Die*Harder:*

A Gnu Public License Random Number Tester

by

Robert G. Brown

Duke University Physics Department
Durham, NC 27708-0305
rgb@phy.duke.edu

Copyright Notice
Copyright Robert G. Brown 2006
Current version: 3.31.1

Notice

This is documentation for the *Dieharder* random number generator test suite. As the project itself is uses a Gnu Public License (GPL), the documentation uses an Open Publication License (OPL). The precise terms of the OPL can be read in an appendix at the end of the manual, and should be adhered to if you wish to make changes to this work.

The current snapshot of this documentation is available for free online at:

http://www.phy.duke.edu/∼rgb/General/dieharder.php

It will be made available in an inexpensive print version (probably via Lulu press) as soon as it is in a sufficiently polished and complete state.

As a "live" document describing a dynamic and sophisticated tool, this book no doubt has errors great and small and may or may not be up to date (and hence correctly describe the latest release of *Dieharder*) at any given time. I apologize for this – my only excuse is that I'm very busy and it's all I can do to keep up with the bug reports, suggestions for extensions, and my own plans for it, and sometimes documentation takes a back seat to progress. It is, of course, an open source project and any reader who is so motivated is welcome to join in.

I cherish good-hearted communication from *Dieharder* users pointing out errors or suggesting new content (and have in the past done my best to implement many such corrections or suggestions).

Open source projects of this sort generate useful collaborations. Although many people have contributed to *Dieharder* at this point with corrections, new tests, new generators to test, and permission to use their code, my greatest debt is to my co-developers and partners, Dirk Eddelbeuttel (who maintains the ready-to-install Debian version of *Dieharder* as well as the R interface) and David Bauer, who has contributed numerous fixes to critical components (such as the ks-test program that is at the heart of the way *Dieharder* works with many of the tests incorporated from their descriptions in diehard and other sources), the "gold standard" AES and Threefish RNGs that are of great use in testing *Dieharder* , and many new tests.

Contents

1 Introduction **9**

2 Testing Random Number Generators **15**

3 Evaluating p-values **19**

3.1 Xtest – A Single Expected Value 19

3.2 Vtest – A Vector of Expected Values 19

3.2.1 Kolmogorov-Smirnov Test 22

3.3 The Test Histogram 22

4 Diehard **25**

4.1 The Original Diehard 25

4.2 The Dieharder Modifications 27

5 Dieharder's Modular Test Structure **29**

6 Dieharder Extensions **33**

6.1 STS Tests 36

6.2 New Tests 37

6.3 Future (Proposed or Planned) Tests 40

7 Results for Selected Generators **43**

7.0.1 A Good Generator: mt19937_1999 43

7.0.2 Comments 72

7.1 A Bad Generator: randu 75

7.2 An Ugly Generator: slatec 101

8 Conclusions **127**

A License Terms **129**

A.1 General Terms 129

A.2 OPEN PUBLICATION LICENSE Draft v0.4, 8 June 1999 129

Chapter 1

Introduction

Random numbers are of tremendous value in many computational contexts. In importance sampling Monte Carlo, random numbers permit the sampling of the relevant part of an extremely large (high-dimensional) phase space in order to determine (for example) the thermal equilibrium properties of a physical system near a critical point. In games, random numbers ensure a unique and fair (in the sense of being unbiased) playing experience. Random numbers play a critical role in cryptography, which is more or less the fundamental basis or *sine qua non* of Internet commerce. Random numbers are of considerable interest to mathematicians and statisticians in contexts that range from the purely theoretical to the very much applied.

There is, alas, a *fundamental problem* with this, and several related sub-problems. The fundamental problem is that it is not possible to *generate* truly random numbers by means of any mathematical algorithm. The very term "random number generator" (RNG) is a mathematical or computational oxymoron.

Even in physics, sources of true randomness are rare. There is a very, very old argument about whether even quantum experiments produce results that are truly *random* at a fundamental level or whether experimental results in quantum theory that produce seemingly random outcomes reflect the *entropy* inherent in the measuring apparatus. This is a non-trivial problem with no simple or obviously true answer even today, since it is fundamentally connected to whether the Universe is open or closed. Both relativity theory and the Generalized Master Equation that is perhaps the most general way of describing the quantum measurement process of an open system embedded in a closed quantum universe suggest that what appears to be irreversibility and randomness in laboratory experiments is due to a projection of the *stationary* quantum description of the universe onto the much smaller quantum description of the "system" that is supposed to produce the random result (such as photon emission due to spontaneous decay of an excited atom into the ground state).

The "randomness" of the apparent result follows from taking a statistical trace over the excluded degrees of freedom, which introduces what amounts to a random phase

approximation that washes out the actual correlations inherent in the extended fully correlated state. Focussing on issues of "hidden variables" within any give quantum subsystem obscures the actual problem, which is *strictly* the impossibility of treating *both* the quantum subsystem being studied *and* the (necessarily classical) measuring apparatus (which is really the rest of the quantum mechanical universe) on an equal quantum mechanical footing. If one does, all trace of randomness disappears as the quantum time evolution operator for the complete system is fully deterministic.

Ultimately, it seems to be difficult to differentiate true randomness in physical processes from mere entropy, a lack of knowledge of some aspect or another of the system. Only by systematically analyzing a series of experimental results for "randomness" can one make a judgement on whether or not the underlying process is truly random, or merely *unpredictable*.

Note well that unpredictable and random are often used as synonyms, but they are not really the same thing. A thing may be unpredictable due to entropy – our lack of the data required to make it predictable. Examples of this sort of randomness abound in *classical* statistical mechanics or the theory of deterministic chaos. We will therefore leave the question about whether any given physical process is in fact random open, a thing to be *experimentally addressed* by applying tests for randomness, and not a fundamental given.

For this reason, purists often refer to software-based RNGs as *pseudo*-random number generators to emphasize the fact that the numbers they produce are not, in fact, "random". As one can see from the discussion above, hardware-based RNGs are equally susceptible to being "pseudo" in their randomness or (however unpredictable they might be) they may well have a distribution that is not perfectly flat and hence does not conform to the requirements of an ideal RNG. At the very least they are as likely to need to be subjected to randomness tests as software generators and should not be assumed to be "truly random".

The purpose of *Dieharder* is to provide a suite of tests, as systematic as possible, to which "random number generators" of all sorts can be subjected. For this reason we will, for brevity's sake, omit the word "pseudo" when discussing RNGs but it should nevertheless be understood.

Another problem associated with random numbers in the context of modern computing is that numerical simulations can consume a *lot* of e.g. random uniform deviates, random unsigned integers, random bits, random numbers of all sorts. Simulations on a large compute cluster can consume close to Avogadro's number of uniform deviates in a single extended computation over the course of months to years. Over such a long sequence, problems can emerge even with generators that appear to pass many tests that sample only a few millions of random numbers (less than a billion bits, say). Many random number generators are in fact state-periodic and repeat a single sequence after a certain number of returns. Older generators often had a very short period. This meant that simulations that consumed more random numbers than this period in fact reproduced the same sample sequence over and over again instead of generating the independent,

identically distributed (*iid*) samples that the author of the simulation probably intended.

A related issue is associated with the dimensionality of the correlation. Many generators produce numbers that are subtly patterned (e.g. distributed on hyperplanes) but only in a space of high dimensionality. A number of tests only reveal a lack of randomness by constructing a statistic that measures the non-uniformity of the distribution of random coordinate N-tuplets in an N dimensional space. This non-uniformity can only be resolved when the space begins to be filled with points at some density. Unfortunately, the number of points required to *fill* such a space scales like the power of the dimension, meaning that it is very difficult to resolve this particular kind of correlation by filling a space of more than a very few dimensions.

For all of these reasons, the development and implementation of tests for the randomness of number sequences produced by various RNGs with real or imagined virtues is an important job in statistical computation and simulation theory. For roughly a decade, the most often cited test suite of this sort is one developed by George Marsaglia known as the "Diehard Battery of Tests of Randomness"[**?**]. Indeed, a random number generator has not been thought to be strong unless it "passes Diehard" – it has become the defining test of randomness, as it were.

This reputation is not undeserved. Diehard contains a number of tests which test for very subtle non-randomness – correlations or patterns in the numbers produced – from the bit-sequence level to the level of distributions of uniform deviates. It has not proven *easy* for RNGs to pass Diehard, which has made it a relatively strong and lasting suite of tests. Even so-called "truly random" sources such as hardware RNGs based on e.g. thermal noise, entropy, and other supposedly "random" electromechanical or even quantum mechanical processes have been demonstrated to contain subtle non-random patterning by virtue of failing Diehard.

One *weakness* of Diehard has been its relative lack of good tests for bitlevel randomness and cryptographic strength. This motivated the development, by the National Institute of Standards and Technology (NIST) of the *Statistical Test Suite* (STS): a Diehard-like collection of tests of the bitlevel randomness of bit sequences produced by RNGs[**?**]. There is small bit of redundancy with Diehard – both include binary rank tests, for example – but by in large the tests represent an extension of the methodology utilized extensively in Diehard to look for specific kinds of bitlevel correlations.

In addition to these two well-known suites of tests, there are a number of other tests that have been described in the literature or implemented in code in various contexts. Perhaps the best-known remaining source of such tests is in Knuth's *The Art of Programming*[**?**], where he devotes an entire section to both the generation and testing of random numbers. Some of the Diehard and STS tests are described here, for example.

A second weakness in Diehard has been its lack of parametric variability. It has been used as a *standard* for RNGs to a *fault*, rather than being viewed as a tool for exploring the properties of RNGs in a systematic way. Anyone who actually works with any of the RNG testers to any significant extent, however, knows that the quality of a RNG is not such a cut and dried issue. A generator that is, in fact, weak can easily "pass Diehard"

(or at least, pass any given test in Diehard) by virtue of producing p-values that are not less than 0.01 (or whatever else one determines the cut-off for failure to be). Of course a *good* RNG produces such a value one in a hundred trials, just as a *bad* RNG might well produce p-values greater than 0.01 98 out of 100 times for any given test size and still, ultimately be bad.

To put it another way, although many tests in the Diehard suite are quite *sensitive* and are capable of demonstrating a lack of randomness in generators with particular kinds of internal correlations, it lacks the power of *clearly discriminating* the failures because in order to increase the discrimination of a test one usually has to increase sample sizes for the individual tests themselves and impose a Kolmogorov-Smirnov test on the distribution of p-values that results from many *independent* runs of the test to determine whether or not it is uniform. This is clearly demonstrated below – parameterizing Diehard (where possible) and increasing its power of discrimination is a primary motivation of this work.

The STS suite publication describes this quite elegantly, although it still falls short when it comes to implementing its tests with a clear mechanism for systematically improving the resolution (ability to detect a given kind of correlation as a RNG failure) and discrimination (ability to clearly and unambiguously and reproducibly demonstrate that failure for any given input RNG that does, in fact, possess one of the kinds of correlation that leads to failure). A strong requirement for this sort of parametric variation to achieve discrimination is that the test suite *integrate* any software RNG being tested so that it can be freely reseeded and so that sequences of random numbers of arbitrary length can be generated. Otherwise a test may by chance miss a failure that occurs only for certain seed moduli, or may not be able to generate enough samples within a test or repeat a test enough times to be able to clearly resolve a marginal failure.

The remaining purpose of this work is to provide a readily available source code distribution of a *universal*, *modifiable* and *extensible* RNG test suite. Diehard was clearly copyrighted work of George Marsaglia, but the *licensing* of the actual program that implemented the suite (although it was openly distributed from the FSU website for close to a decade) was far from clear. STS is a government-sponsored NIST publication and is therefore explicitly in the public domain. Knuth's various tests are described in prose text but not implemented in any particular piece of code at all.

In order to achieve the goals of universality, extensibility, and modifiability, it is essential that a software implementation of a RNG test suite have a very clear *public license* that explicitly protects the right of the user to access and modify the source, and that further guarantees that modifications to the source in turn become part of the open source project from which they are derived and cannot be "co-opted" into a commercial product.

These, then, are the motivations for the creation of the *Dieharder* suite of random number tests – intended to be the Swiss Army Knife of RNG tests or (if you prefer) the "last suite you'll ever wear" as far as RNG testing is concerned. Dieharder is from the beginning a *Gnu Public Licensed* (GPL)[**?**] project and is hence guaranteed to be and remain an open source toolset. There can be no surprises in Dieharder, and for better

or for worse the code is readily available for all to inspect or *modify* as problems are discovered or as individuals wish to experiment with new things.

Dieharder contains all of the diehard tests, implemented wherever possible with variables that control the size of the sample space per test that contribute to the test's p-value, or the number of p-values that contribute to the final test on the *distribution* of p-values returned by many *independent* runs. Dieharder has as a design goal the encapsulation of all of the STS tests *as well* in the single consistent test framework. Dieharder will also implement selected tests from Knuth that thus far have failed to be implemented in either Diehard or the STS.

Finally, Dieharder implements a *timing* test (as the cost in CPU time required to generate a uniform deviate is certainly highly relevant to the process of deciding which RNG to implement in any given piece of code), various tests invented by the author to investigate specific ways a generator might fail (documented below) and has a templated interface for "user contributed" tests where, basically, anybody can add tests of their own invention in a consistent way to the suite. These latter tests clearly demonstrate the *extensibility* of the suite – it took only a few hours of programming and testing to add a simple test to the suite to serve as a template for future developers.

Dieharder is tightly integrated with the *Gnu Scientific Library* (GSL), another GPL project that provides a universal, modifiable, and extensible numerical library in open source form. In particular, the GSL contains over 60 RNGs pre-encapsulated in a single common call environment, so that code can be written that permits any of these RNGs to be used to generate random numbers in any given block of code at run time by altering the value of a single variable in the program. Routines already encapsulated include many well-known generators that fail one or more Diehard tests, as well as several that are touted as having *passed* Diehard.

As we shall see, that is a somewhat optimistic assertion – it is rather fairer to say that Diehard could only rather weakly resolve their *failure* of certain tests. The GSL also provides access to various distributions and to other functions that are essential to any random number generator – the error function or incomplete gamma function, for example – and that are often poorly implemented in code when programmed by a non-expert. A final advantage of this integration with the GSL is that the GSL random number interface is easily *extensible* – it is fairly straightforward to implement any proposed RNG algorithm inside the GSL RNG function prototypes and add new generators to the list of generators that can be selected within the common call framework by means of the runtime RNG index.

The rest of the paper is organized as follows. In the next section the general methodology for testing a RNG is briefly described, both in general and specifically as it is implemented in Dieharder to achieve its design goals. This section is deliberately written to be easy to understand by a non-expert in statistics as conceptually testing is very simple. Diehard is then reviewed in some detail, and the ways the Diehard tests are extended in Dieharder are documented. Dieharder's general program design is then described, with the goal of informing individuals who might wish either to use Dieharder

as is to test the generators already implemented in the GSL for their suitability for some purpose or to help guide individuals who wish to write their own tests or implement their own generators within its framework. A section then describes the *non*-Diehard tests thus far implemented (a selection subject to change as new tests are ported from e.g. the STS or the literature or invented and added). Finally the results of applying the test suite to a few selected RNGs are presented, demonstrating its improved power of discrimination.

Chapter 2

Testing Random Number Generators

The basic idea of testing a RNG is very simple. Choose a process that uses as input a sequence of random numbers (in the form of a stream of bits e.g. 10100101..., a stream of integers in some range e.g. 12 17 4 9 1..., a stream of uniform deviates e.g. 0.273, 0.599, 0.527, 0.981, 0.194...) and that creates as a result a number or vector of numbers that are *known* if the sequence of numbers used as inputs is, in fact, random according to some measure of randomness.

For example, if one adds t uniform deviates (double precision random numbers from the range $[0, 1)$) one expects (on average) that the mean value of the sum would be $\mu = 0.5t$. For large t, the means for many independent, identically distributed (*iid*) sums thus formed should be normally distributed (from the Central Limit Theorem, (CLT)) with a standard deviation of $\sigma = \sqrt{t/12}$ (from the properties of the uniform distribution).

Each such sum numerically generated with a RNG therefore makes up an experiment. Suppose the value of the sum for t samples is x. The *probability* of obtaining this value for the mean from a *perfect* RNG (and actual random sequence) is determined according to the CLT from the error function as:

$$p = \text{erfc}\left(\frac{|\mu - x|}{\sigma\sqrt{2}}\right) \tag{2.1}$$

This is the p-value associated with the *null hypothesis*. We *assume* that the generator is good, create a statistic based on this assumption, determine the probability of obtaining that value for the statistic if the null hypothesis is correct, and then interpret the probability as success or failure of the null hypothesis.

If the p-value is very, very low (say, less than 10^{-6}) then we are pretty safe in rejecting the null hypothesis and concluding that the RNG is "bad". We could be wrong, but the chances are a million to one against a good generator producing the observed value of p. This is really the *only* circumstance that leads to a relatively unambiguous conclusion

concerning the RNG. But suppose it isn't so close to 0. Suppose, in fact, that p for the trial is a perfectly reasonable value. What can we conclude then?

By itself the p-value from a single trial tells us little in most cases. Suppose it is 0.230. Does this mean that the RNG is good or bad? The correct answer is that it does not tell us that the RNG is likely to be bad. It is (if you prefer) insufficient evidence to reject the null hypothesis, but it is *also insufficient* to cause us to *accept* the null hypothesis as proven. That is, it is *incorrect* to assert that it means that the RNG is in fact "good" (unbiased) on the basis of this single test.

After all, suppose that we repeated the test and got 0.230 a second time, and then repeated it a third time and got 0.241, and repeated it a hundred more times and got p-values that consistently lay within 0.015 or so of 0.230! In that case we'd be very safe in concluding that the RNG was a *bad* one that (for the given value of t) *always* summed up to pretty much the same number that is distributed *incorrectly*. We might well *reject* the null hypothesis.

Suppose instead that we got 0.230, 0.001, 0.844, 0.569, 0.018, 0.970... as values for p. Once again, it is not at all obvious from looking at this whether we should conclude that the generator is good or bad. On the one hand, one of these values only occurs once in roughly 1000 trials by chance, and another occurs only one in maybe 50 trials – it seems unlikely that they'd be in a sequence of p-values. On the other hand, it isn't *that* unlikely. One in a thousand chances happen, especially given some unknown number of tries! Given *enough* tries they are nearly *certain* to happen. Did they "just happen" this time, or are they indicative of a problem in the generator? It is difficult to know for sure.

What we would like to do is take the guesswork out of our decision process. What is the probability that this particular sequence of p-values might occur *if* the underlying distribution of p-values is in fact uniform (as a *new* null hypothesis)? To answer this we apply a *Kolmogorov-Smirnov* (KS) test to the p-values observed to determine the probability of obtaining them in a random sampling of a uniform distribution. This is *itself* a p-value, but now it is a p-value that applies to the entire *series* of *iid* trials.

This testing process now gives us two parameters we can tweak to obtain an unambiguous answer – one that is very, very low, consistently – or not. We can increase t, which increases the mean value relative to σ and makes systematic deviations from the mean probability of 0.5 more obvious (but which makes a localized non-random clustering of values for *small* sample sizes *less* obvious) or we can increase the number of *iid* trials to see if the distribution of p-values for the sample size t we're already using is not uniform. In either case, once we discover a combination of t and the number of trials that consistently yields very low overall p-values (visible, as it were, as the p of the distribution of p-values of the distribution of p-values of the experiment) we can safely reject the null hypothesis. If we cannot find such a set of parameters, we are at last tentatively justified in concluding that the RNG passes our very simple test.

This does *not mean* that the null hypothesis is correct. It just means that we cannot prove it to be *incorrect* even though we worked pretty hard trying to do just that!

This is the basic idea of nearly all RNG testers. Some tests generate a single number,

normally distributed. Other tests generate a vector of numbers, and we might determine the p-value of the vector from the χ^2 distribution according to the number of degrees of freedom represented in the vector (which in many cases will be smaller than the number of actual numbers in the vector). A few might generate numbers or vectors that are *not* normally distributed (and we might have to work very hard in these cases to generate a p-value – the KS test itself is just such a case).

In all cases in Dieharder, the p-values from any small sample of *iid* tests is held to be suspect in terms of supporting the acceptance or rejection of the null hypothesis unless and until a KS test of the uniformity of the distribution of p itself yields an unambiguous p-value in a challenging application of the test method. In most cases it would be considered to be worthwhile to play with the parameters described above (number of samples, number of trials) to see if the p-value returned can be made to *consistently* exhibit failure with a very high degree of confidence, making at least the for-cause rejection of the null hypothesis a very safe bet.

There is one test in Dieharder that does not generate a p-value per se. The *bit persistence* test is a bit-level test that basically does successive exclusive-or tests of succeeding (e.g.) unsigned integers returned by a RNG. After a remarkably few trials, the result of this is a *bitmask* of all bits that did not change from the value 1 throughout the sequence. A similar process is used to identify bit positions that a value of 0 that does not change.

This test is actually quite useful (and is very fast). There are a number of generators that (for some seeds) have *less than* e.g. 32 bits that vary. In some cases the generators have fixed bits in the least significant portion of the number. in some cases they have fixed bits in the high end, or perhaps return a positive signed integer (31 bits) instead of 32. In any of these cases it is worthwhile to identify this very early on in the testing process as some of these problems will *inevitably* make the RNG fail later tests, often quite badly. If a test permits the number of significant bits in a presumed random integer to be varied or masked, one can even use the information to perform tests on the *significant* part of the numbers returned.

Chapter 3

Evaluating p-values

Tests used in *Dieharder* can produce a variety of statistics that can be used to produce a p-value

3.1 Xtest – A Single Expected Value

3.2 Vtest – A Vector of Expected Values

It is appropriate to use a Vtest to evaluate the p-value of a single trial test (consisting as usual of *tsamples iid* samples generated using a RNG presumed good according to H_0) in *Dieharder* when the test produces a related vector of statistics, such as a set of observed frequencies – the number of samples that turned out to be one of a finite list of possible discrete target values.

A classic example would be for a simulated die – generate *tsamples* random integers in in the range 1-6. For a "perfect" (unbiased) die, an H_0 die as it were, each integer should should occur with probability $P[i] = 1/6$ for $i \in [1, 6]$. One therefore expects to observe an *average* of $tsamples/6$ in each bin over many runs of *tsamples* each. Of course in any given random trial with a "perfect" die one would usually observe bin frequencies that vary somewhat from this in integer steps.

This variation can't be too great or too small. Obviously observing all 6's in a large trial ($tsamples \gg 1$) would suggest that the die was "loaded" and not truly random because it is pretty unlikely that one would roll (say) twenty sixes in a row with an unbiased die. It can happen, of course – one in about 3.66×10^{15} trials, and $tsamples = 20$ is still pretty small.

It is less obvious that observing *exactly* $tsamples/6 = 1,000,000$ in all bins over (say) $tsamples = 6,000,000$ rolls would ALSO suggest that the die was not random, because there are so many more ways for at least some fluctuation to occur compared to this very

special outcome.

The chi^2 distribution counts these possibilities once and for all for vector (binned) outcomes and determines the probability distribution of observing any given excursion from the expected value if the die is presumed to be an unbiased perfect die. From this one can determine the probability of having observed any given pattern of outcomes in a single trial subject to the null hypothesis H_0 – the p-value.

Evaluating chi^2 and p-value in a Vtest depends on the number of degrees of freedom in the vector – basically how "related" the bin events are. Generally speaking, there is always at least one constraint, since the total number of throws of the die is tsamples, which must therefore equal the sum of all the bin frequencies. The sixth frequency is is therefore not an independent quantity (or in general, the contents of the nth (last) bin is not independent of the contents of the $n-1$ bins preceding it), so the default number of degrees of freedom is at most $n-1$.

However, the number of degrees of freedom in the chi^2 distribution is tricky – it can easily be *less* than this if the expected distribution has long "tails" – bins where the expected value is approximately zero. The binned data only approaches the chi^2 distribution for bins that are have an expected value greater than (say) 10. The code below enforces this constraint, but in many tests (for example, the *Greatest Common Denominator* test) there may be a lot of *weight* aggregated in the neglected tail (of greatest common denominator frequencies for the larger factors). In these cases it is necessary to take further steps to pass in a "good" vector and not get an erroneous p-value. A common strategy is to *summing* the observed and expected values over the tail(s) of the distribution at some point where the bin frequencies are still larger than the cutoff, and turn them all into a single bin that now has a much *greater* occupancy than the cutoff.

Ultimately, the p-value is evaluated as the incomplete gamma function for the observed chi^2 and either an input number of degrees of freedom or (the default) number of bins that have occupancy greater than the cutoff (minus 1). Numerically evaluating the incomplete gamma function *correctly* (in a way that converges efficiently to the correct value for all ranges of its arguments) is actually *not trivial to do* and is often done incorrectly in homemade code. This is one place where using the GSL is highly advantageous – its routines were written and are regularly used and tested by people who know what they are doing, so its incomplete gamma function routine is relatively reliable and efficient.

