theoretic RNG test procedure to replace the naive p-value testing currently embodied in `mrandom`. I also benefited from an over-the-net discussion of RNGs with Eugene D. Brooks of Lawrence Livermore Labs.

Jesse Chisholm and Jon Bentley provided RNG source codes. Ajay Shah entered `mrandom` in his `netlib` archive: file `pub/C-numanal`, available by anonymous ftp from `usc.edu`. Ignatios Souvatzis and Liz Johnson tested `mrandom 2.0`. Using this version, Profs. Bill Knight and Greg Shannon of Indiana U. discovered a flaw in the least-significant bit of the output of the lagged-Fibonnacci generators in both 4.3bsd Unix `random()` and the Knuth/Bentley generator `lprand()` [3].

Robert Plotkin, an MIT undergraduate, developed `mrandom 3.0` as part of his BS thesis research. This version made it easier to install new RNGs, and it contains additional features to allow parallelization. Aubrey Jaffer made several useful suggestions during our design process.

References

[1] J. Baglivo, D. Oliver, and M. Pagano. Methods for exact goodness-of-fit tests. *Journal of*

the American Statistical Association, 82(418):464–469, 1992.

[2] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language*. Wadsworth and Brooks/Cole, 1988.

[3] J. L. Bentley. The software exploratorium: Some random thoughts. *UNIX Review*, April 1992.

[4] M. Blum and O. Goldreich. Towards a computational theory of statistical tests (extended abstract). In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 406–416. IEEE, October 1992.

[5] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, 1987. ISBN 0-387-96467-3.

[6] W. G. Cochran. The χ^2 test of goodness of fit. *Annals of Mathematical Statistics*, 23:315–345, 1952.

[7] N. Cressie and T. R. Read. Multinomial goodness-of-fit tests. *Journal of the Royal Statistical Society, Series B*, 46(3):440–464, 1984.

[8] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–4, December 7 1992.

[9] J. E. Gentle. Portability considerations for random number generation. In W. F. Eddy, editor, *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface*, pages 158–164. Springer-Verlag, Berlin and New York, 1981.

[10] I. Good, T. Gover, and G. Mitchell. Exact distributions for X^2 and for the likelihood-ratio statistic for the equiprobable multinomial distribution. *Journal of the American Statistical Association*, 65(329):267–282, 1970.

[11] S. H. Haberman. A warning on the use of chi-squared statistics with frequency tables with small expected cell counts. *Journal of the American Statistical Association*, 83(402):555–560, 1988.

[12] R. Holt, R. Dudley, D. Gao, and L. Pakula. Handbook and "tables" of classic probabilities. Manuscript, 1992.

[13] N. L. Johnson and S. Kotz. *Distributions in Statistics: Discrete Distributions*. Houghton Mifflin, 1969.

[14] N. L. Johnson and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions, Volume 1*. Houghton Mifflin, 1970.

[15] M. G. Kendall and A. Stuart. *The Advanced Theory of Statistics, Volume 2*. Charles Griffin and Company, 1967. Second Edition.

[16] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981. Second Edition.

[17] K. J. Koehler and K. Larntz. An empirical investigation of goodness-of-fit statistics for sparse multinomials. *Journal of the American Statistical Association*, 75(370):336–344, 1980.

[18] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–774, June 1988.

[19] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, Mar. 1991.

[20] H. Mann and A. Wald. On the choice of the number of class intervals in the application of the chi square test. *Annals of Mathematical Statistics*, 13:306–317, 1942.

[21] G. Marsaglia. A current view of random number generators. In L. Billard, editor, *Computer Science and Statistics: The Interface*, pages 3–10. Elsevier Science Publishers, 1985.

[22] G. Marsaglia and A. Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1(3):462–480, 1991.

[23] U. M. Maurer. A universal statistical test for random bit generators. *J. Cryptology*, 5:89–105, 1992.

[24] J. F. Monahan. Accuracy in random number generation. *Mathematics of Computation*, 45(172):559–568, Oct. 1985.

[25] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[26] W. H. Press and S. A. Teukolsky. Portable random number generators. *Computers in Physics*, 6(5):522–524, Sep/Oct 1992.

[27] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[28] StatSci, Seattle WA. *S-Plus Reference Manual Version 3.0*, September, 1991.

[29] M. Wise. Multinomial probabilities and the χ^2 and X^2 distributions. *Biometrika*, 50:145–154, 1963.

[30] M. Wise. A complete multinomial distribution compared with the X^2 approximation and an improvement to it. *Biometrika*, 51:277–281, 1964.

[31] D. Zelterman. Goodness-of-fit tests for large sparse multinomial distributions. *Journal of the American Statistical Association*, 82(398):624–629, 1987.