8-2001

Empirical Spectral Analysis of Random Number Generators

David Zeitler
Western Michigan University

EMPIRICAL SPECTRAL ANALYSIS OF RANDOM NUMBER GENERATORS

by

David Zeitler

A Dissertation
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Doctor of Philosophy
Department of Statistics

Western Michigan University
Kalamazoo, Michigan
August 2001

EMPIRICAL SPECTRAL ANALYSIS OF RANDOM NUMBER GENERATORS

David Zeitler, Ph.D.

Western Michigan University, 2001

Computer simulation procedures have become a staple of research and development in many fields, including statistics. The generation of pseudo random number sequences is the core of computer simulation procedures. Validity of research results often depend on the underlying validity of the generator being used.

In this work we develop the machinery for a class of tests of spatial uniformity based on a multi-dimensional Fourier transform of the empirical probability density function. The test can be adapted to specific requirements and has the added advantage that it has computational complexity that is relatively independent of the number of data points being analyzed.

UMI Number: 3019532

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Joseph W. McKean and Dr. John A. Kapenga for their exceptional patience, guidance and support throughout this effort. Thanks also to my committee members Dr. Daniel P. Mihalko, Dr. Joshua D. Naranjo and Dr. Robert J. Buck.

I would also like to thank my wife Meredith for her continued patience with me while working on this dissertation, and also my children Nathaniel, Cassandra and Jessica to whom Ph.D has come to mean 'Permanently hiding Daddy' during this past six months.

Finally I'd like to acknowledge Smiths Industries for their support while I worked for them early in this effort and Siemens Dematic for whom I worked during the completion of this work.

David Zeitler

Contents

List of Tables

List of Figures

CHAPTER I

INTRODUCTION

This dissertation presents the theory and application of a powerful new family of tests, the Empirical Spectral Test (EST), for the "randomness" of a random number sequence. The EST detects many types of multidimensional structure and has a wide choice of options within the family.

Pseudo random number sequences and random number sequences have many important applications. They are critical in protocols and systems in numerous areas, including cryptography[**13**], networking, and communications. Generating "good" pseudo random number sequences is also at the core of computer simulation procedures, which have become a fundamental tool in many fields, including Statistics [**47, 48, 40**], Computer Science, Physics [**26**], Operations Research, Optimization [**8**], Computational Chemistry, and Environmental Science among others. We will only briefly refer to some of these applications in order to show how the EST can be applied.

What follows is a summary of the main points in the chapters of this work.

Chapter 2, Background, will provide background information on random and pseudo random sequences, and a few of their applications, as well as references to some of the tests that have been developed for these sequences. This presentation will not be complete, but will provide what we need to motivate our development of the EST.

Chapter 3, The Empirical Spectral Test (EST), will present the EST algorithm, which is based on a multidimensional FFT. The complexity of the EST algorithm is discussed and a comparison with two well known tests, the Spectral Test of Coveyou and MacPherson[**19**] and Discrepancy test of Niederreiter[**78**] is given. Computer code for the EST is provided in Appendix B.

Chapter 4, The EST Distribution, will present the asymptotics for the distribution of the EST, which is the basis for using the EST as a test statistic. This is the major result of this dissertation.

Chapter 5, Applying the EST, will discus the implications of the three parameters in the EST family of tests. Those parameters are: the number of dimensions (k), cells per dimension (b), and sample size (N). It will also address repeating the test with multiple seeds, using the test on special subsequences of the random number stream and what it means when a random number generator fails an EST test.

Chapter 6, A Comparison of Some RNGs, will present the results of a number of EST runs on several well known pseudo random number generators (PRNGs). A larger listing of EST runs is included in Appendix A. We see here that the EST is a useful test. It is very easy to make up a new test for a random number stream, but very hard to make up a new test that is useful and practical.

Chapter 7, Summary and Future Work, will end this text with a few remarks on what we have achieved in this work and what should be done next.

CHAPTER II

BACKGROUND

This chapter presents background information on random and pseudo random sequences: a few of their applications, the most common generators for them, desirable properties they should have, and some of the tests that have been developed for these sequences. This presentation is not intended to be exhaustive, but will provide sufficient information to motivate our development of the EST and references for the reader to explore further.

Of special interest will be a well known example of a pseudo random number generator (PRNG), RANDU. RANDU was a very widely used PRNG that was eventually found to have a startling non-random structure. This discovery lead to a major theoretical breakthrough in the understanding of the entire class of linear congruential generators (LCGs). In chapter 6 we will show that the EST detects problems with RANDU. This indicates that EST can be a useful test, since RANDU easily passes many tests used on PRNGs.

Two of the tests for PRNGs presented here are also of direct interest, the Spectral Test of Coveyou and MacPherson[**19**] and Discrepancy Tests of Niederreiter[**78**]. These tests are considered some of the most powerful known, but they only apply to specific (but important) classes of PRNGs. The EST can be expected to test for the same structures, and more, that

these tests detect and the EST can be applied to any PRNG. Because of this, the EST should be widely accepted as a new tool in RNG testing.

2.1. Uses of Random Numbers

In this section some of the common uses for random number streams will be presented. First we need to define what a random number sequence is. This is actually a very hard and subtle thing to do, and we will have more to say about this in section 2.5.

DEFINITION 2.1.1. Let U be a uniform [0,1) random variable (this is denoted as U[0,1)). A sequence $u_1, u_2, u_3, \ldots$ of independent identically distributed (iid) samples from U[0,1) is called a U[0,1) random sequence.

Note that after the sampling is done there is nothing random about the sequence, each u_i is a fixed known real number. Hence a random number sequence is not itself random, but simply a sequence of samples from a random process. This difference is important.

A pseudo random number generator (PRNG) is a deterministic algorithm, which is used to generate a pseudo random sequence. When we use PRNGs, we are *not* sampling from a random process, but from a process which has been carefully designed to provide sequences of numbers with the important properties of random sequences.

We have just touched upon the surface of what a random sequence is. For more discussion see Knuth[**49**] section 3.5

2.1.1. Simulation Models. Computer simulation has come to be used in almost every field. In many cases physical phenomena are not directly

observable, such as in astrophysics or in the realms of particle physics where humans will never be able to go to or see into physical processes. For these cases, simulation is used to explore and validate phenomena that are unobservable. Even when the system being studied is observable, it is often less expensive to use simulation than to build instruments capable of observing with sufficient detail. In either case, many external and internal factors cannot, either because of lack of knowledge or excessive computational cost, be readily modeled. A standard approach is to replace the detailed model with a statistical replacement which will provide inputs or behavior equivalent to apparent randomness of the real world.

In these modeling situations, uniformity of the underlying generator must be assured to prevent inadequate coverage of the input or behavior space being modeled. Independence in multiple dimensions must be assured to prevent hidden correlations in the generator from causing unrealistic correlations in model behavior. These difficulties become more troublesome as modeling extends into parallel computational environments[**35**]. General references here are numerous, but a good place to start would be an introductory Operations Research text such as Hillier and Lieberman[**38**].

2.1.2. Statistical Techniques.

2.1.2.1. *Distribution Samples.* The core of both statistical sampling and simulations of stochastic processes is the generation of random sequences from various target distributions such as the Normal, Gamma, Beta, Binomial, Poisson and others. These techniques are discussed briefly in section 2.2.1. A good source for methods of generating different sample distributions from U[0,1) sources can be found in Kennedy and Gentle[**47**].

If the underlying U[0,1) generator fails in either uniformity or independence, sampling plans can be skewed, producing biased results. When simulating stochastic processes or data sets to test estimation and testing schemes, failure of either uniformity or independence will also cause biased results.

2.1.2.2. *Resampling.* Resampling is a statistical technique for estimation of bias, variance and other measures of error which can be used when theoretical derivation of these quantities are impractical. The two major techniques in this class are the Jackknife and the Bootstrap.

These techniques are based on the simple idea of starting from a single random sample (or random sequence) and generating new sequences from this sample via random sampling. The desired estimate is then formed from the analysis of these new random sequences.

Good starting points for the interested reader would be Efron[**23**] or the collection edited by LePage and Billard[**60**] and Kleijnen et. al.[**48**].

2.1.3. Monte Carlo Integration. Monte Carlo integration treats numerical integration as a statistical sampling problem, allowing a statistical estimate of the value of the integral rather than a numerical approximation. The technique is most often used in high dimensional integrals where standard techniques are too slow, or integrals over regions with singularities which cause numerical problems. In this class of problems, "spatial uniformity" of n-tuples is more important than some other measures or randomness, such as independence of adjacent n-tuples. Also here the common

lattice methods which are "too uniform" to be random are actually an advantage in integration, resulting in a lower expected integration error. A good starting point for further material would be Niederreiter [**78**].

2.1.4. Optimization. Numerical optimization is a class of techniques for finding the minimum of a multidimensional surface. Many of these techniques use a variation of gradient descent and are not themselves randomized (even if the underlying surface may be stochastic). There are several areas of optimization which use random numbers.

One area is simulated annealing, where random sequences are used to help the algorithm to pull out of local optima. This is analogous to the annealing process used to produce maleable metals by heating and then slowly cooling. A discussion of simulated annealing can be found in Numerical Recipes[**1, 50**] among others.

A second area is genetic algorithms, which is usually applied in situations where the underlying search is over a discrete space. Randomized recombinations of an underlying key or code are performed, searching for a maximum of some measure of the quality of the system specified by the code. This is analogous to how evolution is believed to progress through the random recombination of DNA strands. A recent discussion of genetic algorithms can be found in [**8, 31, 5, 6**].

In both cases above, correlated random numbers or insufficient variation in the randomization can cause a degradation or complete failure of the algorithm to achieve it's goals.

2.1.5. Cryptography and Security. Cryptography and computer security make heavy use of random number generators including applications

in stream ciphers (commonly using LFGs for speed), digital signatures and key exchange. These applications are primarily concerned with the the ability (or preferably the lack thereof) to 'crack' the code being used. One measure of this is the Kolmogorov complexity which is the length of the Turing machine required to generate the sequence or code.

A good source of additional information for cryptography would be Schneier[**13**]. For more information on Kolmogorov complexity, see Li and Vitanyi[**63**]. A discussion of a PRNG specifically designed with cryptographic use in mind [Yarrow-160] can be found in [**46**].

2.2. RNG, PRNG, TRNG (and What is a "True" Random Number)

The definition given for a random number sequence, in section 2.1, begs the question of how such a sequence is captured. There are two approaches to this problem. The first approach is to generate a random sequence with a physical device which we will call a random number generator (RNG).

The second approach is to use a deterministic algorithm, called a pseudo random number generator (PRNG), to generate what is called a pseudo random sequence in [0,1). The hope is that the pseudo random sequence provides the same usefulness as a random sequence would.

A random number generator (RNG) is a means for generating a random number sequence. Note that no computer program or other method that produces a predictable or reproducible sequence of values can be used as an RNG. Or as John von Neumann said,

> "Anyone who considers using arithmetical methods of producing random digits is, of course, in a state of sin."

Only a suitable physical device capable of "true" (sic) random behavior can be used as an RNG. Some authors refer to such devices as "true" (sic) random number generators (TRNGs). We will not use the redundant term "true" with random number sequences and generators. This will be expanded on in the next section.

A pseudo random number generator (PRNG) is a deterministic algorithm, which is used to generate a pseudo random sequence. Again, as mentioned above, a PRNG can not generate a random sequence. The advantages of a PRNG over an RNG are many (speed, reproducibility, cost etc...) and these will be discussed in the section on PRNGs that follows. The question is "Does the fact that the pseudo random sequence is not random adversely effect its use in a specific application?"

2.2.1. Other Distributions. Although this work concentrates on U[0,1) random and pseudo random sequences, this is not really a restriction because all the commonly required random sequences with distribution other then U[0,1) can be effectively generated from U[0,1) sequences. Such distributions include Z[0,1], Z[0,n}, N($\mu\,\sigma^2$), P(λ), etc. Many of the available library PRNGs for these non-uniform distributions take this approach. Next two simple, but very important, examples of this follow.

As a first example, given a u[0,1) random sequence, $u_1, u_2, \ldots$. One method of producing a uniform z[0,m) random sequence is to take, ,$b_1, b_2, \ldots$, where $b_i = \lfloor m u_i \rfloor$. Here $\lfloor x \rfloor$ is the floor function, returning the largest integer not greater than its argument.

As another example, if F^{-1} is the inverse of the probability distribution function F, then the sequence $x_1, x_2, \ldots$, where, $x_i = F^{-1}(u_i)$, is a random sequence with distribution F.

When F^{-1} can not be explicitly expressed, the inverse can sometimes be computed sufficiently well numerically for each u_i. This even applies when F is not monotonic, with a proper interpretation for F^{-1}.

Many more involved procedures exist for the efficient generation of specific non-uniform random sequences from random and pseudo random uniform sequences. It is not our purpose to catalog them here, for further study the reader is directed to Kennedy and Gentle[**47**] and Knuth[**49**].

2.3. RNGs From Physical Devices

As already noted, no algorithm can be used as an RNG. There are however several common physical devices that are used as sources for randomness. We will mention a few of them here and then point out that even with a physical device based RNG, there is still a need for testing the RNG.

2.3.1. Distilling Randomness. The physical systems mentioned below all are used as sources of randomness to construct RNGs. These physical systems all may have non-uniform distributions. The usual RNG based on a physical system will collect single random binary bits in a stream $b_1, b_2, \ldots$ that will be hopefully near independent, but may not be nearly uniformly distributed. These non-uniform bits will then be used in an algorithm to produce a bit stream $c_1, c_2, \ldots$ that will have a uniform or near uniform distribution.

It will usually require more than one bit in the non-uniform sequence to produce one bit in the more uniform sequence. Some types of dependence in a bit sequence can be reduced at the cost of extra bits as well. Two simple examples are:

EXAMPLE. Assume (b_i) is binomial (iid), with an unknown bias ε, that is $Pr[b_i = 0] = 0.5 - \varepsilon$, with $-0.5 < \varepsilon < 0.5$. Now let $c_i = b_{2i-1} \wedge b_{2i}$where "$\wedge$" is the XOR boolean operator. It is simple to see that the bias in c_i is now $2\varepsilon^2$. If four terms were XORed together instead of two, the bias would be only $8\varepsilon^4$. If the b_i were iid, and thus had no correlations, then the c_i would also be iid.

Using bits from (b_i) can completely eliminate bias in the (c_i) (when the bias is not 0.5 or -0.5) by applying the following:

```
j=1
for(i=1; ; i++) {
  while (b_{2j-1}== b_{2j}) {
    j = j+1;
  }
  c_i = b_{2j-1};
}
```

Note that the number of bits required to generate each c_i is non-deterministic, but at least 2.

There are many other methods that have been proposed for distilling randomness, including reducing bias and correlation. Some of these methods include the use of: transition mappings, compression and the FFT. You

can also design an RNG that combines the outputs of several separate physical processes. Many more methods have been proposed than is appropriate to detail here. These would be good starting points for more information [**87, 21**].

2.3.2. Some Physical Sources of Randomness. One source of randomness is radioactive decay. According to quantum theory, the time it takes a radioactive isotope to decay is a random event with a negative exponential distribution. So measuring the time between the decay of particles or the number of decays in a fixed time for a radioactive sample is a source of randomness.

Given a sample of a radio isotope with a reasonably long half life, if you set a time of 1 minute and observe about 600 decays in each minute, then setting the binary bit b_i to 0 or 1 based on whether the number of decays is even or odd will be a reasonable, but not quite uniformly distributed binary sequence. If you only observe about 2 decays, the randomness of the sequence will be less satisfactory.

Commercial random number generator chips using noise diodes are available. An Intel Pentium support chip set[**42**] includes noise diodes and machine language instruction support specifically for random bit sequence generation. Other forms of electronic noise generating devices can be used as well. Since these devices use low level electronic phenomena for randomness, they should be very carefully shielded, since they can be sensitive to external EMFs.

Hardware related events in computers, especially with human or other nondeterministic event drivers, have often been used as a source of randomness. Many computer games used to use the low order bits of high resolution timing between keystrokes as a source of randomness. Hardware events such as computer disk drive events [**43**] or system events, as used in the Linux operating systems /dev/random facility, are becoming commonly available. These random facilities are needed to support secure communications.

2.3.3. The Need to Test RNGs. Although we have listed several sources for random bits, and indicated that the randomness of a sequence can be "improved", it should be noted that designing and constructing reliable RNGs is fraught with problems. We have to address the initial performance of the RNG and, since it is based on a physical process that will at some point break down, the RNG must be continuously monitored.

The EST presented here can be used initially to provide many tests for non-random behavior for an RNG. In some cases, when non-random structure is detected, an understanding of the FFT may provide the designer with clues to methods for improving the RNG, with either redesigned hardware or a suitable post processing procedure for the bits collected. It is expected that the EST will be effective in detecting short term correlations and biases that may be present in RNGs.

The EST can also be used to monitor the performance of the RNG over time, looking for changes in the spectral signature of the generator. This can be more sensitive that simply testing the generator for acceptability again

at each time and has the potential of predicting failures rather than simply detecting them after they occur. This will be the subject of future work.

The EST could be applied to the initial random bit sequence in several ways, as well as being applied to the output of an RNG (which may have resulted from distilling procedures applied to the initial bit sequence).

2.4. Pseudo RNGs (PRNG)s

Here we present a general description of a PRNG and then describe the two of the most common PRNGs, the linear congruential generator (LCG) and the lagged Fibonacci generator (LFG). Finally we mention a common method for combining two generators to produce a new generator. For a far more complete presentation on these and many other issues of choosing and testing generators, one can not go wrong in starting with Knuth[**49**].

Most PRNGs now in use have a relatively simple algorithm, with very carefully chosen constants. This is the result a long history of complicated looking algorithms producing very non-random looking pseudo random sequences. A simple algorithm that has been studied and analyzed, so that it's shortcomings are thought to be known and acceptable, should be preferred to one that has not been as well analyzed and may have severe hidden problems. This falls back to the question of "What is a random sequence and how can we test for one?", which we open in Section 2.5.

Serious non-randomness in a PRNG may be well hidden, until a sequence is used in a specific application, that exposes it. In effect each application that uses a PRNG is a test for the PRNG. Some quantum Monte-Carlo

simulation codes have been used as PRNG tests[**91**], for cases when theoretical results for the simulation are known, before trying the simulation on cases where theoretical results are not know.

While good, this approach should be used with care. There is an implicit assumption that a case which a specific PRNG is acceptable can be extrapolated to new cases. As with any extrapolation, this is not always true. Potential serious hidden problems with the randomness of PRNGs is the reason that when doing simulation studies it is very prudent to repeat the study a couple times using different types of generators for both the theoretically known and unknown cases.

2.4.1. Anatomy of a PRNG. Generally, a PRNG consists of two functions that can be seen by the user, get_urand() and init_urand(), and finite spaces SEED and STATE, where:

- get_urand: STATE-> STATE x [0,1)
- set_urand: SEED -> STATE

The user first uses set_urand(), with some choice of seed, to set an internal state S_0. Then each u_i is generated by successive calls to get_urand() as in:

```
/* choose a seed */
set_urand(seed); /* set STATE S_0 */
for(i=1; i<=n; i++) {
  u_i = get_urand();
}
set_urand(seed);    /* set STATE S_0 */
for(i=1; i<=n; i++) {
```

u_i = get_urand();

}

Note that get_urand() has no arguments, it accesses a "hidden" STATE variable implicitly. Internally, get_urand() has two functions:

- state_to_real: STATE -> [0,1)
- next_state: STATE -> STATE

With these, get_urand is simply

```
get_urand() {
  external STATE state;
  real     u;
  u     = state_to_real(state);   /* u_i = state_to_real(S_i) */
  state = next_state(state);   /* S_{i+1} = next_state(S_i) */
  return(u);
}
```

Most PRNGs actually produce an integer from the state first, using boolean and integer operations, and then transform the integer to the real to be returned. This would imply there are two functions,

- state_to_sinteger() and
- sinteger_to_real(),

that decompose the function state_to_real as:

state_to_real() = sinteger_to_real(state_to_sinteger())

This will be used in presenting a combination generator shortly. It is also true that using state_to_sinteger() in get_urand(), instead of state_to_real(), would provide a generator that would return an integer directly.

There are some issues with the above description. We have not indicated how many bits there are in SEED, STATE or real. In any actual PRNG, these will be of specific sizes.

Since STATE is finite, there is a limit to the number of u_i that can be generated before S_i will repeat, and of course when the S_i repeats the u_i will repeat. To formalize the repeating structure we use a r as the point at which the repetition occurs in the sequence given a particular initial seed giving us a sequence $u_1, u_2, \ldots, u_r, u_{r+1}, \ldots, u_{r+s}, u_r, u_{r+1}, \ldots$ from the generator. If the PRNG is parameterized appropriately, each possible S_i will be visited exactly once before there is a repeat. For example, a PRNG with a 32 bit S could step through 2^{32} unique states before it repeats. Thus a PRNG could have $r = 0$ and $s = 2^B$, where B is the number of bits in the PRNG state S, giving us $u_1, u_2, \ldots, u_s, u_1, u_2, ..., u_s, u_1, u_2, \ldots$. Some common generators such as the LCG and LFG (defined later) have $s = 2^B - 1$.

Note that since STATE is mapped to u_i and the state generally has more bits than u_i. It is possible for u_i to repeat, within $u_1, \ldots, u_s$, without S_i repeating. In fact, when S_i has a larger bit size than u_i, there must be repeats in the u_i. This cycling and repeating clearly deviates from a true random sequence.

Since SEED is finite, there are a limited number of separate starting points for the sequence starts. It may be very hard to insure that choosing two seeds and generating two sequences of size n will result in non-overlapping sequences. The use of overlapping sequences in an application could be a real problem. This will be addressed in a later section on subsequences.

All PRNGs will return a real with a finite number of bits of precision. Typically reals are represented in 32 or 64 bit words, with 23 or 54 bits of precision respectively. This means that a PRNG with a 32 bit state generating a 32 bit floating point number will necessarily loose precision. This is another deviation from our definition of a random sequence. See Kennedy and Gentle[**47**] for further discussion of floating point representations.

All PRNGS are designed to have the highest order bits returned appear "random". Some, such as many LCGs, have very little apparent "randomness" in the low order bits. For example, the low order bits of the UNIX drand() generator are 0,1,0,1... .

2.4.2. Linear Congruential Generators (LCGs). The most thoroughly studied class of generators is the linear congruential. It is a simple linear recursion with an applied modulus.

DEFINITION 2.4.1. A linear congruential generator is defined in terms of the next_state and state_to_real equations:

$$s = (as + c) \mod m$$

$$u = float(s)/float(m)$$

where the a and c parameters are carefully chosen to produce good properties in u.

These generators have m states and can have a full period of $m-1$. This period is only achieved when the parameters are carefully chosen. These generators as a class also tend to fail the runs tests as well as tests for randomness of the low order bits. Good sources for further LCGs theory and analysis are [**49, 84, 65, 13**].

2.4.3. Lagged Fibonacci Generators (LFGs). Another common generator, that is thoroughly studied, is the lagged Fibonacci.

DEFINITION 2.4.2. A lagged Fibonacci generator defined in terms of the next_state and state_to_real equations:,

$$X_n = (X_{n-a} + X_{n-b}) \mod m$$

$$u = float(X)/float(m)$$

where $a < b \leq i$ with a and b chosen very carefully. Note the state $S_i = \{X_{i-b}, \ldots, X_i)\}$.

This is just a variation of the Fibonacci sequence. This form has the the advantage of each successive generate depending on two of the past b elements of the sequence. If the X_i are 32 bit integers, the state is $b2^{32}$ bits, allowing very long period with a very simple and efficient computational form. Further discussion is available in Knuth[**49**] and Schneier[**13**].

2.4.4. Combining PRNGs. Given two or more PRNGs there are many methods of combining them to produce a new PRNG [**57, 55, 86, 47, 13**].

The design goals of such a combination generator usually are to overcome shortcomings in the original generators and produce a longer period. We indicate only one simple but important combination method here.

First, consider two RNGs (not PRNGs) that both return integers uniformly distributed in the range [0,m). We can then get a random sequence drawn from each:

$a_1, a_2, \ldots$

$b_1, b_2, \ldots$

Consider the new combination sequence $c_1, c_2, \ldots$ with $c_i = a_i \wedge b_i$ where the operator "$\wedge$" is the XOR (bitwise exclusive or) boolean operator applied bitwise to the binary representation of its arguments.

The sequence c_i is now also a random uniform [0,m) sequence (e.g .it can be formally treated as drawn from a uniform [0,m) distribution). An important observation is that if sequence a_i is random then the result still holds whether the sequence b_i is random or not! In fact, some of the 'random' data provided with Marsaglia's DIEHARD[**66**] are the result of combining both PRNG sequences with non-random sequences like bit mapped images.

Now we will apply this to one method for combining PRNGs. Given two PRNGS:

- get_urand_1() with set_urand_1() and
- get_urand_2() with set_urand_2().

Without loss of generality, assume that $SEED_1 = SEED_2$. Let the periods of the generators be P_1 and P_2. The general form for this combined generator is:

```
set_urand(SEED seed) {
  set_urand_1(seed);    /* set state_1 */
  set_urand_2(seed);    /* set state_2 */
}

get_urand() {
  external STATE_1 state_1;
  external STATE_2 state_2;
  integer i_1, i_2;
  real u;
  i_1 = state_to_sinteger_1(state_1);
  i_2 = state_to_sinteger_2(state_2);
  u = sinteger_to_real( i_1 ^ i_2);
  state_1 - next_state_1(state_1);
  state_2 - next_state_2(state_2);
  return(u);
}
```

We assume both state_to_sinteger_1 and state_to_sinteger_2 produce pseudo random integers in the same range, [0,m). Then sinteger_to_real() might be either sinteger_to_real_1() or sinteger_to_real_2().

If P_1 and P_2 are relatively prime, then the period of the combined generator is $P_1 P_2$. Also, if the two generators are not of the same type, say a LCG and a LFG, then we would "hope" that they might mask each others defects.

Combination generators are the basis for many efficient PRNGs and PRNGs that provide multiple streams, which are discussed in the next section. The use of the XOR combination technique in particular is at the heart of a number of cryptographic applications.

2.4.5. Multiple Streams of PRNGs. Multiple streams (or sequences) of random numbers are usually thought of in relation to parallel computation or computation involving spatial information. One need not be doing either of these to be using multiple streams of random numbers, all it takes is for there to be two or more calculations in your code which use the random number generator and you're effectively using multiple streams. For example a statistician which runs a series of simulation studies, each simulation taking a sequence of random numbers from the generator in succession. This is a block partitioning of the generator output to the successive runs, like is used in chapter 6 for multiple runs of the EST against a number of generators.

Many schemes for generating distributions other than uniform use multiple uniform generates in combination to obtain the desired distribution. This is an implicit leapfrog partitioning of the generator output.

2.4.5.1. *Use of Different PRNGs.* One approach is to use a different generator for each stream of numbers needed. This would clearly not be easily extended to additional processes and the quality of the numbers generated may not be consistent across streams.

TABLE II-1. Leap Frog Partitioning Example: k=2, n=5

Generator	u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9
Subsequences	$v_{0,0}$	$v_{0,1}$	$v_{1,0}$	$v_{1,1}$	$v_{2,0}$	$v_{2,1}$	$v_{3,0}$	$v_{3,1}$	$v_{4,0}$	$v_{4,1}$

2.4.5.2. *Use of a 'Family' of Generators.* Rather than actually using different generators, it's possible to use different parameterizations of a single class of generators or a 'family' of generators. The separate parameterizations must be chosen with care to be certain that each stream of random numbers is of sufficient quality, but this is easier than getting consistency between different generators. An example of this is the ACORN generators which are discussed in [**103, 102**].

2.4.5.3. *Trees.* Fredrickson et. al. [**27**] presents a dynamic allocation scheme for using a pair (left and right) of congruential generators to produce arbitrary trees of random numbers. Branching occurs at any point in a right sequence by using the left generator to transition from the current state rather than the right. If the two generators are chosen carefully, this process produces disjoint right sequences within the state space defined by the bit size of the generators.

2.4.5.4. *Leap Frog.* A leap frog stream partitioning is simply taking successive vectors from the original stream. To allocate generates from the generator to k different streams, every k-th generate is allocated to a particular stream as is illustrated in Table II-1 for k=2.

DEFINITION 2.4.3. A k stream leap frog partitioning of a stream $\{u_i\}$ into j streams $\{v_{i,j}\}$ is defined as $v_{i,j} = u_{k*i+j}$ where $i = 0, \ldots, n, \ldots$ and $j = 0, \ldots, k$-1.

TABLE II-2. Block Partitioning Example: k=2, n=5

Generator	u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9
Subsequences	$v_{0,0}$	$v_{1,0}$	$v_{2,0}$	$v_{3,0}$	$v_{4,0}$	$v_{1,0}$	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$

An example of a leap frog algorithm is presented by Aluru et.al.[**3**] for the generalized feedback shift register algorithm. This algorithm uses stream dependent initialization with no additional computational cost for generation of each number. Congruential generators can also be easily leap frogged for small numbers of streams.