Dieharder attempts to standardize as many aspects of performing a RNG test as possible, so that there are relatively few things to debug or validate. A Vtest therefore has a standardized "Vtest object" associated with it – a struct defined in Vtest.h as:

```
typedef struct {
  unsigned int nvec;  /* Length of x,y vectors */
  unsigned int ndof;  /* Number of degrees of freedom, default nvec-1
  double *x;          /* Vector of measurements */
  double *y;          /* Vector of expected values */
  double chisq;       /* Resulting Pearson's chisq */
```

```
  double pvalue;      /* Resulting p-value */
} Vtest;
```

There are advantages associated with making this data struct into an "object" of sorts that is available to all tests, but not (frankly) to the point where its contents are opaque[1] The code below thus contains simple constructor and destructor routines that can be used to allocate all the space required for a Vtest in one call, then free the allocated space in just the right order to avoid leaking memory.

This can be done by hand, of course, and some tests involve vectors of Vtests and complicated things and may even *do* some of this stuff by hand, but in general this should be avoided whereever possible and it is nearly always possible.

In summary, the strategy for a Vtest involves the following very generic steps, clearly visible in the actual code of many tests:

- Create/Allocate the Vtest struct(s) required to hold the vector of test outcomes. Note that there may be a vector of Vtests generated within a single test, if you like, if you are a skilled coder.

- Initialize the expected/target values, e.g

  ```
  for(i=0;i<nv;i++){
    vtest->y[i] = tsamples*p[i];
  }
  ```

  This can be done at any time before evaluating the trial's p-value.

- Run the trial. For example, loop *tsamples* times, generating as a result a bin index. Increment that bin.

  ```
  for(t=0;t<tsamples;t++){
    index = make_distributed_number_randomly();
    vtest->x[index]++;
  }
  ```

  Note again that there may well be some logic required to handle e.g. bin tails, evaluate the `p[i]`'s (or they may be input as permanent data from the test include file). Or the test statistic may not be a bin frequency at all but some other number for which a Pearson χ^2 is appropriate.

- Call `Vtest_eval()` to transform the test results into the trial p-value.

- As always, the trial is *repeated psamples* times to generate a *vector* of p-values. As we noted above, any given trial can generate any given p-value. If you run a trial

[1] Discussion of this point ultimately leads one into the C vs C++ wars. rgb is an unapologetic C-coder, but thinks that objects can be just lovely when they can be as opaque as you like when programming, not as opaque as the compiler designer thought they should be. 'Nuff said.

enough times, you will see very small p-values occur, very rarely. You will also see very large p-values, very rarely. In fact, you should *on average* see *all* p-values, *equally* rarely. p itself should be distributed *uniformly*. To see if this *happened* within the limits imposed by probability and reason, we subject the distribution of p to a final *Kolmogorov-Smirnov Test* that can reveal if the RNG produced results that were (on average) *too good* to be random, *too bad* to be random, or *just right* to be random[2].

3.2.1 Kolmogorov-Smirnov Test

A Kolmogorov-Smirnov (KS) test is one that computes how much an observed probability distribution differs from a hypothesized one. Of course this isn't very useful – *all* of the routines used to evaluate test statistics do precisely the same thing. Furthermore, it isn't terribly easy to turn a KS result into an actual p-value – it tends to be more sensitive to one end or the other of an empirical distribution and has other difficulties[3].

For that reason, the KS statistic for the uniform distribution is usually evaluated with the *Anderson-Darling* goodness-of-fit test. Anderson-Darling KS is used throughout *Diehard*, for example, and it is similarly used in *Dieharder* , but to a much higher degree. Note well that a final KS test on a *large* set (at least 100) of trial p-values is the *essential* last step of almost any *Dieharder* test. It is otherwise simply impossible to look at p from a single trial alone and assess whether or not the test "fails" unless that p-value is very, very low. Many of the original *Diehard* tests generated only a very few p-values (1-20) and "passed" many RNGs that in fact *Dieharder* fails with a very obvious (in retrospect) non-uniformity in the final distribution of p. Some of its tests were also flawed (for example, the operm5 test) but the flaw was only visible if one rand the test many times and studied the distribution of p with a KS test.

3.3 The Test Histogram

Although a KS test provides an objective and mathematically justified p-value for the entire test series, the human eye and human judgement are invaluable aids in the process of obtaining an unambiguous result for any test and for evaluating the quality of success or failure. For this reason *Dieharder* also presents a visible histogram of the final p-value distribution.

In the ASCII (text-based) version of *Dieharder* this histogram is necessarily rather crude – it presents binned deciles of the distribution in an autoscaling graph. Nevertheless, it makes it easy to see *why* the p-value of a test series is small. Sometimes it is obvious – because all of the p-values are near zero because the RNG egregiously fails the test in every trial. Other times it is *very subtle* – the test series produces p-values with a

[2] Think of it as "The Goldilocks Test".

[3] See for example the remarks at
http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm

slight bias towards one end or the other of the region, nearly flat, that resolves into an unambiguous failure only when the number of trials contributing p-values is increased to as many as 500 or 1000.

Here one has to judge carefully. Such an RNG isn't *very* bad with respect to the test at issue – one has to work quite hard to show that it fails at all. Many applications might be totally insensitive to the small deviations from true randomness thus revealed.

Others, however, might *not*. Modern simulations use a *lot* of random numbers and accumulate a *lot* of samples. If the statistic being sampled is "like" the one that fails the final KS test, erroneous results can be obtained.

Usually it is fairly obvious when a test is marginal or likely to fail on the basis of a mix of the histogram and final KS p-value. If the latter is low, it may mean something or it may mean nothing – visible inspection of the histogram helps one decide which. If it *might* be meaningful, usually repeating the test (possibly with a larger number of p-values used in the KS test and histogram) will usually suffice to make the failure unambiguous or (alternatively) show that the deviations in the first test were not systematic and the RNG actually does not fail the test[4].

[4]Noting that we carefully refrain from asserting that *Dieharder* is a test suite that can be *passed*. The null hypothesis, by its nature, can never be proven to be true, it can only fail to be observed to fail. In this it greatly resembles both life and science: the laws of inference generally do not permit things like the Law of Universal Gravitation to be "proven", the best that we can say is that we have yet to observe a failure. *Dieharder* is simply a numerical experimental tool that can be used empirically to develop a degree of confidence in any given RNG, not a "validation" tool that proves that any given RNG is suitable for some purpose or another.

Chapter 4

Diehard

4.1 The Original Diehard

The *Diehard Battery of Random Number Tests* consists of the following individual tests:

1. Birthdays
2. Overlapping 5 Permutations
3. 32x32 Binary Rank
4. 6x8 Binary Rank
5. Bitstream
6. Overlapping Pairs Sparse Occupance (OPSO)
7. Overlapping Quadruples Sparse Occupance (OQSO)
8. DNA
9. Count the 1s (stream)
10. Count the 1s (byte)
11. Parking Lot
12. Minimum Distance (2D Spheres)
13. 3D Spheres (minimum distance)
14. Squeeze
15. Sums
16. Runs

The tests are grouped, to some extent, in families when possible; in particular the Binary Rank tests are similar, the Bitstream, OPSO, OQSO and DNA tests are very similar, as are the Parking Lot, the Minimum Distance, and the 3d Spheres tests.

Nevertheless, one reason for the popularity of Diehard is the *diversity* of the kinds of correlations these tests reveal. They test for raw imbalances in the random numbers generated; they test for long and short distance autocorrelations; there are tests that will likely fail if a generator distributes points on 2 or 3 dimensional hyperplanes, there are tests that will fail if the generator is not random with respect to quite complex conditional patterns (such as those required to win a game of craps).

The tests are not without their weaknesses, however. One weakness is that (as implemented in Diehard) they often utilize partially overlapping sequences of numbers to increase the number of "samples" one can draw from a relatively small input *file* of random numbers. Because they strictly utilize file-based random number sources, it is not easy to generate more random numbers if the number in the file turns out not to be adequate for any given test.

Diehard has no adjustable parameters – it was written to be a kind of a "benchmark" that would give you a pass or fail outcome per test per generator, not a testing tool that could be manipulated looking for an elusive failure or trying to resolve a marginal failure.

Many of the tests in Diehard had no concluding KS test (or had a KS test based on a very small number of *iid* p-values and were hence almost as ambiguous as a single p-value would be unless the test series was run repeatedly on new files full of potential rands from the same generator.

Diehard seems more focussed on validating relatively small files full of random numbers than it is on validating RNGs per se that are capable of generating many orders of magnitude more random numbers in far less time than a file can be read in and without the overhead or hassle of storing the file.

A final criticism of the original Diehard program is that, while it was freely distributed, it was written in Fortran. Fortran is not the language of choice for programs written to run under a Unix-like operating system (such as Linux), and the code was not well structured or adequately commented *even* for fortran, making the understanding or modification of the code difficult. It has subsequently been ported to C[**?**] with somewhat better program structuring and commenting. Alas, both the original sources and the port are very ambiguous about their licensing. No explicit licensing statement occurs in the copyrighted code, and both the old diehard site at Florida State University and the new one at the Center for Information Security and Cryptography at the University of Hong Kong have (or had, in the case of FSU) distinctly commercial aspects, offering to sell one a CD-ROM with many pretested random numbers and the diehard program on it.

4.2 The Dieharder Modifications

Dieharder has been deliberately written to try to fix most of these problems with Diehard while preserving all of its tests in default forms that are at *least* as functional as they are in Diehard itself. To this end:

- All *Dieharder* Diehard tests have an outcome based on a single p-value from a KS test of the uniformity of many p-values returned by individual runs of each basic test.

- All *Dieharder* tests have an adjustable parameter controlling the number of individual test runs that contribute p-values to the final KS test (with a default value of 100, much higher than any of the Diehard tests).

- All *Dieharder* tests can optionall generate a simple histogram of these p-values so that their uniformity (or lack of it) can be *visually* assessed. The human eye is very good at identifying potentially significant patterns of deviation from uniformity, especially from several sequential runs of a test.

- Many of the basic Diehard tests in *Dieharder* that have a final test statistic that is a computable function of the number of samples now have the number of samples as an adjustable parameter. Just as in the example above, one can increase or decrease the number of samples in a test and increase or decrease the number of test results that contribute to the final KS p-value. However, some Diehard tests to not permit this kind of variation, at least without a lot more work and the risk of a loss of resolution without warning.

- All tests are integrated with GSL random number generators and use GSL functions (where possible) that are thoroughly tested and supported by experts. For example, *Dieharder* uses GSL versions of the error function, the incomplete gamma function, or to evaluate a binomial distribution of outcomes for some large space to use as a χ^2 target vector. This presumably increases the reliability and maintainability of the code, and certainly increases its speed and flexibility relative to file based input.

- File based random number input is still possible in a number of formats, although the user should be aware that the (default) use of larger numbers of samples per test and larger numbers of tests per KS p-value requires far more random numbers and therefore far larger files than Diehard. If an inadequate number of random numbers is provided in a file, then (to avoid a time-consuming crash) it is automatically rewound mid-trial (and the rewind count recorded in the trial output as a warning). This, in turn, introduces a rather obvious sort of correlation that can lead to incorrect results!

- Certain tests which had additional numbers that could be parameterized as test variables were rewritten so that those variables could be set on the command line (but still default to the Diehard defaults, of course).

- Dieharder tests are *modularized* – they are very nearly boilerplate *objects*, which makes it very easy to create new tests or work on old tests by copying or otherwise using existing tests as templates.

- All code was completely rewritten in well-commented C without direct reference to or the inclusion of either the original fortran code or any of the various attempted C ports of that code. Whereever possible the rewrite was done strictly on the basis of the prose test description. When that was not possible (because the prose description was inadequate to completely explain how to generate the test statistic) the original fortran Diehard code was examined to determine what test statistic actually was but was then implemented in original C. Tabular data and parametric data from the original code was reused in the new code, although of course it was not copied per se as a functional block of code.

- This code is packaged to be RPM installable on most Linux systems. It is also available as a compressed tar archive of the sources that is build ready on most Unix-like operating systems, subject only to the availability of the GSL on the target platform.

- The *Dieharder* code is both copyrighted and 100% Gnu Public Licensed – anyone in the world can use it, resell it, modify it, or extend it – as long as they obey the well-known terms of the license.

As one can easily see, *Dieharder* has significantly extended the parametric utility of the original Diehard program (and thereby considerably increased its ability to discriminate marginal failures of many of the tests). It has done so in a clean, easy to build, publically licensed format that should encourage the further extension of the *Dieharder* test suite.

Next, let us look at the modular program design of *Dieharder* to see how it works.

Chapter 5

Dieharder's Modular Test Structure

Diehard's program organization is very simple. There is a toplevel program shell that parses the command line and initializes variables, installs additional (user added) RNGs so that the GSL can recognize them, and executes the primary work process. That process either executes each test known to Dieharder, one at a time, in a specific order or runs through a case switch to execute a single test. In the event that all the tests are run (using the `-a` switch), most test parameters are ignored and a set of defaults are used. These standard parameters are chosen so that the tests will be "reasonably" sensitive and discriminating and hence can serve as a comparative RNG performance benchmark on the one hand and as a starting point for the parametric exploration of specific tests afterwards.

A *Dieharder* test consists of three subroutines. These test are named according to the scheme:

```
diehard\_birthday()
diehard\_birthday\_test()
help\_diehard\_birthday()
```

(collected into a single file, e.g. `diehard_birthday.c`, for the Diehard Birthday's test). These routines, together with the file `diehard_birthday.h`, and suitable (matching) prototypes and enums in the program-wide include file `dieharder.h`, constitute a complete test.

`diehard_birthday.h` contains a test struct where the test name, a short test description, and the two key default test parameters (the number of samples per test and number of test runs per KS p-value) are specified and made available to the test routines in a standarized way. This file also can contain any test-specific data in static local variables.

The toplevel routine, `diehard_birthday()`, is called from the primary work routine

executed right after startup if the test or is explicitly selected or the `-a` flag is given on the command line. It is a very simple shell for the test itself – it examines how it was started and if appropriate saves the two key test parameters and installs its internal default values for them, it allocates any required local memory used by the test (such as the vector that will hold the p-values required by the final KS test), it rewinds the test file if the test is using file input of random numbers instead of one of the generators, it prints out a standardized test header that includes the test description and the values of the common test parameters, and calls the main sampling routine. This routine calls the actual test routine `diehard_birthday_test()` which evaluates and returns a single p value and stores it in `ks_pvalue`, the vector of p values passed to the final KS test routine. When the sample routine returns, a standard test report is generated that includes a histogram of the obtained values of p, the overall p-value of the test from the final KS test, and a tentative "conclusion" concerning the RNG.

The workhorse routine, `diehard_birthday_test()`, is responsible for running the actual test a single time to generate a single p-value. It uses for this purpose built-in data (e.g. precomputed values for numbers used in the generation of the test statistic) and parameters, common test variable parameters (where possible) such as the number of samples that contribute to the test statistic or user-specified parameters from the command line, and of course a supply of random numbers from the specified RNG or file.

As described above, a very typical test uses a transformation and accumulation of the random numbers to generate a number (or vector of numbers) whose expected value (as a function of the test parameters) is known and to compare this expected value with the value "experimentally" obtained by the test run in terms of σ, the standard deviation associated with the expected value. This is then straightforward to transform into a p-value – the probability that the experimental number was obtained *if* the null hypothesis (that the RNG is in fact a good one) is correct. This probability should be uniformly distributed on the range $[0, 1)$ over many runs of the test – significant deviations from this expected distribution (especially deviations where the test p-values are uniformly very small) indicate *failure* of the RNG to support the null hypothesis.

The final routine, `help_diehard_birthday()`, is completely standardized and exists only to allow the test description to be conveniently printed in the test header or when "help" for the test is invoked on the command line.

Dieharder provides a number of utility routines to make creating a test easier. If a test generates a single test statistic, a struct can be defined for the observed value, the expected value, and the standard deviation that can be passed to a routine that transforms it into a p-value in an entirely standard way using the error function. If a test generates a vector of test statistics that are expected to be distributed according to the χ^s distribution (independently normal for each degree of freedom for some specified number of degrees of freedom, typically one or two less than the number of participating points) there exists a set of routines for creating or destroying a struct to hold e.g. the vector of expected values or experimentally obtained values, or for evaluating the p-value of the experiment from this data.

A set of routines is provided for performing bitlevel manipulations on bitstrings of specified length such as dumping a bit string to standard output so it can be visually examined, extracting a set of $n < m$ bits from a string of m bits on a ring (so that the $m-1$ bit can be thought of as wrapping around to be adjacent to the 0 bit), starting at a specified offset. These routines are invaluable in constructing bit-level tests of randomness both from Diehard and from the STS (which spends far more time investigating bit-level randomness than does Diehard). A routine is provided to extract an unpredictable (but not necessarily uncorrelated) seed from the entropy-based hardware generator provided by e.g. the Linux operating system and others like it (`/dev/random`) if available, and in general the selected software random number generator is reseeded one or more times during the course of a test as appropriate.

This behavior can be overridden by specifying a seed on the command line that is then used throughout all tests to obtain a standard and reproducible result (useful for re-validating a test after significant modifications while debugging).

Last, a simple timing harness is provided that is used to make it easy to *time* any installed RNG. There are many ways to take a bad but fast RNG and improve it by using the not terribly random numbers it generates to generate new, much more random numbers. The catch is that these methods invariably require many of the former to generate one of the latter and take more time. There is an intrinsic trade-off between the speed of a RNG (measured in how many random numbers per second one can generate) and their quality. Since the time it takes to generate a random number is an important parameter to any program design process that consumes a *lot* of random numbers (such as nearly any stochastic numerical simulation, e.g. importance sampling Monte Carlo), Dieharder permits one to search through the library of e.g. GSL random number generators and select one that is "random enough" as far as the tests are concerned but still fast enough that the computation will complete in an acceptable amount of time.

Chapter 6

Dieharder Extensions

As noted in the Introduction, Dieharder is intended to develop into a "universal" suite of RNG tests, providing a consistently controllable interface to all commonly accepted suites of tests (such as Diehard and STS), to specific tests in the literature that are not yet a standard feature of existing suites (e.g. certain tests from Knuth), and to *new* tests that might be developed to challenge RNGs in specific ways, for example in ways that might be expected to be relevant to specific random number consuming applictions.

This is an open-ended task, not one that is likely to ever be "finished". As computer power in all dimensions continues to increase, the demands on RNGs supplying e.g. numerical simulations will increase as well, and tests that were perfectly adequate to test RNGs for applications that would consume at most (say) 10^{12} uniform deviates are unlikely to still suffice as applications consume (say) 10^{18} or more uniform deviates, at least without the ability to parametrically "crank up" the rigorousness of any given test to reveal relevant flaws. Cryptographic applications that were "secure" a decade ago (given the computer power available at that time to attempt to crack them) may well not be secure a decade from now, when Moore's Law and the advent of readily available cluster computing resources can bring perhaps a million times as many cycles per second to bear on the problem of cracking the encryption.

In order to remain relevant and useful, a RNG tester being used to determine the suitability of a RNG for *any* purpose, be it gaming, simulation, or cryptography, has to be relatively simple to scale up to the new challenges presented by the changing landscape of computing.

Another feature of RNG testers that would be very desireable to those seeking to test an RNG to determine its suitability for use in some given application would be sequences of tests that validate certain statistical properties of a given RNG *systematically*. Right now it is *very difficult* to interpret the results of e.g. Diehard or many of the STS tests. If a RNG fails (say) the Birthdays test or the Overlapping 5-Permutations test when pushed to it by increasing test parameters, what does that say about the cryptographic strength of the generator? What does it say about the suitability of the RNG for gaming,

for numerical simulation, to drive a state lottery system?

It is entirely possible, after all, to *pass* some Diehard or STS tests and *fail* others, so failure in some test is not a universal predictor of the unsuitability of the RNG for all purposes. Unfortunately there is little theoretical guidance connecting failure of any given test and suitability for any given purpose.

Furthermore, there is little sense of analysis in RNG tests that might be used to rigorously provide such a theoretical connection. If one is evaluating the suitability of some functional basis to be used to expand some empirically known function, there is a wealth of methodology to help one determine its completeness and convergence properties. One can often state with complete confidence that if one keeps (say) the first five terms in the expansion then one's results will be accurate to within some predetermined fraction.

It is not similarly possible to rank RNGs as (for example) "random through the fifth order" in a series of systematically more demanding tests in some specific dimensional projection of "randomness" and thereby be able to claim with some degree of confidence that the generator will be suitable for use in Monte Carlo computations based on the Wolff cluster method[**?**], or heat bath methods[**?**], or even plain old Metropolis[**?**].

This leaves physicists who utilize these methods in theoretical studies in a bit of a quandry. There exist famous examples of "bad results" in simulation theory brought about by the use of a poor RNG, but (as the testing methodology described above makes clear) the "poverty" of a RNG is known only *ex post facto* – revealed by failure to get the correct result! This makes its quality difficult to determine in an application looking for an answer that is not already known.

One method that is used is to vary the RNG used (keeping other aspects of the computation constant) and see if the results obtained are at least *consistent* across the variation within the numerical experimental resolution of the computation(s). This gives the researcher an uneasy sort of confidence in the result – uneasy because one can easily use suites like Dieharder to demonstrate that there are tests that nearly *all* tested RNGs will fail quite consistently, including some otherwise excellent generators that pass the rest of the tests handily.

Ultimately the question is: Is my application "like" this test that can be failed consistently and silently by otherwise good RNGs or is it like the rest of the tests that are being passed. There is no general analytical way to answer this question at this time. Consequently numerical simulationists often speak bravely during the day of their confidence in their answers but have bad dreams at night.

The situation is not hopeless, of course. Very similar considerations apply to numerical benchmarks in general as predictors of the performance of various kinds of code. What the numerical analysts routinely do is to try to empirically and analytically connect the code whose performance they wish to predict on the basis of a benchmark with a specific constellation of performance on a suite of benchmarks, looking especially at two kinds of numbers: *microbenchmarks* that measure specific low level rates that are known to be broadly proportional to performance on specific kinds of tasks, and application benchmarks selected from applications that are *like* the application whose performance

is being predicted, at least in certain key respects. Benchmark toolsets like the *lmbench* suite[**?**] or netpipes[**?**] provide the former; application benchmark suites such as the SPEC suite[**?**] provide a spectrum of numbers representing the latter.

In this sense, Diehard is similar to SPEC – it provides a number of very different, very complex measures of RNG performance that one can at least hope to relate to certain aspects of RNG usage in certain classes of application. STS is in turn similar to lmbench or netpipe – one can more or less independently test RNGs for very specific measures of low level (bit-level) randomness.

However, there are major holes in RNG testing at both the microbenchmark and application benchmark level. SPEC includes a Monte Carlo computation in its suite, for example, so that people doing Monte Carlo can get some idea of a system's probable performance on that kind of application. Diehard, on the other hand, provides no direct test of a Monte Carlo simulated result that can be easily mapped into similar code. Netpipe permits one to measure average network bandwidth and latency for messages containing a 1, 2, 3...1500 or more bytes, but STS lacks a bit-level test that systematically validates RNGs on a regular series of degrees of parametric correlation.

A final issue worthy of future research in this regard is that of systematic *dependency* of RNG tests. This is connected in turn with that of some sort of decomposition of randomness in a moment expansion. Here it suffices to give an example.

The STS "runs" test counts the total number of 0 runs + total number of 1 runs across a sample of bits. To identify a 0 run one searches for its *necessary* starting bit pair 10 and its corresponding ending pair 01. Suppose we label the count of these bit pairs observed as we slide a window two bits wide around a ring of the m bit sample being tested (where, recall, the $m-1$ bit is considered adjacent to the 0 bit on the circular ring) n_{10} and n_{01}, respectively. Similarly we can imagine counting the 11 and 00 bit pairs, n_{11} and n_{00}.

A moment of reflection will convince one that $n_{10} = n_{01}$. If one imagines starting with a ring consisting only of 0's, any time one inserts a substring of 1's one creates precisely one 01 and one 10 pair. Similarly, $n_{11} = n_{00}$ as a constraint of the construction process. If the requirement of periodic boundary conditions is relaxed the only change is that $n_{10} = n_{01} \pm 1, 0$ as there can now exist a single 10 bit pair that isn't paired with 01 at the end or vice versa. However, the validity of the test should in no way be reduced by including the periodic wraparound pair.

Suddenly the "runs" test doesn't look like it is counting runs at all. It is counting the frequency of occurence of just two bit pairs, e.g. 01 and 00, with the frequency of the other two possible bit pairs 10 and 11 obtained from the symmetry of the construction process and ring. In the non-periodic case, it is counting the frequencies of 01 and 10 pairs where they are *constrained* to be within one of one another.

This is clearly equivalent to, and *less sensitive than* a *direct* measurement of *all four* bitpair frequencies and comparison of the result with the expected *distribution* of bitpair frequencies on a string of m (or $m-1$) bits sampled two bits at a time. That is, if 01 bit pairs, 10 bit pairs, 00 bit pairs and 11 bit pairs all occur with the expected frequencies,

the runs test *must* be satisfied and vice versa. The runs test is precisely equivalent to examining the frequency of occurence of the four binary numbers 00, 01, 10 and 11 in overlapping (or not, as a test option) pairs of bits drawn from m-bit words with or without periodic wraparound! However, it is *much easier to understand* in the latter context, as one can do a KS test or χ^2 test to see if these digits are indeed distributed on m-bit samples correctly, each according to a binomial distribution with $p = 0.25$.

This leads us in a natural way to a description of the two STS tests thus far implemented and to a discussion of new tests that are introduced to attempt to systematize and clarify what is being tested.

6.1 STS Tests

While the program's design goals include having all of the STS tests incorporated into its general test launching and reporting framework, at the time of this writing only the first two STS test, the *monobit* (bit frequency) and *runs* tests are incorporated. In both cases the tests were originally written from the test descriptions in NIST SP800-22; although there are in this case no restrictions on the free (re)use of the actual code provided by the NIST STS website, it is still convenient to have a version of the code that is clearly open source according to the GPL. No code provided by NIST was therefore used in Dieharder.

Rewriting the algorithms provided to be a useful exercise in any event. As one can see from the discussion immediately preceding, the process of implementing the actual algorithm for runs led one inevitably to the conclusion that the test really measured the frequency distribution of "runs" of 0's and 1's only *indirectly*, where the *direct* measurement was of the frequency and distribution the four two bit integer numbers 0-3 in overlapping two bit windows slid down the sampled random bit strings of length m with or without periodic boundary conditions.

In addition, it permitted us to parameterize the tests according to our standard description above. Two parameters were introduced; one to control the number of random numbers sampled in a single test to produce a test p-value, and the other to control how many *iid* tests contributed p-values to a final KS test for uniformity of the distribution of p, producing a single p-value upon which to base the classification of the RNG with regard to "failing" the test (rejecting the null hypothesis).

The monobit test measures the mean frequency of 1's relative to 0's across a long string of bits. n_0 and n_1 are evaluated by literally counting the 1 bits (incrementing a counter with the contents of a window of length 1 slid along the the entire length m of the bit string). Clearly a "good" RNG should produce equal numbers of 0's and 1's – $n_0 \approx 0.5 * m \approx n_1$. This makes it simple to create a test statistic expected (for large m) to be normally distributed and hence easily transformable into a p-value.

In the context of Dieharder, the monobit test *subsumes* the STS *frequency* test as well. The frequency test is equivalent to running many independent monobit tests on blocks of m bits and doing a χ^2 test to see if the mean 1 bit frequency is properly distributed

around a mean value of $m/2$. But this is exactly what Dieharder *already* does with monobit, where a KS test for the uniformity for the individual p-values takes the place of the χ^2 test for the distribution of independent measurements. Obviously these are two different ways of looking at and computing the same thing – the p-values returned must be at least asymptotically the same.

The runs test has already been described above – clearly it is equivalent to counting the frequency n_{01} of the occurence of 01 bit pairs in the test sequence of length m with periodic wraparound, which by symmetry yields n_{10}, n_{00} and n_{11}. Indeed, $n_{01} = n_{10} \approx m * 0.25$, with $n_{11} \approx n_{00} \approx m * 0.25$ as well. This test actually has a three degrees of freedom (two of which are ignored) and converting n_{01} alone measured for a run of length $m \gg 1$ into a p-value via the error function is straightforward.

It is generally performed in the STS only after a monobit/frequency is performed on the same bit string, since if the string has an egregiously incorrect number of 1 bits then it clearly *cannot* have the correct distribution of 00, 01, 10 and 11 bit pairs. Similarly, *even* if the monobit test is satisfied we can still fail the runs test. However, if we *pass* the runs test we *also* must pass the monobit test.

From this we learn two things. First of all, we see that there are clearly logical dependencies connecting the monobit and runs test, although the SP800-22 misses several important aspects of this. Passing monobit is a necessary but not sufficient condition for passing runs. Passing runs is a sufficient but not necessary condition for passing monobit! Second of all, when we interpret runs *correctly* as a simple test for the binomial distribution of n_{01} with $p = 0.25$ for a set of samples of length m bits and hence structurally *identical* to the monobit test, we realize that there is an *entire hierarchy* of related tests that differ only in the number of bit in the windows being sampled.

This motivated the development of *new* tests, which subsume *both* STS monobit and STS runs, but which are clearly part of a systematic series of tests of bitwise randomness.

6.2 New Tests

Three entirely new tests have been added to Dieharder. The first is a straightforward timing test that returns the number of random numbers a generator can return per second (measured in wall-clock time and hence subject to peak value error if the systems is heavily loaded at the time of measurement). This result is extremely useful in many contexts – when deciding which RNG to use of several possibilities that all behave about as well according to the full Dieharder test series, when estimating how long one has to get coffee before a newly initiated test series completes (which in the case of e.g. /dev/random might well be "longer than you want to wait" unless your system has many sources of entropy it can sample).

The second is a relatively weak test, but one that is important for informational purposes. This is the *bit persistence* test described earlier, which examines successive e.g. unsigned integers returned by the RNG and checks for bits that do not change for at least

some values of the seed. Bit positions that do not change over a large number of samples (enough to make the probability that each bit has changed at least once essentially unity) are cumulated as a mask of "bad bits" returned by the generator. A surprising number of early RNGs fail this test, in the sense that a number of the least significant bits do not change for at least some seeds! It also quickly reveals RNGs that only return (say) 31 or 24 random bits instead of the full unsigned integer's worth. This can easily cause the RNG to *fail* certain tests that effectively assume 32 random bits to be returned per call even while the numbers returned are *otherwise* highly random. bit_persist

The third is the most interesting – the *bit distribution* test. This is not a single test, it is a systematic *series* of tests. This test takes a very long set of e.g. unsigned integers and treats the whole thing like a string of m bits with cyclic wraparound at the ends. It then slides a window of length n along this string a bit at a time (with overlap), incrementing a frequency counter indexed by the n-bit integer that appears in the window. The integer frequencies thus observed should be equal and distributed around the mean value of $\frac{m}{2^n}$. They are not all independent – the number of degrees of freedom in the test is roughly $2^n - 1$. A simple χ^2 test converts the distribution observed into a p-value.

This test is equivalent to the STS *series* test, but we now see that there is a clear *hierarchical relationship* between this test and several other tests. Ssuppose n and n' are distinct integers describing the size of bit windows used to generate the test statistics p_n, $p_{n'}$. Then passing the test at $n > n'$ is *sufficient* to conclude that the sequence will also pass the test at n'. If a sequence has all four bit integers occurring with the expected frequencies ($\approx \frac{m}{16}$) within the bounds permitted by perfect randomness, then it *must* have the right number of 1's and 0's, the right number of 00, 01, 10, and 11 pairs, and the right number of 000, 001, 010, 011, 100, 101, 110 and 111 triplets, all within the bounds permitted by perfect randomness.

The converse is *not* true – we cannot conclude that if we pass the test at $n < n'$ we will also pass it at n'. Passing at n is a *necessary condition* for passing at $n' > n$, but is not sufficient.

From this we can conclude that if we accept the null hypothesis for the bit distribution test for $n = 4$ (hexadecimal values), we have *also accepted the null hypothesis* for the STS monobit test ($n = 1$), the STS runs test (slightly weaker than the bit distribution test for $n = 2$) and the bit distribution test for $n = 3$ (distribution of octal values). We have also satisfied a *necessary* condition for the $n = 8$ bit distribution test (uniform distribution of all random bytes, integers in the range 0-255), but of course the two hexadecimal digits that occur with the correct overall frequences could be *paired* in a biased way.

The largest value $n_{\max}$ for which an RNG passes the bit distribution test is therefore an important descriptor of the quality of the RNG. We expect that we can *sort* RNGs according to their values of $n_{\max}$, saying that RNG A is "random up to four bits" while RNG B is "random up to six bits". This seems like it will serve as a useful *microbenchmark* of sorts for RNGs, an easy-to-understand test with a hierarchy of success or failure that can fairly easily be related to at least certain patterns of demands likely to be placed on an RNG in an actual application.

The *mode* of failure is also likely to be very useful information, although Diehard is not yet equipped to prove it. For example it would be very interesting to sort the frequencies by their individual p-values (the probability of obtaining the frequency as the outcome of a binomial trial for just the single n-bit number) and look for potentially revealing patterns.

It is also clear that there are *similar* hierarchical relations between the bit distribution test and a number of *other* tests from Diehard and the STS. For example, the DNA test looks at sequences of 20 bits (10 2 bit numbers). There are 1048576 distinct bit sequences containing 20 bits. Although it is memory intensive and difficult to do a bitdist test at this size, it is in principle possible. Doing so is a waste of time, however – all RNGs will almost certainly fail, once the test is done with enough samples to be able to clearly resolve failure.

Diehard instead looks at the number of *missing* 20 bit integers out of 2^{21} samples pulled from a bitstring a bit larger than this, with overlap. If the frequencies of all of the integers were correct, then of course the number of missing integers would come out correct as well. So passing the bit distribution test for $n = 20$ is a *sufficient* condition for passing Diehard's DNA test, while passing the DNA test is a necessary condition for passing the 20 bit distribution test.

The same is more or less true for the other related Diehard tests. Bitstream, OPSO and OQSO all create overlapping 20 bit integers in slightly different ways from from a sample containing a hair over 2^{21} such integers and measure the number of numbers missing after examining all of those samples. Algorithmically they differ *only* in the way that they overlap and hence have the same expected number of missing numbers over the sample size with slightly different variances.

Count the 1s is the final Diehard test related to the bitstream tests in a hierarchical way. It processes a byte stream and maps each byte into one of five numbers, and then create a five digit base 5 number out of the stream of those numbers. The probability of getting each of the five numbers out of an unbiased byte stream is easily determined, and so the probabilities of obtaining each of the 5^5 five digit numbers can be computed. An (overlapping) stream of bytes is processed and the frequency of each number within that stream (compared to the expected value) for four digit and five digit words is converted into a test statistic.

Clearly if the byte stream is random in the bit distribution test out to $n = 40$ (five bytes) then the Count the 1s test will be passed; a RNG that fails the Count the 1s test cannot pass the $n = 40$ bit distribution test. However here it is very clear that *performing* an $n = 40$ bit distribution test is all but impossible unless one uses a cluster to do so – there are 2^40 bins to tally, which exceeds the total active memory storage capacity of everything but a large cluster. However, such a test would never be necessary, as all RNGs currently known would almost certainly fail the bit distribution test at an n considerably less than 40, probably as low as 8.

6.3 Future (Proposed or Planned) Tests

As noted above, eventually Dieharder should have *all* the STS and Diehard tests (where some effort may be expended making the the set "minimal" and not e.g. duplicating monobit and runs tests in the form of a bit distribution (series) test. Tests omitted from both suites but documented in e.g. Knuth will likely be added as well.

At that point development and research energy will likely be directed into two very specific directions. First to discover additional hierarchical test series like the bit distribution test that provide very specific information about the *degree* to which a RNG is random and also provides some specific insight into the nature of its *failure* when at some point the null hypothesis is unambiguously rejected. These tests will be developed by way of providing Dieharder with an embedded microbenchmark suite – a set of tests that all generators fail but that provide specific measures of the point at which randomness fails as they do so.

Several of the STS tests (such as the *discrete Fourier transform* test) appear capable of providing this sort of information with at most a minor modification to cause them to be performed systematically in a series of tests to the point of failure. Others, such as a straightforward autocorrelation test, do not appear to be in any of the test suites we have examined so far although a number of complex tests are hierarchically related to it.

The second place that Dieharder would benefit from the addition of new tests is in the arena of *application* level tests, specifically in the regime of Monte Carlo simulation. Monte Carlo often relies on several distinct measures of the quality of a RNG – the uniformity of deviates returned (so that a Markov process advances with the correct local frequencies of transition), autocorrelations in the sequence returned (so that transitions one way or the other are not "bunched" or non-randomly patterned in other ways in the Markov process), sometimes even in patterning in random site selection in a high-dimensional space, the precise area of application where many generators are *known* to subtly fail even when they pass most tests for uniformity and local autocorrelation.

Viewing a RNG as a form of iterated map with a discrete chaotic component, there may exist long-period cycles in a sufficiently high dimensional space such that the generator's state becomes weakly correlated after irregular but deterministic intervals, correlations that are only visible or measureable in certain projections of the data. It would certainly help numerical simulationists to have an application level tests series that permit them to at least weakly rank RNGs in terms of their likelihood of yielding a valid sampled result in any given computational context.

The same is true for cryptographic applications, although the tendency in the STS has been to remove tests at this level and rely instead on microbenchmarks presumably redundant with the test for randomness represented by the application.

Long before all of this ambitious work is performed, though, it is to be hoped that the Dieharder package produces the *real* effect intended by its author – the provision of a useable testbed framework for researchers to write, and ultimately contribute, their own RNG tests (and candidate RNGs). Diehard and the STS *both* suffer from their very

success – they are "finished products" and written in such a way that makes it very difficult to *play* with their code or add your own code and ideas to them. Dieharder is written to *never be finished*.

The framework exists to easily and consistently add new software generators, with a simple mechanism for merging those generators directly into the GSL should they prove to be as good or better (or just different) than existing generators the GSL already provides.

The framework exists to easily and consistently add new tests for RNGs.

Since nearly *any* random distribution can be used as the basis for a cleverly constructed test, one expects to see the framework used to build tests on top of pretty much all of the GSL built in random distribution functions, to simultaneously test RNGs used as the basic source of randomness and to test the code that produces the (supposedly) suitably distributed random variable. Either end of this proposition can be formulated as a null hypothesis, and the ability to trivially switch RNGs and hence sample the output distributions compared to the theoretical one for many RNGs adds an important dimension to the validation process both ways.

The framework exists to tremendously increase the ability of the testing process to use available e.g. cluster computing resources to perform its tests. Many of the RNG tests are trivially partitionable or parallelizable. A single test or series of tests across a range can be initiated with a very short packet of information, and the return from the test can be anything from a single p-value to a vector of p-values to be centrally accumulated and subjected to a final KS test. The program thus has a fairly straightforward development path for a future that requires much more stringent tests of RNGs than are currently required or possible.

Chapter 7

Results for Selected Generators

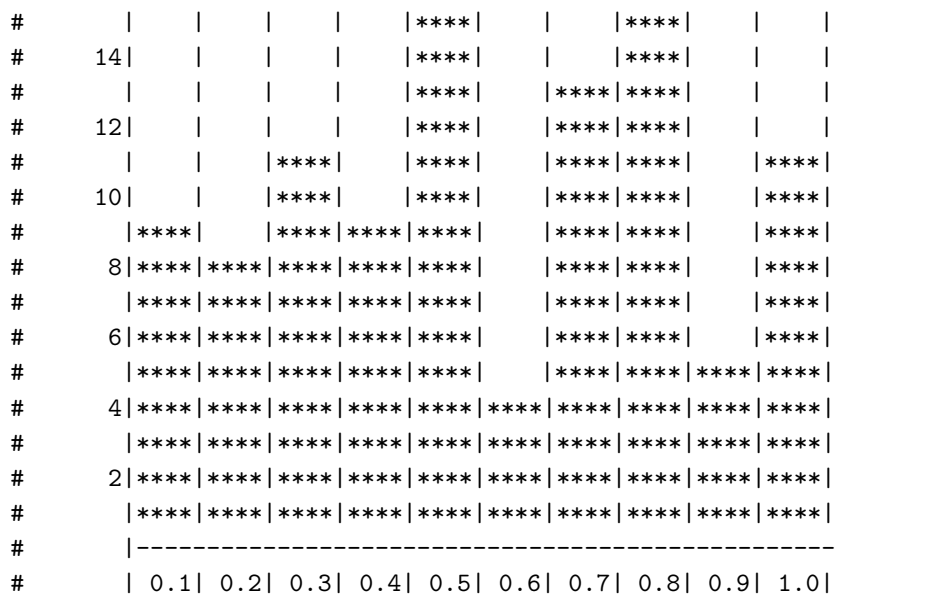
The following are results from applying the full suite of tests to three generators selected from the ones prebuilt into the GSL – a good generator (mt19937_1999), a bad generator (randu) and an *ugly* generator (slatec).

7.0.1 A Good Generator: mt19937_1999

The following is the output from running `dieharder -a -g 13`:

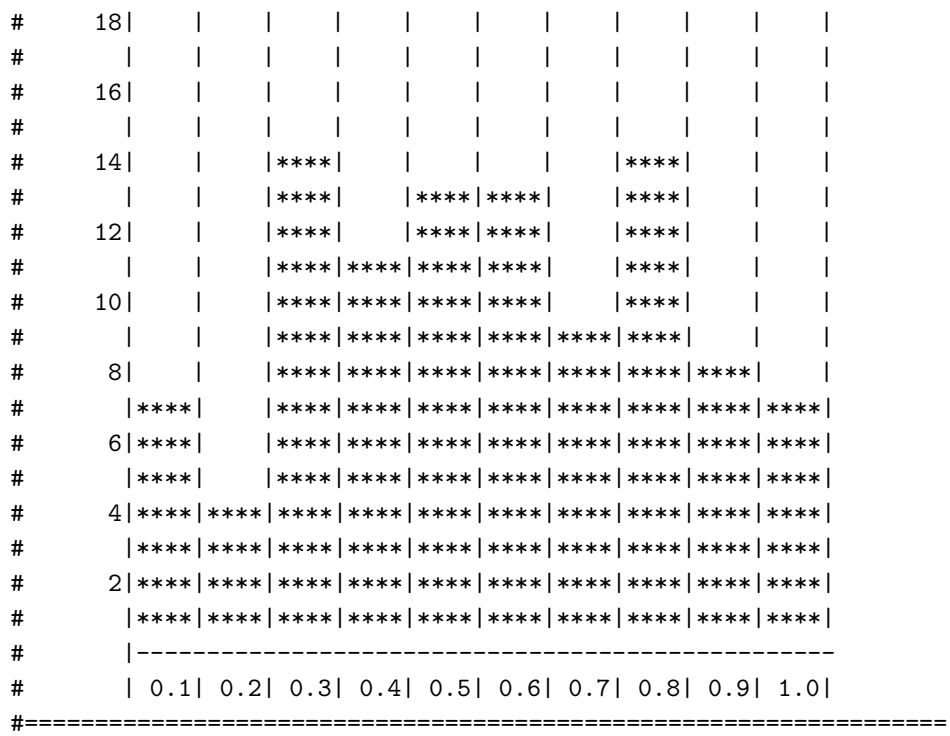
```
#==================================================================
#                         rgb_timing
# This test times the selected random number generator, only.
#==================================================================
#==================================================================
# rgb_timing() test using the mt19937_1999 generator
# Average time per rand = 3.530530e+01 nsec.
# Rands per second = 2.832436e+07.

#==================================================================
#                   RGB Bit Persistence Test
# This test generates 256 sequential samples of an random unsigned
# integer from the given rng.  Successive integers are logically
# processed to extract a mask with 1's whereever bits do not
# change.  Since bits will NOT change when filling e.g. unsigned
# ints with 16 bit ints, this mask logically &'d with the maximum
# random number returned by the rng.  All the remaining 1's in the
# resulting mask are therefore significant -- they represent bits
# that never change over the length of the test.  These bits are
# very likely the reason that certain rng's fail the monobit
# test -- extra persistent e.g. 1's or 0's inevitably bias the
```

```
# total bitcount.  In many cases the particular bits repeated
# appear to depend on the seed.  If the -i flag is given, the
# entire test is repeated with the rng reseeded to generate a mask
# and the extracted mask cumulated to show all the possible bit
# positions that might be repeated for different seeds.
#=================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 256 (test default is 256)
# P-values in final KS test = 1 (test default is 1)
# Samples per test run = 256, tsamples ignored
# Test run 1 times to cumulate unchanged bit mask
#=================================================================
#                          Results
# Results for mt19937_1999 rng, using its 32 valid bits:
# (Cumulated mask of zero is good.)
# cumulated_mask =          0 = 00000000000000000000000000000000
# randm_mask     = 4294967295 = 11111111111111111111111111111111
# random_max     = 4294967295 = 11111111111111111111111111111111
# rgb_persist test PASSED (no bits repeat)
#=================================================================

#=================================================================
#                 RGB Bit Distribution Test
# Accumulates the frequencies of all n-tuples of bits in a list
# of random integers and compares the distribution thus generated
# with the theoretical (binomial) histogram, forming chisq and the
# associated p-value.  In this test n-tuples are selected without
# WITHOUT overlap (e.g. 01|10|10|01|11|00|01|10) so the samples
# are independent.  Every other sample is offset modulus of the
# sample index and ntuple_max.
#=================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
# Testing ntuple = 1
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
```

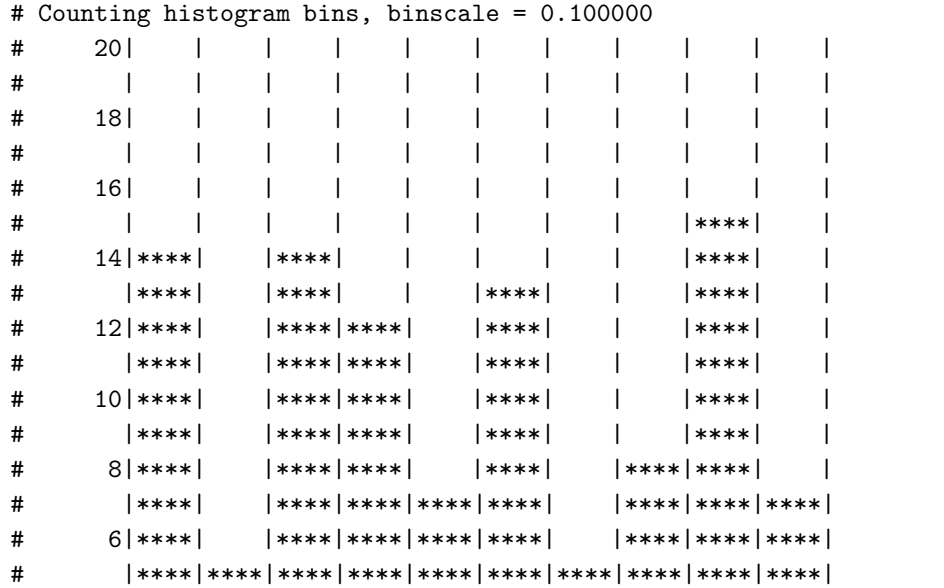


```
#       |    |    |    |    |****|    |    |****|    |    |
#     14|    |    |    |    |****|    |    |****|    |    |
#       |    |    |    |    |****|    |****|****|    |    |
#     12|    |    |    |    |****|    |****|****|    |    |
#       |    |    |****|    |****|    |****|****|    |****|
#     10|    |    |****|    |****|    |****|****|    |****|
#       |****|    |****|****|****|    |****|****|    |****|
#      8|****|****|****|****|****|    |****|****|    |****|
#       |****|****|****|****|****|    |****|****|    |****|
#      6|****|****|****|****|****|    |****|****|    |****|
#       |****|****|****|****|****|    |****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-----------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.940792 for RGB Bit Distribution Test
# Assessment:
# PASSED at > 5%.
# Testing ntuple = 2
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |****|    |    |
#     14|    |    |    |    |****|    |    |****|    |    |
#       |    |    |    |****|****|    |    |****|    |    |
#     12|    |****|    |****|****|    |    |****|    |    |
#       |****|****|    |****|****|    |    |****|    |    |
#     10|****|****|****|****|****|    |    |****|    |    |
#       |****|****|****|****|****|    |    |****|    |****|
#      8|****|****|****|****|****|    |    |****|****|****|
#       |****|****|****|****|****|    |    |****|****|****|
#      6|****|****|****|****|****|    |    |****|****|****|
#       |****|****|****|****|****|    |    |****|****|****|
#      4|****|****|****|****|****|    |****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
```

```
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.300792 for RGB Bit Distribution Test
# Assessment:
# PASSED at > 5%.
# Testing ntuple = 3
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     16|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     14|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     12|    |****|****|    |****|    |    |    |    |    |
#       |    |****|****|    |****|****|    |    |    |    |
#     10|****|****|****|    |****|****|    |    |    |    |
#       |****|****|****|    |****|****|    |****|    |****|
#      8|****|****|****|****|****|****|    |****|    |****|
#       |****|****|****|****|****|****|    |****|****|****|
#      6|****|****|****|****|****|****|    |****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.331776 for RGB Bit Distribution Test
# Assessment:
# PASSED at > 5%.
# Testing ntuple = 4
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```



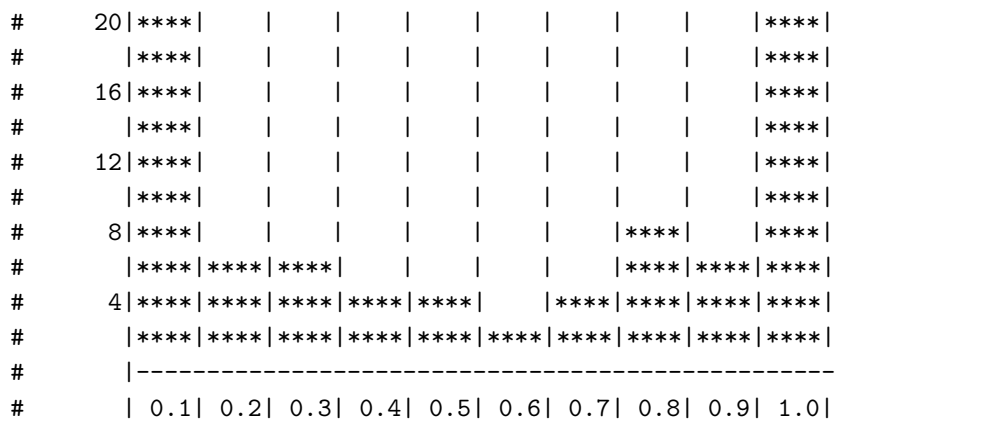