2.4.5.5. *Blocks.* Block partitioning allocates successive elements in the original sequence to each stream as illustrated in Table II-2 where blocks of size 4 are allocated to two streams (usually blocks will be much larger to avoid streams overlapping).

DEFINITION 2.4.4. A k stream block partitioning of a stream $\{u_i\}$ into k streams is defined as $v_{i,j} = u_{k*j+i}$ where where $i = 0, \ldots, n, \ldots$ and $j = 0, \ldots, k$-1.

The RANLIB package[**12**]**,** which is available from statlib, uses a combination of a pair of congruential generators to provide an adequately long sequence which is then partitioned into 32 blocks of contiguous random numbers which can be allocated to processes as necessary. Setting the generators to the beginning of the blocks is performed using algorithms developed by L'Ecuyer . Table II-2 illustrates a block partitioning for k=2.

2.4.6. Packages for PRNGs. Finding a PRNG is never a problem. Every modern operating system and most software packages which expect to do numerical calculation (except maybe financial packages) will include

at least one and usually a few. In addition, these packages will generally provide appropriate transformations from the base U[0,1) generator to appropriate distributions for the application area of the software.

The GNU Scientific library[**95**] provides an extensive set of PRNGs. SPRNG[**73**] provides PRNGs for parallel computation. Since we will be looking closer at the generators in R in chapter 6, we will give a little more detail here on the facilities it provides.

2.4.6.1. *Brief Descriptions of the Five Generators in R.* R is an open source statistical package patterned after the commercial package S-Plus. It's developed and distributed under the GNU Public License (GPL) and is freely available for most common computing platforms. The base uniform random number generator actually allows the user to choose from 6 generators, the last being a user supplied generator.

Details and references for these generator implementations come from the R reference manual[**40**].

Marsaglia-Multicarry[**69**]. A multiply-with-carry RNG, recommended by George Marsaglia. It has a period of over 2^{60} and it has reportedly passed all tests. This is the default generator for R.

Super-Duper. This is essentially the same as Super above. It is the original version which does not pass the MTUPLE test of the Diehard battery. It has a period of 4.6×10^{18} for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd). R uses the implementation by Reeds et.al. (1982). This generator is reportedly not exactly the same as that in recent versions of S-PLUS.

Wichmann-Hill. The Wichmann-Hill generator has a cycle length of 6.9536e12, see Applied Statistics (1984) 33, 123 which corrects the original article.

Mersenne-Twister. From Matsumoto and Nishimura: This is a twisted Generalized Feedback Shift Register (GFSR) with period $2^{19937} - 1$ and equidistribution in 623 consecutive dimensions (over the whole period).

Knuth-TAOCP[**49**]. This is a Generalized Feedback Shift Register (GFSR) generator using lagged Fibonacci sequences with subtraction. That is, the recurrence used is $X[j] = (X[j - 100] - X[j - 37]) \mod 2^{30}$ and the 'seed' is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around 2^{129}.

2.5. Desirable Properties of a RNG/PRNG

Niederreiter[**78**] classifies pseudo random number generator (PRNG) requirements as structural, statistical, complexity-theoretic and computational. Knuth[**49**] goes farther, attempting to carefully define a random sequence. For our purposes we will concentrate here on the more concrete properties and avoid delving into an almost philosophical discussion that ensues when attempting to define a random sequence as fully as Knuth does. We will also discuss some desirable computational properties which are unique to RNGs.

2.5.1. Structural and Statistical Properties of the Generated Sequences. We will lump these together rather than attempting to separate them as Niederreiter does. Simple structural properties include the generator period, resolution, range, etc. which are directly related to comparable

statistical properties. Uniformity and independence are the other two major properties in this group. Uniformity is a major concern when the generator is used to drive Monte-Carlo integration techniques, where the structure of the generated data set is critical to the accuracy of the numerical integration. It is also key to independence when the property is applied to multidimensional data sets gathered from the generator.

The uniformity of the data in various dimensions is also related to the single dimensional properties of short and long term correlations. These properties are in fact what the spectral test and discrepancy which we discuss in section 2.7.5 were designed to test, and what the EST was designed to illuminate.

Note that while uniformity is simple to achieve, it is not easy to achieve simultaneous with independence. There is also a problem with being too uniform, resulting in reduced variation in the statistics.

Whenever we use an RNG, we are taking a subsequence from the generated random sequence. This may be a contiguous or non-contiguous sequence. In either case, the structural and statistical properties of the original sequence must be retained to assure validity of the application. Of course we cannot demand that all subsequences or combinations of subsequences of a PRNG have the desired properties since the sequences are not actually random and there will always exist subsequences under which the properties will fail. We should however try to insure that these failing subsequences be relatively pathological (for example, requiring knowledge of the underlying recursion).

2.5.2. RNG & PRNG Implementation Properties. Properties of RNGs which we must consider for implementation include the cost of the hardware, number of bits provided in u_i, physical space required, and reliability. Hardware generators tend to only produce a few bits per second of random data, so the speed of data generation is also an issue. Common with PRNGs, hardware RNGs ideally will be portable across computing platforms. PRNG specific implementation properties include the computational complexity of the generator in terms of both memory and execution time, portability to different hardware and software environments and reproducibility of the sequence. Suitable references are [**49, 47, 13, 90, 92, 76**].

2.5.3. Complexity Theoretic Properties. Finally there are complexity theoretic considerations for generators which will be used in cryptographic or security applications. We will leave this for ourselves and readers to investigate later[**13, 90, 76, 92**].

2.6. RANDU and Other LCGs

2.6.1. History. RANDU (a LCG defined as X_0 odd; $X_{n+1} = (65539X_n) \mod 2^{31}$) is perhaps the most infamous PRNG in existence. Hopefully only existing today in a state similar to certain deadly viruses that are kept for research purposes after having been eradicated from the population. R provides a relatively short sequence from RANDU stored as a data set for study. The infamy of RANDU is due to the fact that it was used in many scientific computing facilities for many years without researchers knowing its problems.

RANDU had passed all known tests of the day and was considered a good generator, up until the mid to late 60's when a simulation was carried out which deviated far from theory. This motivated several years study which resulted in Marsaglia's article 'Random Numbers Fall Mainly on the Planes'[**65**] and the development of the spectral test[**19**]. It was at this point that hidden structure in random sequences became widely appreciated.

In Figure II-1 we have on the left panel a matrix of orthogonal views of 10,000 points from RANDU plotted in the three dimensional hypercube. This is effectively the view of the data taken by tests before the existence of the hyper-planes in RANDU (and all LCGs) was discovered. The right panel shows the same data rotated from the orthogonal orientation to illustrate the parallel planes upon which all of the data lies. These hyper-planes cannot be seen either visually or mathematically by techniques limited to the orthogonal axes of the space like the chi-squared or other tests available at the time. The spectral test was the first technique developed which allowed this hidden structure to be illuminated.

2.6.2. All LCGs (Such as Unix rand()). So are LCGs like RANDU all banished to oblivion? Not at all. All LCGs will have parallel equally spaced hyper-planes. But if the planes are close enough and not at a critical angle in 3-space, the LCG may still be useable. A quick review of the literature today would find numerous cases of LCGs in use. Indeed, not all LCGs are this bad and even RANDU is still quite adequate for many applications. However they are not blindly trusted anymore. There is now a solid body of theory and tests in sources such as Knuth[**49**] and others

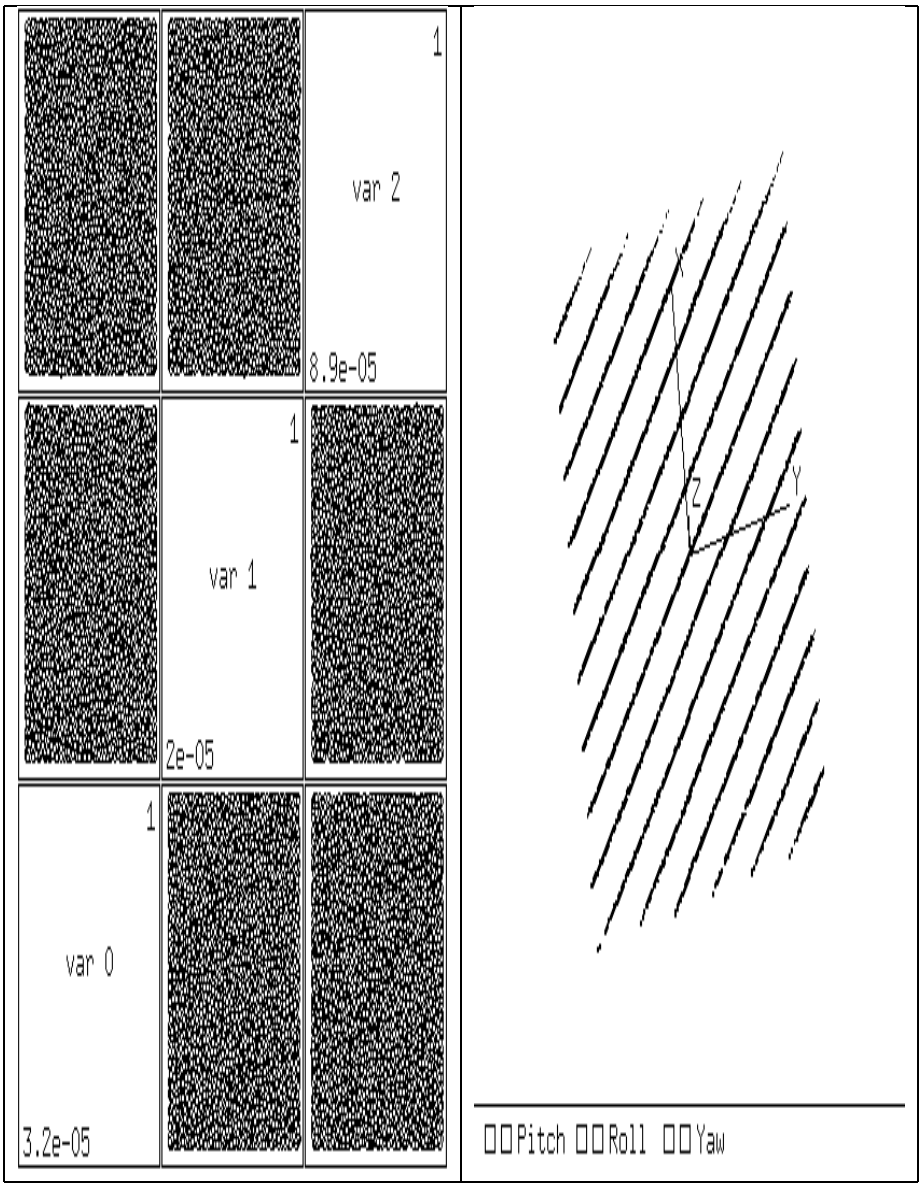


FIGURE II-1. RANDU Hyper-planes in 3 Dimensions

[**24, 25, 56, 59, 70, 85**] which allow the user to carefully choose or reject parameterizations of LCGs for their particular use.

2.7. Testing RNGs and PRNGs

Assurance of statistical quality of the generator can only come from testing outputs against theoretical results. These can be either specific results for the problem being studied, or more general results for the generator itself. A good starting point for testing RNGs and PRNGs would be either Knuth[**49**] or Kennedy and Gentle[**47**].

2.7.1. Philosophy and Testing a RNG or PRNG. There can be no definitive finite test suite for RNGs or PRNGs since the space of possible failures (H_1 or alternative hypotheses) is of infinite dimension. We will discuss this a little more later. This however doesn't prevent us from doing what we can. There are three basic types of RNG and PRNG tests: theoretical, empirical and application. The first are fairly obvious and will be discussed in more detail below. The last was actually alluded to already in this dissertation and is a blending of both theoretical and empirical tests tailored to the specific application where the generators will be used. Effectively these tests run the application under conditions for which theoretical results are already known and then the results are compared with theory.

The EST combines the power of the strongest theoretical tests, spectral and discrepancy, and can be applied to sequences shorter than full period, for any RNG or PRNG. It can also detect wide classes of multidimensional structures. This makes it good candidate for broad application to non-application specific generator testing situations.

2.7.2. Theoretical Tests. The availability of theoretical proof of correct statistical properties is the most desirable situation. We are then left

only with concerns about the actual implementation of the generator. Most generators have proof of the uniformity of the output of the full sequence and generally limited subsets before they are are even considered. If not, it is probably not worth even starting with the generator. Paired correlations and low order independence proofs are also available for many of the better studied generators. Only congruential generators and generators with a lattice structure have higher order theoretical tests available in the form of the spectral test and discrepancy tests respectively.

2.7.3. Empirical Tests. The limited availability of theoretical tests makes empirical tests necessary. RNGs also must be tested empirically. Good discussions of empirical tests can be found both in Knuth[**49**] and in Kennedy and Gentle[**47**]. There is no limit to the possibilities for testing a sequence. Effectively, every computation using random number generation with theoretical results available is an empirical test. Knuth describes in [**49**] 12 separate empirical tests. Some are common statistical tests such as the a Chi-squared or Kolmogorov-Smirnov tests for uniformity. Others such as Serial, Runs, Gaps, Poker and Birthday spacings are based on specific scenarios aimed at detecting patterns in the numbers generated. In each case the tests have associated discrete probability theory available from which to construct a testing framework and decision rule.

2.7.4. Diehard. George Marsaglia's DIEHARD[**66**] suite deserves separate mention here. Dr. Marsaglia has been working with PRNGs for over 40 years. During this long career he has amassed an incredible amount of

experience. He has collected those tests which he feels are the most effective into a suite called DIEHARD[**66**] which is available freely on the internet at [http://stat.fsu.edu/pub/diehard] along with hundreds of megabytes of pseudo random data generated from various sources and which pass the test suite. The DIEHARD suite includes:

- Birthday Spacings Test
- Overlapping 5-Permutation Test
- The Bitstream Test
- Overlapping Pairs & Quadruples Sparse Occupancy Tests
- DNA test considers an alphabet of 4 letters:: C,G,A,T,
- Count The 1's Test on a stream of bytes and for specific bytes
- A Parking Lot Test
- Minimum Distance Test
- 3D Spheres Test
- Squeeze test
- Overlapping Sums
- Craps Test

2.7.5. Spectral Test and Discrepancy. Perhaps the two most trusted and stringent single tests available are the spectral test of Coveyou and MacPherson [**19**] and Neiderreiter's [**78**] discrepancy tests. Both are theoretical tests designed to find specific structures in specific classes of PRNGs (LCGs and lattice structure generators respectively) and are applicable only to the PRNGs which fall into these two classes. They are also theoretical tests applied to full period generators.

The spectral test analyzes a linear congruential generator (LCG) using an FFT of the theoretical PDF of the generator. This theoretical PDF can only be derived from the recurrence relation of the generator and then only when the form is sufficiently tractable. This has only been worked out and coded into software for LCGs. For a perfectly independent uniform generator this FFT, when applied to the full period of the generator, will correspond exactly to 1 for all frequencies equal to zero and zero otherwise. Deviation from this indicates lack of uniformity. Knuth provides a geometric interpretation of this as the separation between parallel hyper-planes. More detailed discussions of the test are available in [**19, 47, 49**].

Discrepancy[**78**] tests look for the supremum of the deviation from expected 'flatness' over various classes of rectangular subregions of the unit hypercube. The test is aimed primarily at determining the maximum deviation from uniformity over a lattice, which can then be directly related to integration error of a quasi-Monte Carlo integration. The test criteria is then stated in terms of a maximum bound on the discrepancy based on the users criteria (usually an integration error bound). Statistical bounds for discrepancy based on asymptotics have been developed by vanHameren[**99**].

A modification of the spectral test called the weighted spectral test [Diaphony] was developed by Hellekalek and Niederreiter[**33**]. This test is also shown to be related to discrepancy[**61**] with equivalent bounds available and is not limited to generators with a lattice structure. There is also some work on the asymptotic distribution available from Leeb[**58**].

2.7.6. Statistical Decision Framework for RNG and PRNG Testing. In this section we will explore the use of statistical testing in a context which

is quite non-standard. A statistical test is an apriori decision rule derived from some assumed (or hypothesized) state of 'nature' which statisticians call H_0 or the NULL hypothesis. The rule specifies under what conditions the experimenter should reject this assumed state of nature based upon the null hypothesis and a chosen risk level α representing the probability of erroneously rejecting H_0. This is usually expressed in the usual Type I (erroneously reject H_0) and Type II (erroneously accept H_0) errors for the statistical decision framework. The normal risk structure used in this framework is predicated on the idea that we would like to be relatively certain when we make statements which go 'against nature', or reject the null hypothesis.

There are a couple of factors which make RNG and PRNG testing somewhat different than this standard framework. First we're primarily looking for understanding of the generator, not making life or death decisions. This makes the negative impact of a Type I error rather small. Second, we are not limited to the currently available data set. New data to perform extended or additional analysis is readily available. Together these factors suggest a somewhat different view of statistical testing for RNGs and PRNGs than that used in a standard statistical context.

2.7.6.1. *What Does it Mean to 'Fail' a Test?* In the case of testing an RNG or PRNG, the 'state of nature' upon which we base our selection of H_0 is the desired properties of the generator. If our statistical test rejects H_0 then we have determined that the assumed state of nature is not true based upon the data we are analyzing. Stating it another way, we have found significant structure in the batch of data. In a standard statistical decision framework we would then declare that we have evidence that the original

state of nature or H_0 is false, confident that there is only a small chance that we might be proven wrong (or worse in the case of many life critical decisions).

Note that in neither case can we be certain that the state of nature is not true (or in our case that the generator is not k-dimensional uniform).

2.7.6.2. *The Power of a Test.* The opposite side of the Type I error coin is the Type II error condition. To get at this, we need to know something about particular alternatives to H_0 and the distribution of our test statistic under these alternatives. The possibilities here are infinite and without some form of theory about what they might be, we cannot even begin this type of development. In fact this is effectively what has been to date in developing most RNG and PRNG tests. That is to first hypothesize an alternative (a particular structure) and then design a test for this particular alternative against an H_0 of uniformity or independence, etc. It will be seen that our proposed test (EST) detects very broad range of structures. The nature of the type of structure being detected by the EST makes identifying and developing distribution theory for alternatives a daunting task at best.

A better approach to determining the power of a test like this is empirical evidence. The application of the test to a broad range of types and implementations of RNGs and PRNGs will give us the assurance that we can keep our Type II errors low. Chapter 6 will begin this work by applying the EST to over half a dozen generators.

2.7.6.3. *What Should be Done When a Rejection is Found?* First, remember a note in the documentation with Marsaglia's DIEHARD suite: "p happens". This is alluding to the fact that if you run a test enough, it will

eventually reject H_0. And with any PRNG test, it will be run numerous times by its very nature. Marsaglia also recommends that the normal statistical risk structure be set aside for PRNG testing and that the researcher demand p values which are zero to numerous places before rejecting a generator. We would go a bit farther and suggest that a generator should never be rejected until the source of the detected structure has been identified.

We have chosen a statistical decision framework as the basis for our test. Others testing PRNGs take a different approach, not attempting to use a standard statistical decision framework, but rather using calculated bounds on errors which might result from the generator structure. Under this framework the rejection clearly implies a high probability of error greater than the users declared acceptable bounds and either another generator needs to be used or the subject generator needs to be modified to meet the bounds criteria.

Things are not that simple when a statistical framework is utilized. First we need to look at the strength of the rejection. If p is moderately low (this is intentionally vague) we should raise a yellow flag. More testing is indicated. If p is extremely small, say zero to 3 or more decimal places, raise a red flag. In either case further examination of the generator and the relationship of the test to the desired use is in order. Additional runs should be made, using the same test on other generators to see if they all exhibit the behavior. The test should also be run on different subsequences from the same generator. If possible other tests like DIEHARD should be applied. Finally keep in mind that the failure does not tell us what the source of the problem is.

In chapter 6 we will present a testing sequence for our new EST, which applies some of these ideas.

2.8. Summary

We should always use random number generators with caution, especially when working in a complex computational environment. In [**70**] Marsaglia suggests that whatever the generators, the user should test the code with a number of different generators to help validate the results. With packages like R that provide a number of generators with a common interface or Thomborson's mrandom [**96**], this is quite easy. Ferrenberg, et.al. [**26**] suggest that the researcher test the random number generator in the software being developed against theoretical results. Both approaches are a good idea when the results are critical.

There is a strong need for a computationally manageable test with the power of the spectral test which can be used for any generator. This is the emphasis of our research, and the reason for developing the EST.

CHAPTER III

EMPIRICAL SPECTRAL TEST

This chapter will introduce the Empirical Spectal Test (EST), discuss briefly the software implementation and compare it to previous tests for RNGs and PRNGs. The EST itself is relatively simple and is presented in flowchart form in section 1. Section 2 will examining the computational complexity of the EST. In section 3 we will discuss the motivation for the EST, comparing it to previous tests. Finally section 4 will discuss the applicability of the EST to sequences of random integers.

3.1. Basic EST

Our proposed testing scheme applies a discrete Fourier transform to the empirical PDF of the generator output constructed from a pseudo random sequence of N points in k-dimensional space, where from the random sequence $u_0, u_1, \ldots$ we form vectors $\boldsymbol{v_i}$ via $\boldsymbol{v_i} = (u_{ki+1}, u_{ki+2}, \ldots, u_{ki+k})$. These vectors are used to accumulate a k-dimensional matrix of cell counts over the unit hypercube, and then dividing each by N to get relative cell frequencies. A k-dimensional FFT is then applied to the relative frequencies and the coefficients are statistically analyzed for non-uniform structure using the asymptotic results of the next chapter.

The statistical tests applied to the k-dimensional Fourier coefficients include a Fisher PLSD scheme using an initial test for overall significance

based on the asymptotic Chi-squared distribution of a quadratic form, followed by individual element tests if overall significance is determined. A Bonferonni multiple comparison is also applied if the overall test fails to reject.

3.1.1. Testing Algorithm. Figure III-1 shows the EST algorithm. In the software (presented in Appendix B) this algorithm is not fully automated. Decisions on which intervals to apply to the individual coefficients are left to the user.

3.1.2. R Code. An example test was implemented in R to provide a more familiar example for readers not familiar with the C programming language and to obtain better graphical output for the C implementation. This code consists of the files testRNG.R, tview.R and mdftt.R.

An R function is also provided to generate random sequences for the C code as well as plot the resulting outputs. This is in testRgen.R.

3.1.3. C Code. For efficiency and greater memory capacity, the FFTs and numerical computations of the tests were implemented in C. This code outputs data which can then be loaded into the R functions for plotting. This C code actually was the first version implemented and contains a number of older random number generators packaged into a wrapper function for easier interface to the tests. The code for these generators and the wrapper function are included should any one find them useful. In the testing presented in chapter 6, only the RANDU, SUPER and RANLIB generators from our implementation was used in the analysis.

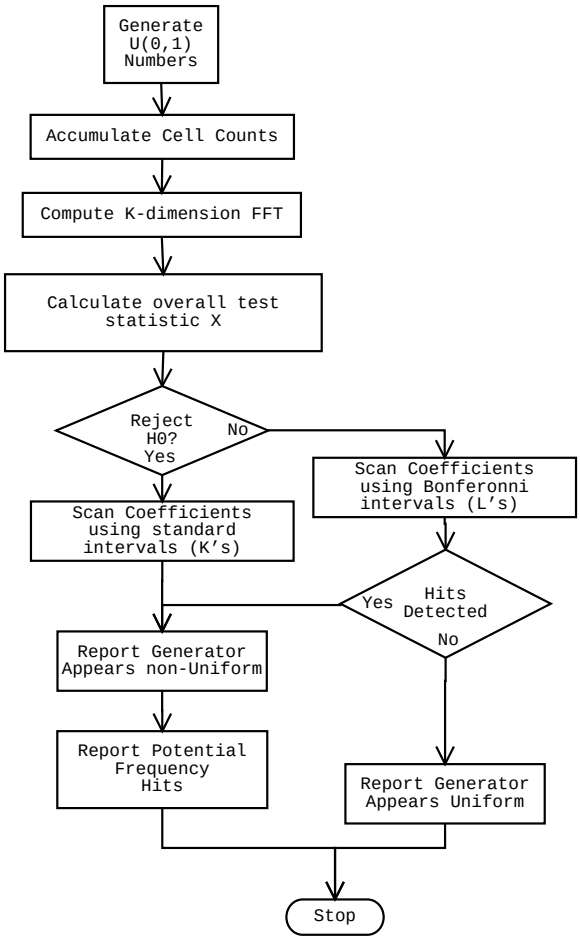


FIGURE III-1. Testing Algorithm

All C code (in Appendix A) is written in ANSI C and compiled with gcc (the Gnu ANSI C compiler) on an X86 Linux workstation and on a SUN Sparcstation. Graphics and 2 dimensional examples were produced using R.

3.1.3.1. *Random Number Generators.* There are many random number generators available. They all have different interface requirements and return different types and ranges of outputs. The first purpose of the package is to provide a shell about these generators which allows the user to easily

use any of the generators without changing more than a constant in an initialization call. The mrandom package of Thomborson [**96**] does this for a collection of serial generators.

It is also desirable to be able to use different generators simultaneously for testing or even for different situations within a single program. We do this with random streams. A stream of random numbers can be selected by making a call to the initialization routine. Numbers from the stream are then returned by calling a single routine with the stream identifier. The C interface specification for the initialization and generator are shown below.

(1) RANLIB Algorithm. This algorithm is based on Pascal and Fortran versions derived from a paper by L'Ecuyer, P. and Cote, S. [**55**].

(2) RANDU Algorithm. The RANDU is a pure linear congruential generator know to exhibit significant structural problems in 3 or more dimensions (see section 2.6.1). This can easily be seen by rotating a scatter plot of the data in 3-d appropriately.

(3) SUPER Algorithm. This is Marsaglia's super-duper algorithm consisting of the exclusive-or of two good generators, a Tausworthe (linear feedback shift register) and a good linear congruential generator.

3.1.3.2. *Transform Algorithm.* The FFT is computed using the Numerical Recipes routine 'fourn', which computes a multidimensional transform using recursive applications of the FFT and returning the transform in the input array. This algorithm is documented in 'Numerical Recipes in C'[**86**]. It takes an input vector of complex numbers stored such that adjacent float type cells contain the real and imaginary parts of the numbers. The successive FFTs are applied to the array without copying elements to a working

area first, thus minimizing data shuffling and memory utilization. Input and output arrays are stored in a vector such that the k-dimensional array indices are incremented with the last index changing fastest and the first least frequently. The bin identification algorithm below is designed to translate a k-dimensional point in the unit hyper cube directly into this structure.

3.1.3.3. *Bin Identification.* The counts are stored in a one dimensional structure of bins indexed to match the requirements of Numerical Recipe's fourn.c routine. Bins are identified by a k-vector $\boldsymbol{d} = (d_0, , ..., d_{k-1})'$ where $0 <= d_i <= (B-1)$ and B is the number of bins/dimension. Bin $(d_0, ..., d_{k-1})$ is stored at location $l = \sum_i (d_i * B^{(k-i)})$. Taking advantage of the fact that we are working with points in the unit hyper cube, we know that $0 \leq x[i] < 1.0 \Rightarrow 0 \leq truncate(x[i] * B) < B$. We can then take $d_i = truncate(x[i] * B)$.

3.1.3.4. *Distribution Functions.* We have used cdflib from netlib for both the chi-squared distribution and normal distribution function evaluations. Implementation in the test code was verified by a spot check against values calculated with R.

3.1.4. Plots of EST Coefficients and Q-Q plots. To initially motivate the potential effectiveness of our test, we examined a couple of well known generators using the FFT of the empirical PDF of a subset of the generator output. If the data is uniform, the coefficients should be exactly 1 for the left most (all zero frequency) entry and zero otherwise. Early exploratory work with histograms of the coefficients suggest a normal distribution, so normal Q-Q plots were also generated.