```
#==================================================================
#                          Results
# Kuiper KS: p = 0.083786 for RGB Bit Distribution Test
# Assessment:
# PASSED at > 5%.
# Testing ntuple = 5
#==================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```



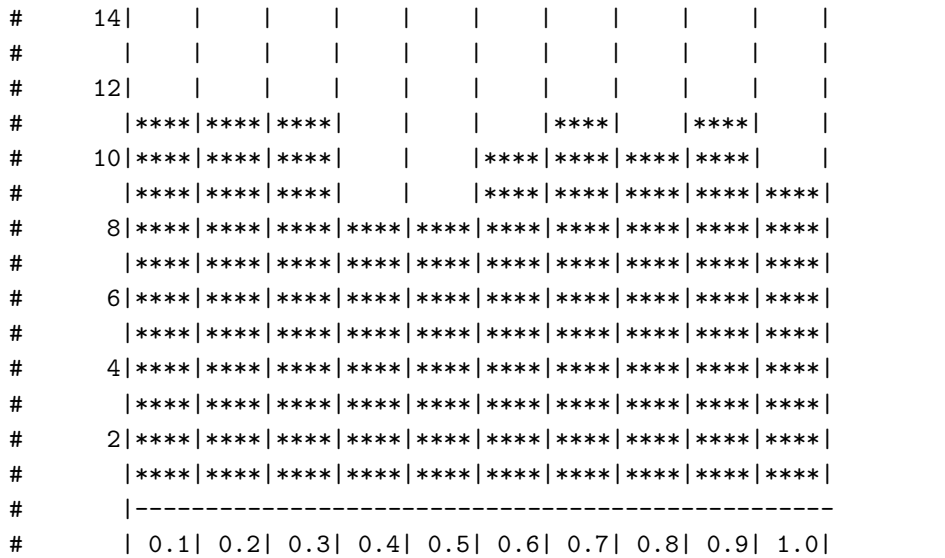
```
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.789288 for RGB Bit Distribution Test
# Assessment:
# PASSED at > 5%.
# Testing ntuple = 6
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for RGB Bit Distribution Test
# Assessment:
# FAILED at < 0.01%.
# Generator mt19937_1999 FAILS at 0.01% for 6-tuplets.  rgb_bitdist terminating.

#=================================================================
```

```
#                Diehard "Birthdays" test (modified).
# Each test determines the number of matching intervals from 512
# "birthdays" (by default) drawn on a 24-bit "year" (by
# default).  This is repeated 100 times (by default) and the
# results cumulated in a histogram.  Repeated intervals should be
# distributed in a Poisson distribution if the underlying generator
# is random enough, and a a chisq and p-value for the test are
# evaluated relative to this null hypothesis.
#
# It is recommended that you run this at or near the original
# 100 test samples per p-value with -t 100.
#
# Two additional parameters have been added. In diehard, nms=512
# but this CAN be varied and all Marsaglia's formulae still work.  It
# can be reset to different values with -x nmsvalue.
# Similarly, nbits "should" 24, but we can really make it anything
# we want that's less than or equal to rmax_bits = 32.  It can be
# reset to a new value with -y nbits.  Both default to diehard's
# values if no -x or -y options are used.
#=================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# 512 samples drawn from 24-bit integers masked out of a
# 32 bit random integer.  lambda = 2.000000, kmax = 6, tsamples = 100
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     14|****|****|    |****|    |    |    |    |    |    |
#       |****|****|    |****|    |    |****|    |****|    |
#     12|****|****|    |****|    |    |****|    |****|    |
#       |****|****|    |****|    |    |****|    |****|    |
#     10|****|****|    |****|    |    |****|    |****|    |
#       |****|****|****|****|****|    |****|    |****|    |
#      8|****|****|****|****|****|    |****|    |****|    |
#       |****|****|****|****|****|    |****|    |****|****|
#      6|****|****|****|****|****|    |****|    |****|****|
#       |****|****|****|****|****|    |****|    |****|****|
```

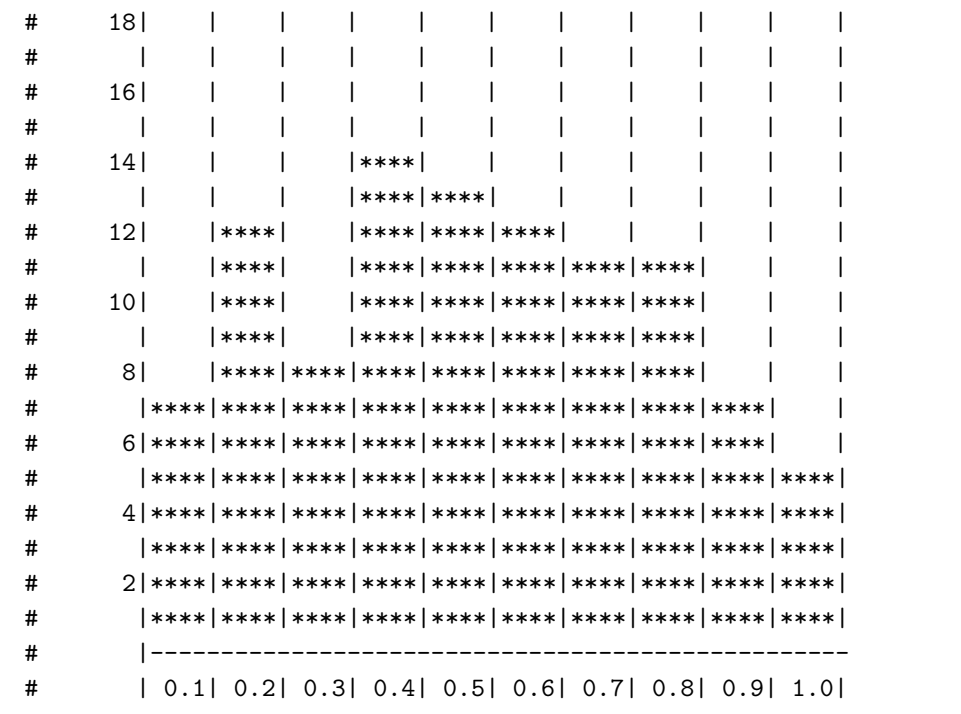
```
#      4|****|****|****|****|****|    |****|    |****|****|
#       |****|****|****|****|****|    |****|    |****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.056099 for Diehard Birthdays Test
# Assessment:
# PASSED at > 5%.

#================================================================
#          Diehard Overlapping 5-{}Permutations Test.
# This is the OPERM5 test.  It looks at a sequence of one mill-
# ion 32-bit random integers.  Each set of five consecutive
# integers can be in one of 120 states, for the 5! possible or-
# derings of five numbers.  Thus the 5th, 6th, 7th,...numbers
# each provide a state. As many thousands of state transitions
# are observed,  cumulative counts are made of the number of
# occurences of each state.  Then the quadratic form in the
# weak inverse of the 120x120 covariance matrix yields a test
# equivalent to the likelihood ratio test that the 120 cell
# counts came from the specified (asymptotically) normal dis-
# tribution with the specified 120x120 covariance matrix (with
# rank 99).  This version uses 1,000,000 integers, twice.
#================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 1000000 (test default is 1000000)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^28 for 100 samples.
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     32|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     28|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
```




```
#=============================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Overlapping 5-{}permutations Test
# Assessment:
# FAILED at < 0.01%.

#=============================================================
#               Diehard 32x32 Binary Rank Test
# This is the BINARY RANK TEST for 31x31 matrices. The leftmost
# 31 bits of 31 random integers from the test sequence are used
# to form a 31x31 binary matrix over the field {0,1}. The rank
# is determined. That rank can be from 0 to 31, but ranks< 28
# are rare, and their counts are pooled with those for rank 28.
# Ranks are found for (default) 40,000 such random matrices and
# a chisquare test is performed on counts for ranks 31,30,29 and
# <=28.
#
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#=============================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 40000 (test default is 40000)
# P-values in final KS test = 100 (test default is 100)
#=============================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```




```
#=================================================================
#                          Results
# Kuiper KS: p = 0.929434 for Diehard 32x32 Rank Test
# Assessment:
# PASSED at > 5%.

#=================================================================
#             Diehard 6x8 Binary Rank Test
# This is the BINARY RANK TEST for 6x8 matrices.  From each of
# six random 32-bit integers from the generator under test, a
# specified byte is chosen, and the resulting six bytes form a
# 6x8 binary matrix whose rank is determined.  That rank can be
# from 0 to 6, but ranks 0,1,2,3 are rare; their counts are
# pooled with those for rank 4. Ranks are found for 100,000
# random matrices, and a chi-square test is performed on
# counts for ranks 6,5 and <=4.
#
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#=================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```




```
#==========================================================================
#                           Results
# Kuiper KS: p = 0.364548 for Diehard 6x8 Binary Rank Test
# Assessment:
# PASSED at > 5%.

#==========================================================================
#                  Diehard Bitstream Test.
# The file under test is viewed as a stream of bits. Call them
# b1,b2,... .  Consider an alphabet with two "letters", 0 and 1
# and think of the stream of bits as a succession of 20-letter
# "words", overlapping.  Thus the first word is b1b2...b20, the
# second is b2b3...b21, and so on.  The bitstream test counts
# the number of missing 20-letter (20-bit) words in a string of
# 2^21 overlapping 20-letter words.  There are 2^20 possible 20
# letter words.  For a truly random string of 2^21+19 bits, the
# number of missing words j should be (very close to) normally
# distributed with mean 141,909 and sigma 428.  Thus
#  (j-141909)/428 should be a standard normal variate (z score)
# that leads to a uniform [0,1) p value.  The test is repeated
# twenty times.
#
# Note that of course we do not "restart file", when using gsl
# generators, we just crank out the next random number.
# We also do not bother to overlap the words.  rands are cheap.
```

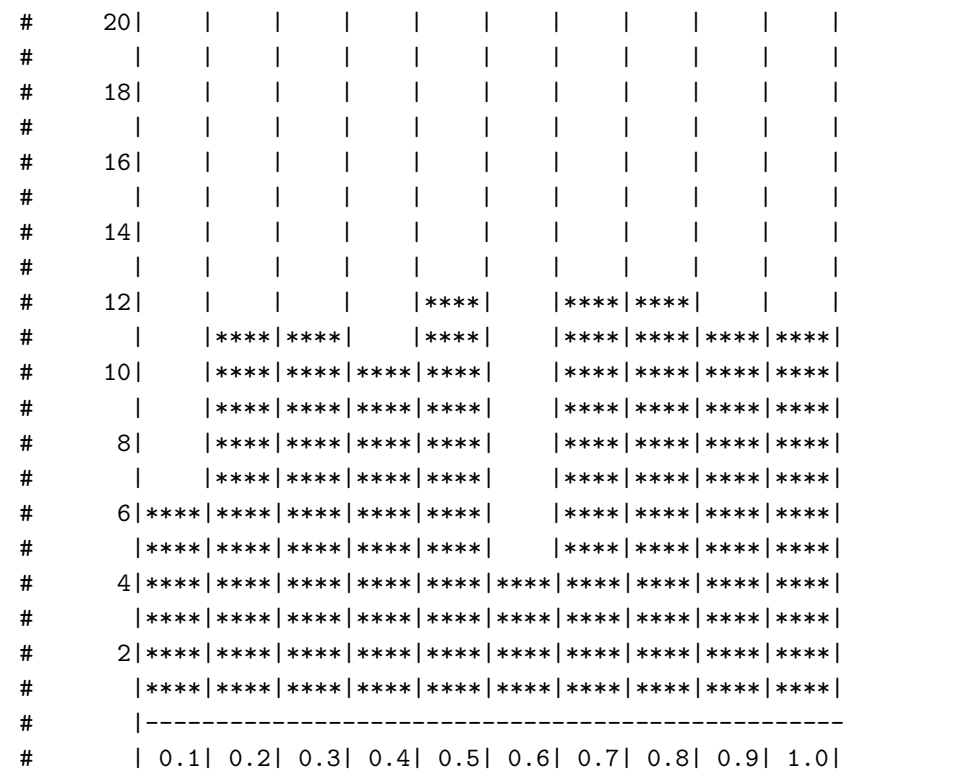
```
# Finally, we repeat the test (usually) more than twenty time.
#=================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     28|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     24|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |****|    |    |
#     20|    |    |    |    |    |    |    |****|    |    |
#       |    |    |    |    |    |    |    |****|    |    |
#     16|    |    |    |    |    |    |    |****|    |    |
#       |    |    |    |    |    |    |****|****|    |****|
#     12|    |    |    |    |    |****|****|****|    |****|
#       |    |    |    |    |    |****|****|****|    |****|
#      8|    |    |****|    |****|****|****|****|    |****|
#       |    |****|****|****|****|****|****|****|****|****|
#      4|    |****|****|****|****|****|****|****|****|****|
#       |    |****|****|****|****|****|****|****|****|****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.004456 for Diehard Bitstream Test
# Assessment:
# POOR at < 1%.
# Recommendation:  Repeat test to verify failure.

#=================================================================
#        Diehard Overlapping Pairs Sparse Occupance (OPSO)
# The OPSO test considers 2-letter words from an alphabet of
# 1024 letters.  Each letter is determined by a specified ten
# bits from a 32-bit integer in the sequence to be tested. OPSO
```

```
# generates  2^21 (overlapping) 2-letter words  (from 2^21+1
# "keystrokes")  and counts the number of missing words---that
# is 2-letter words which do not appear in the entire sequence.
# That count should be very close to normally distributed with
# mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should
# be a standard normal variable. The OPSO test takes 32 bits at
# a time from the test file and uses a designated set of ten
# consecutive bits. It then restarts the file for the next de-
# signated 10 bits, and so on.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#=================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |****|    |    |    |
#     14|    |    |    |    |    |****|****|    |    |    |
#       |    |    |    |    |****|****|****|    |    |    |
#     12|    |    |    |    |****|****|****|    |    |    |
#       |    |    |****|    |****|****|****|    |****|    |
#     10|    |    |****|    |****|****|****|    |****|    |
#       |    |    |****|    |****|****|****|    |****|    |
#      8|****|    |****|    |****|****|****|    |****|    |
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
```

```
# Kuiper KS: p = 0.378389 for Diehard OPSO Test
# Assessment:
# PASSED at > 5%.

#=================================================================
#   Diehard Overlapping Quadruples Sparce Occupancy (OQSO) Test
#
#  Similar, to OPSO except that it considers 4-letter
#  words from an alphabet of 32 letters, each letter determined
#  by a designated string of 5 consecutive bits from the test
#  file, elements of which are assumed 32-bit random integers.
#  The mean number of missing words in a sequence of 2^21 four-
#  letter words,  (2^21+3 "keystrokes"), is again 141909, with
#  sigma = 295.  The mean is based on theory; sigma comes from
#  extensive simulation.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#=================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     14|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |****|    |    |    |
#     12|    |    |    |    |****|    |****|    |    |    |
#       |    |****|    |    |****|    |****|    |****|    |
#     10|    |****|    |    |****|    |****|    |****|    |
#       |    |****|    |****|****|****|****|    |****|    |
#      8|****|****|****|****|****|****|****|****|****|    |
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
```

```
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.393770 for Diehard OQSO Test
# Assessment:
# PASSED at > 5%.

#================================================================
#                    Diehard DNA Test.
#
#   The DNA test considers an alphabet of 4 letters::  C,G,A,T,
# determined by two designated bits in the sequence of random
# integers being tested.  It considers 10-letter words, so that
# as in OPSO and OQSO, there are 2^20 possible words, and the
# mean number of missing words from a string of 2^21  (over-
# lapping)  10-letter  words (2^21+9 "keystrokes") is 141909.
# The standard deviation sigma=339 was determined as for OQSO
# by simulation.  (Sigma for OPSO, 290, is the true value (to
# three places), not determined by simulation.
#
# Note 2^21 = 2097152
# Note also that we don't bother with overlapping keystrokes
# (and sample more rands -- rands are now cheap).
#================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     16|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     14|    |    |    |    |****|    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     12|    |    |    |    |****|    |    |    |    |    |
```

```
#       |****|    |    |    |****|    |    |    |    |    |
#     10|****|    |    |****|****|****|    |    |    |****|
#       |****|****|****|****|****|****|****|    |****|****|
#      8|****|****|****|****|****|****|****|    |****|****|
#       |****|****|****|****|****|****|****|    |****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=============================================================
#                          Results
# Kuiper KS: p = 0.696510 for Diehard DNA Test
# Assessment:
# PASSED at > 5%.

#=============================================================
#          Diehard Count the 1s (stream) (modified) Test.
# Consider the file under test as a stream of bytes (four per
# 32 bit integer).  Each byte can contain from 0 to 8 1's,
# with probabilities 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the stream of bytes provide a string of overlapping  5-letter
# words, each "letter" taking values A,B,C,D,E. The letters are
# determined by the number of 1's in a byte::  0,1,or 2 yield A,
# 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus
# we have a monkey at a typewriter hitting five keys with vari-
# ous probabilities (37,56,70,56,37 over 256).  There are 5^5
# possible 5-letter words, and from a string of 256,000 (over-
# lapping) 5-letter words, counts are made on the frequencies
# for each word.   The quadratic form in the weak inverse of
# the covariance matrix of the cell counts provides a chisquare
# test::  Q5-Q4, the difference of the naive Pearson sums of
# (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.
#=============================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#=============================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```




```
#==========================================================================
#                           Results
# Kuiper KS: p = 0.985250 for Diehard Count the 1s (stream)
# Assessment:
# PASSED at > 5%.

#==========================================================================
#         Diehard Count the 1s Test (byte) (modified).
#     This is the COUNT-THE-1's TEST for specific bytes.
# Consider the file under test as a stream of 32-bit integers.
# From each integer, a specific byte is chosen , say the left-
# most::  bits 1 to 8. Each byte can contain from 0 to 8 1's,
# with probabilitie 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the specified bytes from successive integers provide a string
# of (overlapping) 5-letter words, each "letter" taking values
# A,B,C,D,E. The letters are determined  by the number of 1's,
# in that byte::  0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D,
# and  6,7 or 8 ---> E.  Thus we have a monkey at a typewriter
# hitting five keys with with various probabilities::  37,56,70,
# 56,37 over 256. There are 5^5 possible 5-letter words, and
# from a string of 256,000 (overlapping) 5-letter words, counts
# are made on the frequencies for each word. The quadratic form
# in the weak inverse of the covariance matrix of the cell
```

```
# counts provides a chisquare test::  Q5-Q4, the difference of
# the naive Pearson  sums of (OBS-EXP)^2/EXP on counts for 5-
# and 4-letter cell counts.
#
# Note: We actually cycle samples over all 0-31 bit offsets, so
# that if there is a problem with any particular offset it has
# a chance of being observed.  One can imagine problems with odd
# offsets but not even, for example, or only with the offset 7.
# tsamples and psamples can be freely varied, but you'll likely
# need tsamples >> 100,000 to have enough to get a reliable kstest
# result.
#==================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |****|    |    |    |
#     14|    |    |    |    |    |    |****|    |    |    |
#       |    |    |    |****|    |    |****|    |    |    |
#     12|    |    |    |****|    |    |****|    |    |    |
#       |    |    |    |****|    |    |****|****|    |    |
#     10|    |****|    |****|    |****|****|****|    |    |
#       |    |****|****|****|****|****|****|****|    |****|
#      8|    |****|****|****|****|****|****|****|****|****|
#       |    |****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#==================================================================
#                           Results
# Kuiper KS: p = 0.748440 for Diehard Count the 1s (byte)
```

```
# Assessment:
# PASSED at > 5%.

#==================================================================
#             Diehard Parking Lot Test (modified).
# This tests the distribution of attempts to randomly park a
# square car of length 1 on a 100x100 parking lot without
# crashing.  We plot n (number of attempts) versus k (number of
# attempts that didn't "crash" because the car squares
# overlapped and compare to the expected result from a perfectly
# random set of parking coordinates.  This is, alas, not really
# known on theoretical grounds so instead we compare to n=12,000
# where k should average 3523 with sigma 21.9 and is very close
# to normally distributed.  Thus (k-3523)/21.9 is a standard
# normal variable, which converted to a uniform p-value, provides
# input to a KS test with a default 100 samples.
#==================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 0 (test default is 0)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     14|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |
#     12|****|    |****|    |****|    |    |    |    |    |
#       |****|****|****|    |****|****|    |****|    |    |
#     10|****|****|****|    |****|****|    |****|    |****|
#       |****|****|****|    |****|****|****|****|    |****|
#      8|****|****|****|    |****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|    |****|
#      6|****|****|****|****|****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|    |****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
```

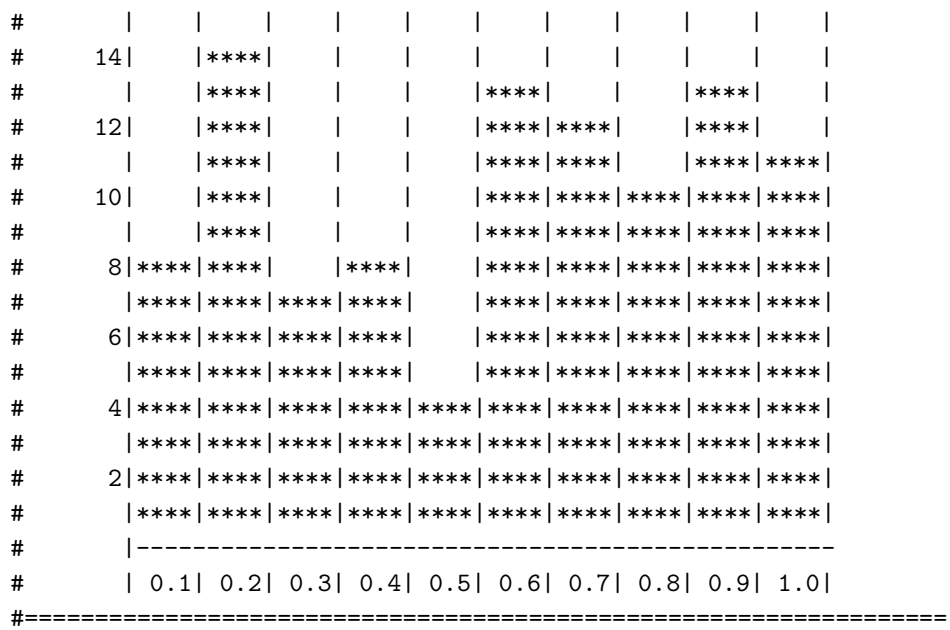
```
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.379171 for Diehard Parking Lot Test
# Assessment:
# PASSED at > 5%.