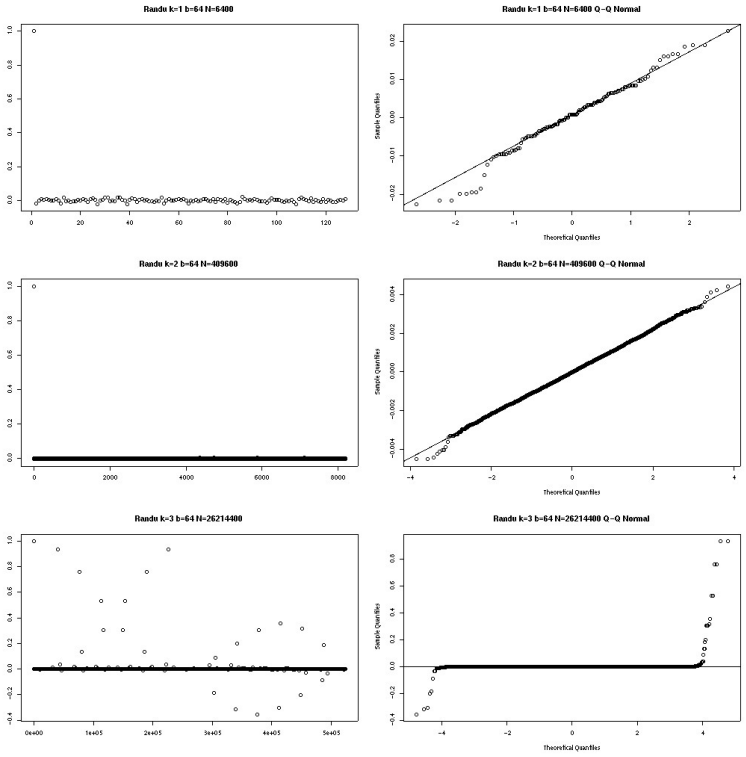


FIGURE III-2. RANDU Coefficients & Q-Q Normal Plots

Of the generators examined, RANDU is know to be bad. The plots on the left side of Figure III-2 show Fourier coefficients generated from the RANDU generator in the first three dimensions. Note the radically different behavior in three dimensions where the parallel hyper-planes first show up.

The right side of Figure III-2 shows Q-Q Normal plots of these three data sets with the first element having the value 1.0 subtracted off. We will justify this in the next chapter. It is clear from these plots that the generator is relatively well behaved in the first two dimensions, but changes drastically in three as expected.

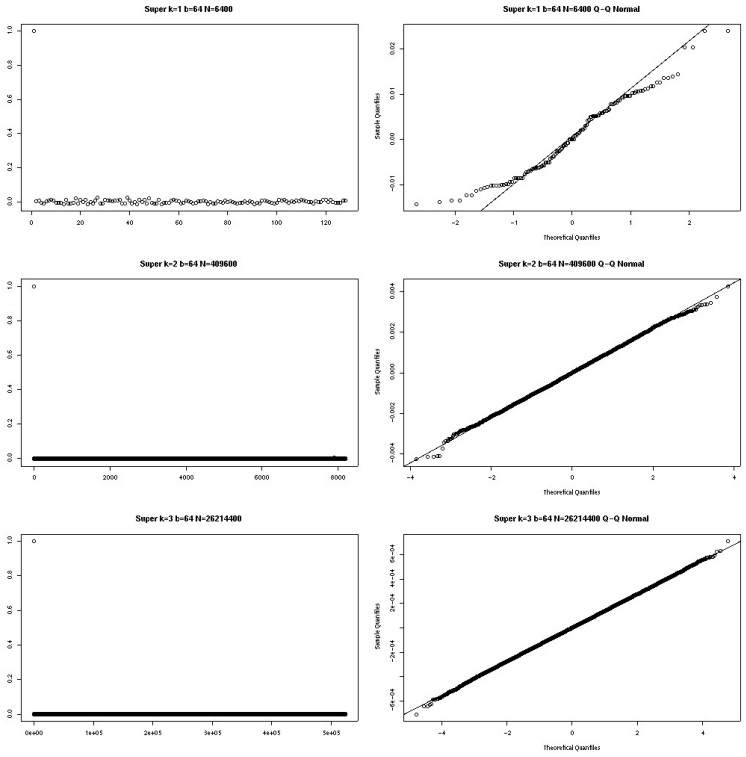


FIGURE III-3. Super Coefficients & Q-Q Normal Plots

Figures III-3 and III-4 show the same plots for Marsaglia's Super-Duper generator and for the RANLIB generator. Both of which are know to be much better than RANDU.

3.2. Complexity Analysis

There are three major components to the EST algorithm. The first is the collection of the cell counts. This operation requires kN operations, each requiring either two differences or a single division and an integer truncation. Thus we have $2kN$ floating point differences, or kN division and integer truncate operations. The later is how the current code is implemented.

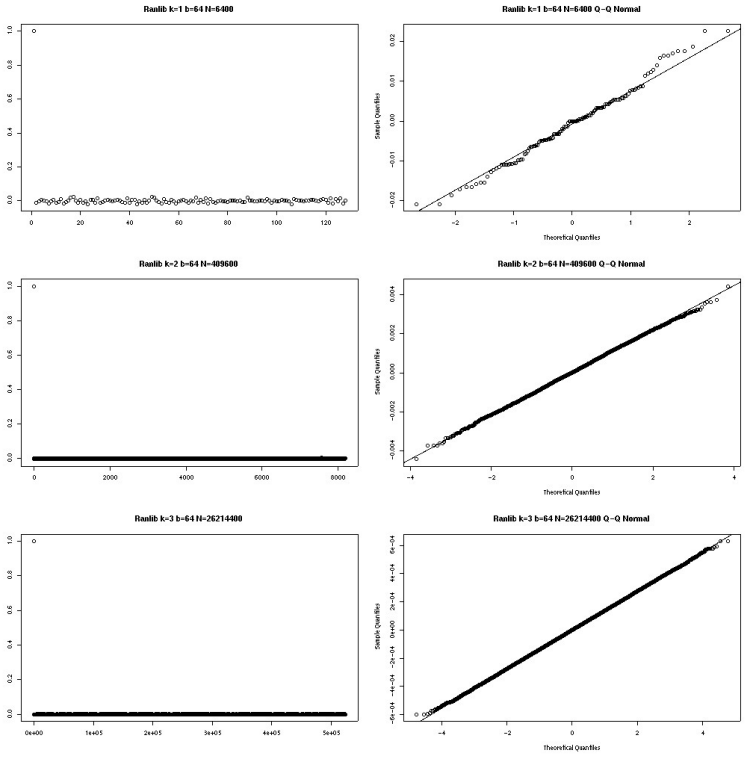


FIGURE III-4. RANLIB Coefficients & Q-Q Normal Plots

In either case we have an $o(kN)$ process. This step can be reduced to integer mask and shift operations with no floating point if we either use an integer output from the generator, or are willing to make the code specific to the particular floating point format in the implementation.

The second operation is the FFT and will be addressed last. Third is the calculation of the test statistic X. This operation is

$$X = 2N \sum_{l=0}^{2b^k-1} z_l^2 - 1$$

(z_l is the output of the FFT and is defined later in chapter 4) which is a summation of $2b^k$ elements, exactly $2b^k - 1$ of them requiring a floating

point multiply. So we have a complexity here of $o(b^k)$. This term will be easily dominated by either the FFT or the cell counting operations.

A discrete Fourier transform is defined in terms of each of the b^k combinations of $\boldsymbol{n} = (n_1, \ldots, n_k)$ Fourier frequencies. For each frequency combination $\boldsymbol{n}$ the associated coefficient is calculated as a sum over the observation bins as:

(III-1) $$f_{\boldsymbol{n}} = \sum_{l_1=0}^{b-1} \cdots \sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b} n_1 l_1} \cdots e^{i\frac{2\pi}{b} n_k l_k} \cdot y_{\boldsymbol{l}}$$

where $\boldsymbol{l} = (l_1, \ldots, l_k)$ denotes the vector of summation indices.

This definition shows that the DFT can be computed directly as b^k sums (one for each f_n) of b^k terms. Each term is the product of an exponential and constant and the exponential has a couple of products in it. So the complexity of this brute force method is $o(b^k b^k)$.

The k dimensional FFT is known to be an $O(b^k \log_2 b^k)$ complexity process[**81, 86, 104**] when b is a power of 2. In the Numerical Recipes code currently being used, b must be a power of 2.

For values of b that are not powers of 2, several approaches have been developed for recursively building up the DFT. These have complexities that usually depend directly on the prime factorization of b and fall between those for the brute force method and the FFT. Generally for these recursive methods, a b with many small factors is much better than one with a small number of large factors. An example of this can be found in the FFTW papers by Frigo[**28, 29**].

This analysis assumes a single processor architecture. Parallel implementations of the FFT exist and would have markedly better performance. The free FFTW package [**28**] has parallel options.

Most algorithms can compute Real FFTs in about 1/2 the time of complex FFTs when b is a power of 2. This advantage disappears when p is a large prime, or a product of large primes.

Pulling this together we have that the EST is $o(kN)$ for the cell counting processand for the FFT and test statistic computations $o(b^k b^k)$, or $O(b^k \log_2 b^k)$ when b is a power of 2.

Memory requirements for the EST as currently implemented consists primarily of a single $2b^k$ floating point array which stores the initial cell counts, and is then used to compute the FFT in place using the Numerical Recipes FFT routines. Not all FFT algorithms compute in place and the memory requirements can be as much as twice this in some implementations.

Note that even a brute force DFT implementation with floating point operations in the cell count operations can have an advantage over previous spectral tests with a complexity of only $o(kN) + o(b^k \log_2 b^k)$, compared to the weighted spectral test with complexity $O(kN^2)$, if we use optimal algorithms and bin sizes.

3.3. Potential Effectiveness of the Test

3.3.1. Structures Detected by the EST. Effectively all of the tests for RNGs and PRNGs are looking for structure of some kind. The spectral test is looking for the hyper-planes in an LCG. Discrepancy is looking for the greatest non-flatness of the pdf, etc. Empirical tests are generally looking

in the generator output for a specific type of structure hypothesized by the test developer. In each case, the test is designed to look for a specific type of structure in the generated sequence. The FFT is known to provide an effective means of extracting arbitrary structure from noisy environments. Thus the EST detects the existence of and helps in the identification of many structures in the empirical pdf of the generator.

3.3.2. The EST Compared with a Chi-square Test on the Original Cell Counts. The EST first gathers cell counts in a k dimensional hyper-cube, just as a chi-square test for uniformity would. In the chi-square test the asymptotic normality of these cell counts is used to detect significant deviation from the expected cell counts in the data. However the form of the test statistic effectively limits the chi-square test to orthogonal views of the data similar to the left panel in Figure II-1. The FFTs application of combinations of frequencies in k-dimensions allows it to effectively rotate its view of the data automatically allowing the determination of structure(s) not visible from orthogonal views of the data. It also would allow the detection of k-dimensional structures such as an embedded sphere in a 3d analysis. This makes the EST more sensitive to a wider class of structures than the chi-square test.

3.3.3. The EST Compared to Spectral and Discrepancy Tests. This approach has similarities to the spectral test of Coveyou and MacPherson [**19**] and Knuth[**49**], the weighted spectral test or Diaphony of Hellekalek and Niederreiter[**33**] and discrepancy tests of Niederreiter[**78**]. However

each of these approaches are aimed at structural limitations which are important for Monte Carlo integration techniques, using measures of the resolution of the point set as the test criteria. As such, they are generally not as sensitive to statistical properties as we would like for assurance of the statistical quality of simulations.

In the case of the Coveyou and MacPherson and the Knuth spectral tests the full period analytic output of the generator is analyzed to ascertain the spacing between covering sets of parallel hyper-planes by calculating the minimum wavelength at a given dimensionality. This is a theoretical test requiring the mathematical form of the generator and using computation for calculation of the derived numerical relationships. A note in Coveyou & MacPherson indicates that this is the discrete equivalent of the Weyl theorem for uniformity. Our tests were specifically motivated by the desire for an empirical equivalent to this spectral test which needs neither the form of the generator or necessitated full period output. What we do goes somewhat farther than the original spectral test, in that it examines more than the minimum wavelength.

Neiderrieter's discrepancy test looks at the supremum of the discrepancies from expected cell frequencies over differing classes of subsets in the unit cube using the analytical form of the generator. These cells consist of rectangular subregions of k-dimensional space. This test is examining statistics closely related to the empirical PDF and so it would appear more closely related to our work than the spectral test. However discrepancy is examining the supremum of the discrepancies over rectangular subregions, not the structure of the discrepancies over disjoint subregions.

Hellekalek's weighted spectral test (or Diaphony), is perhaps the closest to our empirical spectral test. Unlike either the spectral test or discrepancy, this test is applied directly to the data, allowing application to non-lattice structure generators and to subsequences of the generator output. Diaphony is a numerical analysis procedure originally developed as an analog to discrepancy in special quasi-Monte Carlo integration methods. The idea here is to use a weighted version of the spectral test, emphasizing small frequencies which the authors state are more important indicators of uniformity. This appears to be a somewhat subjective criteria. This weighted form of the spectral tests corresponds to Diaphony.

The FFT analysis of the empirical PDF provides not only a different view of the uniformity of the generated data, but also has the benefit of all of the special purpose hardware and software developed during the decades of use of FFTs for other purposes. It can also be more easily tailored to the level of confidence the user requires in the data set, by adjusting the number of cells the empirical PDF is collected over. If high resolution is desired, but compute power is limited, the test can be applied on a reduced grid (subset) of cells with the desired resolution. Even though the number of generated points must be high in this case, the much heavier computation required for the analysis is limited to the chosen subset of cells and thus can be tailored to the available compute power of the test machine. Thus our test is configurable to allow examination of the generator in a wide variety of ways with user controllable computational loads.

3.4. EST Applies Equally to $Z[0, m]$ and $Z[0, 1]$ Streams

In all of our discussion to this point we've been talking about testing U[0,1) generators. What about integer or even binary streams? A closer look at the EST will reveal that what we have done is to reduce an arbitrary sequence to a lattice structure on the unit hypercube. Note also that one means of obtaining integer streams is to derive them from a U[0,1) sequence as was discussed in the first example of section 2.2.1. This is the same method used to reduce the arbitrary U[0,1) sequence down to a lattice for application of the FFT. Hence for $Z[0, m],\ m \geq 1$ streams we can apply the EST with our bin size (b) set to m and the remainder of the test follows through.

CHAPTER IV

THE EST DISTRIBUTION

We will now go into the actual distribution of the EST coefficients which are the basis of the test. First in section 1 we will lay the groundwork by getting the matrix form of the multidimensional DFT of an empirical PDF derived from the random sequence. In section 2 we will then derive the distribution of the coefficients output by the DFT. From this distribution we will develop in sections 3 and 4 the overall test and the confidence intervals for examining individual coefficients in the EST.

4.1. Multidimensional FFT of the Empirical PDF of a k-dim RNG

An empirical PDF can be obtained from a sample of N points in k dimensions from a uniform $[0, 1)$ random number generator by producing a grid of cells on the unit hyper cube and counting the occurrences of points within each cell. These cell frequencies are then divided by N to produce an empirical PDF. If the underlying generator is truly independent uniform, the resulting k-dimensional PDF will have equal probability for all cell frequencies. The transformation of the empirical PDF is achieved using the k-dimensional discrete Fourier transform (DFT) to the k-dimensional empirical PDF cell frequency array. If the underlying generator generates independent uniforms, the expected transform consists of the mean in the $(0, 0, \ldots, 0)$ frequency (transformed) cell and zeros in all others. If not, the structure will show up as frequency information in one or more dimensions.

We first need some machinery to be able to readily work with a k-dimensional transform of the empirical PDF. Since we know that the Fourier transform is linear, we would like to be able to use standard linear theory. As will be shown, the transform can be viewed as a complex linear function of the form $\boldsymbol{f} = \boldsymbol{A}\,\boldsymbol{y}$. This will allow us to deal with the arbitrary dimensionality and resolution of the test in a general fashion.

NOTE. We denote vectors by bold lowercase ($\boldsymbol{a}$), matrices bold uppercase ($\boldsymbol{A}$), and scalars with standard font (a).

4.1.1. Construction of the Empirical PDF.. We start by first constructing the empirical PDF on a $\frac{1}{b} \times \cdots \times \frac{1}{b}$ or $\left(\frac{1}{b}\right)^k$ resolution grid of hypercubes. The dimension of the space we are working with will be denoted k. That is, we are generating points in $\Re^k$ distributed uniformly on the unit hypercube $[0,1)_1 \times [0,1)_2 \times \cdots \times [0,1)_k$. Using a leap-frog partitioning scheme we will take disjoint k-tuples from the generator as our vectors in $\Re^k$. Let u_j = the j^{th}univariate generate from a uniform random number generator. If we sequentially assign these univariate generates to the elements of a k-dimensional vector $\boldsymbol{u}$ such that the sequence of k generates for the l^{th} generated vector is the $u_j's$ with $j = \left\{ \begin{array}{cccc} kl & kl+1 & \cdots & kl+k-1 \end{array} \right\}$ giving $\boldsymbol{u}_{\,l} = (u_{kl}, u_{kl+1}, ..., u_{kl+k-1}), l = 0, \ldots, N-1$.

Partition the unit hypercube into equivolume hypercubes (cells) using b bins in each dimension, producing $T = b^k$ equivolume cells. Each cell is then a $\left(\frac{1}{b}\right)^k$ hypercube within which we will be counting points. Let $\boldsymbol{t} = (t_1, t_2, \ldots, t_k)$, where cell membership is determined by taking the largest integer $\leq u_j b$ designated as $t_j = \lfloor u_j b \rfloor$,[$t_j = 0 \ldots b-1$]. This scheme provides the basis for an algorithm to quickly allocate points to cells

as they are generated with only a single multiply and an integer truncation as well as the means to roll out a k-dimensional grid into a vector for simpler notation later on.

The cells are rolled out in row order (column or rightmost index changing fastest) into a single vector. Each of the T cells are then numbered using the mapping function

(IV-1) $$c_{\boldsymbol{t}} = \sum_{j=1..k} t_j b^{(k-j)}$$

where $c_{\boldsymbol{t}} \in 0...T-1$. For k=2, b=8, the $T = 8^2 = 64$ cells are shown in Figure IV-1,

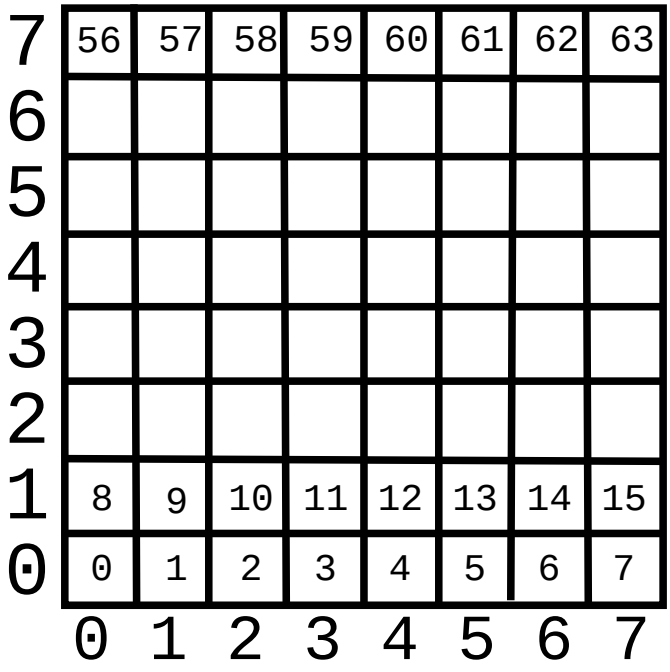
NOTE. When using the cell numbers $c_{\boldsymbol{t}}$ as indices we will denote them by $\boldsymbol{t}$ to reduce the complexity of our notation. So $\boldsymbol{X}_{c_{\boldsymbol{t}}} \equiv \{\boldsymbol{X}\}_{\boldsymbol{t}} \equiv \boldsymbol{X}_{\boldsymbol{t}}$.

The empirical PDF of a k-dimensional random number generator using counts of points falling into the b^k equal volume k dimensional cells is a T element vector function with each element $f_N(\boldsymbol{t})$ defined as:

(IV-2) $$f_N(\boldsymbol{t}) = \frac{1}{N} \sum_{l=0..N-1} I_{\boldsymbol{t}}(\boldsymbol{u}_l),$$

where the function I is an indicator function on the hypercube cells

$$I_{\boldsymbol{t}}(\boldsymbol{u}_l) = \begin{cases} 1 & \boldsymbol{u}_l \in \boldsymbol{t} \\ 0 & else \end{cases}.$$



(a) Grid for k=2, b=8

FIGURE IV-1. Two Dimensional Grid

There are T of these values, one for each value of $\boldsymbol{t}$. We will work with them as a vector $\boldsymbol{y} = \{f_N(\boldsymbol{t})\}$. The transform will be applied to this vector.

4.1.2. Discrete Fourier Transform (DFT) of the Empirical PDF. We now turn to the transform of the empirical PDF IV-2. The discrete Fourier transform (DFT) of the k-dimensional empirical PDF is an orthogonal transformation which maps spatial information in the unit hypercube in $\mathfrak{R}^k$ to frequency information in $\mathcal{C}^k$. If we let $\boldsymbol{s} = (s_{n_1}, \ldots, s_{n_k})$ denote the

Fourier frequencies, $s_{n_j} = \frac{2\pi n_j}{b}, n_j = 0 \ldots b-1, j = 1 \ldots k$, then the vector $\boldsymbol{s}$ consists of a set of "crossed" orthogonal frequencies in k dimensions, allowing us to extract "waves" of length 1 down to length $\frac{1}{b}$ in the data from any direction, not just along a single axis as with a single dimension FFT.

DEFINITION 4.1.1. The discrete Fourier transform is defined in terms of each of the T combinations of $\boldsymbol{n} = (n_1, \ldots, n_k)$ Fourier frequencies. For each frequency combination $\boldsymbol{n}$ the associated coefficient is calculated as a sum over the observation bins as:

$$f_{\boldsymbol{n}} = \sum_{l_1=0}^{b-1} \cdots \sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b} n_1 l_1} \cdots e^{i\frac{2\pi}{b} n_k l_k} \cdot y_{\boldsymbol{l}} \tag{IV-3}$$

where $\boldsymbol{l} = (l_1, \ldots, l_k)$ denotes the vector of summation indices.

NOTE. It is useful for computation and conceptualization of the multi-dimensional transform process to formulate this as a recursive DFT by pulling the exponentials through the iterated sums.

$$f_{\boldsymbol{n}} = \sum_{l_1} e^{i\frac{2\pi}{b} l_1 n_1} \left[\cdots \left[\sum_{l_k} e^{i\frac{2\pi}{b} l_k n_k} \cdot y_{\boldsymbol{l}} \right] \right] \tag{IV-4}$$

The following theorem provides us with a compact form for the multi-dimensional transform. A similar matrix formulation can be found in section 14.4 of Garg[**30**].

THEOREM 4.1.2. *The Fourier transform of the empirical PDF with resolution* $\left(\frac{1}{b}\right)^k$ *of a k-dimensional independent uniform pseudo random number generator expressed as a complex linear transformation is*

(IV-5) $$\boldsymbol{f} = \boldsymbol{A}\,\boldsymbol{y}$$

where

$\boldsymbol{f}$ *is a* $b^k \times 1$ *complex vector Fourier coefficients,*

$\boldsymbol{y}$ *is a* $b^k \times 1$ *real vector of relative cell frequencies and*

$\boldsymbol{A}$ *is a* $b^k \times b^k$ *complex matrix of Fourier multipliers with each row corresponding to a single frequency.*

PROOF. Equation IV-3 can be rewritten in terms of inner products of the frequency and summation index vectors $\boldsymbol{n}$ and $\boldsymbol{l}$ by first reorganizing the exponents in the product of exponentials inside the summation and then recognizing this summation as an inner product.

(IV-6)
$$f_{\boldsymbol{n}} = \sum_{l_1=0}^{b-1} \cdots \sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b}[l_1 n_1 + l_2 n_2 + \cdots + l_k n_k]} \cdot y_{\boldsymbol{l}} = \sum_{l_1=0}^{b-1} \cdots \sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b}\boldsymbol{l}'\boldsymbol{n}} \cdot y_{\boldsymbol{l}}$$

This makes it clear that for each combination of Fourier frequencies we are working with a b^k or T element summation of complex exponentials which are functions of the inner product of the frequency vector and the cell location vector. We now define a T element vector of Fourier multipliers $\boldsymbol{a}_{\boldsymbol{n}} = \{\boldsymbol{a}_{\boldsymbol{n}}\}_{\boldsymbol{l}} = e^{i\frac{2\pi}{b}\boldsymbol{l}'\boldsymbol{n}}$ which contains all of the multipliers in the T element summation in (IV-6). If we now use the cell identification

function (IV-1) to do a change of summation variables with associated ordering of $\boldsymbol{a}_{\boldsymbol{n}}$ to go from the iterated summation over each of the l_j's to a single summation over the cell identifiers, we can write the equation for each coefficient (frequency combination) as an inner product of a complex vector and the real data vector; that is,

(IV-7) $$f_{\boldsymbol{n}} = \boldsymbol{a}'_{\boldsymbol{n}}\, \boldsymbol{y}$$

From this form we can now build a matrix $\boldsymbol{A}$ with the $\boldsymbol{a}_{\boldsymbol{n}}$'s forming the rows. Each row corresponds to the k-iterated summation for each coefficient for a particular frequency combination. Columns correspond to individual relative cell frequencies across the combinations of frequencies in the transform.

This then gives us the entire k-dimensional Fourier transform expressed as a complex linear transformation.

$$\boldsymbol{f} = \boldsymbol{A}\, \boldsymbol{y}$$

with the desired properties. □

4.1.3. Structure of the $\boldsymbol{A}$ Matrix. The $\boldsymbol{A}$ matrix is critical to further work, so we should take a closer look at it. First we need to make the notation clearer by pulling the exponentials outside the matrix. We do this by expressing (as illustrated in Equation IV-8) an arbitrary sized matrix of exponentials as an exponential raised to a matrix power where the original

matrix of exponentials and the matrix power of e are the same size.

$$\text{(IV-8)} \qquad \begin{bmatrix} e^{M_{0,0}} & e^{M_{0,1}} & \cdots & e^{M_{0,k}} \\ e^{M_{1,0}} & e^{M_{1,1}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ e^{M_{j,0}} & \cdots & \cdots & e^{M_{j,k}} \end{bmatrix} = e^{\boldsymbol{M}}$$

Using this, we have $\boldsymbol{A}$ expressed as $\left(e^{i\frac{2\pi}{b}}\right)^{\boldsymbol{M}}$. Since the rows of $\boldsymbol{A}$ are formed from the iterated sums which generate the coefficients of the Fourier transform, each element of $\boldsymbol{M}$ is a dot product of a frequency vector and a summation index vector $M_{i,j} = \boldsymbol{n}_i' \boldsymbol{l}_j$ where the value of $\boldsymbol{n}_i$ is constant across each of the rows and the value of $\boldsymbol{l}_j$ ranges through all T possible combinations of the summation indices. Conversely the columns are a specific combination of summation indices $\boldsymbol{l}_j$ across the T possible combinations of frequencies $\boldsymbol{n}_i$. As long as we remain consistent with the ordering of our cell frequency vector, we can roll these out from the summations in any order we like. Since we have already rolled out the cell frequency vector using a relationship between a k-vector of integer values in the range $0 \ldots b-1$ defined in Equation IV-1, we will use that to roll out the columns of this matrix. The ordering of the rows is arbitrary, so we will use the same relationship to order them as we do the columns. Since the vector components are the same size and range over the same values, we will have a symmetric matrix.

a. The first row has all $\boldsymbol{n}_i = 0$ and the first column is all $\boldsymbol{l}_j = 0$.

b. For $i > 1$ we select the $\boldsymbol{n}_i$ and for the column select the corresponding $\boldsymbol{l}_i$.

M-matrix k=2, b=4

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
1	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**	**00**
2	**00**	01	02	03	**00**	01	02	03	**00**	01	02	03	**00**	01	02	03
3	**00**	02	04	06	**00**	02	04	06	**00**	02	04	06	**00**	02	04	06
4	**00**	03	06	09	**00**	03	06	09	**00**	03	06	09	**00**	03	06	09
5	**00**	**00**	**00**	**00**	01	01	01	01	02	02	02	02	03	03	03	03
6	**00**	01	02	03	01	02	03	04	02	03	04	05	03	04	05	06
7	**00**	02	04	06	01	03	05	07	02	04	06	08	03	05	07	09
8	**00**	03	06	09	01	04	07	10	02	05	08	11	03	06	09	12
9	**00**	**00**	**00**	**00**	02	02	02	02	04	04	04	04	06	06	06	06
10	**00**	01	02	03	02	03	04	05	04	05	06	07	06	07	08	09
11	**00**	02	04	06	02	04	06	08	04	06	08	10	06	08	10	12
12	**00**	03	06	09	02	05	08	11	04	07	10	13	06	09	12	15
13	**00**	**00**	**00**	**00**	03	03	03	03	06	06	06	06	09	09	09	09
14	**00**	01	02	03	03	04	05	06	06	07	08	09	09	10	11	12
15	**00**	02	04	06	03	05	07	09	06	08	10	12	09	11	13	15
16	**00**	03	06	09	03	06	09	12	06	09	12	15	09	12	15	18

TABLE IV-1. Example M-matrix

That is $\boldsymbol{n}_i = \boldsymbol{l}_i$ and thus $\boldsymbol{n}_i' \boldsymbol{l}_j = \boldsymbol{n}_j' \boldsymbol{l}_i$.