#================================================================
#         Diehard Minimum Distance (2d Circle) Test
# Generate 8000 points in a 10000^2 square.  Determine the
# the shortest nearest neighbor distance R.  This should generate
# p = 1.0 - exp(-R^2/0.995).  Repeat for lots of samples, apply a
# KS test to see if p is uniform.
#
# The number of samples is fixed -- tsamples is ignored.
#================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |****|    |    |    |    |    |    |
#     14|****|    |    |****|****|    |    |    |    |    |
#       |****|    |    |****|****|    |    |    |    |    |
#     12|****|    |    |****|****|    |    |    |****|    |
#       |****|    |    |****|****|    |    |    |****|    |
#     10|****|    |    |****|****|    |    |****|****|    |
#       |****|    |    |****|****|    |    |****|****|    |
#      8|****|    |****|****|****|    |    |****|****|    |
#       |****|****|****|****|****|    |****|****|****|    |
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
```

```
#==================================================================
#                          Results
# Kuiper KS: p = 0.774383 for Diehard Minimum Distance (2d Circle) Test
# Assessment:
# PASSED at > 5%.
#==================================================================
#          Diehard 3d Sphere (Minimum Distance) Test
# Choose  4000 random points in a cube of edge 1000.  At each
# point, center a sphere large enough to reach the next closest
# point. Then the volume of the smallest such sphere is (very
# close to) exponentially distributed with mean 120pi/3.  Thus
# the radius cubed is exponential with mean 30. (The mean is
# obtained by extensive simulation).  The 3DSPHERES test gener-
# ates 4000 such spheres 20 times.  Each min radius cubed leads
# to a uniform variable by means of 1-exp(-r^3/30.), then a
#  KSTEST is done on the 20 p-values.
#
# This test ignores tsamples, and runs the usual default 100
# psamples to use in the final KS test.
#==================================================================
#                        Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 4000 (test default is 4000)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     14|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     12|    |    |****|****|    |    |    |****|    |    |
#       |****|    |****|****|    |    |****|****|    |    |
#     10|****|    |****|****|    |    |****|****|    |****|
#       |****|    |****|****|****|    |****|****|    |****|
#      8|****|****|****|****|****|    |****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
```

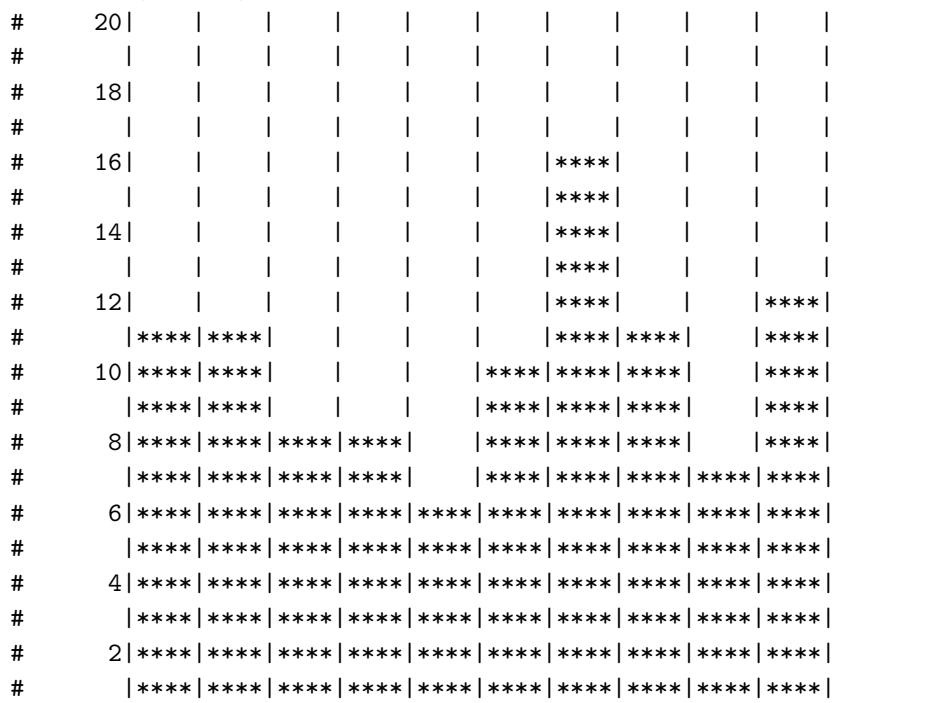
```
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.989677 for Diehard 3d Sphere (Minimum Distance) Test
# Assessment:
# PASSED at > 5%.
#================================================================
#                 Diehard Squeeze Test.
#  Random integers are floated to get uniforms on [0,1). Start-
#  ing with k=2^31=2147483647, the test finds j, the number of
#  iterations necessary to reduce k to 1, using the reduction
#  k=ceiling(k*U), with U provided by floating integers from
#  the file being tested.  Such j's are found 100,000 times,
#  then counts for the number of times j was <=6,7,...,47,>=48
#  are used to provide a chi-square test for cell frequencies.
#================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |****|    |    |
#     14|    |    |    |    |    |    |    |****|    |****|
#       |    |    |    |    |    |    |    |****|    |****|
#     12|    |    |    |****|    |    |    |****|    |****|
#       |    |    |    |****|    |    |    |****|    |****|
#     10|****|****|****|****|    |    |    |****|    |****|
#       |****|****|****|****|    |    |    |****|****|****|
#      8|****|****|****|****|    |    |****|****|****|****|
#       |****|****|****|****|    |****|****|****|****|****|
#      6|****|****|****|****|    |****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
```

```
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.807374 for Diehard Squeeze Test
# Assessment:
# PASSED at > 5%.

#=================================================================
#                  Diehard Sums Test
# Integers are floated to get a sequence U(1),U(2),... of uni-
# form [0,1) variables.  Then overlapping sums,
#   S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed.
# The S's are virtually normal with a certain covariance mat-
# rix.  A linear transformation of the S's converts them to a
# sequence of independent standard normals, which are converted
# to uniform variables for a KSTEST. The  p-values from ten
# KSTESTs are given still another KSTEST.
#
# Note well:  -O causes the old diehard version to be run (more or
# less).  Omitting it causes non-overlapping sums to be used and
# directly tests the overall balance of uniform rands.
#=================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     16|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     14|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     12|    |    |    |    |    |    |    |    |    |****|
#       |    |****|    |    |    |    |****|    |    |****|
#     10|    |****|    |****|    |    |****|    |    |****|
#       |****|****|    |****|    |****|****|****|    |****|
```

```
#      8|****|****|****|****|    |****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=============================================================
#                           Results
# Kuiper KS: p = 0.557686 for Diehard Sums Test
# Assessment:
# PASSED at > 5%.
#=============================================================
#                    Diehard Runs Test
#  This is the RUNS test.  It counts runs up, and runs down,
# in a sequence of uniform [0,1) variables, obtained by float-
# ing the 32-bit integers in the specified file. This example
# shows how runs are counted:  .123,.357,.789,.425,.224,.416,.95
# contains an up-run of length 3, a down-run of length 2 and an
# up-run of (at least) 2, depending on the next values.  The
# covariance matrices for the runs-up and runs-down are well
# known, leading to chisquare tests for quadratic forms in the
# weak inverses of the covariance matrices.  Runs are counted
# for sequences of length 10,000.  This is done ten times. Then
# repeated.
#
# In Dieharder sequences of length tsamples = 100000 are used by
# default, and 100 p-values thus generated are used in a final
# KS test.
#=============================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=============================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
```



```
#                           Results
# Kuiper KS: p = 0.472238 for Runs (up)
# Assessment:
# PASSED at > 5%.
```



```
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```

```
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.682452 for Runs (down)
# Assessment:
# PASSED at > 5%.

#================================================================
#                   Diehard Craps Test
#  This is the CRAPS TEST. It plays 200,000 games of craps, finds
#  the number of wins and the number of throws necessary to end
#  each game.  The number of wins should be (very close to) a
#  normal with mean 200000p and variance 200000p(1-p), with
#  p=244/495.  Throws necessary to complete the game can vary
#  from 1 to infinity, but counts for all>21 are lumped with 21.
#  A chi-square test is made on the no.-of-throws cell counts.
#  Each 32-bit integer from the test file provides the value for
#  the throw of a die, by floating to [0,1), multiplying by 6
#  and taking 1 plus the integer part of the result.
#================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 200000 (test default is 200000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     14|    |    |    |    |    |****|****|    |    |    |
#       |    |    |    |    |    |****|****|    |    |****|
#     12|    |    |    |    |    |****|****|    |    |****|
#       |    |    |    |****|    |****|****|    |    |****|
#     10|    |    |****|****|    |****|****|    |    |****|
#       |    |    |****|****|    |****|****|    |****|****|
#      8|    |****|****|****|    |****|****|****|****|****|
#       |    |****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
```

```
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.872210 for Craps Test (mean)
# Assessment:
# PASSED at > 5%.
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     14|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |****|    |    |    |    |    |    |
#     12|****|    |    |****|****|    |    |    |    |    |
#       |****|    |****|****|****|    |    |    |****|    |
#     10|****|    |****|****|****|    |    |****|****|    |
#       |****|    |****|****|****|    |    |****|****|    |
#      8|****|    |****|****|****|    |****|****|****|    |
#       |****|    |****|****|****|****|****|****|****|    |
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.945661 for Craps Test (freq)
# Assessment:
# PASSED at > 5%.

#================================================================
#                      STS Monobit Test
# Very simple.  Counts the 1 bits in a long string of random uints.
# Compares to expected number, generates a p-value directly from
```

```
# erfc().  Very effective at revealing overtly weak generators;
# Not so good at determining where stronger ones eventually fail.
#=================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |    |
#     14|****|    |    |    |****|    |    |    |    |    |    |
#       |****|    |    |    |****|    |    |    |    |    |    |
#     12|****|****|****|    |****|    |    |****|    |    |    |
#       |****|****|****|    |****|    |    |****|    |    |    |
#     10|****|****|****|    |****|    |    |****|****|    |    |
#       |****|****|****|    |****|    |    |****|****|    |    |
#      8|****|****|****|    |****|    |    |****|****|    |    |
#       |****|****|****|****|****|    |    |****|****|****|    |
#      6|****|****|****|****|****|    |****|****|****|****|    |
#       |****|****|****|****|****|****|****|****|****|****|    |
#      4|****|****|****|****|****|****|****|****|****|****|    |
#       |****|****|****|****|****|****|****|****|****|****|    |
#      2|****|****|****|****|****|****|****|****|****|****|    |
#       |****|****|****|****|****|****|****|****|****|****|    |
#       |----------------------------------------------------    |
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|    |
#=================================================================
#                           Results
# Kuiper KS: p = 0.286818 for STS Monobit Test
# Assessment:
# PASSED at > 5%.

#=================================================================
#                       STS Runs Test
# Counts the total number of 0 runs + total number of 1 runs across
# a sample of bits.  Note that a 0 run must begin with 10 and end
# with 01.  Note that a 1 run must begin with 01 and end with a 10.
# This test, run on a bitstring with cyclic boundary conditions, is
# absolutely equivalent to just counting the 01 + 10 bit pairs.
```

```
# It is therefore totally redundant with but not as good as the
# rgb_bitdist() test for 2-tuples, which looks beyond the means to the
# moments, testing an entire histogram  of 00, 01, 10, and 11 counts
# to see if it is binomially distributed with p = 0.25.
#==================================================================
#                         Run Details
# Random number generator tested: mt19937_1999
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     14|    |    |    |****|    |    |    |    |    |****|
#       |    |    |    |****|****|    |    |    |    |****|
#     12|    |    |    |****|****|    |    |    |    |****|
#       |    |    |****|****|****|    |    |    |    |****|
#     10|****|****|****|****|****|    |****|    |    |****|
#       |****|****|****|****|****|    |****|    |    |****|
#      8|****|****|****|****|****|****|****|    |    |****|
#       |****|****|****|****|****|****|****|****|    |****|
#      6|****|****|****|****|****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|    |****|
#      4|****|****|****|****|****|****|****|****|    |****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#==================================================================
#                           Results
# Kuiper KS: p = 0.337260 for STS Runs Test
# Assessment:
# PASSED at > 5%.
```

7.0.2 Comments

This is the output from what is generally considered a "good" RNG – one that is often touted as passing Diehard. As one can see from the output above, although it "passed" Diehard, it was a bit of a marginal pass and in fact when the number of test samples is increased a number of tests return p-values that are not terribly low but which *also* are not uniformly distributed, usually a sign of eventual failure. Let us quickly review these results and comment.

The mt19937_1999 is quite fast (at 28×10^7 rands per second on a 1.87 GHz laptop). The bit persistence test shows that all of its unsigned integer bits vary, which is good. The sequence of bit distributions tests that follow show that it is 5-bit random – arbitrary pieces of the integers it returns that are 5 bits in length are uniformly distributed across all 32 integers thus represented, but *6* bit chunks are *not* uniformly distributed over the 64 integers thus represented!

This is interesting information. It means that this generator will not produce completely uniform results for any ntuplet of bits with 6 or more bits in it. If used to select random ascii letters, for example, it probably will not cover the alphabet uniformly within the expectations of statistics. It also points to a direction for further study of the generator. Why and how does the generator fail to be 6 bit random? What would happen if one took two mt19937_1999 generators (with independent seeds) and "shuffled" their output in 5 bit chunks?

We then begin on the Diehard tests. The test passes the birthday test, but examining the distribution produced we can see that the pass is a bit "marginal", in the sense that p isn't terribly uniformly distributed. When this happens, one may want to rerun the particular test a few times to see if the features in the histogram vary or are systematic. Alternatively, rerunning it with a larger value of KS samples with e.g. `-p 1000` or more may push it into unambiguous failure and in fact it does. This *is* a marginal result, and a sample of random numbers *far* smaller than what would be used in any numerical simulation will unambiguously fail the Diehard birthdays test.

Surprisingly, we find that the generator *unambiguously* fails the Overlapping 5-Permutations test with the Dieharder defaults already! Note well that the original diehard test only produces *two* p-values, and there is a very good chance, of course, that at least one of those two will be well above the usual $p < 0.01$ criterion for rejecting the null hypothesis. Dieharder reveals a very systematic problem with mt19937_1999 – it has a tendency to produce long stretches of rands that are either a bit "too random" (too likely to precisely balance permutations) or "too ordered" (too likely to favor certain permutations over others). The two are nearly balanced so that *overall* the generator probably *does* balance the total number of permutations nicely, but the *bunching* of the permutations in samples of 1000000 random integers is non-random!

mt19937_1999 handily passes the binary rank tests. This is interesting, given that we have already determined the generator to be 5 bit random (but not 6). Apparently the binary rank tests are less sensitive to overt inhomogeneity in the ntuple bit distribution than the bit distribution test, or at least sample a *different* aspect of bitlevel randomness.

We then begin the four bit distribution tests in Diehard: bitstream, OPSO, OQSO and DNA. All of these test the distribution of overlapping 20-bit integer substrings of an unsigned integer stream of random numbers, but they do so in different ways. mt19937_1999 does "poorly" on the bitstream test, but passes the other three easily. Note once again that running the bitstream test only 20 times (the original Diehard default) and examining the resulting p-values by eye one would almost certainly have passed the generator, but now we have a *cumulative* p-value from an actual KS test less than 1%, a point where most tests recommend rejecting the null hypothesis. The Dieharder recommendation is instead to *rerun the test* a few times (or up the number of samples in the KS test with `-p 500` or the like)! This is especially reasonable given that the RNG passes the *other* three 20-bit tests, something that seems relatively unlikely if it fails bitstream.

Doing so, we note that in fact mt19937_1999 *fails* the Diehard bitstream test quite unambiguously at `-p 500`. The p-values returned from the test are systematically *too good* (strongly biased towards higher values of p)!

This in turn suggests that we rerun the *other* 20-bit tests with larger values of p. Perhaps we just got lucky and there are features in the histograms of p that are systematic but not yet significant compared to the statistical noise still residual with only 100 samples. We try `dieharder -d 6 -t 0 -p 500` to see how OPSO fares with a lot more p-values in the final KS test, and discover that yes, mt19937_1999 *passes* OPSO while failing bitstream at the Dieharder (enhanced Diehard) level. The Diehard tests are *very good* at revealing certain kinds of non-randomness, and are even capable of some fairly subtle discrimination in that regard.

mt19937_1999 also passes the Diehard Count the 1s tests handily. Note that this tests something completely different from the STS monobit test or bit distribution test at $n = 1$ – it is more concerned with detecting midrange bit correlations within the overlapping stream. Because we already know mt19937_1999 is 5 bit random, this isn't a complete surprise – even though the test uses all 8 bits of a byte, it is balanced in 1's overall and has the right frequency of bit patterns out through 5 bits, so a test like this that is primarily sensitive to having the right number of bits only on a bytewise basis seems likely to pass.

The next three tests – Diehard parking lot and minimum distance (in 2d and 3d) are tests that are at least weakly sensitive to the bunching of uniform deviates picked in coordinate ntuples on hyperplanes. The first attempts to cover an (integer) grid with non-overlapping "cars" and it is hoped that one will record an excess of crashes if the coordinates are bunched compared to truly random. It is most sensitive to gross deviations from uniformity, as the tiny deviations associated with hyperplane formation are likely block-averaged by the truncation process converting unsigned integers or uniform deviates into integers over a smaller range. The minimum distance tests are similar but instead of looking for excess crashes it picks random coordinates that are uniform deviates and looking for the minimum distance between all possible pairs. This test should be much *more* sensitive to hyperplanar bunching that occurs at the dimensionality of the fields being so filled, as one might reasonably expect that any such bunching will reduce the average minimum distance observed after some number of trials. Knuth suggests alternative ways of making the same determination (perhaps more accurately and repro-

ducibly), and this *kind* of test should fairly clearly be carried out in a systematic study of higher dimension until a failure is observed as the bit distribution (STS series) test is above. In any event, mt19937_1999 passes the parking lot and minimum distance test in 2d and 3d, suggesting that it is reasonably uniform through three dimensions.

The Diehard squeeze test is another measure of the uniformity of the distribution, computing the *distribution* of the number of multiplications required to reduce the maximum signed integer 2147483647 to 1 by multiplying it by a uniform deviate and rounding the double precision result up to the next highest integer. This test is repeated many times and χ^2 test on the frequency histogram used to generate a test p-value. mt19937_1999 passes this test handily.

The Diehard sums test also tests the distribution of uniform deviates by summing them 100 at a time and computing the distribution of totals, then reducing that distribution to a p-value. Dieharder performs a final KS test as usual on the results from 100 tests instead of the 10 that were the default in Diehard, and uses non-overlapping sums by default as well. mt19937_1999 passes the sums test either way.

The Diehard runs test counts the frequencies of up-runs and down-runs (sequences of random integers or uniform deviates that strictly increase or decrease) in a large sample sequence. Diehard used uniform deviates, but Dieharder uses random unsigned integers as examining them is much faster – uniform deviates usually being formed by performing a division on random unsigned integers – and obviously yield identical (but opposite) information with only the interpretation of up and down runs being interchanged. mt19937_1999 passes the runs tests in both directions easily.

Finally, mt19937_1999 passes both aspects of the Diehard craps test, producing the correct overall probability of winning and the correct distribution for the throws required to end the game.

As one expects, the mt19937_1999 RNG passes both the STS monobit and runs tests, given that we have already determined that it is 5 bit random in the bit distribution test above (equivalent to STS series).

In summary, mt19937_1999 does *not* pass all the Dieharder tests derived from Diehard tests, failing both the bitstream and the overlapping 5-permutations test. In both cases the RNG produces "reasonable" p-values quite a lot of the time (permitting one to conclude that the Diehard suite of tests were all "passed" by this RNG). However, Dieharder has revealed that those p-values were produced systematically in the wrong *proportions* so that the final KS test is not passed and the null hypothesis must be rejected.

Still, it is clear that for nearly all purposes, mt19937_1999 is an excellent RNG – as good as any available in the GSL. We will not make the mistake of stating that it "passes Dieharder" as we do not wish to imply that Dieharder is a benchmark test to be passed but rather a toolset that can be used to explore. Dieharder contains tests that *no* RNG we've tested is able to "pass", and we expect to add more. We would much prefer that Diehard return specific information about where, and how, any given RNG *fails* given that all of them are *pseudo*-random number generators and hence bound to fail in some respect or another.

So much for good RNGs. What about bad ones?

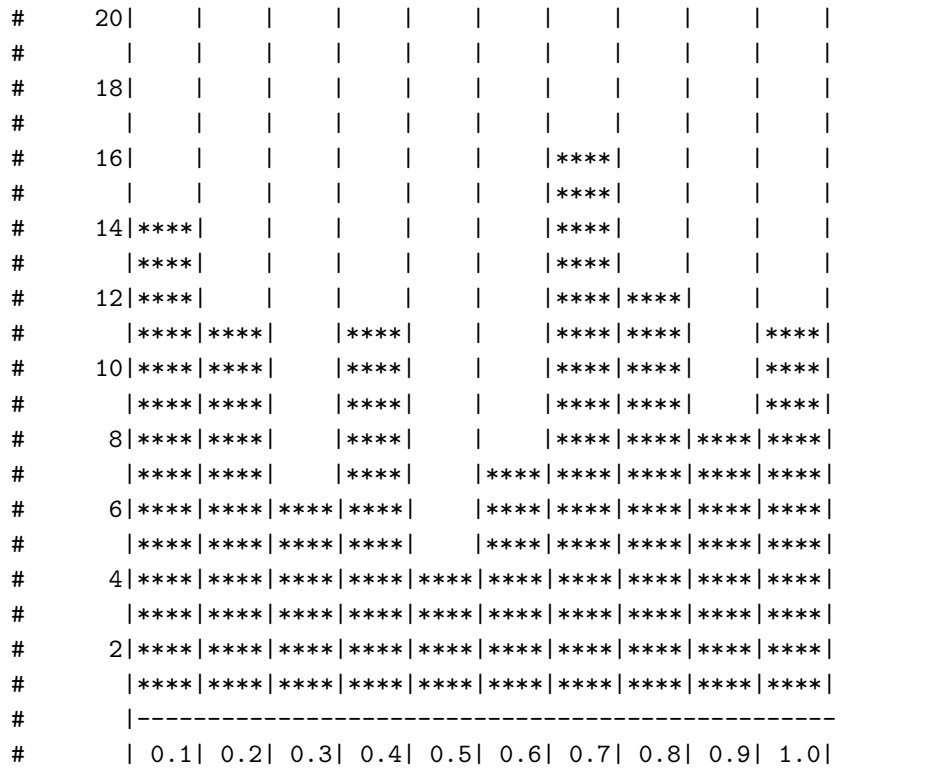
7.1 A Bad Generator: randu

Let us look now at an "infamously bad" random number generator, randu. randu is a linear congruential generator that is *so* bad that it has its own Wikipedia page[**?**] extolling its complete lack of virtue. When successive points in a 3 dimensional space are selected with randu, they all fall on 15 distinct planes. Let us examine randu with Dieharder and see what it tells us about its suitability as a RNG.

```
#==================================================================
#                        RGB Timing Test
#
# This test times the selected random number generator only.  It is
# generally run at the beginning of a run of -a(ll) the tests to provide
# some measure of the relative time taken up generating random numbers
# for the various generators and tests.
#==================================================================
#==================================================================
# rgb_timing() test using the randu generator
# Average time per rand = 2.538020e+01 nsec.
# Rands per second = 3.940079e+07.

#==================================================================
#                    RGB Bit Persistence Test
# This test generates 256 sequential samples of an random unsigned
# integer from the given rng.  Successive integers are logically
# processed to extract a mask with 1's whereever bits do not
# change.  Since bits will NOT change when filling e.g. unsigned
# ints with 16 bit ints, this mask logically &'d with the maximum
# random number returned by the rng.  All the remaining 1's in the
# resulting mask are therefore significant -- they represent bits
# that never change over the length of the test.  These bits are
# very likely the reason that certain rng's fail the monobit
# test -- extra persistent e.g. 1's or 0's inevitably bias the
# total bitcount.  In many cases the particular bits repeated
# appear to depend on the seed.  If the -i flag is given, the
# entire test is repeated with the rng reseeded to generate a mask
# and the extracted mask cumulated to show all the possible bit
# positions that might be repeated for different seeds.
#==================================================================
#                         Run Details
# Random number generator tested: randu
# Samples per test pvalue = 256 (test default is 256)
```

```
# P-values in final KS test = 1 (test default is 1)
# Samples per test run = 256, tsamples ignored
# Test run 1 times to cumulate unchanged bit mask
#=================================================================
#                          Results
# Results for randu rng, using its 31 valid bits:
# (Cumulated mask of zero is good.)
# cumulated_mask =          5 = 00000000000000000000000000000101
# randm_mask     = 2147483647 = 01111111111111111111111111111111
# random_max     = 2147483647 = 01111111111111111111111111111111
# rgb_persist test FAILED (bits repeat)
#=================================================================

#=================================================================
#                 RGB Bit Distribution Test
# Accumulates the frequencies of all n-tuples of bits in a list
# of random integers and compares the distribution thus generated
# with the theoretical (binomial) histogram, forming chisq and the
# associated p-value.  In this test n-tuples are selected without
# WITHOUT overlap (e.g. 01|10|10|01|11|00|01|10) so the samples
# are independent.  Every other sample is offset modulus of the
# sample index and ntuple_max.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
# Testing ntuple = 1
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
```

```
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for RGB Bit Distribution Test
# Assessment:
# FAILED at < 0.01%.
# Generator randu FAILS at 0.01% for 1-tuplets.  rgb_bitdist terminating.

#=================================================================
#                Diehard "Birthdays" test (modified).
# Each test determines the number of matching intervals from 512
# "birthdays" (by default) drawn on a 24-bit "year" (by
# default).  This is repeated 100 times (by default) and the
# results cumulated in a histogram.  Repeated intervals should be
# distributed in a Poisson distribution if the underlying generator
# is random enough, and a a chisq and p-value for the test are
# evaluated relative to this null hypothesis.
#
# It is recommended that you run this at or near the original
# 100 test samples per p-value with -t 100.
#
# Two additional parameters have been added. In diehard, nms=512
# but this CAN be varied and all Marsaglia's formulae still work.  It
# can be reset to different values with -x nmsvalue.
# Similarly, nbits "should" 24, but we can really make it anything
# we want that's less than or equal to rmax_bits = 32.  It can be
# reset to a new value with -y nbits.  Both default to diehard's
# values if no -x or -y options are used.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# 512 samples drawn from 24-bit integers masked out of a
# 31 bit random integer.  lambda = 2.000000, kmax = 6, tsamples = 100
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```




```
#=====================================================================
#                          Results
# Kuiper KS: p = 0.538138 for Diehard Birthdays Test
# Assessment:
# PASSED at > 5%.