Now we look at a simple case with $k = 2$ and $b = 4$. We then have a 4×4 grid, so $\boldsymbol{A}$ is a 16×16 matrix of complex exponentials derived from the Fourier transform equation IV-3:

$$f_{n_1,n_2} = \sum_{l_1=0}^{3} \sum_{l_2=0}^{3} e^{i\frac{2\pi}{4}n_1 l_1} e^{i\frac{2\pi}{4}n_2 l_2} \cdot y_{l_1,l_2} = \sum_{l_1=0}^{3} \sum_{l_2=0}^{3} e^{i\frac{2\pi}{4}[n_1 l_1 + n_2 l_2]} y_{l_1,l_2}$$

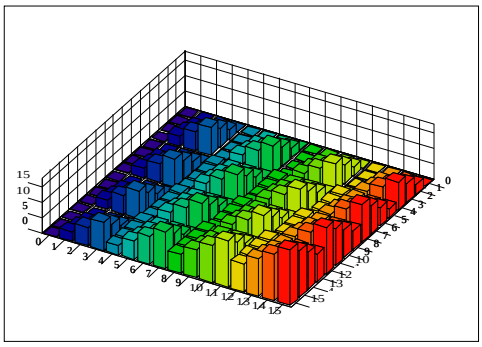
This results in the 16×16 M-matrix illustrated in Table IV-1. Note the zero entries indicated in bold face type. For the zero-zero combination of frequencies, the exponent is zero for all data points. Similarly the first column corresponds to zeroes in both of the summation indices, so they are also zeroes. The first 4 rows of the matrix are detailed in Table IV-2 showing the row and column (i, j) values of $\boldsymbol{n}$ and $\boldsymbol{l}$ along with the indexing function c applied to $\boldsymbol{l}$ and the inner product of the two vectors. In the first four rows of the matrix we see the first frequency component holding at

i	j	n1	n2	l1	l2	c(l1,l2)	l'.n	i	j	n1	n2	l1	l2	c(l1,l2)	l'.n
1	1	0	0	0	0	0	0	3	1	0	2	0	0	0	0
1	2	0	0	0	1	1	0	3	2	0	2	0	1	1	2
1	3	0	0	0	2	2	0	3	3	0	2	0	2	2	4
1	4	0	0	0	3	3	0	3	4	0	2	0	3	3	6
1	5	0	0	1	0	4	0	3	5	0	2	1	0	4	0
1	6	0	0	1	1	5	0	3	6	0	2	1	1	5	2
1	7	0	0	1	2	6	0	3	7	0	2	1	2	6	4
1	8	0	0	1	3	7	0	3	8	0	2	1	3	7	6
1	9	0	0	2	0	8	0	3	9	0	2	2	0	8	0
1	10	0	0	2	1	9	0	3	10	0	2	2	1	9	2
1	11	0	0	2	2	10	0	3	11	0	2	2	2	10	4
1	12	0	0	2	3	11	0	3	12	0	2	2	3	11	6
1	13	0	0	3	0	12	0	3	13	0	2	3	0	12	0
1	14	0	0	3	1	13	0	3	14	0	2	3	1	13	2
1	15	0	0	3	2	14	0	3	15	0	2	3	2	14	4
1	16	0	0	3	3	15	0	3	16	0	2	3	3	15	6
2	1	0	1	0	0	0	0	4	1	0	3	0	0	0	0
2	2	0	1	0	1	1	1	4	2	0	3	0	1	1	3
2	3	0	1	0	2	2	2	4	3	0	3	0	2	2	6
2	4	0	1	0	3	3	3	4	4	0	3	0	3	3	9
2	5	0	1	1	0	4	0	4	5	0	3	1	0	4	0
2	6	0	1	1	1	5	1	4	6	0	3	1	1	5	3
2	7	0	1	1	2	6	2	4	7	0	3	1	2	6	6
2	8	0	1	1	3	7	3	4	8	0	3	1	3	7	9
2	9	0	1	2	0	8	0	4	9	0	3	2	0	8	0
2	10	0	1	2	1	9	1	4	10	0	3	2	1	9	3
2	11	0	1	2	2	10	2	4	11	0	3	2	2	10	6
2	12	0	1	2	3	11	3	4	12	0	3	2	3	11	9
2	13	0	1	3	0	12	0	4	13	0	3	3	0	12	0
2	14	0	1	3	1	13	1	4	14	0	3	3	1	13	3
2	15	0	1	3	2	14	2	4	15	0	3	3	2	14	6
2	16	0	1	3	3	15	3	4	16	0	3	3	3	15	9

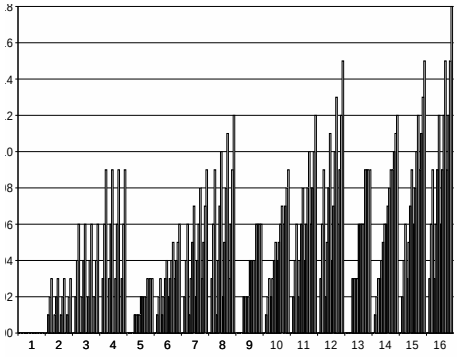
TABLE IV-2. First 4 Rows of M

zero and the second sequencing through the values $\{1, 2, 3\}$. This gives us the relatively simple periodic behavior easily seen in each of Table IV-1and Figure IV-2. As more frequency content comes into play with higher row numbers, the complexity of the rows becomes greater.

The increasing complexity of the periodic structure as row number increases is illustrated in Figure IV-2. This increasing complexity is the strength that this transform has over a single dimensional transform, allowing it to illuminate more detailed and complex behavior in the data.



(a) 3D Bar Chart



(b) Side-by-side Bar Chart

FIGURE IV-2. Periodic Structure of $\boldsymbol{A}$

4.2. Distribution of the Fourier Coefficients

We now have a simple form for the Fourier transform of the empirical PDF (4.1.2) and we are ready to derive the distribution of the transform coefficients from the relative cell frequencies. Empirical results from examining a number of suspected good generators have suggested that they are independent normals with variance $\frac{1}{N}$.

4.2.1. Asymptotic Distribution of the Empirical PDF. First we need to get the asymptotic distribution of the relative cell frequencies. This is an application of well known results for asymptotic normality of cell frequencies. The following lemma however, makes it specific to our context.

LEMMA 4.2.1. *The empirical PDF with resolution* $\frac{1}{b} \times k$ *of an independent uniform pseudo random number generator is*

$$\boldsymbol{y} \sim AN_T(\mathbf{1}\, p, \frac{1}{N}\left(p\boldsymbol{I} - p^2\boldsymbol{J}\right))$$

PROOF. Under the assumption of independent uniformity of the generator, the k-dimensional structure $\boldsymbol{X} = \{g(\boldsymbol{t})\}$ of cell frequencies, $g(\boldsymbol{t}) = Nf_N(\boldsymbol{t})$, are multinomially distributed, with individual cells binomially distributed, $p = \frac{1}{T}$ and $q = (1 - \frac{1}{T})$. The mean of the cell frequencies is $Np = \frac{N}{T}$, while the variance of each cell is $Npq = N\left(\frac{1}{T} - \frac{1}{T^2}\right)$ and the covariance between any two cells is $-Np^2 = \frac{N}{T^2}$.

This k-dimensional structure is difficult to manipulate, so we will 'roll it out' into a T element vector $\boldsymbol{x}$ using the cell indices function (IV-1).

This now gives us a vector $\boldsymbol{x}$ which is multinomially distributed and, hence, by the multivariate central limit theorem has an asymptotically multivariate distribution. More precisely, we know that $\boldsymbol{y} = \frac{1}{N}\boldsymbol{x}$ is asymptotically normal,

$$\boldsymbol{y} \sim AN_T(\mathbf{1}\, p, \frac{1}{N}\boldsymbol{\Sigma})$$

with $\boldsymbol{\Sigma} = \{\sigma_{ij}\}$ and

$$\sigma_{ij} = \begin{cases} pq & i = j \\ -p^2 & i \neq j \end{cases} .$$

Note that $\mathbf{\Sigma}$ has rank $T - 1$ due to the constraints on the cell probabilities $\sum_i p_i = 1$.

A well known shorthand form for this covariance matrix using the identity matrix and $\boldsymbol{J} = \mathbf{1}\,\mathbf{1}'$ is

$$\text{(IV-1)} \qquad \mathbf{\Sigma} = \begin{pmatrix} pq & -p^2 & \dots & -p^2 \\ -p^2 & pq & -p^2 & \vdots \\ \vdots & -p^2 & \ddots & -p^2 \\ -p^2 & \dots & -p^2 & pq \end{pmatrix} = p\boldsymbol{I} - p^2\boldsymbol{J}$$

giving us our desired result. □

4.2.2. Preliminaries. We can now turn to obtaining the distribution of the Fourier transform of the empirical PDF using theory for the distribution of linear transformations and complex normal distributions. A couple of preliminary items are key to this.

The first item is the definition of a complex normal distribution, which provides us with a relationship between multivariate real normals and complex normals as is defined and used for multivariate time series analysis; see Brillinger[**10**] or Brockwell and Davis[**11**] or covered in somewhat greater detail by Goodman [**32**]. Detail of the derivation of the PDF and characteristic function for this distribution is available in Wooding [**105**].

DEFINITION. Complex Multivariate Normal Distribution. Let $\boldsymbol{Y}$ be a complex random vector of length m. We say $\boldsymbol{Y}$ has a complex multivariate normal distribution, denoted by

$$\boldsymbol{Y} \sim N_m^c(\boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

if the folowing conditions hold.

(1) The $m \times m$ matrix $\boldsymbol{\Sigma}$ is a Hermitian covariance matrix with real and imaginary portions $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}_2$ respectively. That is, $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}_1 + i\boldsymbol{\Sigma}_2$ is a complex valued $m \times m$ matrix such that $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^* = \boldsymbol{\Sigma}_1' - i\boldsymbol{\Sigma}_2'$ and $\boldsymbol{a}^*\boldsymbol{\Sigma}\,\boldsymbol{a} \geq 0$ for all $\boldsymbol{a} \in \boldsymbol{C}^m$.

(2) Writing $\boldsymbol{Y}$ into its real and imaginary parts as $\boldsymbol{Y} = \boldsymbol{Y}_1 + i\boldsymbol{Y}_2$, let $\mu_1 = E(\boldsymbol{Y}_1)$ and $\mu_2 = E(\boldsymbol{Y}_2)$. Define $\mu = \mu_1 + i\mu_2$.

(3) Assume that $\begin{pmatrix} \boldsymbol{Y}_1 \\ \boldsymbol{Y}_2 \end{pmatrix}$ has the multivariate normal distribution

$$\begin{bmatrix} \boldsymbol{Y}_1 \\ \boldsymbol{Y}_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \frac{1}{2} \begin{bmatrix} \boldsymbol{\Sigma}_1 & -\boldsymbol{\Sigma}_2 \\ \boldsymbol{\Sigma}_2 & \boldsymbol{\Sigma}_1 \end{bmatrix} \right).$$

Then we say that $\boldsymbol{Y} = \boldsymbol{Y}_1 + i\boldsymbol{Y}_2$ is a complex-valued multivariate normal random vector with mean $\boldsymbol{\mu} = \boldsymbol{\mu}_1 + i\boldsymbol{\mu}_2$ and covariance matrix $\boldsymbol{\Sigma}$.

This definition is isomorphic in the special case of Fourier coefficients. See Wooding [**105**] for details.

The second item concerns the properties of sums of the complex exponentials we are working with.

FACT 4.2.2. *From the period $\frac{2\pi}{N}$of the complex exponential $e^{i\phi}$ and the symmetry of the summation with integer powers,*

$$\sum_{n=0}^{N-1} e^{i\frac{2\pi}{N}nr} = \begin{cases} N & r = mN, m \in integers \\ 0 & else \end{cases}.$$

This relationship can be found in many sources, including Oppenheim & Sheiffer[**81**]and Coveyou and Macpherson[**19**].

The final item is a fact from Wooding[**105**] which gives us the distribution for a linear combination of complex random vectors.

FACT 4.2.3. *If $\boldsymbol{z}$ is $N_m^c(\theta, \Sigma)$ & C is a matrix, then*

$$C\boldsymbol{z} \sim N_m^c(C\theta, C\Sigma C^*).$$

This is a well known result which follows directly from the characteristic function of the complex normal developed in Wooding [**105**]for the special case of the distribution of Fourier coefficients, which is given by,

$$\phi(t) = E\left[e^{iRe(\boldsymbol{t}^*\boldsymbol{z})}\right] = e^{iRe(\boldsymbol{t}^*\boldsymbol{z})-\frac{1}{4}\boldsymbol{t}^*\Sigma\boldsymbol{t}}.$$

4.2.3. Distribution of the Transform. We can now get the distribution of the transform.

THEOREM 4.2.4. *The Fourier coefficients $\boldsymbol{f}$ are distributed as*

$$AN_T^c(\boldsymbol{\delta}_0, \frac{1}{N}(\boldsymbol{I} - \boldsymbol{\delta}_0\boldsymbol{\delta}_0'))$$

under the assumption of an underlying independent uniform generator.

PROOF. From Lemma IV-2 we know that $\boldsymbol{y} \sim AN_T(p\mathbf{1}\,, \frac{1}{N}\boldsymbol{\Sigma}\,)$. We also know that the Fourier transform is a continuous and differentiable function. We will use the well known result that if $X_n \to x$ in distribution and G is continuous, then $G(X_n) \to G(x)$ in distribution. In our case, G is the linear transformation $\boldsymbol{A}$ and hence is continuous. But $\boldsymbol{A}$ is complex so the limit distribution will likely be complex. We first note that $\boldsymbol{v} = \boldsymbol{y} + i\mathbf{0}$, which has mean $\boldsymbol{\mu} + i\mathbf{0}$ and variance $\boldsymbol{\Sigma} + i\mathbf{0}$, which by Definition 4.2.2 has distribution $AN_T^c\left(p\,(\mathbf{1} + i\mathbf{0}\,)\,, \frac{1}{N}\,(\boldsymbol{\Sigma} + i\mathbf{0}\,)\right)$. Then by Fact 4.2.3 we have that $\boldsymbol{f} = \boldsymbol{A}\,\boldsymbol{v}$ is distributed

$$AN_T^c(p\boldsymbol{A}\ (\mathbf{1} + i\mathbf{0}\,)\,, \frac{1}{N}\boldsymbol{A}\ (\boldsymbol{\Sigma} + i\mathbf{0}\,)\,\boldsymbol{A}^{\,*}) = AN_T^c(\boldsymbol{A}\,\mathbf{1}\,p, \frac{1}{N}\boldsymbol{A}\,\boldsymbol{\Sigma}\,\boldsymbol{A}^{\,*})$$

Expanding and rearranging the mean we can use fact 4.2.2 to simplify it as

$$p\boldsymbol{A}\,\mathbf{1} = p\sum_{\boldsymbol{l}} e^{i\frac{2\pi}{b}\left(\boldsymbol{l}\,'\boldsymbol{n}\,\right)} y_{\boldsymbol{l}} = p\sum_{l_1=o}^{b-1}\cdots\left[\sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b}n_k l_k}\right]\cdots e^{i\frac{2\pi}{b}n_1 l_1}.$$

Note that

$$\left[\sum_{l_k=0}^{b-1} e^{i\frac{2\pi}{b}n_k l_k}\right] = \begin{cases} b & n_k = mb, m \in integers \\ 0 & else \end{cases}$$

and $n_k = mb$ for $n_1 = 0$ only.

Now consider the r^{th} element of $\boldsymbol{A}\,\mathbf{1}$, which is

$$\begin{aligned}\{\boldsymbol{A}\,\mathbf{1}\,\}_r &= \boldsymbol{a}\,'_{\boldsymbol{n}\,r}\mathbf{1} = \sum_{\boldsymbol{l}} e^{i\frac{2\pi}{b}\boldsymbol{l}\,'\boldsymbol{n}\,r} \\ &= \left[\sum_{l_1}^{b-1} e^{i\frac{2\pi}{b}l_1\{\boldsymbol{n}\,r\}_1}\right]\cdots\left[\sum_{l_k}^{b-1} e^{i\frac{2\pi}{b}l_k\{\boldsymbol{n}\,r\}_k}\right] \\ &= \left[\begin{cases} b & \{\boldsymbol{n}\,r\}_1 = 0 \\ 0 & else \end{cases}\right]\cdots\left[\begin{cases} b & \{\boldsymbol{n}\,r\}_k = 0 \\ 0 & else \end{cases}\right]\end{aligned}$$

For the first row, $r = 1$, all of the $n_i = 0$; hence, we have the product of k summations, each of which has value b giving us b^k. For $r \neq 1$, we have

at least one $n_i \neq 0$ in each factor and $n_i < b$. By Fact 4.2.2 this factor is 0 and, hence, the product is 0.

Thus the mean is a delta vector ($\boldsymbol{\delta}_0^T$) with the one in the first (zero) position. To simplify notation, we will drop the superscript indicating the length of the delta, zero $\mathbf{0}$ and $\mathbf{1}$'s vectors when their length is clear from context.

Turning to the variance,

$\boldsymbol{A}\,\boldsymbol{\Sigma}\,\boldsymbol{A}^* = \boldsymbol{A}\,(p\boldsymbol{I} - p^2\boldsymbol{J}\,)\boldsymbol{A}^* = p\boldsymbol{A}\,\boldsymbol{A}^* - p^2\boldsymbol{A}\,\boldsymbol{J}\,\boldsymbol{A}^*$

we look first at the elements of $\boldsymbol{A}\,\boldsymbol{A}^*$.

$\{\boldsymbol{A}\,\boldsymbol{A}^*\}_{rs} = \boldsymbol{a}_r^*\boldsymbol{a}_s = \sum_{\boldsymbol{l}} e^{i\frac{2\pi}{b}\boldsymbol{l}'(\boldsymbol{n}_s - \boldsymbol{n}_r)}$

but,

$\{(\boldsymbol{n}_s - \boldsymbol{n}_r)\}_i \in -(b-1), \ldots, 0, \ldots, (b-1)$,

so this is non-zero only for $r = s$.

thus we will get b^k at $r = s$ and zero otherwise, finally giving us

$p\boldsymbol{A}\,\boldsymbol{A}^* = pb^k\boldsymbol{I} = \frac{b^k}{T}\boldsymbol{I}$.

Going to the second term,

$p^2\boldsymbol{A}\,\boldsymbol{J}\,\boldsymbol{A}^* = p^2\,(\boldsymbol{A}\,\mathbf{1}\,)\,(\mathbf{1}'\boldsymbol{A}^*)$

We already know that $\boldsymbol{A}\,\mathbf{1} = \begin{pmatrix} b^k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$ and

$\mathbf{1}'\boldsymbol{A}^* = \left(\bar{\boldsymbol{A}}\mathbf{1}\,\right)' = \begin{pmatrix} b^k & 0 & \cdots & 0 \end{pmatrix}$ by the same argument, but with $-i$ instead of i, since we are just going the opposite direction around

the unit circle. Thus

$$p^2 \boldsymbol{A}\,\boldsymbol{J}\,\boldsymbol{A}^{*} = p^2 \begin{pmatrix} b^k \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} b^k & 0 & \cdots & 0 \end{pmatrix} = \begin{pmatrix} p^2 b^{2k} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{pmatrix}$$

Putting these together we get $\boldsymbol{A}\,\boldsymbol{\Sigma}\,\boldsymbol{A}' = pb^k \boldsymbol{I} - p^2 b^{2k} \boldsymbol{\delta}_0 \boldsymbol{\delta}_0'$

Note now that $b^k = T$ so $pb^k = \frac{1}{T} \cdot T = 1$ and $p^2 b^{2k} = \left(pb^k\right)^2 = 1$

So finally, $\boldsymbol{A}\,\boldsymbol{\Sigma}\,\boldsymbol{A}' = \begin{cases} 0 & i = j = 0 \\ 1 & i = j > 0 \\ 0 & else \end{cases} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 1 \end{pmatrix}$

and we have our variance $\frac{1}{N}\left(\boldsymbol{A}\,\boldsymbol{\Sigma}\,\boldsymbol{A}'\right) = \frac{1}{N}\left(\boldsymbol{I} - \boldsymbol{\delta}_0 \boldsymbol{\delta}_0'\right)$ □

From this we have that the mean is zero everywhere other than the $\mathbf{0}$ frequency element which also has degenerate variance. The remaining elements are independent normals with variance $\frac{1}{N}$.

4.2.4. Distribution of the Re & Im Coefficients. The derivation of section 4.2.3 is for complex coefficients of the form $a + bi$. This is sufficient for analysis of functions of the complex numbers which have real values, such as the magnitude. To get better potential resolution in our test statistics, we would like to be able to work with functions of the real and imaginary components separately. However we do not yet have the distribution of the individual real and imaginary coefficients generated by the FFT algorithms.

THEOREM 4.2.5. *If we expand our distribution as indicated by definition 4.2.2 into a $2T$ normal*

$$\boldsymbol{f} = Re\boldsymbol{f} + iIm\boldsymbol{f}$$

$$\boldsymbol{S} = \frac{1}{N}(\boldsymbol{I} - \boldsymbol{\delta}_0\boldsymbol{\delta}_0') = Re\boldsymbol{S} + iIm\boldsymbol{S}$$

where we have $Im\boldsymbol{S} = \mathbf{0}$, then

$$\text{(IV-2)} \qquad \boldsymbol{z} = \begin{pmatrix} Re\boldsymbol{f} \\ Im\boldsymbol{f} \end{pmatrix} \sim AN_{2T}\left(\boldsymbol{\delta}_0, \frac{1}{2}\begin{pmatrix} \boldsymbol{S} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{S} \end{pmatrix}\right)$$

.

PROOF. Theorem 4.2.3 gives us the asymptotic distribution of $\boldsymbol{f}$ as a complex normal:

$$\text{(IV-3)} \qquad AN_T^c(\boldsymbol{\delta}_0, \frac{1}{N}(\boldsymbol{I} - \boldsymbol{\delta}_0\boldsymbol{\delta}_0'))$$

From definition 4.2.2 and the fact that our mean and variance are strictly real, the result is immediate. □

4.3. Overall Test

We now have the distribution of the Fourier coefficients either as at T element complex vector ($\boldsymbol{f}$) or as a $2T$ real vector ($\boldsymbol{z}$). The obvious test given the asymptotic normality is to use a quadratic form with a Chi-squared distribution. This form would normally be $\boldsymbol{z}'\boldsymbol{\Sigma}^{-1}\boldsymbol{z}$, however in our case the covariance matrix is not full rank and therefore not invertible. This could be approached using a generalized inverse of $\boldsymbol{\Sigma}$ [1], but a very simple

[1] As was suggested by Dr. Daniel Mihalko.

projection seems clearer in our case and makes the actual implementation in software more obvious. We then will be separating the coefficient vector into two portions or subspaces with a simple projection. One the primary subspace will be full rank $(2T-2)$ and the other (our residual) will be a two dimensional space with an asymptotically vanishing variance. The result can then be combined in a normal fashion to obtain a test statistic which includes the full measurement vector and potentially better power under alternatives.

THEOREM 4.3.1. *As in Theorem 4.2.5, let* $\boldsymbol{z} = \begin{pmatrix} Re\boldsymbol{f} \\ Im\boldsymbol{f} \end{pmatrix}$ *where* $\boldsymbol{f} = \boldsymbol{A}\,\boldsymbol{y}$ *. Suppose* $X = 2N\sum_{l=0}^{2T-1} z_l^2 - 1$*. Then* X *has an asymptotic* χ^2 *distribution with* $2T-2$ *degrees of freedom, under the null hypothesis of a uniform independent generator. We write* $X \sim A\chi^2_{2T-2}$*.*

PROOF. Recall from Theorem 4.2.5 that $\boldsymbol{z}$ is

$$AN_{2T}\left(\boldsymbol{\delta}_{\,0}, \frac{1}{2}\begin{bmatrix} \boldsymbol{S} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{S} \end{bmatrix}\right)$$

.

Let $\boldsymbol{g} = \boldsymbol{P}\,\boldsymbol{z}$ where the $2(T-1)\times 2T$ matrix

$$\boldsymbol{P} = \left[\begin{array}{c|c|c|c} \boldsymbol{0}_{\,\frac{r}{2}} & \boldsymbol{I}_{\,\frac{r}{2}x\frac{r}{2}} & \boldsymbol{0}_{\,\frac{r}{2}} & \boldsymbol{0}_{\,\frac{r}{2}x\frac{r}{2}} \\ \boldsymbol{0}_{\,\frac{r}{2}} & \boldsymbol{0}_{\,\frac{r}{2}x\frac{r}{2}} & \boldsymbol{0}_{\,\frac{r}{2}} & \boldsymbol{I}_{\,\frac{r}{2}x\frac{r}{2}} \end{array}\right], r = 2(T-1).$$

This is a simple projection from $\Re^{2T}$ to $\Re^{2(T-1)}$ which strips off the Real and Imaginary coefficients of the $\boldsymbol{0}$ frequency components leaving us with $\boldsymbol{g} \sim AN_{2(T-1)}\left(\boldsymbol{0}\,, \frac{1}{2N}\boldsymbol{I}\,\right)$. The residual of this projection $\boldsymbol{h}$ is the stripped

off components which is $AN_2\left(\boldsymbol{\delta}_{\,0}, \mathbf{0}\right)$, i.e. a degenerate two dimensional vector with $\boldsymbol{h} = \begin{pmatrix} z_0 \\ z_T \end{pmatrix}$.

We now have

$$X_1 = \boldsymbol{g}\,' \left[\frac{1}{2N}\boldsymbol{I}\right]^{-1} \boldsymbol{g} \sim A\chi^2_{2T-2}(0)$$

so the sum

$$X = X_1 + 2N\left(\boldsymbol{h} - \boldsymbol{\delta}_{\,0}\right)^2$$

is also distributed as an $A\chi^2_{2T-2}$ and

$$X = 2N \sum_{l=0}^{2T-1} z_l^2 - 1$$

. □

Our test is then to reject the null hypothesis of a uniform independent generator when X is too great in magnitude, indicating excessive power at some set of frequencies. More formally, our asymptotically level α test of H_0: The generator is uniform ($p_i = \frac{1}{T} \forall i$), is

$$\text{Reject } H_0 \text{ if } X > \chi^2_{\alpha,2T-2}$$

where $\chi^2_{\alpha\,2T-2}$ is the upper α critical point of the χ^2 distribution with $2T-2$ degrees of freedom.

4.4. Testing for Individual Outliers in the Coefficients

Since we expect to find particular unknown combinations of frequencies which match with periodic behavior in the empirical PDF, we can look for these as outliers in the coefficients. Graphics are an obvious method, but highly subjective. We will scan the coefficients for outliers using a Fisher's protected LSD (PLSD) scheme. If the overall test above rejects, we will use

an α level confidence interval, otherwise we will use a Bonferonni interval, assuring that we will retain at least an overall level α test.

4.4.1. Cutoff Based on Individual Real and Imaginary Components. Since our reduced vector $\boldsymbol{g}$ is $AN_{2T-2}(\mathbf{0}, \frac{1}{2N}\mathbf{I})$, we can base our initial test for individual outliers in all but the zero frequency coefficients on the distribution of $\sqrt{2N}g_j$, which is the standard normal. We calculate the critical value K_1 for which any observation with absolute value greater than K_1 implies a likely outlier such that $1 - \Phi(K_1) < \frac{\alpha}{2}$, giving us a probability of seeing an observation with absolute value > $K_1 \leq \alpha$. If the overall test did not reject, we can use $L_1 : 1 - \Phi(L_1) < \frac{\alpha}{4(T-1)}$ for the Bonferonni interval.

4.4.2. Cutoff Based on the Magnitude of Complex Coefficient Magnitudes. The above interval uses the inverse of a hypercube centered at the origin as the critical region in $2(T-1)$ space, applied to $2(T-1)$ elements. If we consider each Real/Imaginary pair, this is equivalent to testing with the inverse of a square region about the origin as our critical region for each pair. However, a circular cutoff region using the magnitude of complex coefficients may actually make more sense, combining both the real and imaginary components at a particular frequency combination. The boundary of the resulting region is then a uniform distance from the origin with respect to each complex pair. This also reduces the number of comparisons to T-1. So alternatively, we can use a circular critical region $|f_j| > C \; \forall j$, using the real and imaginary parts of the complex number simultaneously. For simplicity, we will work with $|f_j|^2 = (Re(f_j)^2 + Im(f_j)^2)$. Standardizing this gives us $\left|\sqrt{2N}f_j\right|^2 \sim AX_2^2$.

We then apply the standard one sided test. That is we declare that the j-th coefficient is an outlier if $|f_j|^2 > \frac{K_2}{2N}$, where K_2 is the upper α X^2 critical point with 2 degrees of freedom. The Bonferonni cutoff L_2 of this, to be used if the overall test did not reject, simply replaces α with $\frac{\alpha}{T-1}$ making L_2 the $1 - \frac{\alpha}{T-1}$ quantile of the X_2^2 distribution.

NOTE: Be careful not to apply both Bonferonni procedures if the overall test fails to reject. If the overall test has rejected H_0 we should be safe applying both forms of the confidence intervals since we are protected by the overall rejection. If we use both Bonferonni tests as specified, we are no longer assured of an overall at most α testing scheme. A particular test must be selected a-priori, or the levels must be adjusted accordingly. In this case it would be necessary to use $\frac{\alpha}{5(T-1)}$ and $\frac{\alpha}{3(T-1)}$ respectively to cover the larger number of comparisons being performed.

4.4.3. Testing the Zero Frequency Coefficients. The two zero frequency coefficients are asymptotically degenerate, but not actually zero for any finite N. We should still be able to test them however, if we knew the convergence rate of the variance. These tests would be similar to those above, but with variance structure depending not only directly on N, but also on the convergence rate of the distribution under the null hypothesis. This is left to future work.

CHAPTER V

APPLYING THE EST

In this chapter, possible interpretations for various parameter choices for the EST will be presented and suggestions for the application of the EST will be given. The EST can be used to test any RNGs or PRNGs, as it is applied to a random sequence or pseudo random sequence sampled from the generator. The EST can detect numerous types of structure in the sequence being tested, including many types of:

- both periodic and non-periodic structures
- both local and global multidimensional spatial structures
- both short term and long term serial correlation

The types of structure that the EST can detect are all ones that would prove unacceptable in some common situations. We don't want to overstate the capabilities of the EST though. While it can detect an extremely broad class of structures, like every other test, it cannot possibly detect all possible structure in the data.

5.1. Interpreting EST Results

If the reader hasn't been through Section 2.7.6 and particularly Section 2.7.6.3 yet, this would be a good time. These sections discuss the difficulties of testing RNGs and PRNGs and suggest some guidelines for interpretation.

For these reasons, the best general rule is to use the EST to compare generators, rather than simply test them in isolation. Using generators that are thought to be "good" to help interpret the p values of the test.

It is also true that it is best to repeat any test of any RNG or a PRNG. For repeating tests on a PRNG you should select new seeds both randomly and systematically.

The normal construction of a test based on a chi-squared distribution (which is the basis for the EST overall test) uses a right tailed rejection region only. This is done based on the asymptotic power of the test against unspecified alternatives (See Stuart & Ord[**93**] section 30.6). However the arguments this is based on are for a quite different experimental and testing environment than that of the EST. It can easily be argued that X too small corresponds as readily to a potential problem in the generator as does X too large. Hence it would be prudent to at least flag these cases or even modify the current EST test to reject on both left and right tails. Actual modification of the test is unnecessary since it outputs a p value which can be tested manually or automatically external to the core test software (as is done later in chapter 6).

5.2. Values Returned by the EST

The EST returns:

- EST statistics: These are the primary results of the test.
  - N: the total number of data points generated in k - dimensions
  - T: b^k
  - a: the value of α used for critical values and confidence limits

  - X: the EST test statistic
  - p: the p-value associated with X
  - C: the test critical value ($X > C$ implies reject H_0)
  - K1: the symmetric $1-\alpha$ confidence limit for Re and Im values
  - K2: the Bonferonni $1-\alpha$ confidence limit for Re and Im values
  - L1: the $1-\alpha$ confidence limit for Rho
  - L2: the $1-\alpha$ Bonferonni confidence limit for Rho

The values N, T and a are simply returned for convenience. The main item to look at first is the p-value. If this is below the predetermined level α, the test has found significant structure. The $K_1 \& L_1$ confidence limits are provided primarily as a visual aide in gauging the scatter in the coefficients. With the large number of coefficients, there will almost certainly always be values outside these relatively tight limits. If the rejection is strong (very low p), there should be points outside the Bonferroni limits $K_2 \& L_2$. This is the starting point for analyzing the nature of the potential failure. The Bonferroni limits are also set at a level such that even without a low p-value, coefficients which lie outside these limits are cause for further study.

- EST FFT coefficients:
  - [Re] a b^k vector of real coefficients
  - [Im] a b^k vector of imaginary coefficients
  - [Rho] a b^k vector of magnitudes of the complex coefficients
  - [Phi] a b^k vector of phase angles of the complex coefficients

The FFT coefficients are provided to facilitate and analysis process. This test is not meant to provide only an accept/reject decision, but rather as a

tool for exploring random sequences for potentially troublesome structure. The search for that structure begins with this data.

5.3. The $< b, k, N, seed >$ Family of ESTs

The EST is a family of tests, parameterized by $< b, k, N, seed >$. The parameters b, k and N are integers and seed is the seed for the generator begin tested, assuming one is needed.

The first parameter is the number of 'cells' allocated in each dimension. It effectively controls the resolution of your search for structure. For example, a value of $b = 4$ would split the unit interval into 4 equal segments within which the cell counts are gathered, allowing structures spanning more than about 0.25 units in any direction to be detected. Increasing b thus improves the EST resolution. The second parameter, k, is the dimensionality of the test. This controls the number of dimensions within which to search the data for structure. For example, planes embedded in higher dimensional space cannot be detected with k<3. This is why the RANDU structure doesn't show up strongly until the test for 3 dimensions and higher is used. N controls the total number of data points to be generated in k dimensions for the test. And finally seed controls the selection of the subsequence from the generator to be examined.

Note that together b & k have the main effect on the memory and execution time requirements of the test which increases porportional to b^k and $b^k \log_2 b^k$ respectively.

5.3.1. b as a Power of 2. For the current implementation we have used the Numerical Recipes[**86**] multidimensional FFT routine 'fourn'. This

routine uses a radix-2 decimation-in-time algorithm which is limited to sequences that are a power of 2. Hence in our applications the reader will see the value of b, which is the length of the fundamental unit upon which the FFT is applied, is always a power of 2. This limitation can be overcome by using software such as FFTW[**28**] or by padding the sequence to a power of 2 length. Otherwise, it's best to use b equal to a power of 2 for efficiency.

5.3.2. The Size of $\frac{N}{b^k}$. This ratio is the expected cell count under H_0. Since our asymptotics depend on the same convergence to multivariate normal as that used in a standard chi-squared test, we will use the same rule of thumb for minimum cell counts. This says that the minimum expect cell count should be 5 or more. Much below this and the asymptotics are inadequate, causing potentially false rejections. This is due to the normal approximation being poor for expected cell counts less than 5. Experience with running the EST and examining the results indicates that the greater the expected cell count, the better the distribution tends to normality for good generators. In practice, it would be best to keep this value at least above 5.

5.3.3. Relationship Between $N,\ k$ and the Number of $u_i's$. When data is gathered for the EST, we currently use the allocation scheme in Section 2.4.5.4. This means that individual $u_i's$ from the generator are not 'reused'. So to get a N k-dimensional vectors we actually need a sequence $u_1, u_2, \ldots, u_{kN}$ of length kN .

5.3.4. Fix kN, Then Increase N & b and k. This scenario is essentially what we are working with in chapter 6. In this case we preselected

a block of data from the target PRNG and then applied varied N & b depending on k to 'use up' the sequence. It is looking for structure within a fixed length output of the generator. Note that we also ran the tests on successive blocks from the generators. This is effectively making repeated runs against a fixed pattern of seeds. Had time permitted, additional runs with randomized seeds would have been appropriate.

5.3.5. Fix b and k and Increase N. This is testing for patterns that appear in various length runs or that become stronger with increasing quantities of data. An example of the later can be found in chapter 6 for the Marsaglia Multicarry generator in R.

5.3.6. Fix b and N and increase k. This test sequence would be searching for patterns in various dimensions and short term correlations of length k. Note that you will need to start with a large N since as k increases $\frac{N}{b^k} \to 0$ and the asymptotics will break down. This is also a fixed resolution test sequence, since we have constant b .

5.3.7. Fix k and N and Increase b. This increases the resolution of the test, making it able to detect smaller structure with increasing b. Note that the computational load goes up with $b^k \log_2 b^k$, even with moderate k, the test will rapidly increase in execution time and memory requirements. The initial N will also need to be relatively large or the asymptotics will break down quickly as noted above.

5.3.8. Fix b and Increase N and k. This sequence would be searching for patterns with a set 'resolution'.

5.3.9. Fix k and Increase b and N. This would be searching for k term correlations and special patterns at increasing resolutions. Since we are varying both b & N, we can keep the cell counts reasonable and prevent the asymptotics from breaking down as the test sequence progresses.

5.4. Repeating With a New Seed

There will always be cases when structure will show up for a some particular subsequences from even the best generator, especially if N is relatively small. In fact if this doesn't happen, we should suspect the generator. The converse is also true, a bad generator can easily have good looking subsequences. It is worth repeating. Run the EST with multiple seeds.

5.5. Comparing With 'Good' Generators.

Having several generators that are considered "good" is very useful when evaluating a new RNG or PRNG. Multiple runs of the "good" generators (for fixed b, k, and N) can provide a benchmark indication of the power of the test and what a rejection of a new generator at a specific p value represents.

5.6. Combining Test Results & Selection of α Level

Throughout this chapter we have been discussing running multiple tests and comparing results. The EST decision criteria is designed to provide an asymptotic level α test for a particular parameterization. Many of the problems indicated by 'p happens' come from not recognizing this. If we are running an exploratory analysis, there is not much that can be done except

careful interpretation of the results. However if we are running a fixed sequence of tests, we can adjust our alpha using a Bonferroni approach to get an asymptotic level α overall. This is assuming our tests are independent. That is not true for many of the scenarios above even under the H_0 . So regardless, we need to keep our wits about as we interpret these tests.

Since the purpose of our testing is to illuminate potentially problematic behavior in the random number generator, we may want to consider our choices for confidence level. Confidence levels are generally set at 90% or higher, reflecting the perceived high cost of stating the existence of structure when none existed (rejection of the null hypothesis when it is actually true). This is predicated on an experimental context which is not necessarily consistent with our goals for the EST.

In this context the normal relative weighting of Type I and Type II errors reflected in the choice of α may not be appropriate. It is suggested that confidence levels for the testing not be the usual levels of 90% or greater, but rather 80-90%. This reflects the greater cost of accepting a generator with poor properties in the light of a large number of alternatives available today and the relative ease with which the testing can be performed.

CHAPTER VI

SOME COMPARITIVE RESULTS

We used SUPER, RANDU and RANLIB in chapter III to motivate the EST. We will take another look at those three generators, now actually using the EST. Finally we will run the EST against the five generators provided with the R statistical package.

NOTE: We offer these results as examples of potential application of the EST. They should not yet be taken to indicate either that a generator is good or bad. As we stated in chapter 2, no test should be declared 'bad' until the underlying cause of the rejection is understood.

It seems desirable to run tests such that the same sequence of random numbers is used, but applied with different dimensionality. We would also like to achieve the highest bin count per dimension to get the maximum resolution from the test. For these example test runs an arbitrary sequence length of 2^{16} was chosen and the values of b and E selected to use as much of the sequence as possible. Note that b is being chosen as a power of 2 to maximize the efficiency of the FFT calculations for the particular FFT implementation we are using. Table VI-1 shows the results of the calculations. In this table E is the target expected frequency for each cell and M is the total number of random variates necessary to generate the N points in k dimensional space. The column resolution indicates the approximate resolution per dimension of the test with respect to the unit hypercube and

TABLE VI-1. Test Parameter Calculations

k	E	b	T	N	M	Resolution	Volume
1	8	8192	8192	65536	65536	0.00012	0.00012
2	8	64	4096	32768	65536	0.01563	0.00024
3	5	16	4096	20480	61440	0.06250	0.00024
4	4	8	4096	16384	65536	0.12500	0.00024
5	12	4	1024	12288	61440	0.25000	0.00098

TABLE VI-2. Overall Test p Values

Generator	k=1	k=2	k=3	k=4	k=5
RANDU	0.003	0.982	0.000	0.000	0.055
Super	0.074	0.733	0.198	0.214	0.062
Ranlib	0.325	0.666	0.125	0.585	0.653

the last column indicates the volume of the unit hypercube of each cell in the test.

6.1. Another Look at RANDU, SUPER and RANLIB

As expected for RANDU the EST rejects H_0 strongly in three dimensions. However we can see from these results in Table VI-2 and the figures in Appendix A pages A-A that RANDU also fails in one dimension. In two dimensions however, the EST did not reject H_0. However, there were coefficients which exceeded the Bonferonni limit. For Super the EST rejects H_0 in 1 and 5 dimensions, but not nearly as strongly as RANDU. For RANLIB the EST did not reject H_0 in any situation.

6.2. Testing the PRNG Functions in R

Tests were also applied to the 5 generators in R. Each was tested up to 5 dimensions. Results are tabulated in Table VI-3 for the initial runs using the parameterizations of Table VI-1. For each of the 25 test scenarios, there is a page with graphics and test summary output in appendix A on pages

TABLE VI-3. R Generator p Values

Generator	k=**1**	**k=2**	**k=3**	**k=4**	k=5
Marsaglia-Multicarry	0.075	0.689	0.057	0.945	0.504
Super-Duper	0.347	0.838	0.847	0.800	0.137
Mersenne-Twister	0.172	0.246	0.912	0.863	0.761
Knuth-TAOCP	0.956	0.467	0.974	0.839	0.452
Wichmann-Hill	0.677	0.870	0.031	0.870	0.418

119-143 of Appendix B. Each figure consists of an array of six panels. The left column is the real and imaginary coefficients and the right column is the magnitudes. Each column has a histogram at the top, a Q-Q plot in the center and a plot of the coefficients (or magnitudes) with confidence limits at the bottom. The confidence limits are depicted with a dashed line for the PLSD limits and a solid line for the Bonferonni limits. In the lower left coefficients plot, both real and imaginary coefficients are plotted by frequency combination index. Real coefficients are displayed with a circle and imaginary coefficients are plotted with a diamond. Test parameterization [k, b, N] are displayed under the upper left panel. The calculated p value is under the left center panel and the value of X and the critical value C is included under the lower left panel.

These first 25 runs are suggestive, but inconclusive. If this were a standard statistical analysis, we would do one of two things. If no further experimentation is available, we would necessarily make an accept or reject decision on the basis of these results. But the preferable course of action would be to gather more data, perhaps refining our hypothesis. In this case additional data is readily available, so the tests were run again. Each was run 100 times using sequential blocks of random numbers from the generators. That is, each of the 100 tests for each condition were run with a unique set

of random numbers from the generator. The results are presented in graphics on pages 144-148 of Appendix B. Each graph has an overall histogram of $X - C$. The remaining 5 panels show the 100 p-values for each dimensionality the generator is tested for. These results appear to show that the p-values are random. A detailed analysis could be performed from this to determine if the number of rejections significantly exceeded expectations for our test, giving us a meta-analysis of the generator. This is straight forward from existing theory and can be done at a later time.

A second set of 25 test runs were made with the same data used above, but instead of 100 multiple applications of the test, the entire dataset was used. This actually takes considerably less time than the multiple applications and since our test complexity is a function of the grid structure and not N, we can relatively easily increase N at will without over taxing the available hardware. Results of these runs are shown in Table VI-4 and in the graphics on page 154 of Appendix B. This graphics simply shows the contents of the table in graphical form with a dashed line indicating the rejection level for each panel. Individual graphics equivalent to those provide for our first run are provided on pages 155-179 of Appendix B. A quick look shows somewhat similar results to the first set of runs with three exceptions. We have p-values which are zero to three decimal places for the Marsaglia Multicarry generator in 3 dimensions, the Knuth TAOCP generator in one dimension and a p-value of 1.000 for Super-Duper in 5 dimensions. Looking at the individual graphics for the 3 dimensional Marsaglia Multicarry test on page 157 we see a clear failure as spectacular as RANDU, but more specific to a couple of frequency combinations. This failure also doesn't

TABLE VI-4. R Generator p Values for Large N Runs

Generator	**k=1**	**k=2**	**k=3**	**k=4**	k=5
Marsaglia-Multicarry	0.084	0.602	0.000	0.489	0.066
Super-Duper	0.035	0.149	0.163	0.039	1.000
Mersenne-Twister	0.721	0.027	0.832	0.034	0.015
Knuth-TAOCP	0.000	0.791	0.997	0.005	0.075
Wichmann-Hill	0.585	0.023	0.602	0.417	0.947

extend into other dimensionality as with RANDU for anything above three dimensions. Clearly further study of this generator is in order. No clear conclusions can be drawn about the other two cases.

Other than for the Marsaglia Multicarry in 3 dimensions, the large N results are still interesting, but not conclusive. As large as the data set is it's still a small fraction of the full period of any of these generators. Even our 100x runs are still only using only $100 \times 2^{16} (< 2^{23})$ generates from any one generator. So one last test sequence was ran. This sequence starts out with 10x the original N and increments in steps of 10 up to 100x the original N or equal to the large N run. The results are shown in R coplots on pages 149-153 of Appendix B. Each page has five panels in the lower half of the page. These five panels correspond to the values of k. The values of $X - C$ are plotted in each panel with respect to increasing N. Since these runs use the same data set from the generator and progressively use a larger subset, each step including the previous set, we have progressive sums of multinomials.

6.3. Summary

From this analysis, we can conclude that there is evidence of problems in the Marsaglia Multicarry generator at k=3 in R and should suspect the

Super-Duper at k=5 and Knuth TAOCP at k=1 and probably the Wichmann Hill at k=2. Further study of these generators is definitely indicated. From this it would also be prudent if you must depend on solid validity of your results, to use the Mersenne Twister generator in R until further analysis can be performed on the other generators. The reader should keep in mind that the EST is a very stringent test. For many applications, generator failures even as severe as indicated in RANDU (while statistically significant) are not significant to the application.

CHAPTER VII

SUMMARY AND FUTURE WORK

7.1. Summary

We have presented the background for RNG and PRNG testing along with the theory and application of the Empirical Spectral Test (EST). The test demonstrated in chapter 6 that it has the potential to find structure both in known situations such as RANDU and where none was know prior to its application, as in the Marsaglia Multicarry. We would be remiss to attempt to state that this is finished work. There is considerable refinement possible both in the theory and in the implementation. We are just getting started with these.

The evident effectiveness of this test makes it desirable to begin more extensive application studies and to fill in better information and software support for determining the nature of structure once it's found. As with the theory, we are just getting started on applying the EST and have barely begun to develop the support tools necessary to fully utilize it.

7.2. Future Work

In this chapter we have collected some of the major ideas for future work.

7.2.1. More Generalized Application. Chapter 4 provides the basic machinery for a wide class of tests for uniformity of random spatial data

against periodic alternatives. Many real world processes exhibit periodic behavior. This suggests that application to a wider range of problems than simply pseudo random number generation may prove fruitful.

7.2.2. Statistical Refinement. It would be desirable to have a better handle on the asymptotics of this test. Here we have only used the common asymptotic results for the multinomial. However a Fourier transform under the right conditions will cause the data to tend toward normality as well. This second level normalization process may improve the convergence, thus allowing better confidence in the results with smaller sample sizes.

The power of this test under alternatives should also be investigated to better understand application to various forms of alternatives. Is this an approximately UMP test against unspecified alternatives? It is, at best, an approximate UMP test. Even the standard chi-squared test is only approximately UMP since the test statistic is only asymptotically normal. We are another transformation away from this yet. The fact that the transformation is invertible suggests that this may still be approximately UMP.

There should be potential for developing a test based on the presence of symmetry in the coefficients which could be more sensitive to certain alternatives. Examination of the graphics in appendix A indicates that there is often apparently significant symmetry in the coefficient values. This is expected behavior for an FFT on data containing certain types of structure. It should be possible to develop a test which takes advantage of this expected behavior to detect specific structures with greater sensitivity.

Since the pseudo random numbers are generated by a numerical recursion, there exists the potential for the data to be 'too uniform'. We have seen

p values over 0.95 on some of test runs. This puts the data close to the left end of the chi-squared distribution. That implies an improbably low variation from expected values and hence the combination of both that the null hypothesis is true and a likely too small variance in the coefficients. This would suggest that a different form of test for variance might be appropriate.

7.2.3. Code Improvements. The FFTW package[**28**] should be used instead of the older and less efficient numerical recipes code used here. Implementation in a cluster computing environment would also be beneficial since the memory and computational resource requirements rise rapidly with the number of dimensions being examined.

7.2.3.1. *Object Oriented Version of R Code.* This implementation was done as a straight forward coding of the test, however conversion to a more general set of object functions could be achieved fairly easily. The current mdftt.R could be modified slightly to generate a new object of type 'mdftt'. Then the tview.R code could be split up and converted into plot.mdftt and summary.mdftt functions which provide summary and plotting functions for the mdftt object.

7.2.4. Relationship to an Empirical Characteristic Function. The process of applying the FFT to the empirical probability density function is an analog to obtaining a characteristic function from the analytic PDF. Formalizing this relationship can give the EST a stronger statistical basis.

7.2.5. Cryptography. The EST can also be used to monitor the performance of the RNG over time, looking for changes in the spectral make up of

the generator. This can be more sensitive that simply testing the generator for acceptability again at each time.

The EST can also be used to monitor the performance of the RNG over time, looking for changes in the spectral signature of the generator. This can be more sensitive that simply testing the generator for acceptability again at each time and has the potential of predicting failures rather than simply detecting them after they occur.

7.2.6. Meta Testing with the EST. In chapter 6 we applied the EST in an ad hoc meta testing framework with apparent success. This work should be formalized, developing the distributional properties and defining appropriate algorithms for application. With proper distributional characteristics available, this type of test can be made considerably more sensitive and much less subjective than our current ad hoc work.

Bibliography

[1] Emile H. Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, John & Sons, Incorporated, January 1989.

[2] William Aiello, Sivaramakrishnan Rajagopalan, and Ramarathnam Venkatesan. Design of practical and provably good random number generators. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–9, 1995.

[3] Srinivas Aluru, G. M. Prabhu, and John Gustafson. A random number generator for parallel computers. *Parallel Comput.*, 18(8):839–847, 1992.

[4] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM*, 45(1):70–122, January 1998.

[5] Thomas Back, Zbigniew Michalewicz, and David B. Fogel, editors. *Evolutionary Computation 1: Basic Algorithms and Operartions, Vol. 1*. Iop Pub, January 1999.

[6] Thomas Back, Zbigniew Michalewicz, and David B. Fogel, editors. *Evolutionary Computation 2: Advanced Algorithms and Operators, Vol. 2*. Iop Pub, January 1999.

[7] David H. Bailey and Paul N. Swarztrauber. The fractional fourier transform and applications. *SIAM Review*, 33:389–404, September 1991.

[8] Eric B. Baum, Dan Boneh, and Charles Garrett. On genetic algorithms. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 230–239, July 1995.

[9] Peter Bloomfield. *Fourier Analysis of Time Series: An Introduction*. Wiley, 1976.

[10] David R. Brillinger. *Time Series Data Analysis and Theory*. Holt, Reinhart & Winston, Inc., 1975.

[11] Peter J. Brockwell and Richard A. Davis. *Time Series: Theory and Methods*. Springer-Verlag New York, Inc., second edition, 1991.

[12] Barry W. Brown and James Lovato. Ranlib library of fortran routines for random number generation base generator documentation, 1991.

[13] Bruce Schneier. *Applied Cryptography*. Wiley, 2 edition, 1996.

[14] Eleanor Chu and Alan George. FFT algorithms and their adaptation to parallel processing. *Linear Algebra Appl.*, 284(1-3):95–124, 1998. ILAS Symposium on Fast Algorithms for Control, Signals and Image Processing (Winnipeg, MB, 1997).

[15] Paul D. Coddington and Sung-Hoon Ko. Techniques for empirical testing of parallel random number generators. 1998.

[16] Raymond Couture and Pierre L'Ecuyer. Distribution Properties of Multiply-With-Carry Random Number Generators. *Mathematics of Computation*, 66(218):591–607, April 1997.

[17] Raymond Couture, Pierre L'Ecuyer, and Shu Tezuka. On the distribution of k-dimensional vectors for simple combined tausworthe sequences. *Mathematics of Computation*, 60:749–761, April 1993.

[18] Raymond Couture, Pierre L'Ecuyer, and Shu Tezuka. Supplement to on the distribution of k-dimensional vectors for simple combined tausworthe sequences. *Mathematics of Computation*, 60:S11–S16, April 1993.

[19] R. R. Coveyou and R. D. Macpherson. Fourier analysis of uniform random number generators. *J. Assoc. Comput. Mach.*, 14:100–119, 1967.

[20] Edward Dudewicz. Entropy-based random number evaluation. Technical report, Ohio State Univ., Columbus. Dept of Statistics, April 1981.

[21] D.E. Eastlake, S.D. Crocker, and J.L. Schiller. Randomness requirements for security, 1994.

[22] Alan Edelman, Peter McCorquodale, and Sivan Toledo. The future fast fourier transform? *SIAM J. SCI. COMPUT.*, 20(3):1094–1114, 1999.

[23] Bradley Efron. *THe Jackknife, the Bootstrap and Other Resampling Plans*. SIAM, Philadelphia, PA, 1982.

[24] Karl Entacher. Bad subsequences of well-known linearcongruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 8(1):61–70, January 1998.

[25] Karl Entacher. Parallel streams of linear random numbers in the spectral test. *ACM Transactions on Modeling and Computer Simulation*, 9(1):31–44, January 1999.

[26] A. Ferrenberg, D.P. Landau, and Y. Wong. Monte carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–84, December 1992.

[27] P. Frederickson, R. Hiromoto, T.L. Jordan, B. Smith, and T. Warnock. Pseudo-random trees in monte carlo. *Parallel Computing*, 1:175–180, 1984.

[28] M. Frigo and S.G. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.

[29] Matteo Frigo. A fast fourier transform compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[30] Hari Krishna Garg. *Digital Signal Processing Algorithms: Number Theory, Convolution, Fast Fourier Transforms, and Applications*. CRC Press, 1998.

[31] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Longman, Inc., October 1989.

[32] N. R. Goodman. Statistical Analysis Based on a Certain Multivariate Complex Gaussian Distribution (An Introduction). *Annals of Mathematical Statistics*, 34(1):152–177, 1963.

[33] H. Hellekalek, P.; Niederreiter. The weighted spectral test: diaphony. *ACM Transactions on Modeling and Computer Simulation*, 8(1), Jan 1998.

[34] Peter Hellekalek. Inversive pseudorandom number generators: Concepts, results and links. In C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, 1995.

[35] Peter Hellekalek. Don't trust parallel monte carlo! In *Proceedings of the 12th workshop on Parallel and distributed simulation*, pages 82–89, May 1998. Workshop on Parallel and Distributed Simulation.

[36] Charles Herring and Julian I. Palmore. Random number generators are chaotic. *SIGPLAN Notices*, 24(11), November 1989.

[37] T.P. Hettmansperger and J.W. McKean. *Robust Nonparametric Statistical Methods*. John Wiley & Sons Inc., 1998.

[38] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw Hill, 1995.

[39] Y. Horibe. Entropy and an optimal random number transformation. (4):527–9, July 1981.

[40] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[41] Tohru Ikeguchi and Kazuyuki Aihara. Lyapunov spectral analysis on random data. *International Journal of Bifurcation and Chaos in Applied Sciences and Engineering*, 7(6):1267–1282, June 1997.

[42] Intel Corporation. *Intel 82802 Firmware Hub: Random Number Generator*, December 1999.

[43] Markus Jakobsson, Elizabeth Shriver, Bruce K. Hillyer, and Ari Juels. A practical secure physical random bit generator. In *Proceedings of the 5th ACM conference on Computer and communications security*, pages 103–111, 1998.

[44] F. James. Chaos and randomness. *Chaos, Solitons & Fractals*, 6(1):221–226, 1995.

[45] Brad C. Johnson. Radix-b extensions to some common empirical tests for pseudorandom number generators. *ACM Transactions on Modeling and Computer SImulation*, 6(4):261–273, October 1996.

[46] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *ixth Annual Workshop on Selected Areas in Cryptography*. Springer Verlag, August 1999.

[47] Jr. Kennedy, William J. and James E. Gentle. *Statistical Computing*. Marcel Dekker, Inc., 1980.

[48] Jack P.C. Kleijnen, Ad J. Feelders, and C.H. Cheng Cheng, Russell. Bootstrapping and validation of metamodels in simulation. In *Proceedings of the 1998 Winter Simulaiton Conference*, 1998.

[49] D.E. Knuth. *The art of computer programming, vol. 2: seminumerical algorithms*, volume 2. Addison Wesley Longman, Reading, MA, 3 edition, 1997.

[50] P. J. Van Laarhoven and Emile H. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, January 1987.

[51] C. Lanczos and B. Gellai. Fourier analysis of random sequences. *Computers and Mathematics with Applications*, 1:269–276, 1975.

[52] Pierre L'Ecuyer. Uniform random number generators. In *Proceedings of the 1998 Winter Simulation Conference*, 1998.

[53] Pierre L'Ecuyer. Good Parameters and Implementations for Combined Multiple Recursive Randome Number Generators. *Operations Research*, 47(1):159–164, January 1999.

[54] Pierre L'Ecuyer. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. *Mathematics of Computation*, 68(225):249–260, January 1999.

[55] Pierre L'Ecuyer and Serge Cote. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, March 1991.