#=====================================================================
#          Diehard Overlapping 5-{}Permutations Test.
# This is the OPERM5 test.  It looks at a sequence of one mill-
# ion 32-bit random integers.  Each set of five consecutive
# integers can be in one of 120 states, for the 5! possible or-
# derings of five numbers.  Thus the 5th, 6th, 7th,...numbers
# each provide a state. As many thousands of state transitions
# are observed,  cumulative counts are made of the number of
# occurences of each state.  Then the quadratic form in the
# weak inverse of the 120x120 covariance matrix yields a test
# equivalent to the likelihood ratio test that the 120 cell
# counts came from the specified (asymptotically) normal dis-
# tribution with the specified 120x120 covariance matrix (with
# rank 99).  This version uses 1,000,000 integers, twice.
#=====================================================================
#                         Run Details
# Random number generator tested: randu
```

```
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^28 for 100 samples.
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     32|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     28|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     20|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#     12|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#      8|****|****|    |****|    |****|    |    |    |****|
#       |****|****|****|****|    |****|    |    |    |****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard Overlapping 5-{}permutations Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#               Diehard 32x32 Binary Rank Test
# This is the BINARY RANK TEST for 31x31 matrices. The leftmost
# 31 bits of 31 random integers from the test sequence are used
# to form a 31x31 binary matrix over the field {0,1}. The rank
# is determined. That rank can be from 0 to 31, but ranks< 28
# are rare, and their counts are pooled with those for rank 28.
# Ranks are found for (default) 40,000 such random matrices and
# a chisquare test is performed on counts for ranks 31,30,29 and
# <=28.
#
```

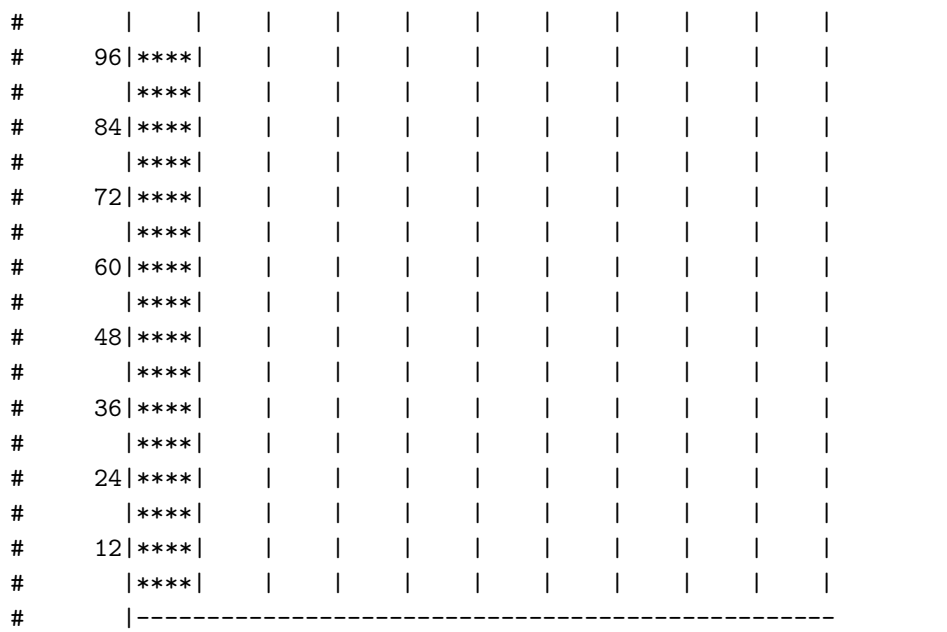
```
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#=================================================================
#                         Run Details
# Random number generator tested: randu
# Samples per test pvalue = 40000 (test default is 40000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard 32x32 Rank Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#              Diehard 6x8 Binary Rank Test
# This is the BINARY RANK TEST for 6x8 matrices.  From each of
# six random 32-bit integers from the generator under test, a
# specified byte is chosen, and the resulting six bytes form a
# 6x8 binary matrix whose rank is determined.  That rank can be
# from 0 to 6, but ranks 0,1,2,3 are rare; their counts are
```

```
# pooled with those for rank 4. Ranks are found for 100,000
# random matrices, and a chi-square test is performed on
# counts for ranks 6,5 and <=4.
#
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     72|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     64|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     56|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     40|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     32|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#      8|****|****|    |    |    |    |    |    |    |    |
#       |****|****|    |    |    |****|    |****|    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard 6x8 Binary Rank Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                  Diehard Bitstream Test.
# The file under test is viewed as a stream of bits. Call them
```

```
# b1,b2,... .  Consider an alphabet with two "letters", 0 and 1
# and think of the stream of bits as a succession of 20-letter
# "words", overlapping.  Thus the first word is b1b2...b20, the
# second is b2b3...b21, and so on.  The bitstream test counts
# the number of missing 20-letter (20-bit) words in a string of
# 2^21 overlapping 20-letter words.  There are 2^20 possible 20
# letter words.  For a truly random string of 2^21+19 bits, the
# number of missing words j should be (very close to) normally
# distributed with mean 141,909 and sigma 428.  Thus
#  (j-141909)/428 should be a standard normal variate (z score)
# that leads to a uniform [0,1) p value.  The test is repeated
# twenty times.
#
# Note that of course we do not "restart file", when using gsl
# generators, we just crank out the next random number.
# We also do not bother to overlap the words.  rands are cheap.
# Finally, we repeat the test (usually) more than twenty time.
#=================================================================
#                         Run Details
# Random number generator tested: randu
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
```

```
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-----------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Bitstream Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#        Diehard Overlapping Pairs Sparse Occupance (OPSO)
# The OPSO test considers 2-letter words from an alphabet of
# 1024 letters.  Each letter is determined by a specified ten
# bits from a 32-bit integer in the sequence to be tested. OPSO
# generates  2^21 (overlapping) 2-letter words  (from 2^21+1
# "keystrokes")  and counts the number of missing words---that
# is 2-letter words which do not appear in the entire sequence.
# That count should be very close to normally distributed with
# mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should
# be a standard normal variable. The OPSO test takes 32 bits at
# a time from the test file and uses a designated set of ten
# consecutive bits. It then restarts the file for the next de-
# signated 10 bits, and so on.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
```

```
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard OPSO Test
# Assessment:
# FAILED at < 0.01%.

#================================================================
#   Diehard Overlapping Quadruples Sparce Occupancy (OQSO) Test
#
#  Similar, to OPSO except that it considers 4-letter
#  words from an alphabet of 32 letters, each letter determined
#  by a designated string of 5 consecutive bits from the test
#  file, elements of which are assumed 32-bit random integers.
#  The mean number of missing words in a sequence of 2^21 four-
#  letter words,  (2^21+3 "keystrokes"), is again 141909, with
#  sigma = 295.  The mean is based on theory; sigma comes from
#  extensive simulation.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#================================================================
#                         Run Details
# Random number generator tested: randu
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
```

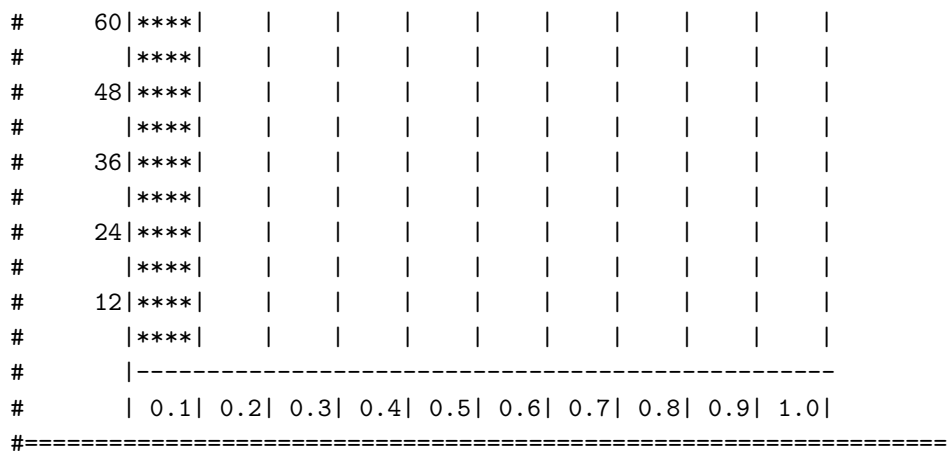



```
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard OQSO Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                   Diehard DNA Test.
#
#   The DNA test considers an alphabet of 4 letters::  C,G,A,T,
# determined by two designated bits in the sequence of random
# integers being tested.  It considers 10-letter words, so that
# as in OPSO and OQSO, there are 2^20 possible words, and the
# mean number of missing words from a string of 2^21  (over-
# lapping)  10-letter  words (2^21+9 "keystrokes") is 141909.
# The standard deviation sigma=339 was determined as for OQSO
# by simulation.  (Sigma for OPSO, 290, is the true value (to
# three places), not determined by simulation.
#
# Note 2^21 = 2097152
# Note also that we don't bother with overlapping keystrokes
# (and sample more rands -- rands are now cheap).
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 2097152 (test default is 2097152)
```

```
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard DNA Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#          Diehard Count the 1s (stream) (modified) Test.
# Consider the file under test as a stream of bytes (four per
# 32 bit integer).  Each byte can contain from 0 to 8 1's,
# with probabilities 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the stream of bytes provide a string of overlapping  5-letter
# words, each "letter" taking values A,B,C,D,E. The letters are
# determined by the number of 1's in a byte::  0,1,or 2 yield A,
# 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus
# we have a monkey at a typewriter hitting five keys with vari-
# ous probabilities (37,56,70,56,37 over 256).  There are 5^5
```

```
# possible 5-letter words, and from a string of 256,000 (over-
# lapping) 5-letter words, counts are made on the frequencies
# for each word.   The quadratic form in the weak inverse of
# the covariance matrix of the cell counts provides a chisquare
# test::  Q5-Q4, the difference of the naive Pearson sums of
# (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Count the 1s (stream)
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#         Diehard Count the 1s Test (byte) (modified).
```

```
#     This is the COUNT-THE-1's TEST for specific bytes.
# Consider the file under test as a stream of 32-bit integers.
# From each integer, a specific byte is chosen , say the left-
# most::  bits 1 to 8. Each byte can contain from 0 to 8 1's,
# with probabilitie 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the specified bytes from successive integers provide a string
# of (overlapping) 5-letter words, each "letter" taking values
# A,B,C,D,E. The letters are determined  by the number of 1's,
# in that byte::  0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D,
# and  6,7 or 8 ---> E.  Thus we have a monkey at a typewriter
# hitting five keys with with various probabilities::  37,56,70,
# 56,37 over 256. There are 5^5 possible 5-letter words, and
# from a string of 256,000 (overlapping) 5-letter words, counts
# are made on the frequencies for each word. The quadratic form
# in the weak inverse of the covariance matrix of the cell
# counts provides a chisquare test::  Q5-Q4, the difference of
# the naive Pearson  sums of (OBS-EXP)^2/EXP on counts for 5-
# and 4-letter cell counts.
#
# Note: We actually cycle samples over all 0-31 bit offsets, so
# that if there is a problem with any particular offset it has
# a chance of being observed.  One can imagine problems with odd
# offsets but not even, for example, or only with the offset 7.
# tsamples and psamples can be freely varied, but you'll likely
# need tsamples >> 100,000 to have enough to get a reliable kstest
# result.
#==============================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#==============================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
```

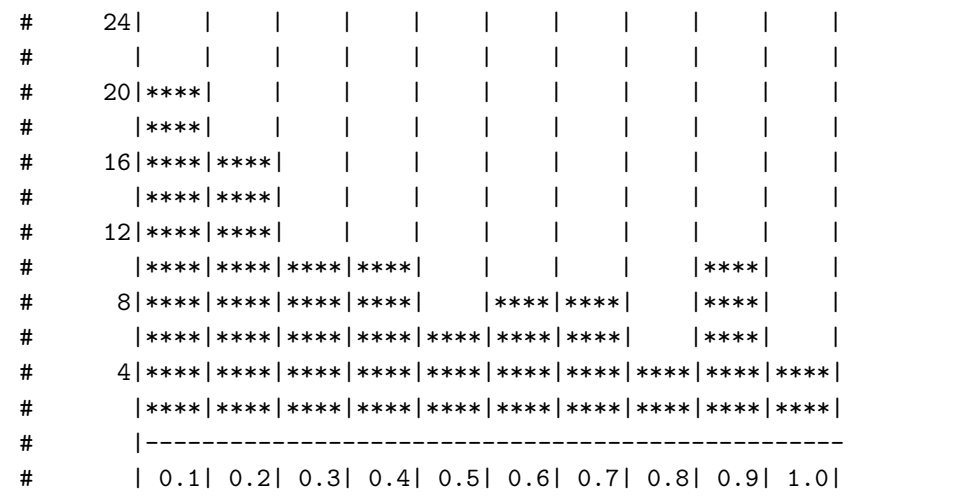



```
#==================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Count the 1s (byte)
# Assessment:
# FAILED at < 0.01%.

#==================================================================
#             Diehard Parking Lot Test (modified).
# This tests the distribution of attempts to randomly park a
# square car of length 1 on a 100x100 parking lot without
# crashing.  We plot n (number of attempts) versus k (number of
# attempts that didn't "crash" because the car squares
# overlapped and compare to the expected result from a perfectly
# random set of parking coordinates.  This is, alas, not really
# known on theoretical grounds so instead we compare to n=12,000
# where k should average 3523 with sigma 21.9 and is very close
# to normally distributed.  Thus (k-3523)/21.9 is a standard
# normal variable, which converted to a uniform p-value, provides
# input to a KS test with a default 100 samples.
#==================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 0 (test default is 0)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     16|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```

```
#     14|    |    |    |****|    |    |    |    |    |    |
#       |    |    |    |****|    |    |    |    |    |    |
#     12|    |    |    |****|    |    |****|    |    |    |
#       |    |    |    |****|    |    |****|    |    |    |
#     10|****|    |****|****|    |    |****|****|****|    |
#       |****|    |****|****|****|    |****|****|****|****|
#      8|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-----------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.977990 for Diehard Parking Lot Test
# Assessment:
# PASSED at > 5%.

#================================================================
#         Diehard Minimum Distance (2d Circle) Test
# Generate 8000 points in a 10000^2 square.  Determine the
# the shortest nearest neighbor distance R.  This should generate
# p = 1.0 - exp(-R^2/0.995).  Repeat for lots of samples, apply a
# KS test to see if p is uniform.
#
# The number of samples is fixed -- tsamples is ignored.
#================================================================
#                         Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     60|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     54|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     48|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     42|    |    |****|    |    |    |    |    |    |    |
```

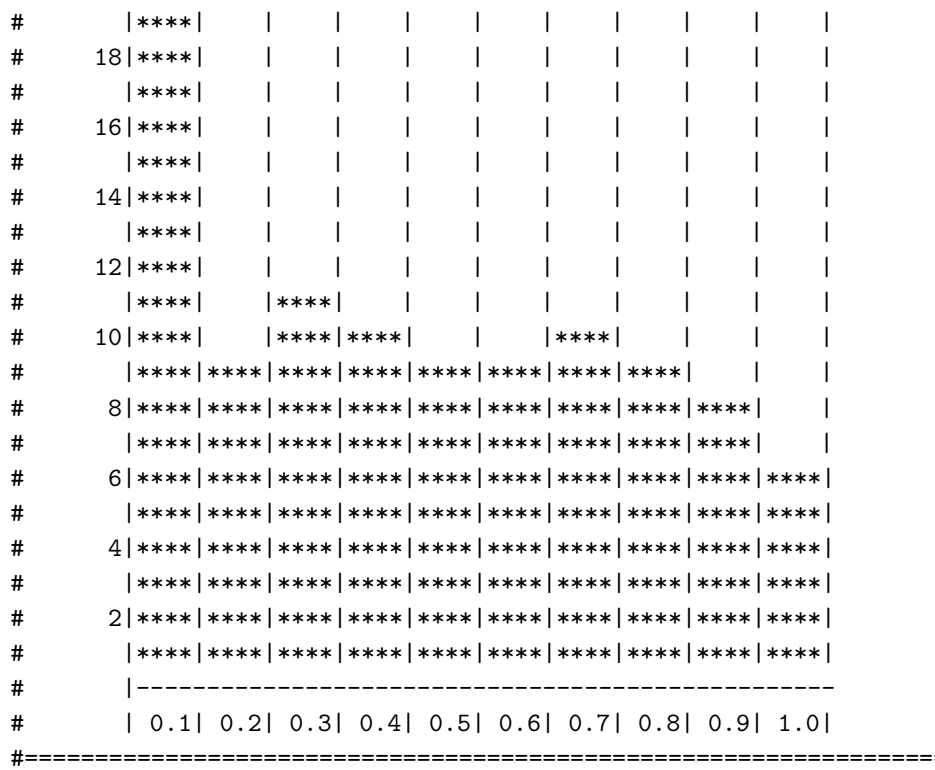
```
#       |    |    |****|    |    |    |    |    |    |    |
#     36|    |    |****|    |    |    |    |    |    |    |
#       |    |    |****|    |    |    |    |    |    |    |
#     30|    |    |****|    |    |    |    |    |    |    |
#       |    |    |****|    |    |    |    |    |    |    |
#     24|    |    |****|    |    |    |    |    |    |    |
#       |    |    |****|    |    |    |    |    |    |    |
#     18|    |    |****|    |    |    |    |    |    |    |
#       |    |    |****|    |    |    |    |    |    |    |
#     12|    |    |****|    |****|    |    |    |****|****|
#       |    |    |****|    |****|    |****|    |****|****|
#      6|    |    |****|    |****|****|****|    |****|****|
#       |    |    |****|    |****|****|****|    |****|****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Minimum Distance (2d Circle) Test
# Assessment:
# FAILED at < 0.01%.

#================================================================
#         Diehard Minimum Distance (2d Circle) Test
# Generate 8000 points in a 10000^2 square.  Determine the
# the shortest nearest neighbor distance R.  This should generate
# p = 1.0 - exp(-R^2/0.995).  Repeat for lots of samples, apply a
# KS test to see if p is uniform.
#
# The number of samples is fixed -- tsamples is ignored.
#================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     28|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```

```
#     24|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     20|****|    |****|    |    |    |    |    |    |    |
#       |****|    |****|    |    |    |    |    |    |    |
#     16|****|    |****|    |    |    |    |    |    |    |
#       |****|    |****|    |    |    |    |    |    |    |
#     12|****|****|****|****|****|    |    |    |    |    |
#       |****|****|****|****|****|    |    |    |    |    |
#      8|****|****|****|****|****|    |    |    |    |    |
#       |****|****|****|****|****|    |    |****|    |    |
#      4|****|****|****|****|****|****|****|****|    |    |
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.000001 for Diehard 3d Sphere (Minimum Distance) Test
# Assessment:
# FAILED at < 0.01%.

#================================================================
#                  Diehard Squeeze Test.
#  Random integers are floated to get uniforms on [0,1). Start-
#  ing with k=2^31=2147483647, the test finds j, the number of
#  iterations necessary to reduce k to 1, using the reduction
#  k=ceiling(k*U), with U provided by floating integers from
#  the file being tested.  Such j's are found 100,000 times,
#  then counts for the number of times j was <=6,7,...,47,>=48
#  are used to provide a chi-square test for cell frequencies.
#================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     28|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```



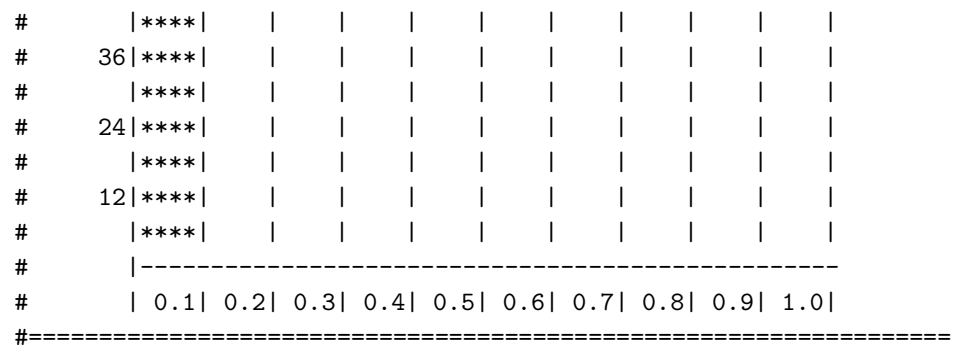

```
#=================================================================
#                          Results
# Kuiper KS: p = 0.013380 for Diehard Squeeze Test
# Assessment:
# POSSIBLY WEAK at < 5%.
# Recommendation:  Repeat test to verify failure.

#=================================================================
#                  Diehard Sums Test
# Integers are floated to get a sequence U(1),U(2),... of uni-
# form [0,1) variables.  Then overlapping sums,
#   S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed.
# The S's are virtually normal with a certain covariance mat-
# rix.  A linear transformation of the S's converts them to a
# sequence of independent standard normals, which are converted
# to uniform variables for a KSTEST. The  p-values from ten
# KSTESTs are given still another KSTEST.
#
# Note well:  -O causes the old diehard version to be run (more or
# less).  Omitting it causes non-overlapping sums to be used and
# directly tests the overall balance of uniform rands.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```

```
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     16|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |****|****|
#     14|    |    |    |    |    |    |    |    |****|****|
#       |    |    |    |    |    |    |    |    |****|****|
#     12|    |    |    |    |    |    |    |    |****|****|
#       |    |    |    |    |    |    |****|    |****|****|
#     10|    |****|    |    |    |    |****|    |****|****|
#       |****|****|    |    |    |    |****|    |****|****|
#      8|****|****|    |****|    |    |****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      6|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.214757 for Diehard Sums Test
# Assessment:
# PASSED at > 5%.

#=================================================================
#                Diehard "runs" test (modified).
# This tests the distribution of increasing and decreasing runs
# of integers.  If called with reasonable parameters e.g. -s 100
# or greater and -n 100000 or greater, it will compute a vector
# of p-values for up and down and verify that the proportion
# of these values less than 0.01 is consistent with a uniform
# distribution.
#=================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
```

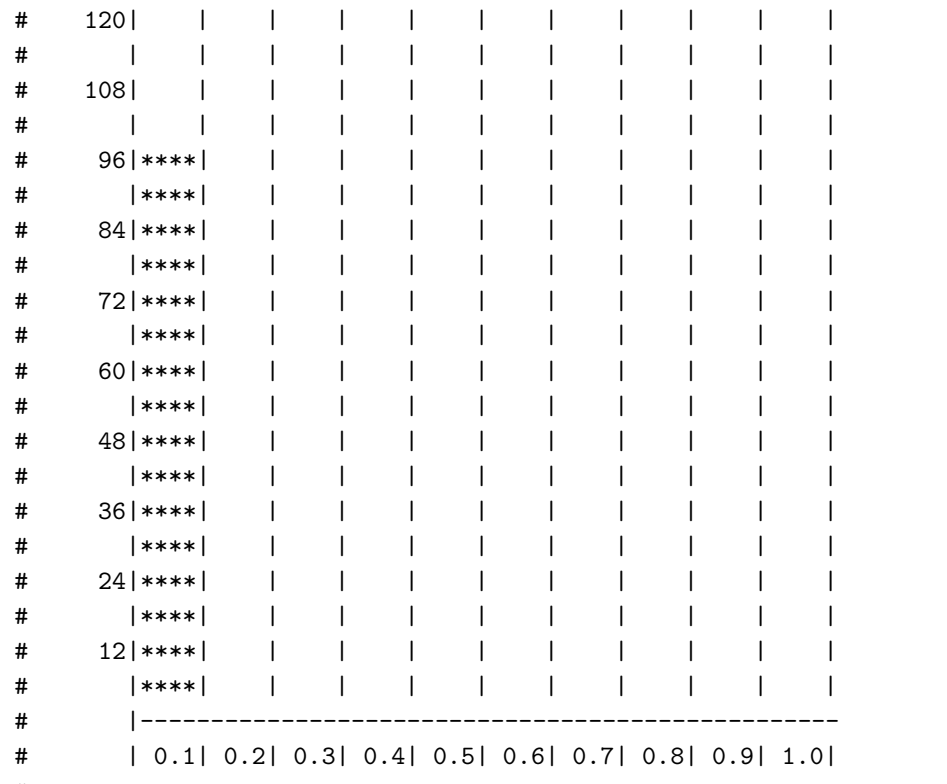



```
#=================================================================
#                          Results
# Kuiper KS: p = 0.599644 for Runs (up)
# Assessment:
# PASSED at > 5%.
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |****|    |    |    |    |    |    |    |
#       |****|    |****|    |    |    |    |    |    |    |
#     14|****|    |****|    |    |    |    |    |    |    |
#       |****|    |****|    |    |    |****|    |    |    |
#     12|****|    |****|    |    |    |****|    |    |    |
#       |****|    |****|    |    |    |****|    |****|    |
#     10|****|    |****|    |    |    |****|    |****|    |
#       |****|****|****|    |    |    |****|    |****|    |
#      8|****|****|****|****|    |    |****|****|****|    |
#       |****|****|****|****|    |****|****|****|****|    |
#      6|****|****|****|****|    |****|****|****|****|    |
#       |****|****|****|****|****|****|****|****|****|****|
```

```
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                           Results
# Kuiper KS: p = 0.316433 for Runs (down)
# Assessment:
# PASSED at > 5%.