[56] Pierre L'Ecuyer and Richard Simard. Beware of linear congruential generators with multipliers of the form $a = +/-2^q +/-2^r$. *ACM Transactions on Mathematical Software*, 25(3):367–374, September 1999.

[57] Pierre L'Ecuyer and Shu Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57:735–746, October 1991.

[58] Hannes Leeb. Weak limits for the diaphony. *Lecture Notes in Statistics*, (127):330–339, 1996.

[59] Hannes Leeb and Stefan Wegenkittl. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Transactions on Modeling and Computer Simulation*, 7(2):272–286, April 1997.

[60] Raoul LePage and Lynne Billard, editors. *Exploring the Limits of Bootstrap*. Wiley, 1992.

[61] Vsevolod F. Lev. On Two Versions of l2-Discrepancy and Geometrical Interpretation of Diaphony. *Acta Math. Hungar.*, 69(4):281–300, 1995.

[62] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the Association for Computing Machinery*, 20(3):456–468, July 1973.

[63] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 1993.

[64] N. M. Maclaren. The generation of multiple independent sequences of pseudorandom numbers. *Applied Statistics*, 38:351–359, 1989.

[65] G. Marsaglia. Random numbers fall mainly on the planes. *Proc Nat. Acad. Sc.*, 61(1):25–28, September 1968.

[66] G. Marsaglia. The marsaglia random number cdrom, with the diehard battery of tests of randomness, 1996. Department of Statistics, Florida State University.

[67] G. Marsaglia. A random number generator for C. Discussion Paper posting on Usenet newsgroup sci.stat.math, September 1997.

[68] G. Marsaglia and Wai Wan Tsang. The monty python method for generating random variables. *ACM Transactions on Mathematical Software*, 24(3):341–50, Sept 1998.

[69] George Marsaglia. Multiply-With-Carry (MWC) Generators. In DIEHARD distribution [http:http://stat.fsu.edu/pub/diehard/].

[70] George Marsaglia. A current view of random number generators. In L. Billar, editor, *XVIth Conference on: Computer Science and Statistics: The Interface*, pages 3–10. Elsevier Science Publishers B.V. (North-Holand), 1985.

[71] George Marsaglia and Arif Zaman. A New Class of Random Number Generators. *The Annals of Applied Probability*, 1(3):462–480, August 1991.

[72] Conrado Martinez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.

[73] M. Mascagni. Sprng: A scalable library for pseudorandom number generation. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, pages 22–24, March 1999.

[74] T. Matsumoto, M.; Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[75] U.M. Maurer. A universal statistical test for random bit generators. *Journal of Cryptology*, 5(2):89–105, 1992.

[76] Alfred J. Menezes, Scott A. Vanstone, and Paul C. van Oorschot. *Handbook of Applied Cryptography*. CRC Press, LLC, October 1996.

[77] Eytan Modiano. Random algorithms for scheduling multicast traffic in wdm broadcast-and-select networks. *IEEE/ACM Transactions on Networking*, 7(3), June 1999.

[78] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, PA, 1992.

[79] Harald Niederreiter. Pseudorandom vector generation by the multiple-recursive matrix method. *Mathematics of Computation*, 64:279–294, January 1995.

[80] Christy Juinsuk Olsen. Spectral analysis of random number generators. Master's thesis, University of Nevada, Las Vegas, 1992.

[81] Alan V. Oppenheim and Ronald W. Shafer. *Digital Signal Processing*. Prentice-Hall, 1975.

[82] Victor Pan. Complexity of computations with matrices and polynomials. *SIAM Review*, 34:225–262, June 192.

[83] Sang Hwan Park, Wook Hyun Kwon, Oh Kyu Kwon, and Myung-Joon Kim. Short-Time Fourier Analysis via Optimal Harmonic FIR Filters. 45(6):1535–1542, June 1997.

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

[85] C. Pierce, M.A.; Bays. Applications of the collision test to produce results consistent with the spectral test. 57(5):335–43, Nov 1991.

[86] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[87] M. Sanaha and U.V. Vazirami. Generating quasi-random sequences from slightly random sources. *Journal of Computer and System Sciences*, 33:75–87, 1986.

[88] P. Shchur, L.N.; Butera. The ranlux generator: resonances in a random walk test. 9(4):607–24, June 1998.

[89] Srivastava and Khatri. *An Introduction to Multivariate Statistics*. New Your: Elsevier-North Holland, 1979.

[90] William Stallings. *Cryptography and Network Security: Principles and Practice, Second Edition*. Prentice Hall, June 1998.

[91] Dietrich Stauffer. Ising Model as Test for Simple Random Number Generators. *International Journal of Modern Physics*, 10(5):807–808, 1999.

[92] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, LLC, March 1995.

[93] Alan Stuart and J. Keith Ord. *Advanced Theory of Statistics Volume 2 (Fifth edition)*. Oxford University Press, New York, 1991.

[94] A. Tajima, S. Ninomiya, and S. Tezuka. Analysis of the anomaly of ran1() generator in monte carlo pricing of financial derivatives. 41(3):387–97, Sept 1998.

[95] James Theiler. The gnu scientific library [http://sources.redhat.com/gsl/].

[96] Clark Thomborson. Tools for randomized experimentation, 1992. (draft copy via anonymous ftp).

[97] M. Tomassini, M. Sipper, M. Zolla, and M. Perrenoud. Generating high-quality random numbers in parallel by cellular automata. volume 16, pages 291–305. Elsevier, Dec 1999.

[98] D. Ugrin-Sparac, G.; Ugrin-Sparac. On a possible error of type ii in statistical evaluation of pseudo-random number generators. 56(2):105–16, 1996.

[99] Andre van Hameren, Ronald Kleiss, and Jiri Hoogland. Gaussian limits for discrepencies I. Asymptotic results. *Computer Physics Communications*, (107):1–20, 1997.

[100] I. D. Wichmann, B. A.; Hill. Algorithm as 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.

[101] I. D. Wichmann, B. A.; Hill. Correcction: Algorithm as 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 33:123, 1984.

[102] R. S. Wikramaratna. ACORN—a new method for generating sequences of uniformly distributed pseudo-random numbers. *J. Comput. Phys.*, 83(1):16–31, 1989.

[103] R. S. Wikramaratna. Pseudo-random number generation for parallel monte carlo– a splitting approach. *SIAM News*, 33(9), 2000.

[104] Shmuel Winograd. *Arithmetic Complexity of Computations*. SIAM, Philadelphia, PA, 1980.

[105] R. A. Wooding. The Multivariate Distribution of Complex Normal Variables. *Biometrika*, 43(1/2):212–215, June 1956.

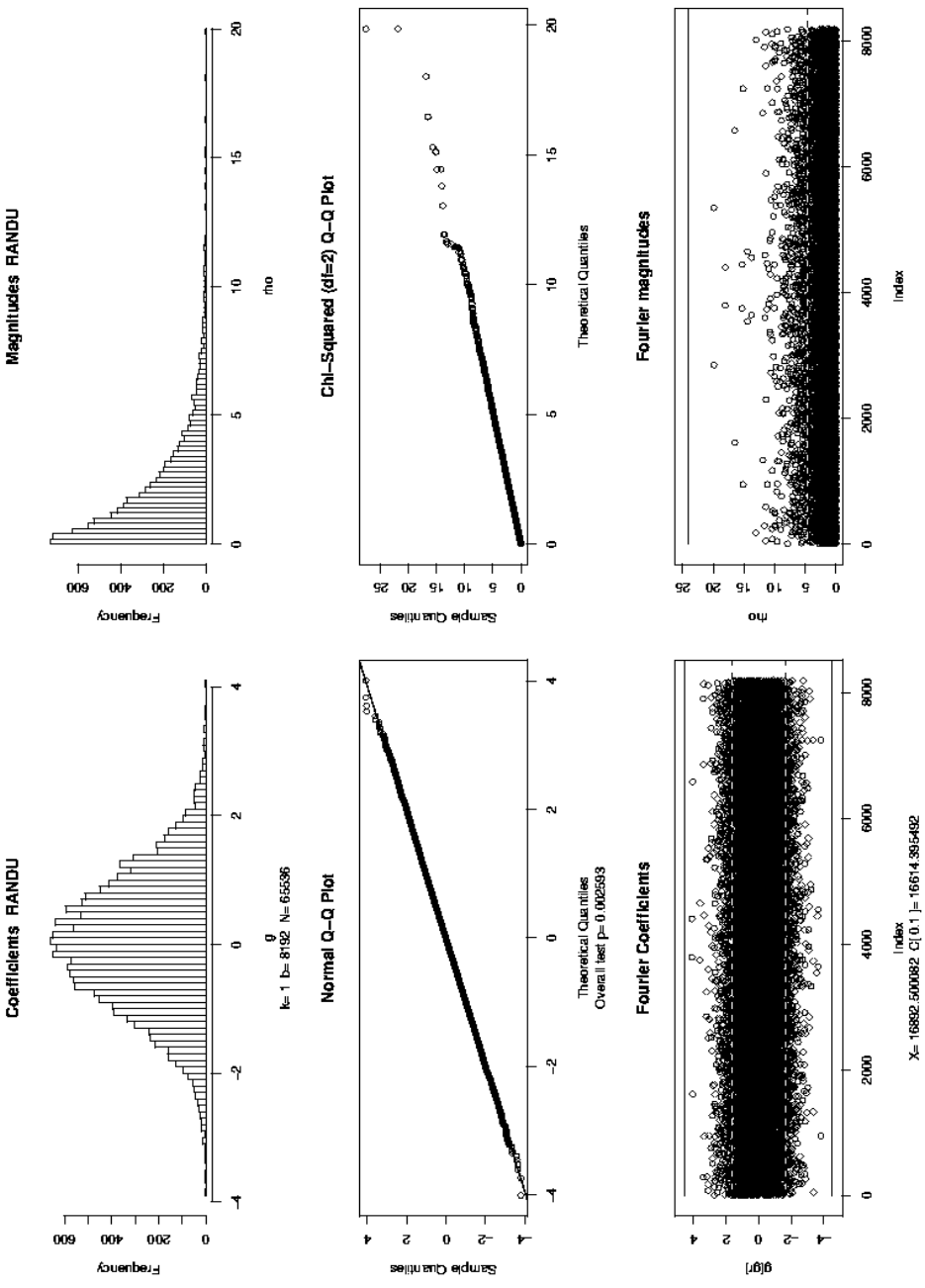
[106] J. Yuan. Tests of gaussianity and linearity for random fields using estimated higher order spectra. *IEEE Transactions on SIgnal Processing*, 46(1):247–250, January 1998.

[107] Chung-Kwong Yuen. Testing random number generators by walsh transform. *IEEE Transactions on Computers*, C-26(4):329–333, April 1977.

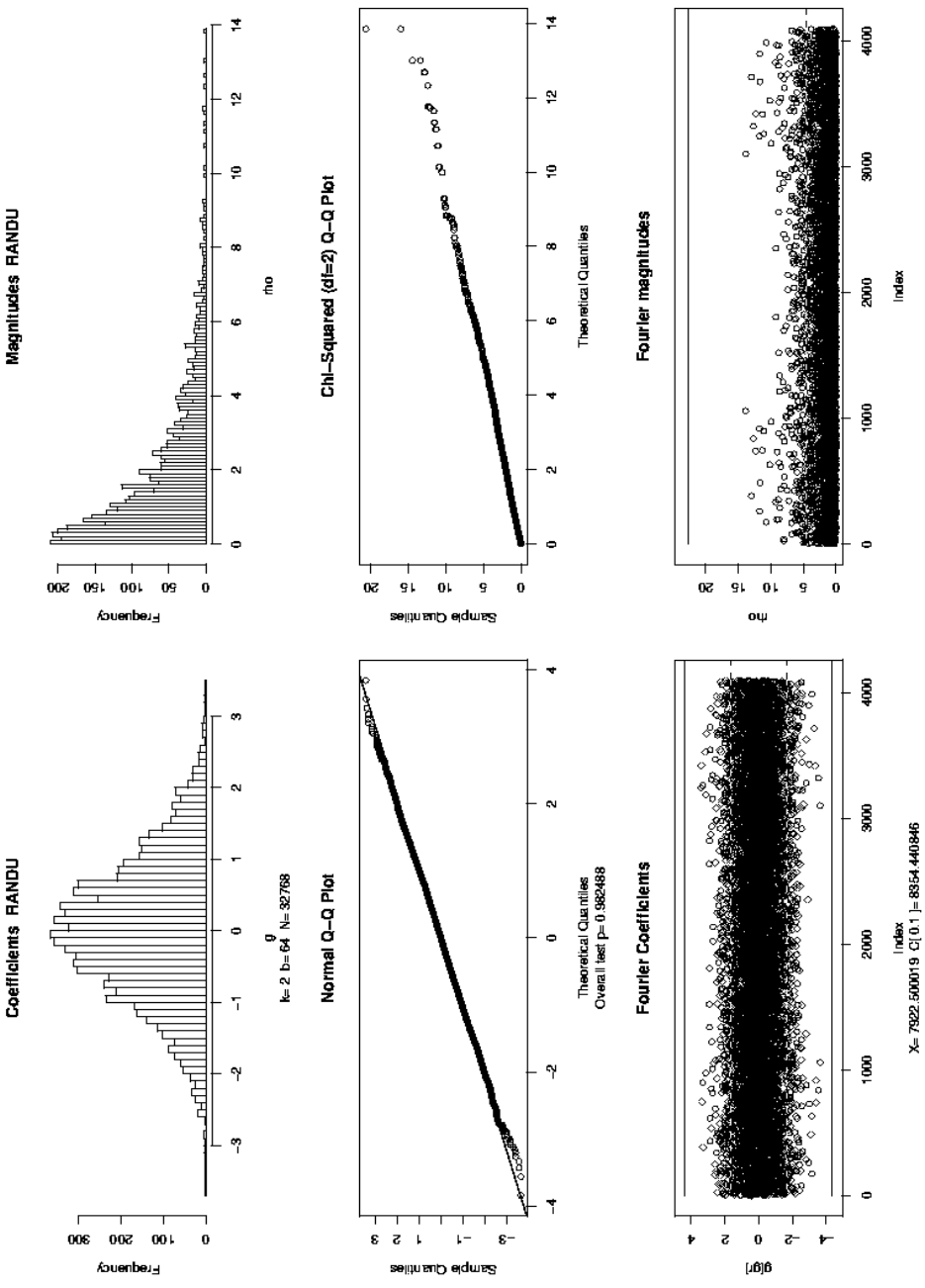
[108] David Zeitler and John Kapenga. Assuring the validity of random number generation for parallel computation. In *Proceedings of the Quality and Productivity Section of the ASA*. ASA, Alexandria, VA, 1993.

APPENDIX A

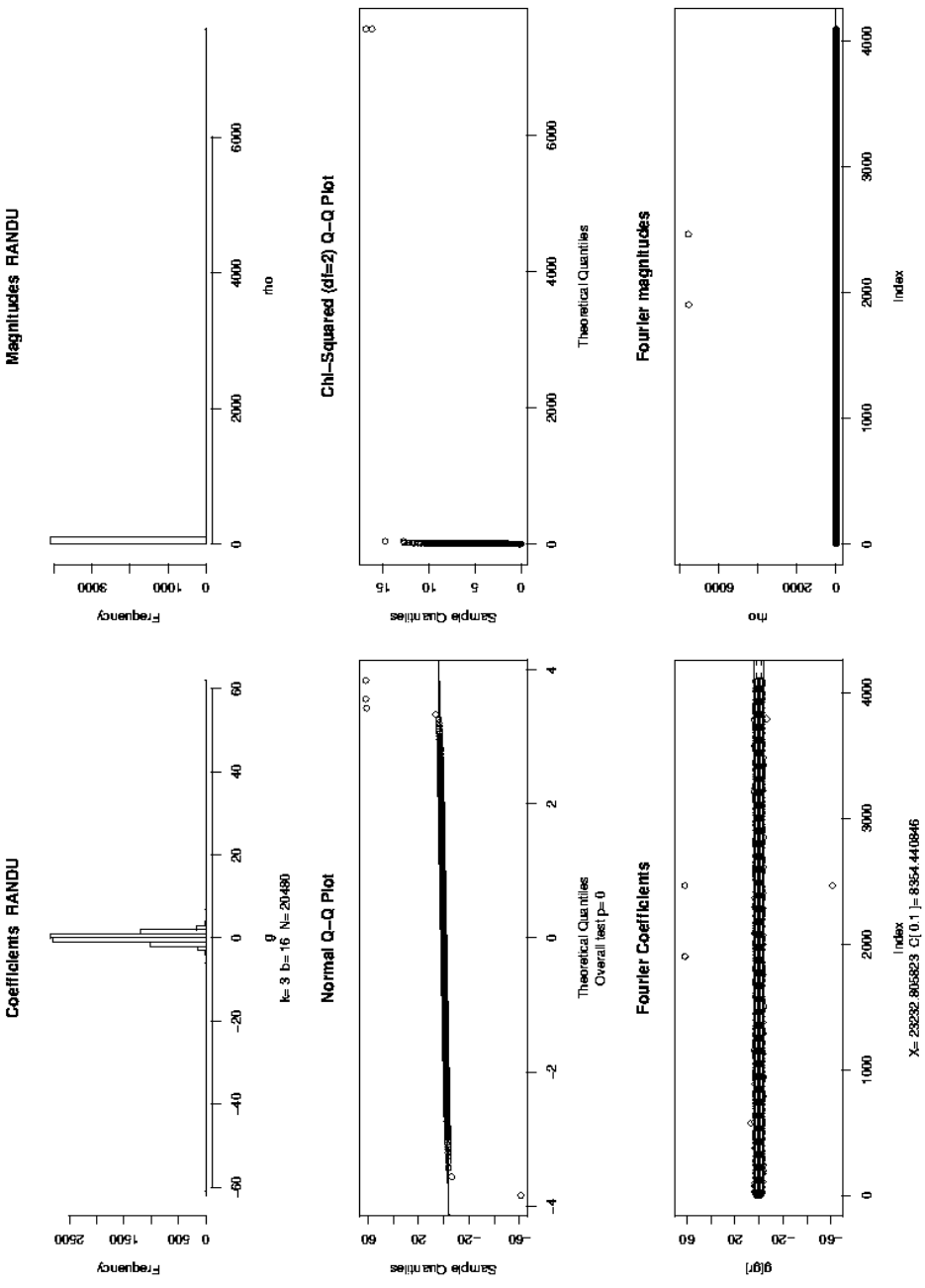
GRAPHICS


Coefficients RANDU
Frequency
g
k= 1 b= 8192 N= 65536
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.002593
Fourier Coefficients
g[gr]
Index
X= 16882.500082 C[0.1]= 16614.395492
Magnitudes RANDU
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
mo
Index

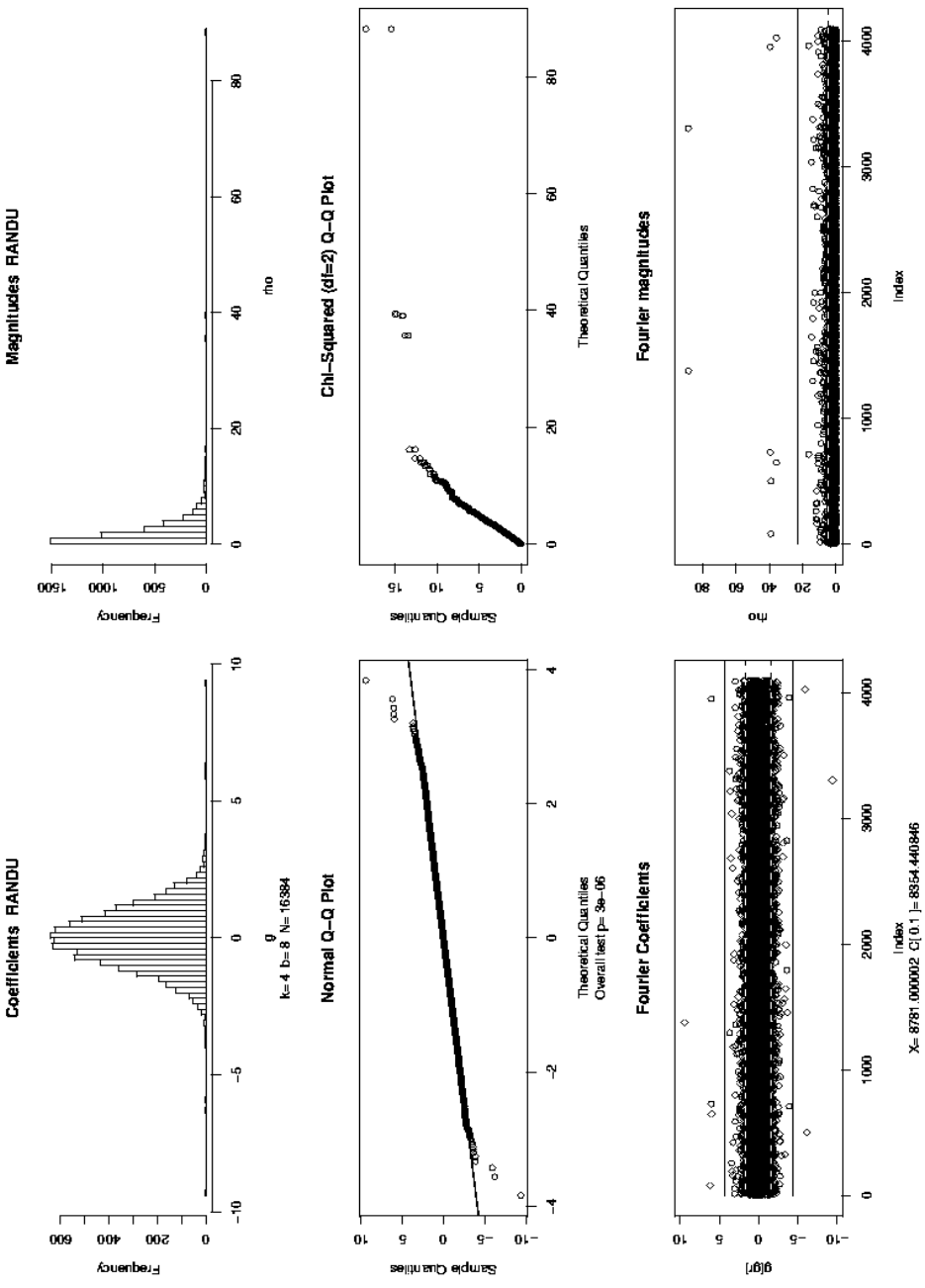
RANDU 1d


Coefficients RANDU
Frequency
g
k= 2 b= 64 N= 32768
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.982488
Fourier Coefficients
g[gr]
Index
X= 7922.500019 C[0.1]= 8354.440846
Magnitudes RANDU
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

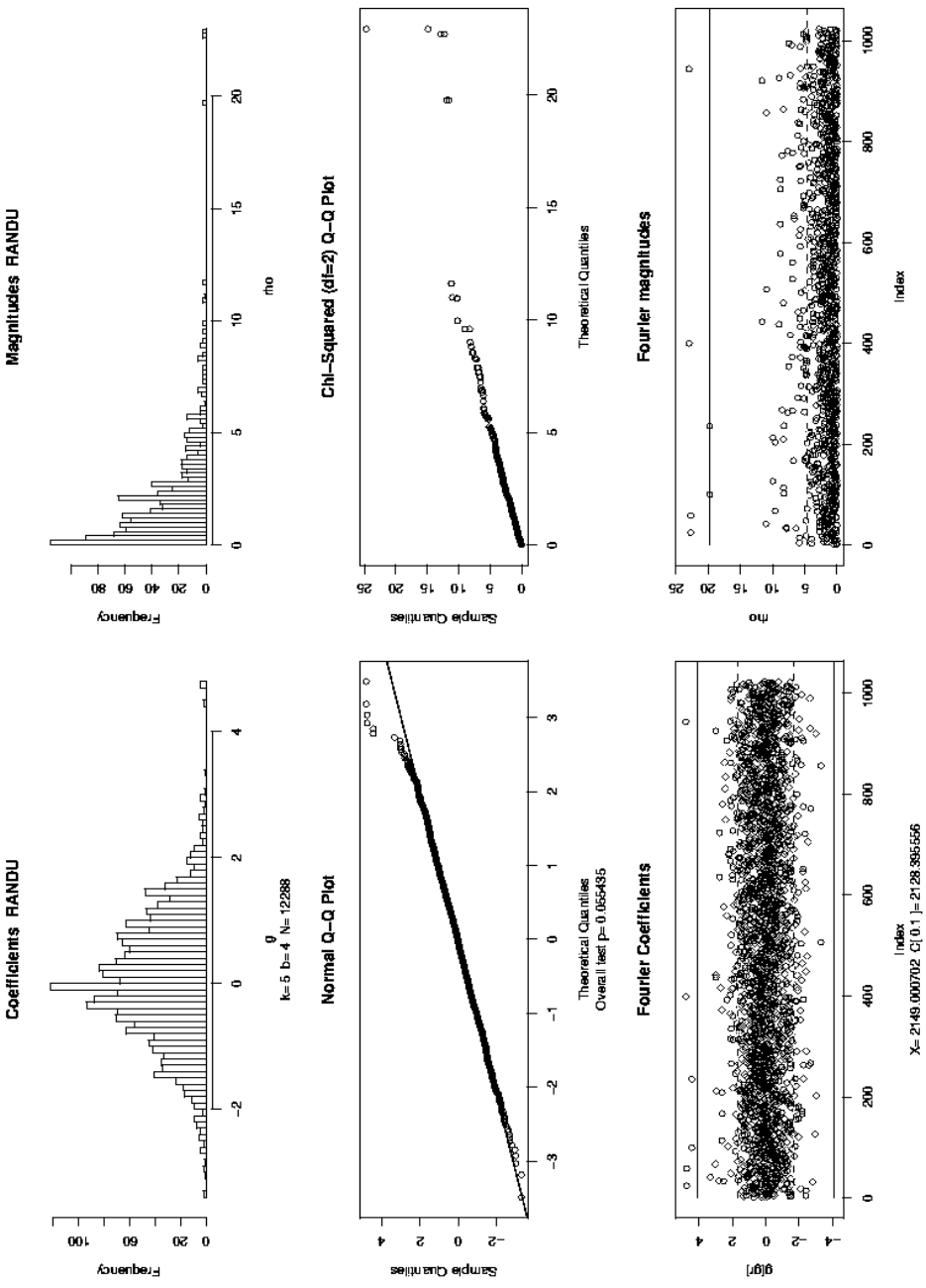
RANDU 2d


Magnitudes RANDU
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
mo
Index
Coefficients RANDU
Frequency
g
k= 3 b= 16 N= 20480
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0
Fourier Coefficients
g[gr]
Index
X= 29232.805823 C[0.1]= 8354.440846

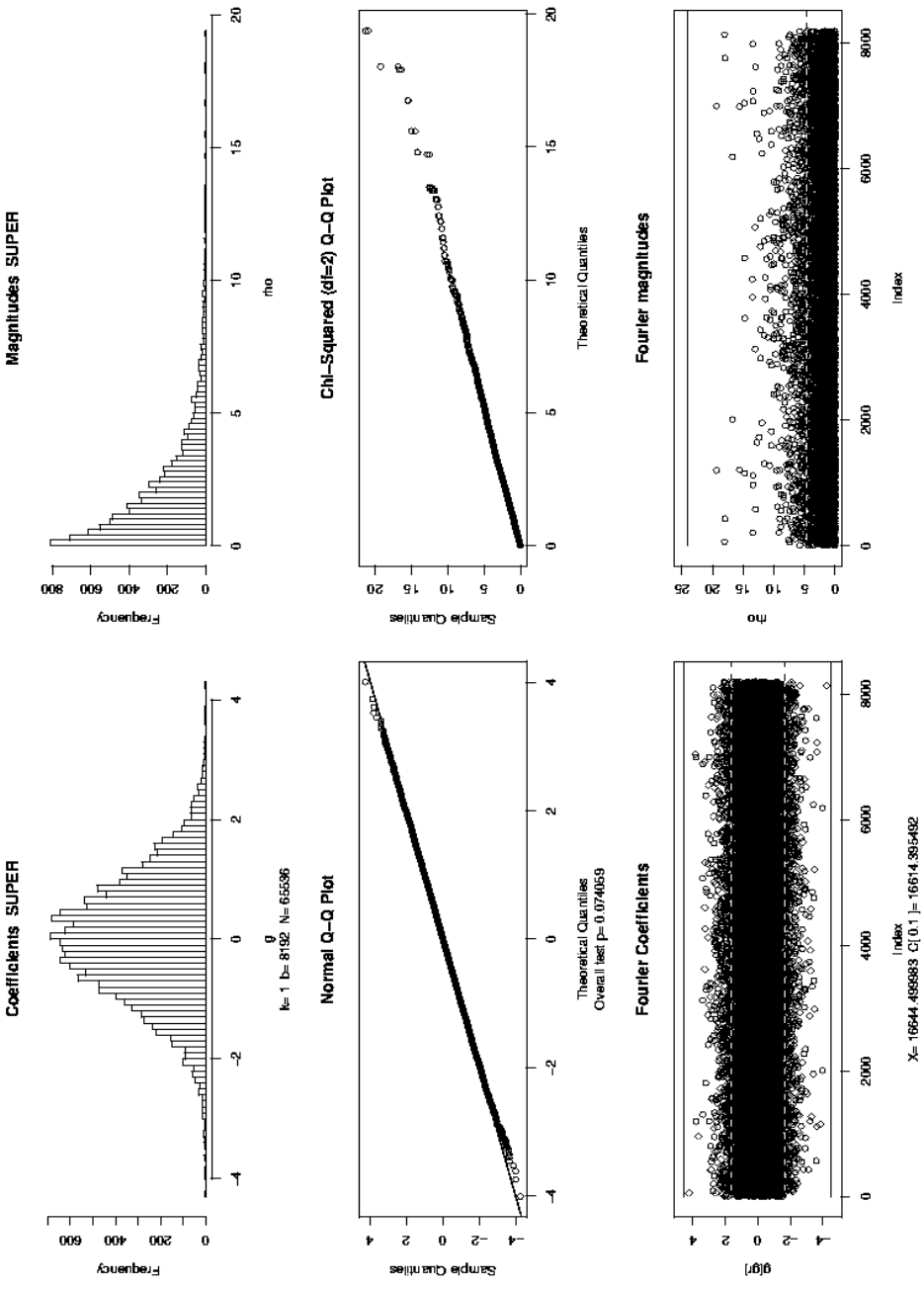
RANDU 3d


Magnitudes RANDU
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index
Coefficients RANDU
Frequency
g
k= 4 b= 8 N= 16384
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 3e–06
Fourier Coefficients
g[g]
Index
X= 8781.000002 C[0.1]= 8354.440846