#================================================================
#                    Diehard Craps Test
#  This is the CRAPS TEST. It plays 200,000 games of craps, finds
#  the number of wins and the number of throws necessary to end
#  each game.  The number of wins should be (very close to) a
#  normal with mean 200000p and variance 200000p(1-p), with
#  p=244/495.  Throws necessary to complete the game can vary
#  from 1 to infinity, but counts for all>21 are lumped with 21.
#  A chi-square test is made on the no.-of-throws cell counts.
#  Each 32-bit integer from the test file provides the value for
#  the throw of a die, by floating to [0,1), multiplying by 6
#  and taking 1 plus the integer part of the result.
#================================================================
#                          Run Details
# Random number generator tested: randu
# Samples per test pvalue = 200000 (test default is 200000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
```




```
#==========================================================================
#                            Results
# Kuiper KS: p = 0.000000 for Craps Test (mean)
# Assessment:
# FAILED at < 0.01%.
#==========================================================================
#                  Histogram of p-values
# Counting histogram bins, binscale = 0.100000
```




```
#==========================================================================
#                            Results
# Kuiper KS: p = 0.000000 for Craps Test (freq)
# Assessment:
# FAILED at < 0.01%.
```

```
#==================================================================
#                      STS Monobit Test
# Very simple.  Counts the 1 bits in a long string of random uints.
# Compares to expected number, generates a p-value directly from
# erfc().  Very effective at revealing overtly weak generators;
# Not so good at determining where stronger ones eventually fail.
#==================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     72|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     64|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     56|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     40|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     32|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#      8|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |****|****|****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#==================================================================
#                          Results
# Kuiper KS: p = 0.000000 for STS Monobit Test
# Assessment:
# FAILED at < 0.01%.

#==================================================================
#                       STS Runs Test
# Counts the total number of 0 runs + total number of 1 runs across
```

```
# a sample of bits.  Note that a 0 run must begin with 10 and end
# with 01.  Note that a 1 run must begin with 01 and end with a 10.
# This test, run on a bitstring with cyclic boundary conditions, is
# absolutely equivalent to just counting the 01 + 10 bit pairs.
# It is therefore totally redundant with but not as good as the
# rgb_bitdist() test for 2-tuples, which looks beyond the means to the
# moments, testing an entire histogram  of 00, 01, 10, and 11 counts
# to see if it is binomially distributed with p = 0.25.
#==================================================================
#                        Run Details
# Random number generator tested: randu
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     72|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     64|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     56|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     40|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     32|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     16|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#      8|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |****|****|    |****|****|****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#==================================================================
#                          Results
# Kuiper KS: p = 0.000000 for STS Runs Test
# Assessment:
# FAILED at < 0.01%.
```

We begin by noting that randu is about 40After all, we should make every effort to

explain why the RNG was quite popular and in widespread use back in the 70's. Then the bad news begins.

The bit persistence test shows that randu has only 31 significant bits (signed integer) which is to be expected and not itself a problem. Howevere it *also* shows that the cumulated mask is not zero! Two of the three least significant bits produced by the randu iterated map *never change*. This is usually a very bad sign for a random number generator, as it means that the generator will very likely fail even the simplest tests for uniformity.

randu does not disappoint us in this regard. It is not even 1 bit random according to the bit distribution series test. To put it bluntly, the generator produces an unbalanced number of 0's and 1's, meaning that it *cannot* produced balanced distributions of higher order numbers of bits.

That does not mean, of course, that it will fail all the rest of the tests. There are usually at least *some* tests that are relatively insensitive to the particular kinds of correlations in nearly any RNG that was random "enough" to have been named and distributed as a RNG (and hence make it into the GSL). Indeed, as we look down the list we see that it *passes* the Diehard birthdays test, the parking lot test, and would have passed the squeeze test by the usual $p > 0.01$ condition (replaced in Dieharder by a "answer cloudy, try again later" sort of message, where one is advised to try again *harder* by increasing the number of samples in the final KS test with `-p 500` or the like). It passes the sums test (apparently) and the runs test. It fails *all of the rest*, including of course both the monobit and STS runs test, most of them by directly producing nothing p-values very close to 0.0, making the final KS test moot.

Even this RNG, well-known to be *terrible* for most purposes, is capable of producing sequences that are "random enough" to pass some of Diehard's *difficult* tests at the default Dieharder values, which in turn are invariably more challenging than those of Diehard. What happens if one cranks up the tests to to where they make a RNG die *harder*? In the case of randu, mostly it dies. For example the sums test fails without any doubt at 500 p-values. runs fails badly at 1000. The squeeze test fails at 300. Even parking lot fails at 2000 p-values. With randu, "passing" Diehard tests appears to be a matter more of how hard you look, not whether or not the result is truly random according to the test measure.

In contrast, mt19937_1999 still *passes* the runs test at *10000* p-samples, although at that point the pass gets to be visibly marginal. We reiterate the previous observation concerning the true purpose of Dieharder with its variable controls: *The point isn't "passing" any given test, the point is determining where it fails in a quantitative way!* When this is done one can *compare* the performance of different RNGs in a meaningful way on a test by test basis.

From the above we see that Dieharder correctly leads us to conclude that randu is a pretty poor generator that might well not even produce a zero-sum *coin flip game*, let alone produce an unbiased result for something like a state lottery. For all of that, randu is *not* the worst RNG in the GSL. Consider the following.

7.2 An Ugly Generator: slatec

slatec is the GSL encapsulation of the RAND function from the SLATEC Common Mathematical Library, still available from www.netlib.org[**?**]. The meaning of the SLATEC acronym is lost in time – one might guess that the "S" stands for "Sandia" or "Scientific", the "LA" likely stands for "Los Alamos" and the "TEC" conceivably refers to its presumed utility for technical applications (the only header information available in the archived sources suggest that it was developed by a consortium of DOE and DOD national laboratories). Again it is a linear congruential generator, this time one from Knuth that was subjected to a spectral test for certain multipliers to pick the "best" one. Again we may safely presume that slatec was used to perform much simulation work in the 80's, quite likely (given its sponsors) in the field of nuclear device design.

Let us see what Dieharder makes of it:

```
#==================================================================
#                       RGB Timing Test
#
# This test times the selected random number generator only.  It is
# generally run at the beginning of a run of -a(ll) the tests to provide
# some measure of the relative time taken up generating random numbers
# for the various generators and tests.
#==================================================================
#==================================================================
# rgb_timing() test using the slatec generator
# Average time per rand = 3.582940e+01 nsec.
# Rands per second = 2.791004e+07.

#==================================================================
#                    RGB Bit Persistence Test
# This test generates 256 sequential samples of an random unsigned
# integer from the given rng.  Successive integers are logically
# processed to extract a mask with 1's whereever bits do not
# change.  Since bits will NOT change when filling e.g. unsigned
# ints with 16 bit ints, this mask logically &'d with the maximum
# random number returned by the rng.  All the remaining 1's in the
# resulting mask are therefore significant -- they represent bits
# that never change over the length of the test.  These bits are
# very likely the reason that certain rng's fail the monobit
# test -- extra persistent e.g. 1's or 0's inevitably bias the
# total bitcount.  In many cases the particular bits repeated
# appear to depend on the seed.  If the -i flag is given, the
# entire test is repeated with the rng reseeded to generate a mask
# and the extracted mask cumulated to show all the possible bit
# positions that might be repeated for different seeds.
```

```
#==================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 256 (test default is 256)
# P-values in final KS test = 1 (test default is 1)
# Samples per test run = 256, tsamples ignored
# Test run 1 times to cumulate unchanged bit mask
#==================================================================
#                           Results
# Results for slatec rng, using its 22 valid bits:
# (Cumulated mask of zero is good.)
# cumulated_mask =          0 = 00000000000000000000000000000000
# randm_mask     =    4194303 = 00000000001111111111111111111111
# random_max     =    4194303 = 00000000001111111111111111111111
# rgb_persist test PASSED (no bits repeat)
#==================================================================

#==================================================================
#                 RGB Bit Distribution Test
# Accumulates the frequencies of all n-tuples of bits in a list
# of random integers and compares the distribution thus generated
# with the theoretical (binomial) histogram, forming chisq and the
# associated p-value.  In this test n-tuples are selected without
# WITHOUT overlap (e.g. 01|10|10|01|11|00|01|10) so the samples
# are independent.  Every other sample is offset modulus of the
# sample index and ntuple_max.
#==================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
# Testing ntuple = 1
#==================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
```

```
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-----------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for RGB Bit Distribution Test
# Assessment:
# FAILED at < 0.01%.
# Generator slatec FAILS at 0.01% for 1-tuplets.  rgb_bitdist terminating.

#=================================================================
#                Diehard "Birthdays" test (modified).
# Each test determines the number of matching intervals from 512
# "birthdays" (by default) drawn on a 24-bit "year" (by
# default).  This is repeated 100 times (by default) and the
# results cumulated in a histogram.  Repeated intervals should be
# distributed in a Poisson distribution if the underlying generator
# is random enough, and a a chisq and p-value for the test are
# evaluated relative to this null hypothesis.
#
# It is recommended that you run this at or near the original
# 100 test samples per p-value with -t 100.
#
# Two additional parameters have been added. In diehard, nms=512
# but this CAN be varied and all Marsaglia's formulae still work.  It
# can be reset to different values with -x nmsvalue.
# Similarly, nbits "should" 24, but we can really make it anything
# we want that's less than or equal to rmax_bits = 32.  It can be
# reset to a new value with -y nbits.  Both default to diehard's
# values if no -x or -y options are used.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# 512 samples drawn from 22-bit integers masked out of a
```

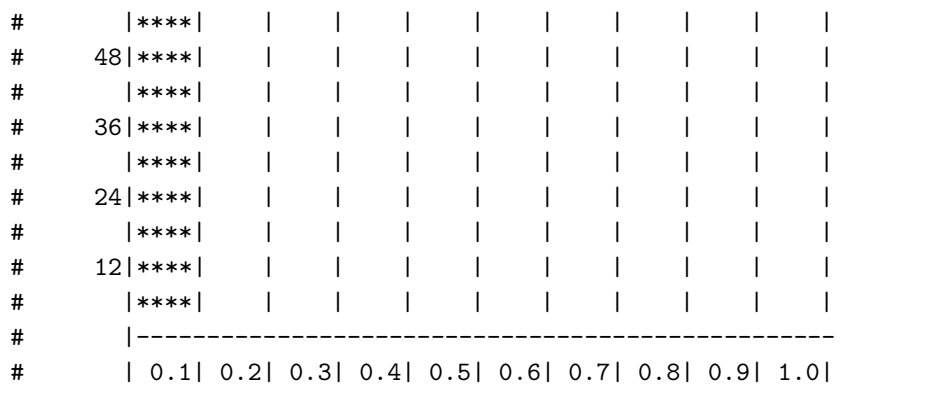
```
# 22 bit random integer.  lambda = 8.000000, kmax = 2, tsamples = 100
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     72|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     64|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     56|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     48|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     40|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     32|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     24|****|    |    |    |    |****|    |    |    |    |
#       |****|    |    |    |    |****|    |    |    |    |
#     16|****|    |    |    |    |****|    |    |    |    |
#       |****|    |    |    |    |****|    |    |    |    |
#      8|****|    |    |    |    |****|    |    |    |    |
#       |****|****|    |    |    |****|    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Birthdays Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#          Diehard Overlapping 5-{}Permutations Test.
# This is the OPERM5 test.  It looks at a sequence of one mill-
# ion 32-bit random integers.  Each set of five consecutive
# integers can be in one of 120 states, for the 5! possible or-
# derings of five numbers.  Thus the 5th, 6th, 7th,...numbers
# each provide a state. As many thousands of state transitions
# are observed,  cumulative counts are made of the number of
# occurences of each state.  Then the quadratic form in the
# weak inverse of the 120x120 covariance matrix yields a test
# equivalent to the likelihood ratio test that the 120 cell
# counts came from the specified (asymptotically) normal dis-
# tribution with the specified 120x120 covariance matrix (with
```

```
# rank 99).  This version uses 1,000,000 integers, twice.
#
# Note that Dieharder runs the test 100 times, not twice, by
# default.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 1000000 (test default is 1000000)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^28 for 100 samples.
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     72|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     64|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |****|
#     56|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     48|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     40|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     32|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     24|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     16|    |    |    |    |    |    |****|    |    |****|
#       |    |    |    |    |    |    |****|    |    |****|
#      8|****|****|    |    |    |    |****|    |    |****|
#       |****|****|    |    |    |    |****|    |    |****|
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Overlapping 5-{}permutations Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#               Diehard 32x32 Binary Rank Test
# This is the BINARY RANK TEST for 31x31 matrices. The leftmost
# 31 bits of 31 random integers from the test sequence are used
```

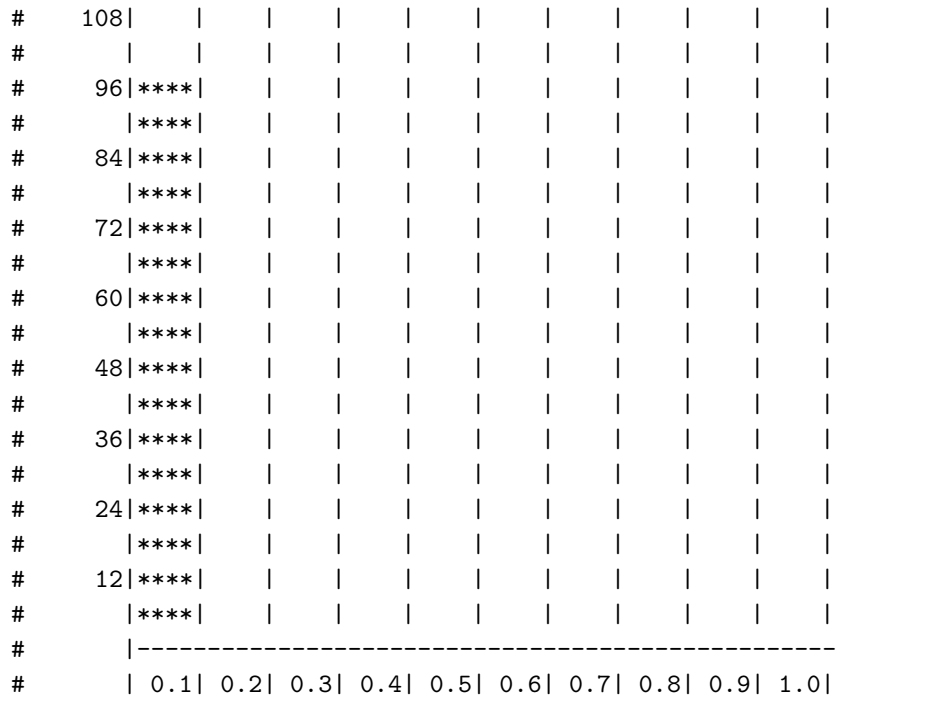
```
# to form a 31x31 binary matrix over the field {0,1}. The rank
# is determined. That rank can be from 0 to 31, but ranks< 28
# are rare, and their counts are pooled with those for rank 28.
# Ranks are found for (default) 40,000 such random matrices and
# a chisquare test is performed on counts for ranks 31,30,29 and
# <=28.
#
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 40000 (test default is 40000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard 32x32 Rank Test
# Assessment:
# FAILED at < 0.01%.
```

```
#================================================================
#              Diehard 6x8 Binary Rank Test
# This is the BINARY RANK TEST for 6x8 matrices.  From each of
# six random 32-bit integers from the generator under test, a
# specified byte is chosen, and the resulting six bytes form a
# 6x8 binary matrix whose rank is determined.  That rank can be
# from 0 to 6, but ranks 0,1,2,3 are rare; their counts are
# pooled with those for rank 4. Ranks are found for 100,000
# random matrices, and a chi-square test is performed on
# counts for ranks 6,5 and <=4.
#
# As always, the test is repeated and a KS test applied to the
# resulting p-values to verify that they are approximately uniform.
#================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    100|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     90|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     80|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     70|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     50|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     40|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     30|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     20|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#     10|****|    |    |    |    |    |    |    |    |****|
#       |****|    |    |    |    |    |    |    |    |****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
```

```
# Kuiper KS: p = 0.000000 for Diehard 6x8 Binary Rank Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                   Diehard Bitstream Test.
# The file under test is viewed as a stream of bits. Call them
# b1,b2,... .  Consider an alphabet with two "letters", 0 and 1
# and think of the stream of bits as a succession of 20-letter
# "words", overlapping.  Thus the first word is b1b2...b20, the
# second is b2b3...b21, and so on.  The bitstream test counts
# the number of missing 20-letter (20-bit) words in a string of
# 2^21 overlapping 20-letter words.  There are 2^20 possible 20
# letter words.  For a truly random string of 2^21+19 bits, the
# number of missing words j should be (very close to) normally
# distributed with mean 141,909 and sigma 428.  Thus
#  (j-141909)/428 should be a standard normal variate (z score)
# that leads to a uniform [0,1) p value.  The test is repeated
# twenty times.
#
# Note that of course we do not "restart file", when using gsl
# generators, we just crank out the next random number.
# We also do not bother to overlap the words.  rands are cheap.
# Finally, we repeat the test (usually) more than twenty time.
#=================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
```




```
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Bitstream Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#        Diehard Overlapping Pairs Sparse Occupance (OPSO)
# The OPSO test considers 2-letter words from an alphabet of
# 1024 letters.  Each letter is determined by a specified ten
# bits from a 32-bit integer in the sequence to be tested. OPSO
# generates  2^21 (overlapping) 2-letter words  (from 2^21+1
# "keystrokes")  and counts the number of missing words---that
# is 2-letter words which do not appear in the entire sequence.
# That count should be very close to normally distributed with
# mean 141,909, sigma 290. Thus (missingwrds-141909)/290 should
# be a standard normal variable. The OPSO test takes 32 bits at
# a time from the test file and uses a designated set of ten
# consecutive bits. It then restarts the file for the next de-
# signated 10 bits, and so on.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```



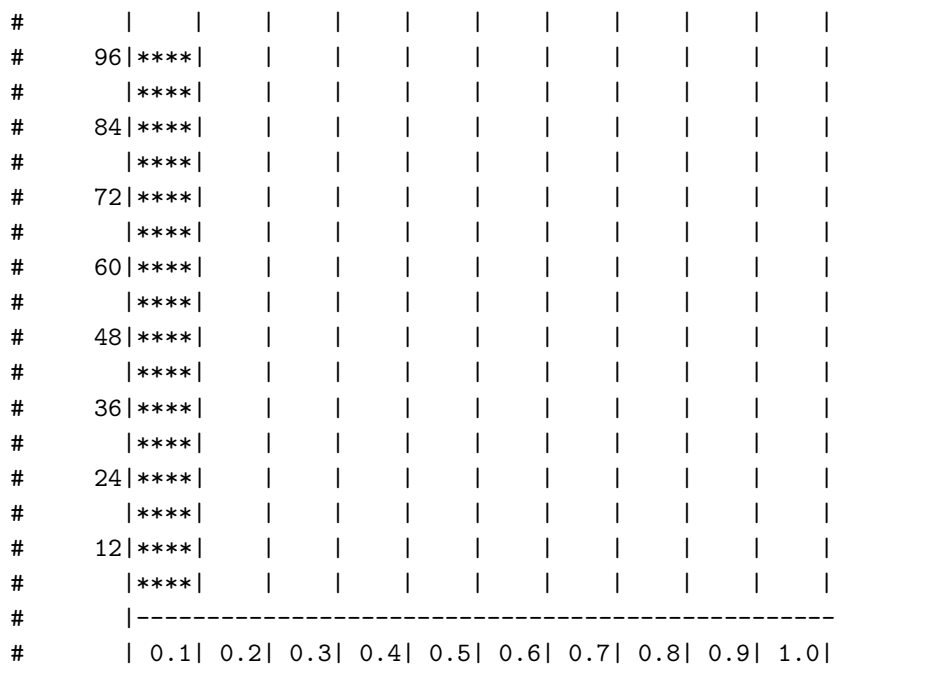

```
#=============================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard OPSO Test
# Assessment:
# FAILED at < 0.01%.

#=============================================================
#   Diehard Overlapping Quadruples Sparce Occupancy (OQSO) Test
#
#  Similar, to OPSO except that it considers 4-letter
#  words from an alphabet of 32 letters, each letter determined
#  by a designated string of 5 consecutive bits from the test
#  file, elements of which are assumed 32-bit random integers.
#  The mean number of missing words in a sequence of 2^21 four-
#  letter words,  (2^21+3 "keystrokes"), is again 141909, with
#  sigma = 295.  The mean is based on theory; sigma comes from
#  extensive simulation.
#
#  Note 2^21 = 2097152, tsamples cannot be varied.
#=============================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
```

```
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard OQSO Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                    Diehard DNA Test.
#
#   The DNA test considers an alphabet of 4 letters::  C,G,A,T,
# determined by two designated bits in the sequence of random
# integers being tested.  It considers 10-letter words, so that
# as in OPSO and OQSO, there are 2^20 possible words, and the
# mean number of missing words from a string of 2^21  (over-
# lapping)  10-letter  words (2^21+9 "keystrokes") is 141909.
# The standard deviation sigma=339 was determined as for OQSO
# by simulation.  (Sigma for OPSO, 290, is the true value (to
# three places), not determined by simulation.
#
```

```
# Note 2^21 = 2097152
# Note also that we don't bother with overlapping keystrokes
# (and sample more rands -- rands are now cheap).
#=================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 2097152 (test default is 2097152)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard DNA Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#          Diehard Count the 1s (stream) (modified) Test.
# Consider the file under test as a stream of bytes (four per
# 32 bit integer).  Each byte can contain from 0 to 8 1's,
```

```
# with probabilities 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the stream of bytes provide a string of overlapping  5-letter
# words, each "letter" taking values A,B,C,D,E. The letters are
# determined by the number of 1's in a byte::  0,1,or 2 yield A,
# 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus
# we have a monkey at a typewriter hitting five keys with vari-
# ous probabilities (37,56,70,56,37 over 256).  There are 5^5
# possible 5-letter words, and from a string of 256,000 (over-
# lapping) 5-letter words, counts are made on the frequencies
# for each word.   The quadratic form in the weak inverse of
# the covariance matrix of the cell counts provides a chisquare
# test::  Q5-Q4, the difference of the naive Pearson sums of
# (OBS-EXP)^2/EXP on counts for 5- and 4-letter cell counts.
#=============================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#=============================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     84|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     72|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     48|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     36|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     24|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     12|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=============================================================
```

```
#                           Results
# Kuiper KS: p = 0.000000 for Diehard Count the 1s (stream)
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#         Diehard Count the 1s Test (byte) (modified).
#     This is the COUNT-THE-1's TEST for specific bytes.
# Consider the file under test as a stream of 32-bit integers.
# From each integer, a specific byte is chosen , say the left-
# most::  bits 1 to 8. Each byte can contain from 0 to 8 1's,
# with probabilitie 1,8,28,56,70,56,28,8,1 over 256.  Now let
# the specified bytes from successive integers provide a string
# of (overlapping) 5-letter words, each "letter" taking values
# A,B,C,D,E. The letters are determined  by the number of 1's,
# in that byte::  0,1,or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D,
# and  6,7 or 8 ---> E.  Thus we have a monkey at a typewriter
# hitting five keys with with various probabilities::  37,56,70,
# 56,37 over 256. There are 5^5 possible 5-letter words, and
# from a string of 256,000 (overlapping) 5-letter words, counts
# are made on the frequencies for each word. The quadratic form
# in the weak inverse of the covariance matrix of the cell
# counts provides a chisquare test::  Q5-Q4, the difference of
# the naive Pearson  sums of (OBS-EXP)^2/EXP on counts for 5-
# and 4-letter cell counts.
#
# Note: We actually cycle samples over all 0-31 bit offsets, so
# that if there is a problem with any particular offset it has
# a chance of being observed.  One can imagine problems with odd
# offsets but not even, for example, or only with the offset 7.
# tsamples and psamples can be freely varied, but you'll likely
# need tsamples >> 100,000 to have enough to get a reliable kstest
# result.
#=================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
```




```
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Count the 1s (byte)
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#             Diehard Parking Lot Test (modified).
# This tests the distribution of attempts to randomly park a
# square car of length 1 on a 100x100 parking lot without
# crashing.  We plot n (number of attempts) versus k (number of
# attempts that didn't "crash" because the car squares
# overlapped and compare to the expected result from a perfectly
# random set of parking coordinates.  This is, alas, not really
# known on theoretical grounds so instead we compare to n=12,000
# where k should average 3523 with sigma 21.9 and is very close
# to normally distributed.  Thus (k-3523)/21.9 is a standard
# normal variable, which converted to a uniform p-value, provides
# input to a KS test with a default 100 samples.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 0 (test default is 0)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
```

```
# Counting histogram bins, binscale = 0.100000
#     60|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     54|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     48|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     42|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |****|    |    |    |
#     36|    |    |    |    |    |    |****|    |    |    |
#       |    |    |    |    |    |    |****|    |    |    |
#     30|    |    |    |    |    |    |****|****|    |    |
#       |    |    |    |    |    |    |****|****|    |    |
#     24|    |    |    |    |    |    |****|****|    |    |
#       |    |    |    |    |    |    |****|****|    |    |
#     18|    |    |    |    |    |    |****|****|    |    |
#       |    |    |    |    |    |    |****|****|    |    |
#     12|    |    |    |    |    |****|****|****|    |    |
#       |    |    |    |****|    |****|****|****|    |    |
#      6|    |    |****|****|    |****|****|****|    |    |
#       |    |    |****|****|    |****|****|****|    |    |
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Parking Lot Test
# Assessment:
# FAILED at < 0.01%.
#=================================================================
#         Diehard Minimum Distance (2d Circle) Test
# It does this 100 times::   choose n=8000 random points in a
# square of side 10000.  Find d, the minimum distance between
# the (n^2-n)/2 pairs of points.  If the points are truly inde-
# pendent uniform, then d^2, the square of the minimum distance
# should be (very close to) exponentially distributed with mean
# .995 .  Thus 1-exp(-d^2/.995) should be uniform on [0,1) and
# a KSTEST on the resulting 100 values serves as a test of uni-
# formity for random points in the square. Test numbers=0 mod 5
# are printed but the KSTEST is based on the full set of 100
# random choices of 8000 points in the 10000x10000 square.
#
# This test uses a fixed number of samples -- tsamples is ignored.
# It also uses the default value of 100 psamples in the final
# KS test, for once agreeing precisely with Diehard.
#=================================================================
```

```
#                          Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 8000 (test default is 8000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                 Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     84|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     72|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     60|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     48|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     36|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     24|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     12|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard Minimum Distance (2d Circle) Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#          Diehard 3d Sphere (Minimum Distance) Test
# Choose  4000 random points in a cube of edge 1000.  At each
# point, center a sphere large enough to reach the next closest
# point. Then the volume of the smallest such sphere is (very
# close to) exponentially distributed with mean 120pi/3.  Thus
# the radius cubed is exponential with mean 30. (The mean is
# obtained by extensive simulation).  The 3DSPHERES test gener-
# ates 4000 such spheres 20 times.  Each min radius cubed leads
# to a uniform variable by means of 1-exp(-r^3/30.), then a
```

```
#  KSTEST is done on the 20 p-values.
#
# This test ignores tsamples, and runs the usual default 100
# psamples to use in the final KS test.
#=================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 4000 (test default is 4000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    120|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#    108|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     96|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     84|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     72|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     60|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     48|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     36|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     24|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     12|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for Diehard 3d Sphere (Minimum Distance) Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                   Diehard Squeeze Test.
#  Random integers are floated to get uniforms on [0,1). Start-
#  ing with k=2^31=2147483647, the test finds j, the number of
#  iterations necessary to reduce k to 1, using the reduction
```