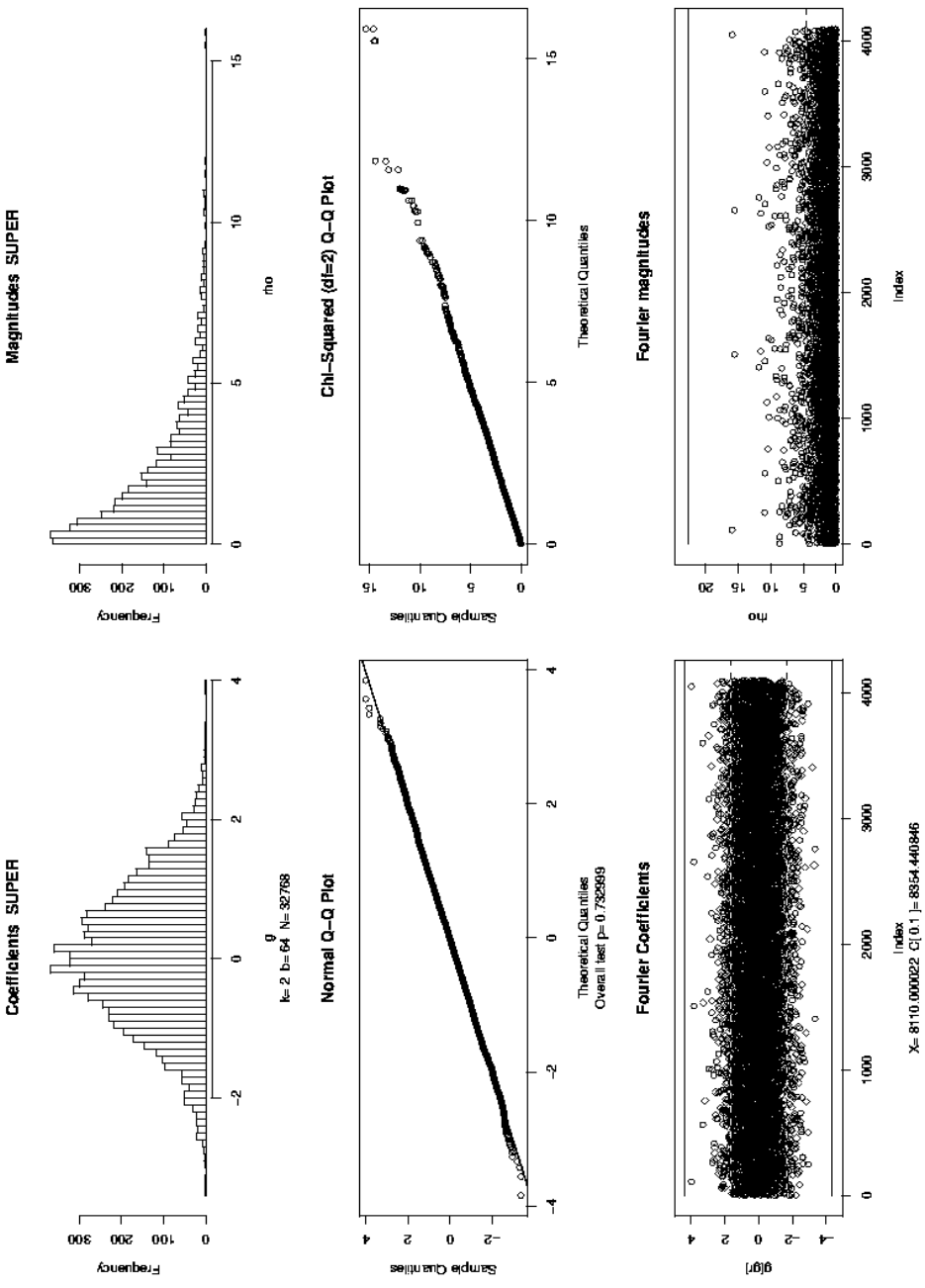
RANDU 4d


Coefficients RANDU
Frequency
g
k= 5 b= 4 N= 12288
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.655435
Fourier Coefficients
g[gr]
Index
X= 2149.000702 C[0.1]= 2128.395556
Magnitudes RANDU
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

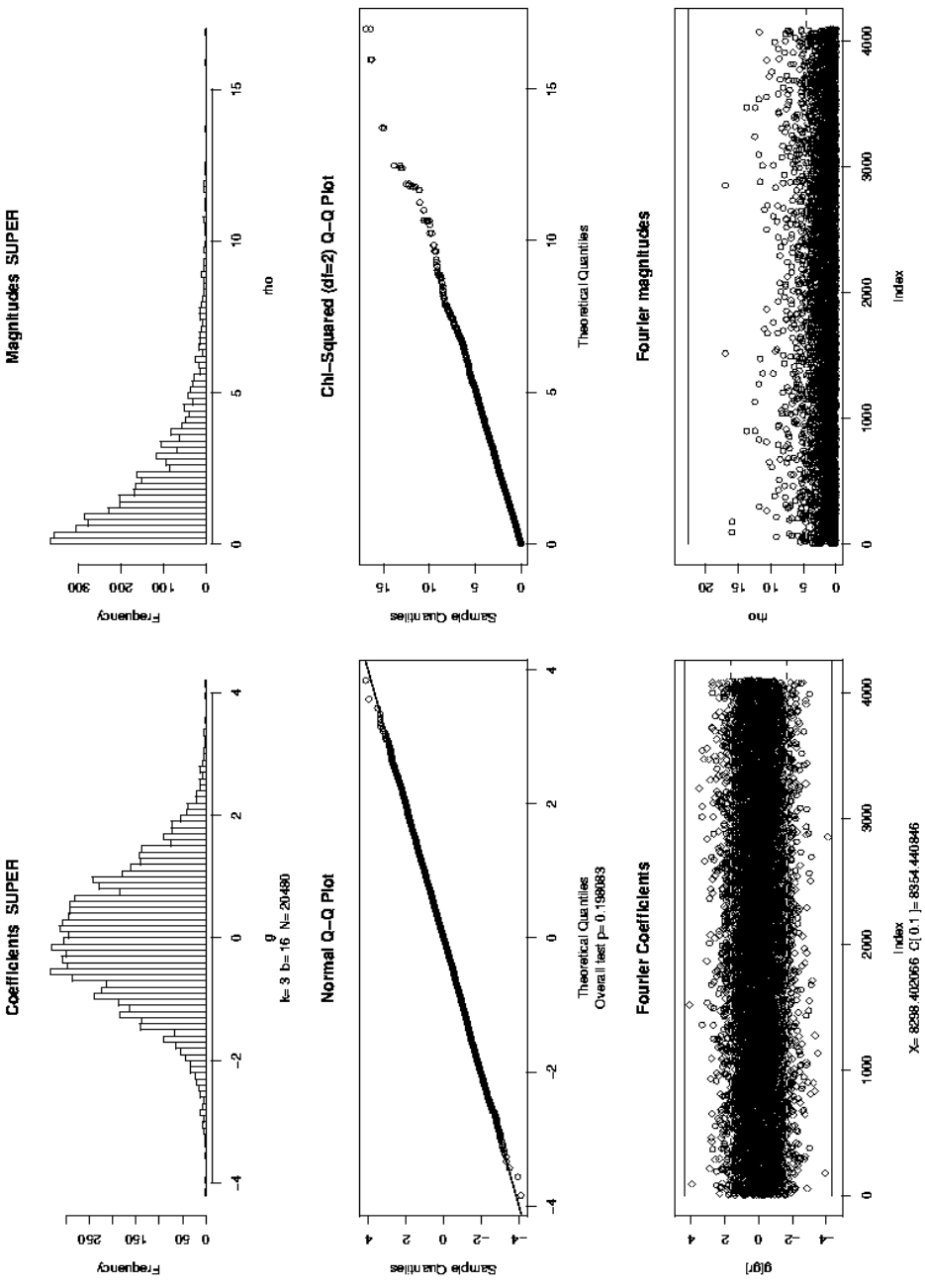
RANDU 5d


Magnitudes SUPER
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
mo
Index
Coefficients SUPER
Frequency
g
k= 1 b= 8192 N= 65536
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.074059
Fourier Coefficients
g[gr]
Index
X= 16644.499883 C[0.1]= 16614.395492

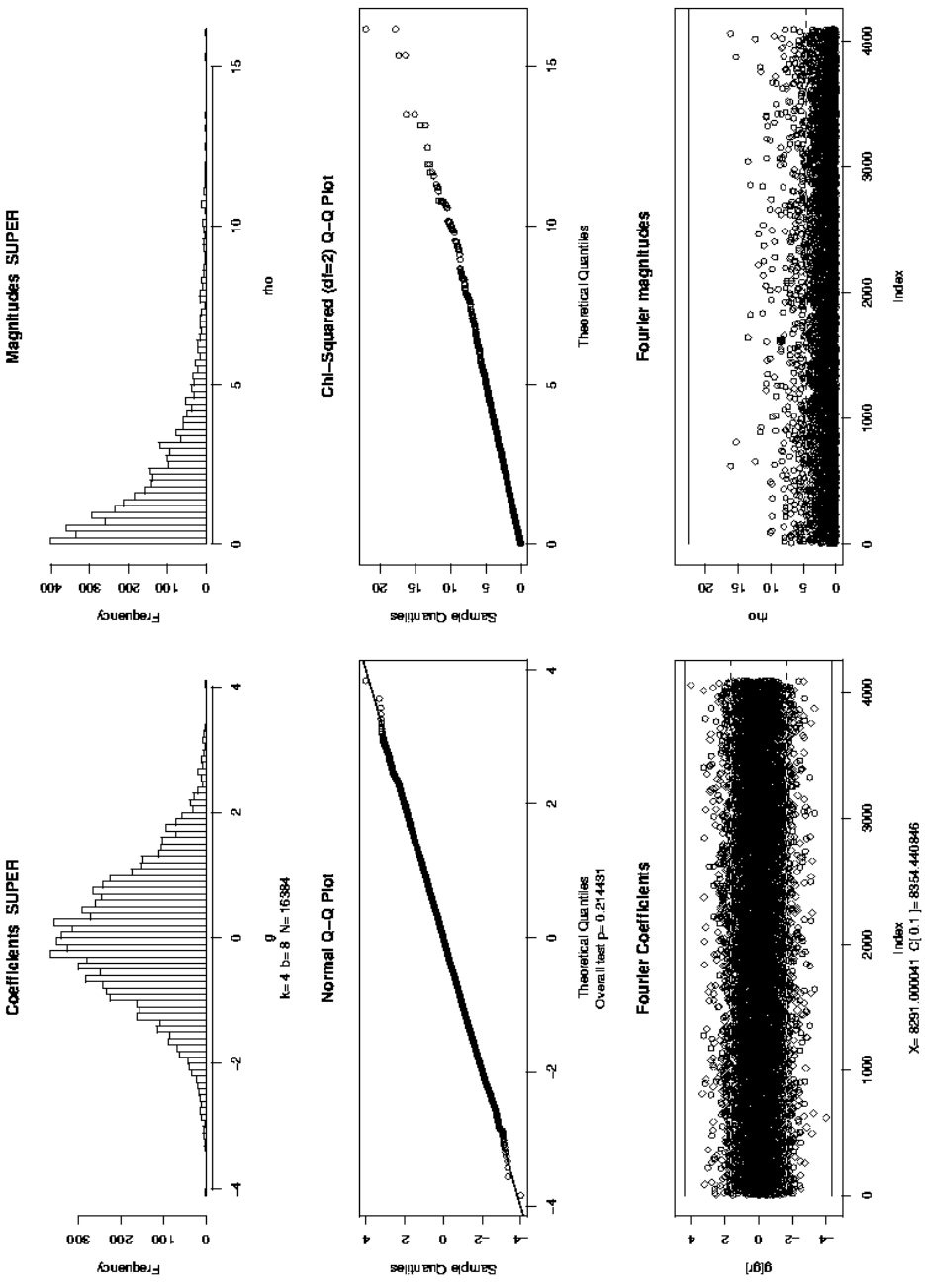
SUPER 1d


Coefficients SUPER
Frequency
g
k= 2 b= 64 N= 32768
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.732999
Fourier Coefficients
g[g]
Index
X= 8110.000022 C[0.1]= 8354.440846
Magnitudes SUPER
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

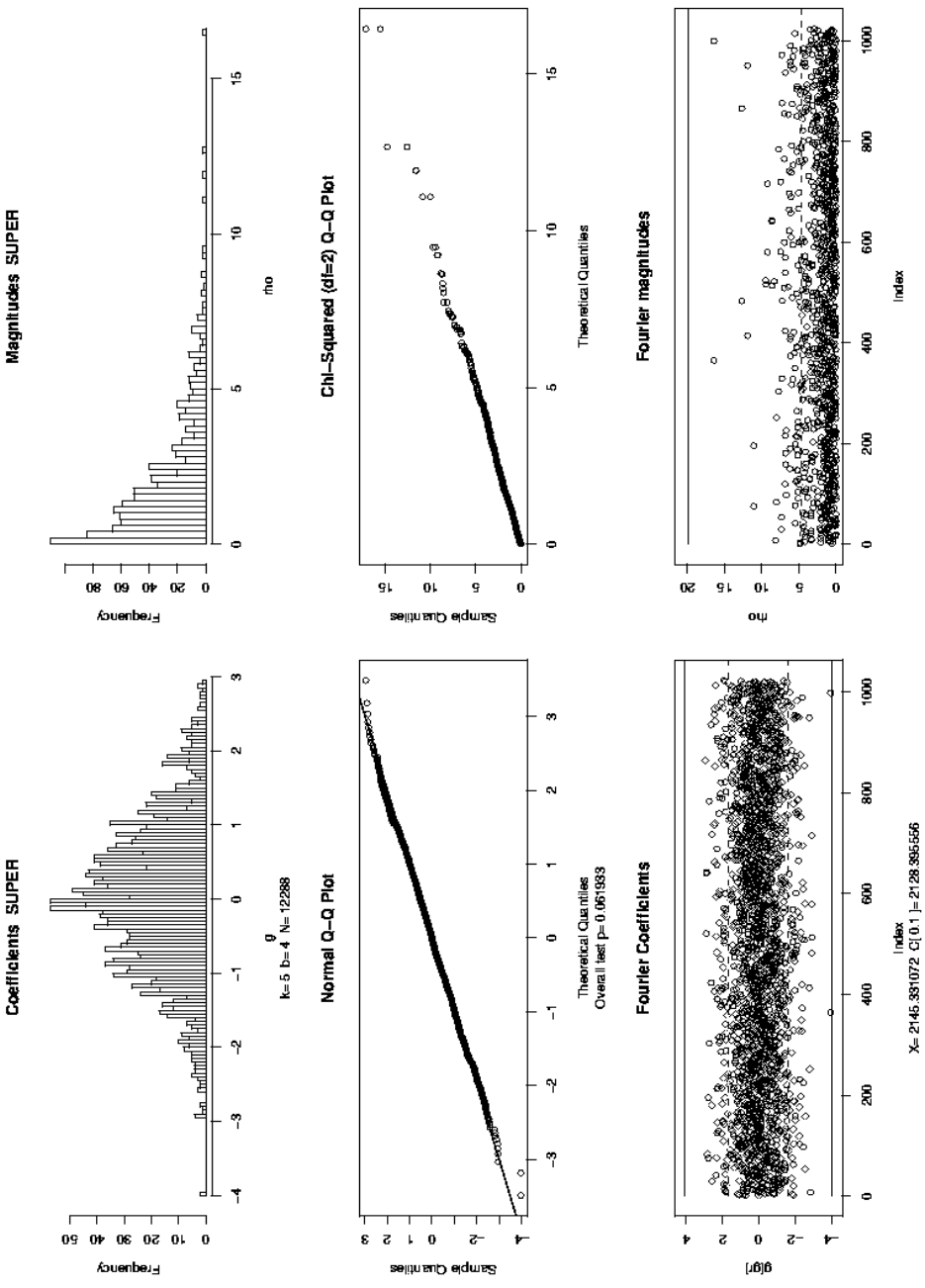
SUPER 2d


Coefficients SUPER
Frequency
g
k= 3 b= 16 N= 20480
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.198083
Fourier Coefficients
g[gr]
Index
X= 8258.402066 C[0.1]= 8354.440846
Magnitudes SUPER
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

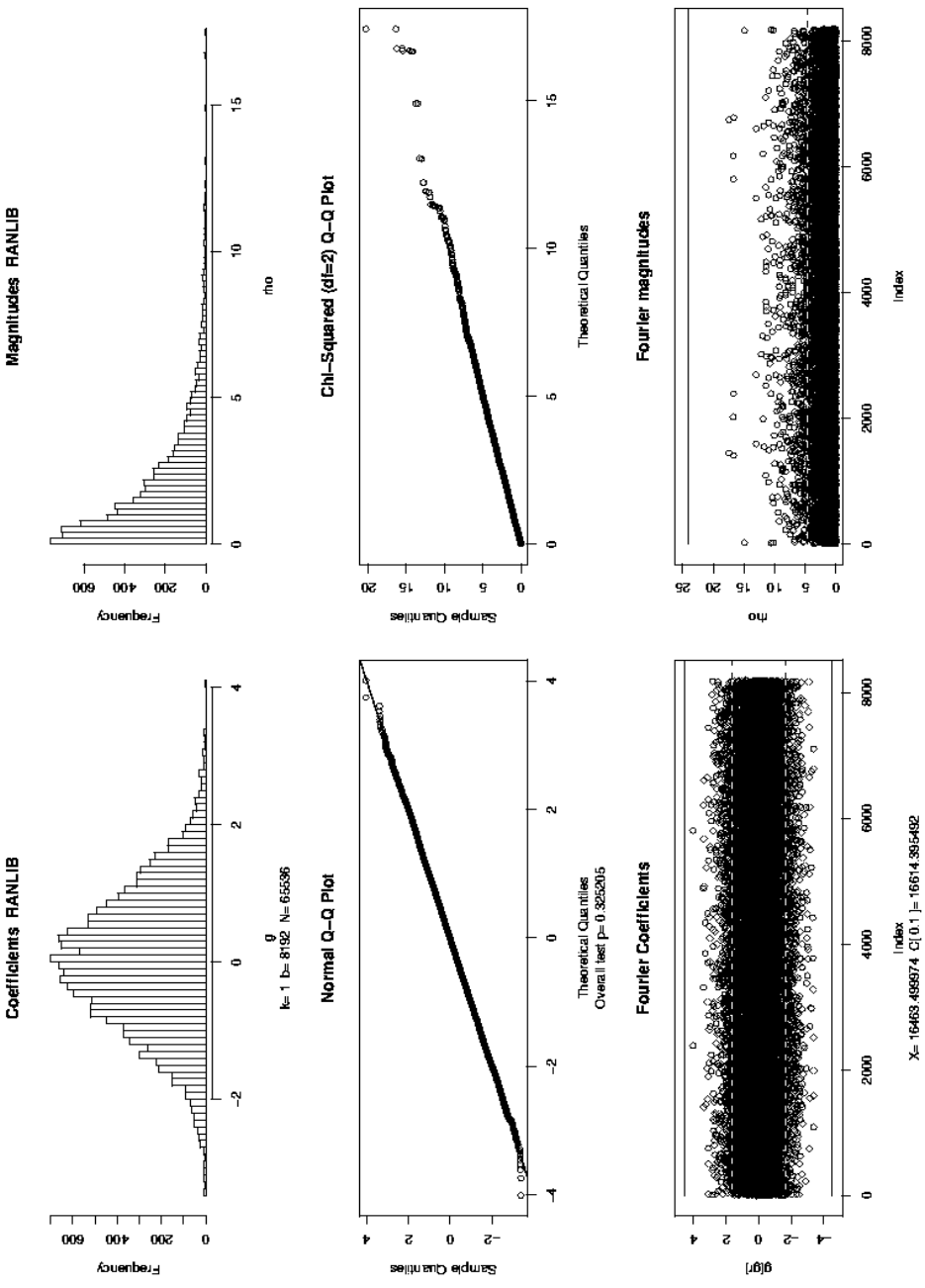
SUPER 3d


Magnitudes SUPER
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index
Coefficients SUPER
Frequency
g
k= 4 b= 8 N= 16384
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.214431
Fourier Coefficients
g[gr]
Index
X= 8291.000041 C[0.1]= 8354.440846

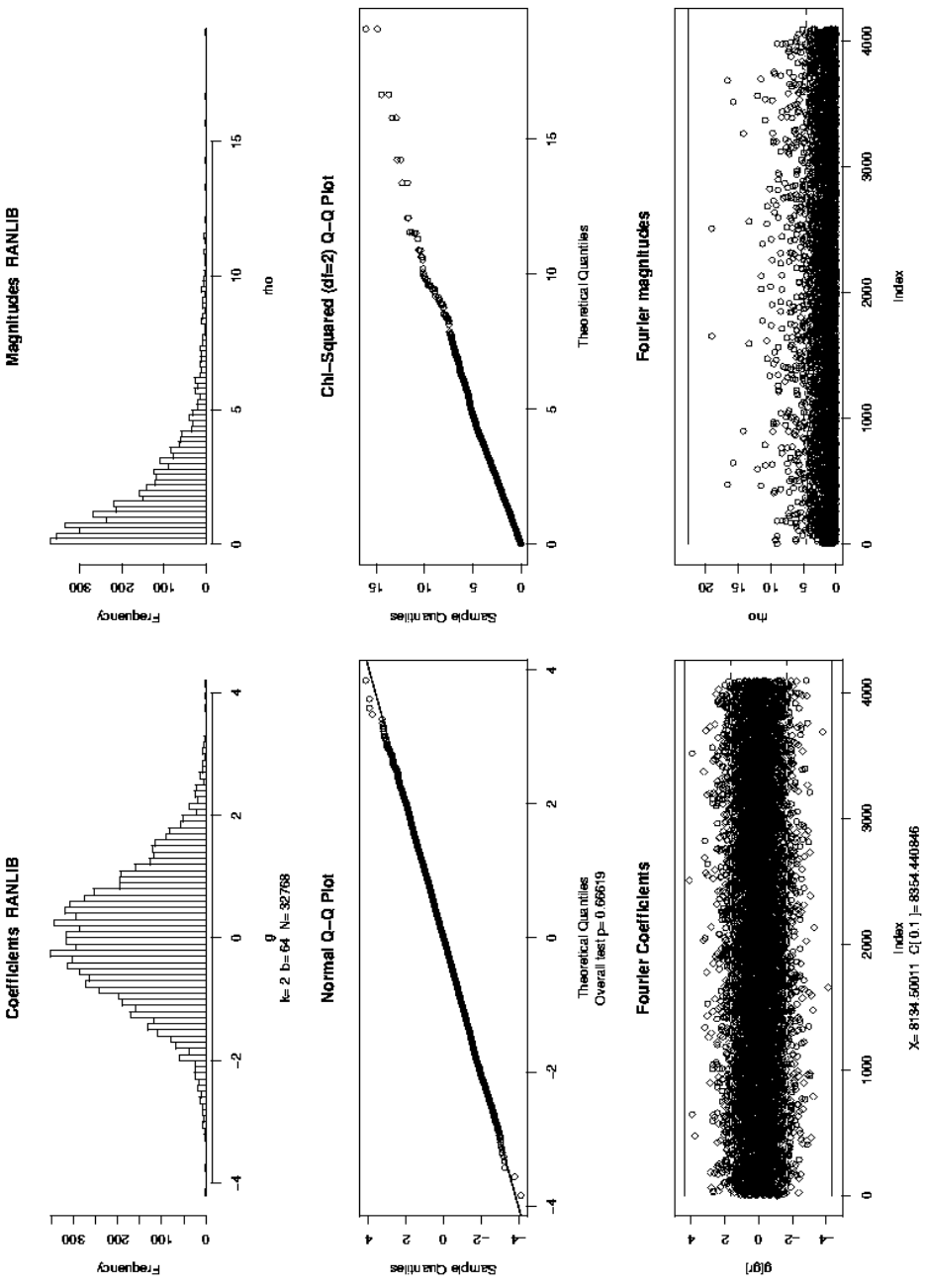
SUPER 4d


Magnitudes SUPER
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
mo
Index
Coefficients SUPER
Frequency
g
k= 5 b= 4 N= 12288
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.061933
Fourier Coefficients
g[gr]
Index
X= 2145.331072 C[0.1]= 21289.395556

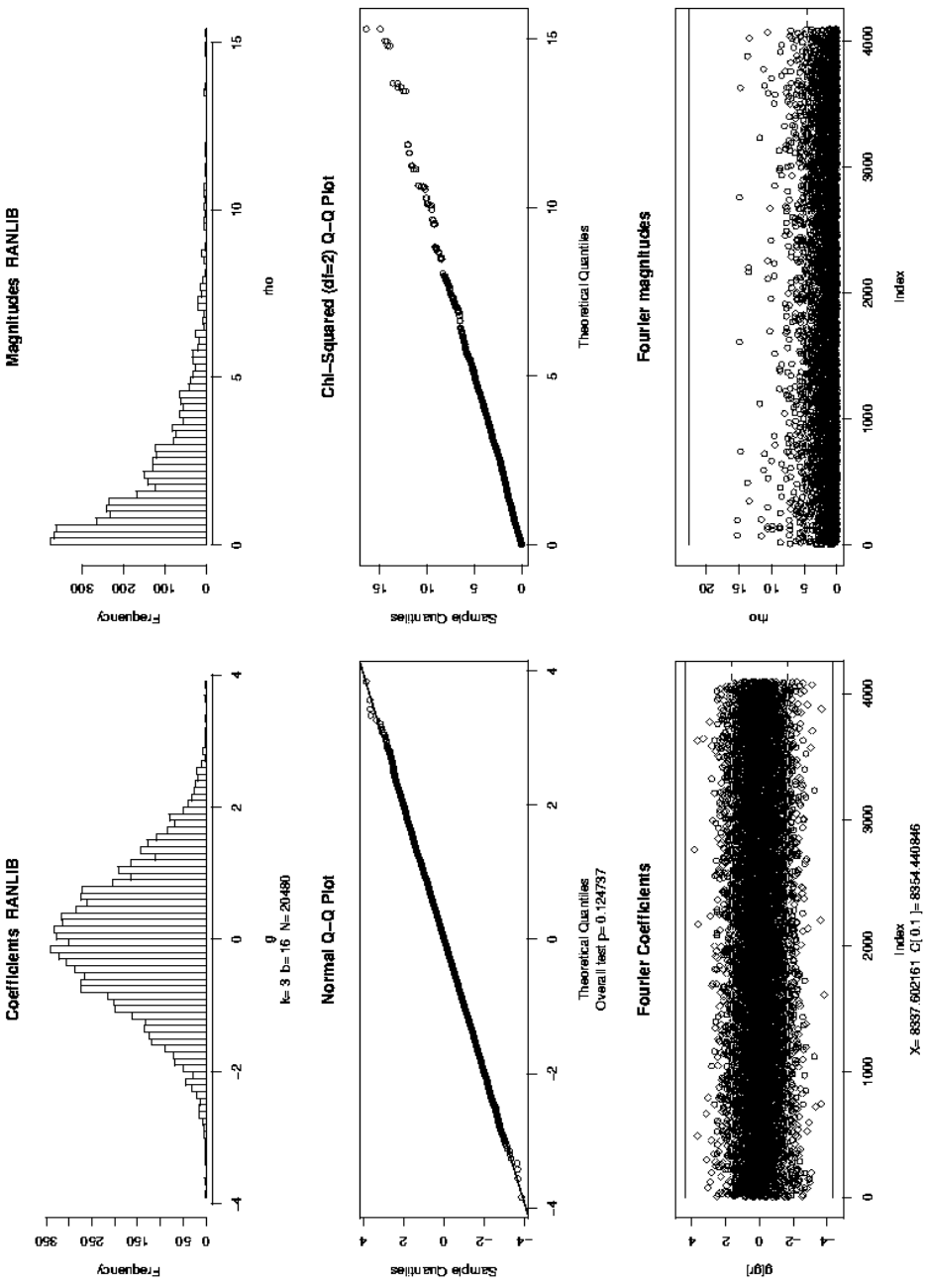
SUPER 5d


Coefficients RANLIB
Frequency
g
k= 1 b= 8192 N= 65536
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.325205
Fourier Coefficients
g[gr]
Index
X= 16463.499974 C[0.1]= 16614.395492
Magnitudes RANLIB
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