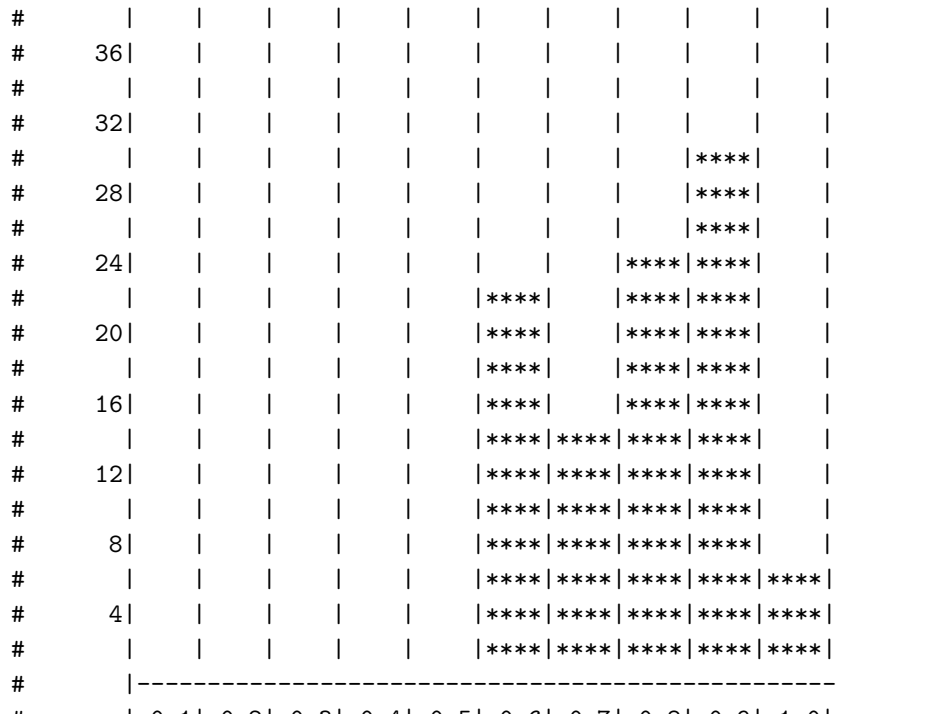
```
#  k=ceiling(k*U), with U provided by floating integers from
#  the file being tested.  Such j's are found 100,000 times,
#  then counts for the number of times j was <=6,7,...,47,>=48
#  are used to provide a chi-square test for cell frequencies.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     28|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |****|
#     24|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     20|    |    |****|    |    |    |    |    |    |****|
#       |    |    |****|    |    |    |    |    |    |****|
#     16|    |    |****|    |    |    |****|    |    |****|
#       |    |    |****|    |    |    |****|****|    |****|
#     12|    |    |****|    |****|    |****|****|    |****|
#       |    |    |****|    |****|    |****|****|    |****|
#      8|    |****|****|    |****|    |****|****|    |****|
#       |    |****|****|    |****|    |****|****|    |****|
#      4|    |****|****|    |****|    |****|****|    |****|
#       |    |****|****|    |****|    |****|****|    |****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Squeeze Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                  Diehard Sums Test
# Integers are floated to get a sequence U(1),U(2),... of uni-
# form [0,1) variables.  Then overlapping sums,
#   S(1)=U(1)+...+U(100), S2=U(2)+...+U(101),... are formed.
```

```
# The S's are virtually normal with a certain covariance mat-
# rix.  A linear transformation of the S's converts them to a
# sequence of independent standard normals, which are converted
# to uniform variables for a KSTEST. The  p-values from ten
# KSTESTs are given still another KSTEST.
#
# Note well:  -O causes the old diehard version to be run (more or
# less).  Omitting it causes non-overlapping sums to be used and
# directly tests the overall balance of uniform rands.
#=================================================================
#                         Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100 (test default is 100)
# P-values in final KS test = 100 (test default is 100)
# Number of rands required is around 2^21 per psample.
# Using non-overlapping samples (default).
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     80|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |****|
#     72|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     64|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     56|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     48|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     40|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     32|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     24|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     16|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |****|****|
#      8|    |    |    |    |    |    |    |****|****|****|
#       |    |    |    |    |    |    |    |****|****|****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Diehard Sums Test
# Assessment:
```

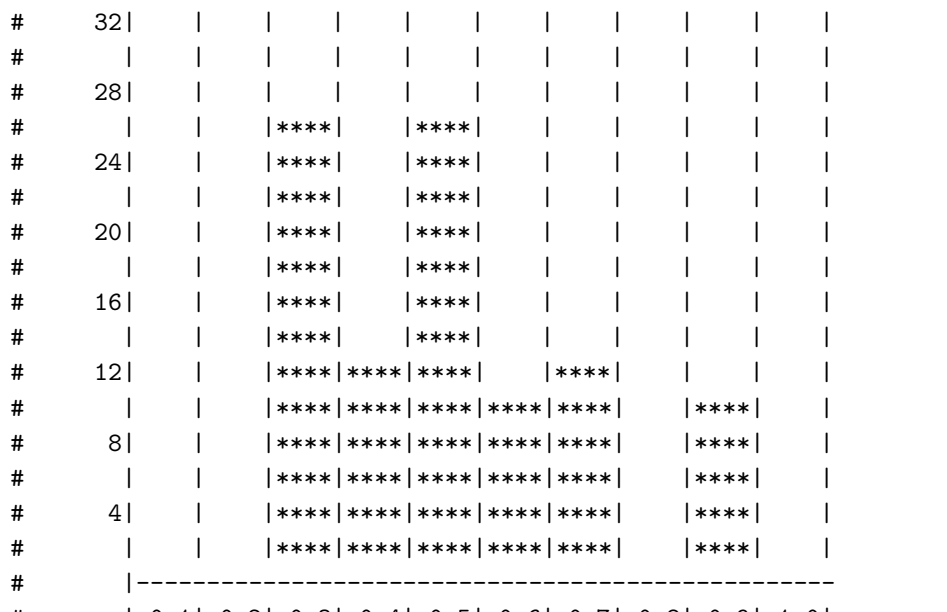
```
# FAILED at < 0.01%.
#=================================================================
#                    Diehard Runs Test
#  This is the RUNS test.  It counts runs up, and runs down,
# in a sequence of uniform [0,1) variables, obtained by float-
# ing the 32-bit integers in the specified file. This example
# shows how runs are counted:  .123,.357,.789,.425,.224,.416,.95
# contains an up-run of length 3, a down-run of length 2 and an
# up-run of (at least) 2, depending on the next values.  The
# covariance matrices for the runs-up and runs-down are well
# known, leading to chisquare tests for quadratic forms in the
# weak inverses of the covariance matrices.  Runs are counted
# for sequences of length 10,000.  This is done ten times. Then
# repeated.
#
# In Dieharder sequences of length tsamples = 100000 are used by
# default, and 100 p-values thus generated are used in a final
# KS test.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |    |
#     28|    |    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |    |
#     24|    |    |    |    |****|    |    |    |    |    |    |
#       |    |    |    |    |****|    |    |    |    |    |    |
#     20|    |    |    |****|****|    |    |****|    |    |    |
#       |    |    |    |****|****|    |    |****|    |    |    |
#     16|    |    |    |****|****|    |****|****|    |    |    |
#       |    |    |    |****|****|    |****|****|    |    |    |
#     12|    |    |    |****|****|    |****|****|    |    |    |
#       |    |    |    |****|****|    |****|****|    |    |    |
#      8|    |    |****|****|****|    |****|****|****|    |    |
#       |    |    |****|****|****|    |****|****|****|    |    |
#      4|    |    |****|****|****|    |****|****|****|    |    |
```

```
#       |    |    |****|****|****|    |****|****|****|    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Runs (up)
# Assessment:
# FAILED at < 0.01%.
#================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     28|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     24|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |    |****|    |    |    |    |
#     20|    |    |    |    |    |****|    |    |    |    |
#       |    |    |    |    |****|****|    |    |    |    |
#     16|    |    |    |    |****|****|    |    |****|    |
#       |    |    |    |    |****|****|    |    |****|    |
#     12|    |    |    |    |****|****|    |****|****|    |
#       |    |    |    |****|****|****|    |****|****|    |
#      8|****|    |    |****|****|****|    |****|****|    |
#       |****|    |    |****|****|****|    |****|****|    |
#      4|****|    |    |****|****|****|    |****|****|    |
#       |****|    |    |****|****|****|    |****|****|    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Runs (down)
# Assessment:
# FAILED at < 0.01%.

#================================================================
#                   Diehard Craps Test
#  This is the CRAPS TEST. It plays 200,000 games of craps, finds
#  the number of wins and the number of throws necessary to end
#  each game.  The number of wins should be (very close to) a
#  normal with mean 200000p and variance 200000p(1-p), with
```

```
#  p=244/495.  Throws necessary to complete the game can vary
#  from 1 to infinity, but counts for all>21 are lumped with 21.
#  A chi-square test is made on the no.-of-throws cell counts.
#  Each 32-bit integer from the test file provides the value for
#  the throw of a die, by floating to [0,1), multiplying by 6
#  and taking 1 plus the integer part of the result.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 200000 (test default is 200000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |****|    |    |    |    |    |    |
#       |    |    |    |****|    |    |    |****|    |    |
#     32|    |    |    |****|    |    |    |****|    |    |
#       |    |    |    |****|    |    |    |****|    |    |
#     28|    |    |    |****|    |    |    |****|    |    |
#       |    |    |    |****|    |    |    |****|    |    |
#     24|    |    |    |****|    |    |    |****|    |    |
#       |    |    |    |****|    |    |    |****|    |    |
#     20|    |    |    |****|    |    |    |****|    |    |
#       |    |    |    |****|    |    |    |****|    |    |
#     16|    |    |    |****|    |    |    |****|    |    |
#       |    |    |    |****|    |    |    |****|    |****|
#     12|    |    |    |****|    |    |    |****|    |****|
#       |    |    |    |****|    |    |    |****|    |****|
#      8|    |    |    |****|    |    |****|****|    |****|
#       |    |    |    |****|    |    |****|****|****|****|
#      4|    |    |    |****|    |    |****|****|****|****|
#       |    |    |    |****|    |    |****|****|****|****|
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Craps Test (mean)
# Assessment:
# FAILED at < 0.01%.
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
```




```
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for Craps Test (freq)
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                     STS Monobit Test
# Very simple.  Counts the 1 bits in a long string of random uints.
# Compares to expected number, generates a p-value directly from
# erfc().  Very effective at revealing overtly weak generators;
# Not so good at determining where stronger ones eventually fail.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#                Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```




```
#=================================================================
#                          Results
# Kuiper KS: p = 0.000000 for STS Monobit Test
# Assessment:
# FAILED at < 0.01%.

#=================================================================
#                       STS Runs Test
# Counts the total number of 0 runs + total number of 1 runs across
# a sample of bits.  Note that a 0 run must begin with 10 and end
# with 01.  Note that a 1 run must begin with 01 and end with a 10.
# This test, run on a bitstring with cyclic boundary conditions, is
# absolutely equivalent to just counting the 01 + 10 bit pairs.
# It is therefore totally redundant with but not as good as the
# rgb_bitdist() test for 2-tuples, which looks beyond the means to the
# moments, testing an entire histogram  of 00, 01, 10, and 11 counts
# to see if it is binomially distributed with p = 0.25.
#=================================================================
#                        Run Details
# Random number generator tested: slatec
# Samples per test pvalue = 100000 (test default is 100000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     40|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
```

```
#     36|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     32|    |    |    |    |    |    |    |    |****|    |
#       |    |    |    |    |    |    |    |    |****|    |
#     28|    |    |    |    |    |    |    |    |****|    |
#       |    |    |    |****|    |    |    |    |****|    |
#     24|    |    |    |****|    |    |    |****|****|    |
#       |    |    |    |****|    |    |    |****|****|    |
#     20|    |    |    |****|    |    |    |****|****|    |
#       |    |    |    |****|    |    |    |****|****|    |
#     16|    |    |    |****|****|    |    |****|****|    |
#       |    |    |    |****|****|    |    |****|****|    |
#     12|    |    |    |****|****|    |    |****|****|    |
#       |    |    |    |****|****|    |    |****|****|    |
#      8|    |    |    |****|****|    |    |****|****|    |
#       |    |    |    |****|****|    |    |****|****|    |
#      4|    |    |    |****|****|    |    |****|****|    |
#       |    |    |    |****|****|    |    |****|****|    |
#       |----------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.000000 for STS Runs Test
# Assessment:
# FAILED at < 0.01%.
```

From the above we see that where randu was merely bad, slatec is downright *ugly*. It is about the same speed as mt19937_1999. It has only *22 bits* in the numbers it returns (a span of only about four million numbers!) It again fails to be even 1-bit random according to the bit distribution test. It then proceeds to fail *all of the tests for randomness!* Even tests that one might expect to be relatively insensitive to its small number of bits (such as the parking lot and runs test) are failed badly.

Note that in a *number* of cases the failure is one that *requires* the final KS test of the returned test p-values. Those p-values themselves are not necessarily poor – in the case of the Diehard sums test, for example, they are all very close to 1.0! They are just completely incorrectly distributed.

The slatec RNG as implemented in the GSL holds a special place in my heart, as it is the perfect generator to use to demonstrate *failure* of the null hypothesis in a random number test. This is actually rather rare – all but a very few of the RNGs incapsulated in the GSL will pass at least a few Diehard(er) tests with the defaults. slatec is a generator that I wouldn't hesitate to list *unsuitable for any purpose* – except, of course, demonstrating the unambiguous failure of an RNG test in Dieharder.

Chapter 8

Conclusions

The *Dieharder* results presented above show the practical utility of adding controls to and generalizing RNG testing suites so that their ability to discriminate failure of a RNG (rejection of the null hypothesis) can be tuned to the needs of the user. *Dieharder* has also showed the danger of treating the passing of any such suite of tests for fixed values and *lacking* such controls as the defining property of a "good" RNG. RNGs will all fail at least some tests for randomness at some point because the sequences they produce are not, in fact, truly random. However, a good RNG may have to be pushed *very hard* to demonstrate a failure of randomness, and even when pushed may only fail *some* tests. Indeed, at some point the validity of the tests themselves may fail because of numerical problems other than the quality of the RNG.

Even in its infancy, *Dieharder* has proven to be a useful tool for studying RNGs, and the encapsulation of RNGs that one might wish to study in the tightly-integrated GSL promises to facilitate many projects that study RNGs or wish to test library-based RNGs for suitability in some numerical application. Because it is a fully GPL tool, both the tool itself and all modifications of the tool that might be distributed must be provided *with immediate access to the source* so that one will never find oneself in the position of using a binary program as a "black box" and therefore uncertain as to whether some particular failure observed is due to a failure of the RNG or rather due to a bug in the program. Access to the code means that one can add input or output statements to any routine as required to *validate* the operation of any test. This can be very important; one might wish to be *certain* that a generator that is "supposed" to be random (perhaps one built on the basis of a quantum process believed to be random on theoretical grounds) but that fails some *Dieharder* test indeed *does* fail that test.

Dieharder is indeed, though, still regrettably incomplete. Although all of the Diehard tests are encapsulated, many STS tests and many tests suggested by Knuth are not yet encapsulated. There is also no practical limit on the number of ways one *can* test RNGs, and the availability of a convenient and consistent interface for encapsulating new tests should, it is hoped, encourange the development of altogether new ones.

In addition, *Dieharder* will eventually be given a graphical user interface (likely inherited from *R*) and the ability to execute tests on a cluster. These two additions will both make the tool easier to play with and use and much faster, so that more complex tests can be performed on longer sequences of numbers. A graphical interface has additional advantages as well – many random number generators fail because they decompose into hyperplanes in a high enough dimensionality. Although this can be tested for numerically, it is certainly desireable to be able to visualize it as well, and visualization may well reveal *new* patterns of RNG failure that are *not* detected by any known tests.

Numerically generated random numbers play an increasingly important role in many statistical applications from business and gaming through physics and mathematics. Sophisticated tests are thereby required to validate RNGs for suitability in many different roles. *Dieharder* is a good platform upon which to develop those tests, in addition to being a pretty good set of tests already.

Appendix A

License Terms

A.1 General Terms

License is granted to copy or use " for "Dieharder, a Gnu Public License Random Number Generator Tester" according to the Open Public License (OPL, enclosed below), which is a Public License, developed by the GNU Foundation, which applies to "open source" generic documents.

There are two modifications to the general OPL given below:

1. Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder. (This is to prevent errors from being introduced which would reflect badly on the author's professional abilities.)

2. Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder. (This is so that the author can make at least some money if this work is republished as a book and sold commercially for – somebody's – profit.)

Electronic versions of the *Dieharder* manual can be freely redistributed, and of course any user of *Dieharder* is welcome to print out such a version for their own use if they so desire. However, if such a user wishes to contribute in a small way to the development of the tool, they should consider buying a paper or electronic copy of the book if and when they are distributed by the author in forms that can be sold.

A.2 OPEN PUBLICATION LICENSE Draft v0.4, 8 June 1999

I. REQUIREMENTS ON BOTH UNMODIFIED AND MODIFIED VERSIONS

The Open Publication works may be reproduced and distributed in whole or in part, in any medium physical or electronic, provided that the terms of this license are adhered to, and that this license or an incorporation of it by reference (with any options elected by the author(s) and/or publisher) is displayed in the reproduction.

Proper form for an incorporation by reference is as follows:

Copyright (c) <year> by <author's name or designee>. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, vX.Y or later (the latest version is presently available at http://www.opencontent.org/openpub/).

The reference must be immediately followed with any options elected by the author(s) and/or publisher of the document (see section VI).

Commercial redistribution of Open Publication-licensed material is permitted.

Any publication in standard (paper) book form shall require the citation of the original publisher and author. The publisher and author's names shall appear on all outer surfaces of the book. On all outer surfaces of the book the original publisher's name shall be as large as the title of the work and cited as possessive with respect to the title.

II. COPYRIGHT

The copyright to each Open Publication is owned by its author(s) or designee.

III. SCOPE OF LICENSE

The following license terms apply to all Open Publication works, unless otherwise explicitly stated in the document.

Mere aggregation of Open Publication works or a portion of an Open Publication work with other works or programs on the same media shall not cause this license to apply to those other works. The aggregate work shall contain a notice specifying the inclusion of the Open Publication material and appropriate copyright notice.

SEVERABILITY. If any part of this license is found to be unenforceable in any jurisdiction, the remaining portions of the license remain in force.

NO WARRANTY. Open Publication works are licensed and provided "as is" without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement.

IV. REQUIREMENTS ON MODIFIED WORKS

All modified versions of documents covered by this license, including translations, anthologies, compilations and partial documents, must meet the following requirements:

1. The modified version must be labeled as such.
2. The person making the modifications must be identified and the modifications dated.
3. Acknowledgement of the original author and publisher if applicable must be retained

according to normal academic citation practices.

4. The location of the original unmodified document must be identified.

5. The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission.

V. GOOD-PRACTICE RECOMMENDATIONS

In addition to the requirements of this license, it is requested from and strongly recommended of redistributors that:

1. If you are distributing Open Publication works on hardcopy or CD-ROM, you provide email notification to the authors of your intent to redistribute at least thirty days before your manuscript or media freeze, to give the authors time to provide updated documents. This notification should describe modifications, if any, made to the document.

2. All substantive modifications (including deletions) be either clearly marked up in the document or else described in an attachment to the document.

Finally, while it is not mandatory under this license, it is considered good form to offer a free copy of any hardcopy and CD-ROM expression of an Open Publication-licensed work to its author(s).

VI. LICENSE OPTIONS

The author(s) and/or publisher of an Open Publication-licensed document may elect certain options by appending language to the reference to or copy of the license. These options are considered part of the license instance and must be included with the license (or its incorporation by reference) in derived works.

A. To prohibit distribution of substantively modified versions without the explicit permission of the author(s). "Substantive modification" is defined as a change to the semantic content of the document, and excludes mere changes in format or typographical corrections.

To accomplish this, add the phrase 'Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.' to the license reference or copy.

B. To prohibit any publication of this work or derivative works in whole or in part in standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

To accomplish this, add the phrase 'Distribution of the work or derivative of the work in any standard (paper) book form is prohibited unless prior permission is obtained from the copyright holder.' to the license reference or copy.

OPEN PUBLICATION POLICY APPENDIX:

(This is not considered part of the license.)

Open Publication works are available in source format via the Open Publication home page at http://works.opencontent.org/.

Open Publication authors who want to include their own license on Open Publication works may do so, as long as their terms are not more restrictive than the Open Publication license.

If you have questions about the Open Publication License, please contact TBD, and/or the Open Publication Authors' List at opal@opencontent.org, via email.