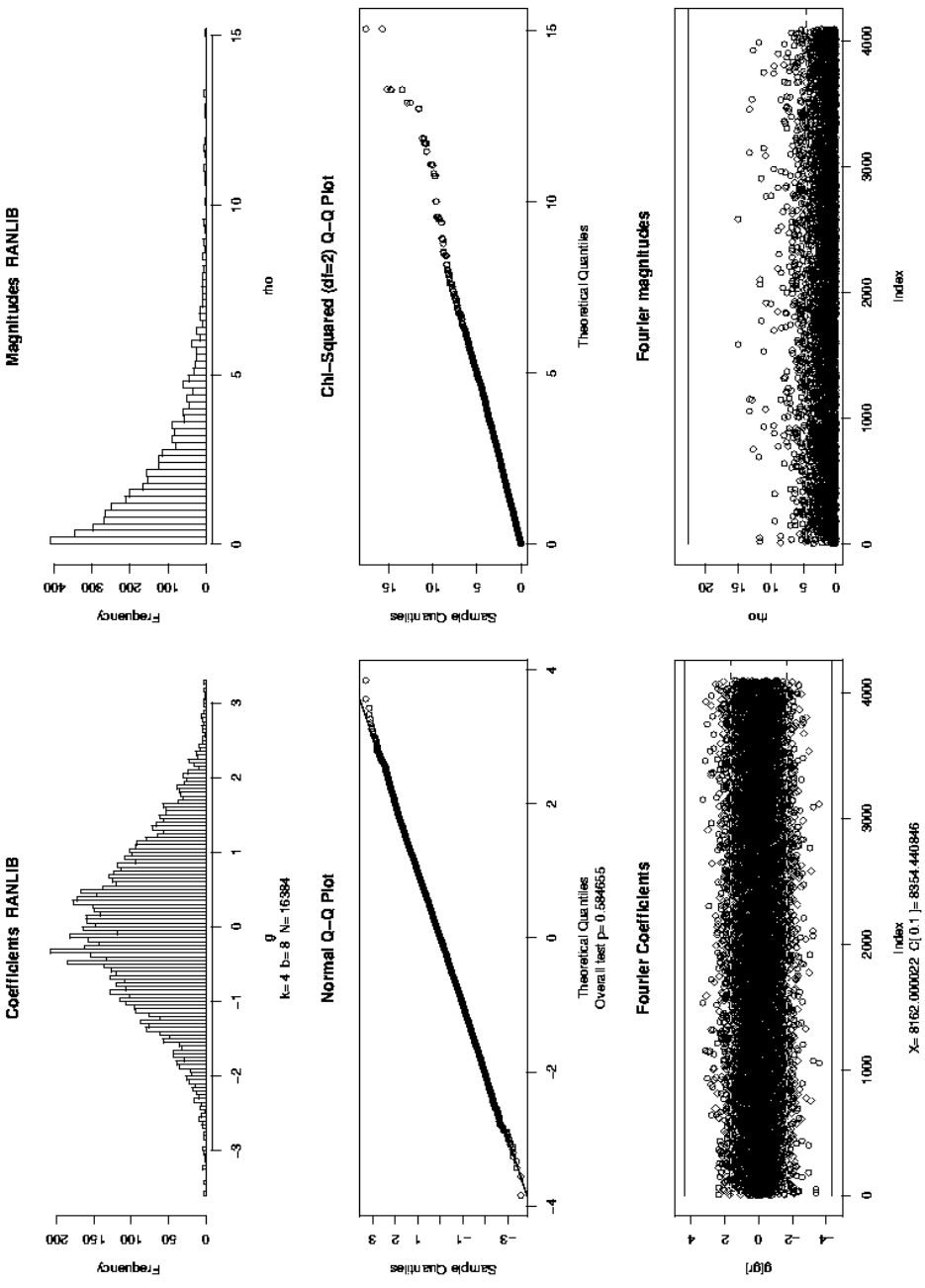
RANLIB 1d


Coefficients RANLIB
Frequency
g
k= 2 b= 64 N= 32768
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.66619
Fourier Coefficients
g[gr]
Index
X= 8134.50011 C[0.1]= 8354.440846
Magnitudes RANLIB
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

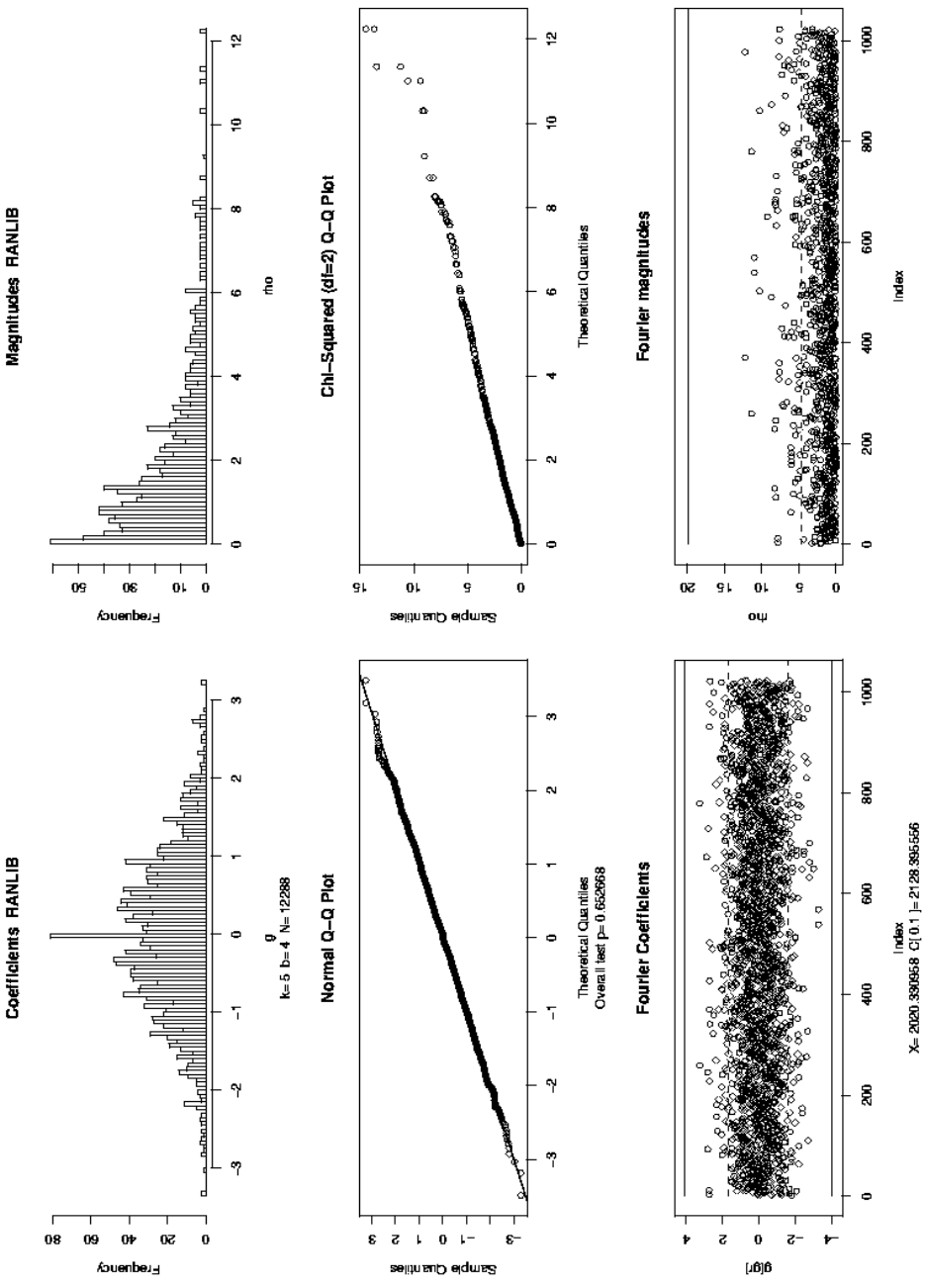
RANLIB 2d


Coefficients RANLIB
Frequency
g
k= 3 b= 16 N= 20480
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.124737
Fourier Coefficients
g[gr]
Index
X= 8337.602161 C[0.1]= 8354.440846
Magnitudes RANLIB
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

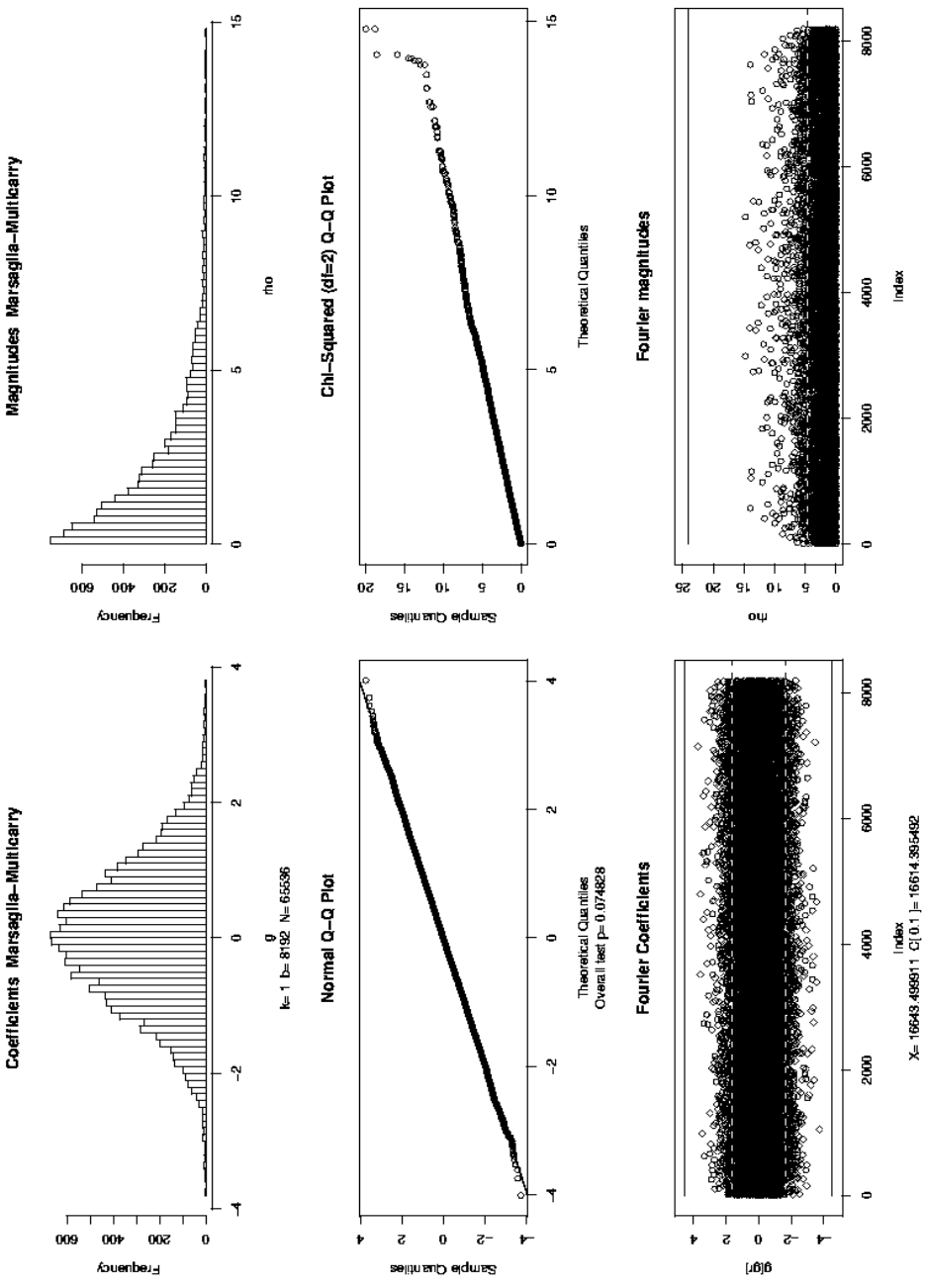
RANLIB 3d


Coefficients RANLIB
Frequency
g
k= 4 b= 8 N= 16384
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.584655
Fourier Coefficients
g[gr]
Index
X= 8162.000022 C[0.1]= 8354.440846
Magnitudes RANLIB
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
rho
Index

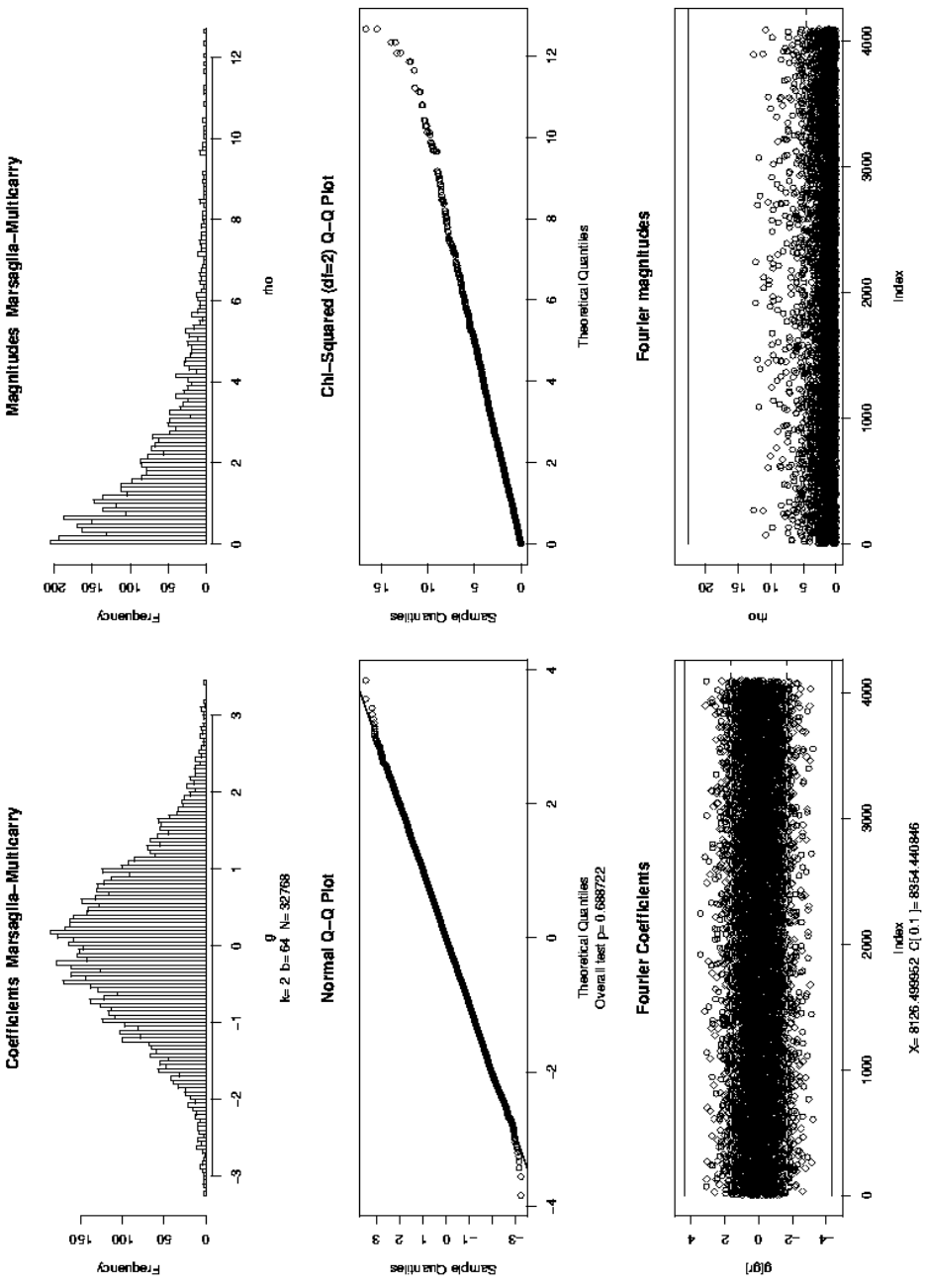
RANLIB 4d


Coefficients RANLIB
Frequency
g
k= 5 b= 4 N= 12288
Normal Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Overall test p= 0.652668
Fourier Coefficients
g[gr]
Index
X= 2020.390958 C[0.1]= 2128.395556
Magnitudes RANLIB
Frequency
rho
Chi–Squared (df=2) Q–Q Plot
Sample Quantiles
Theoretical Quantiles
Fourier magnitudes
mo
Index

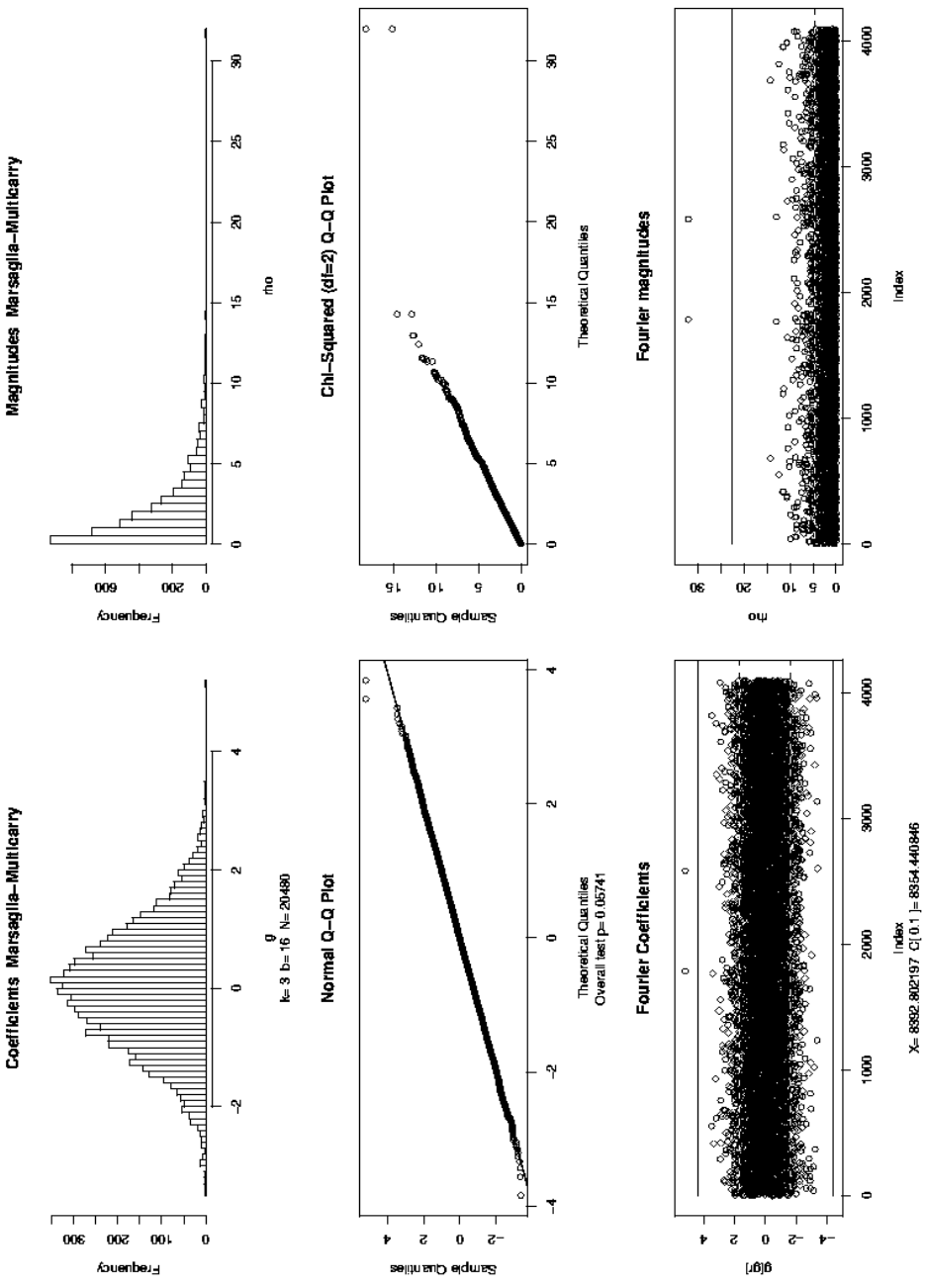
RANLIB 5d



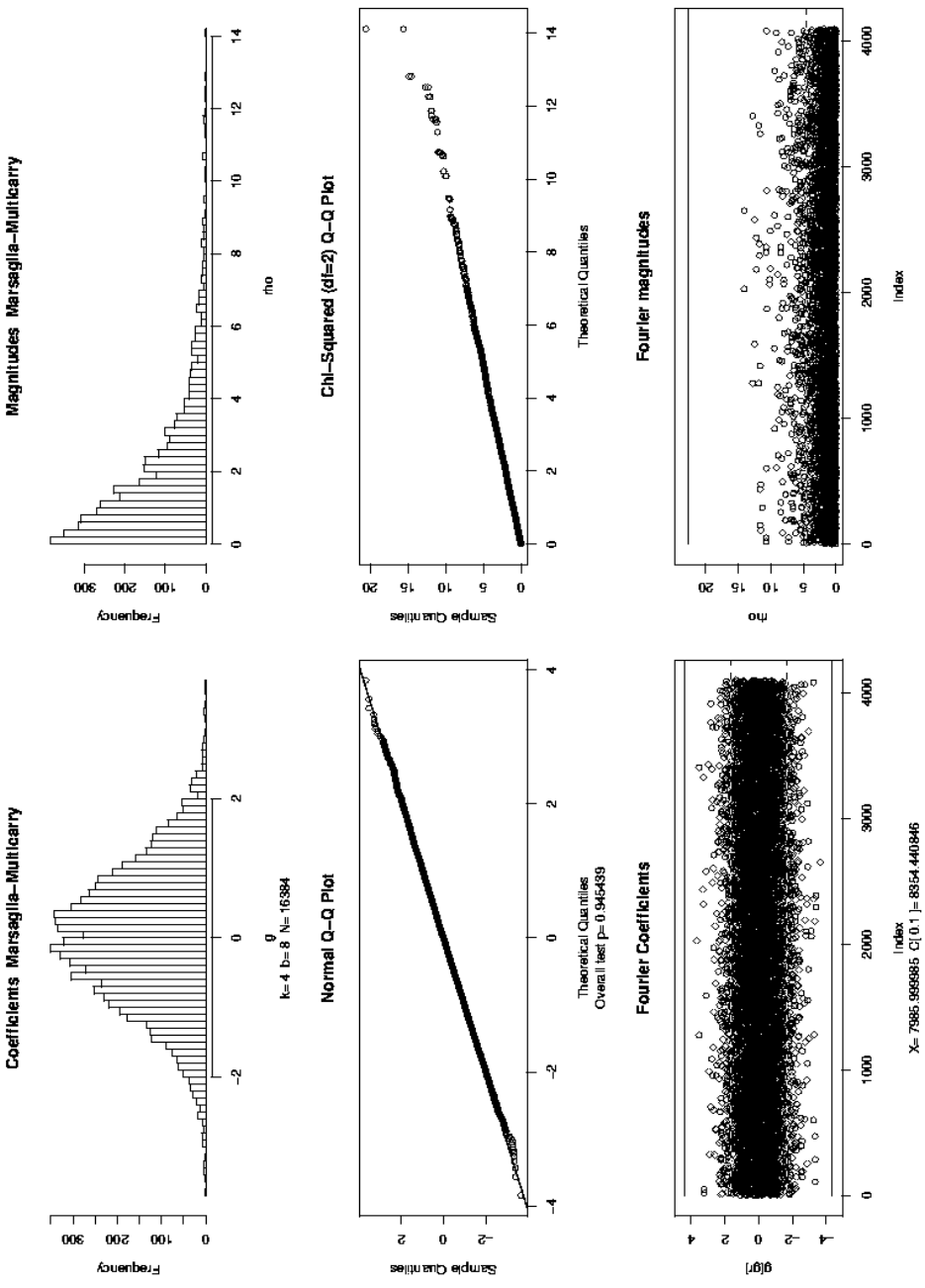
Marsaglia Multicarry k=1



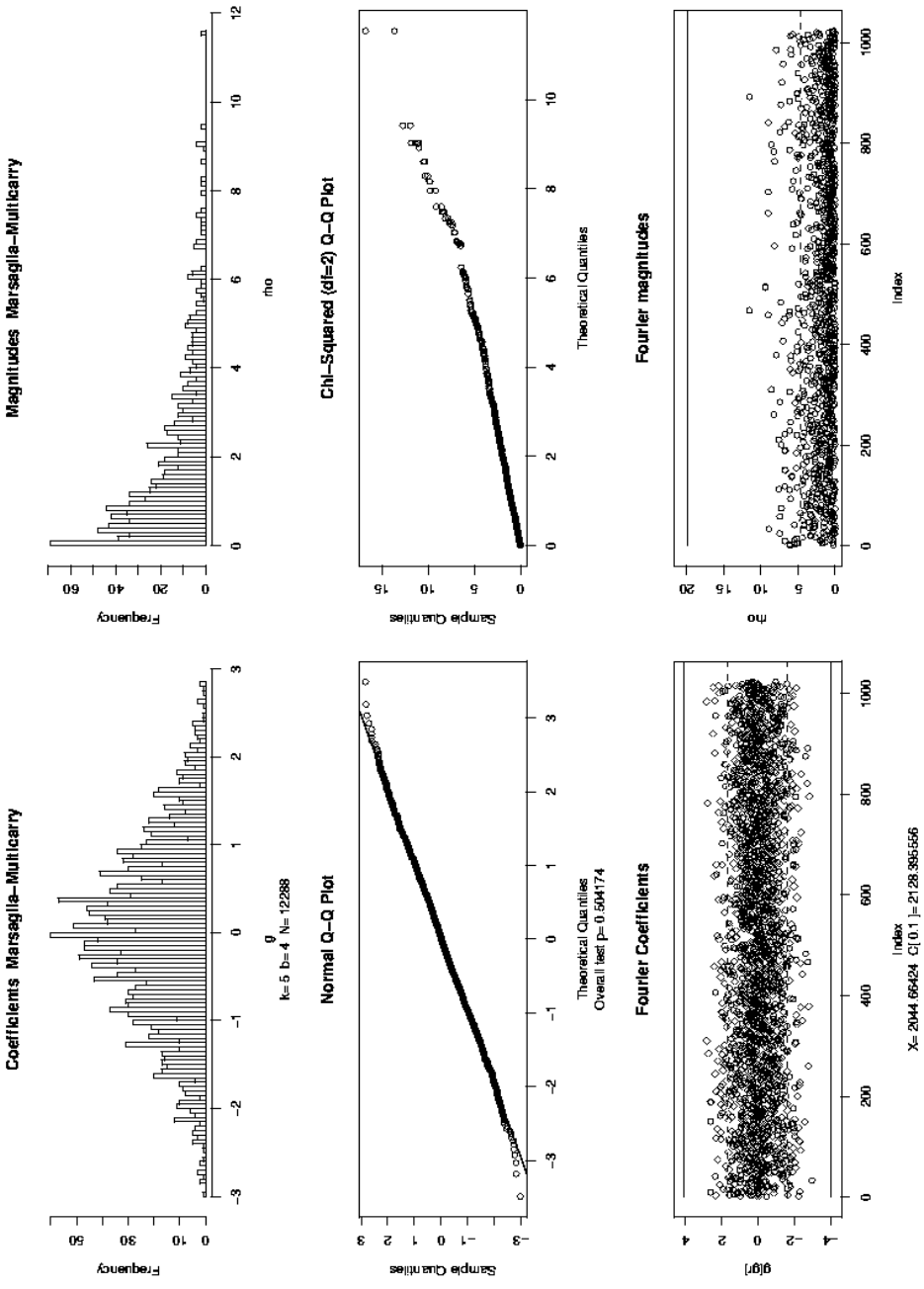
Marsaglia Multicarry k=2



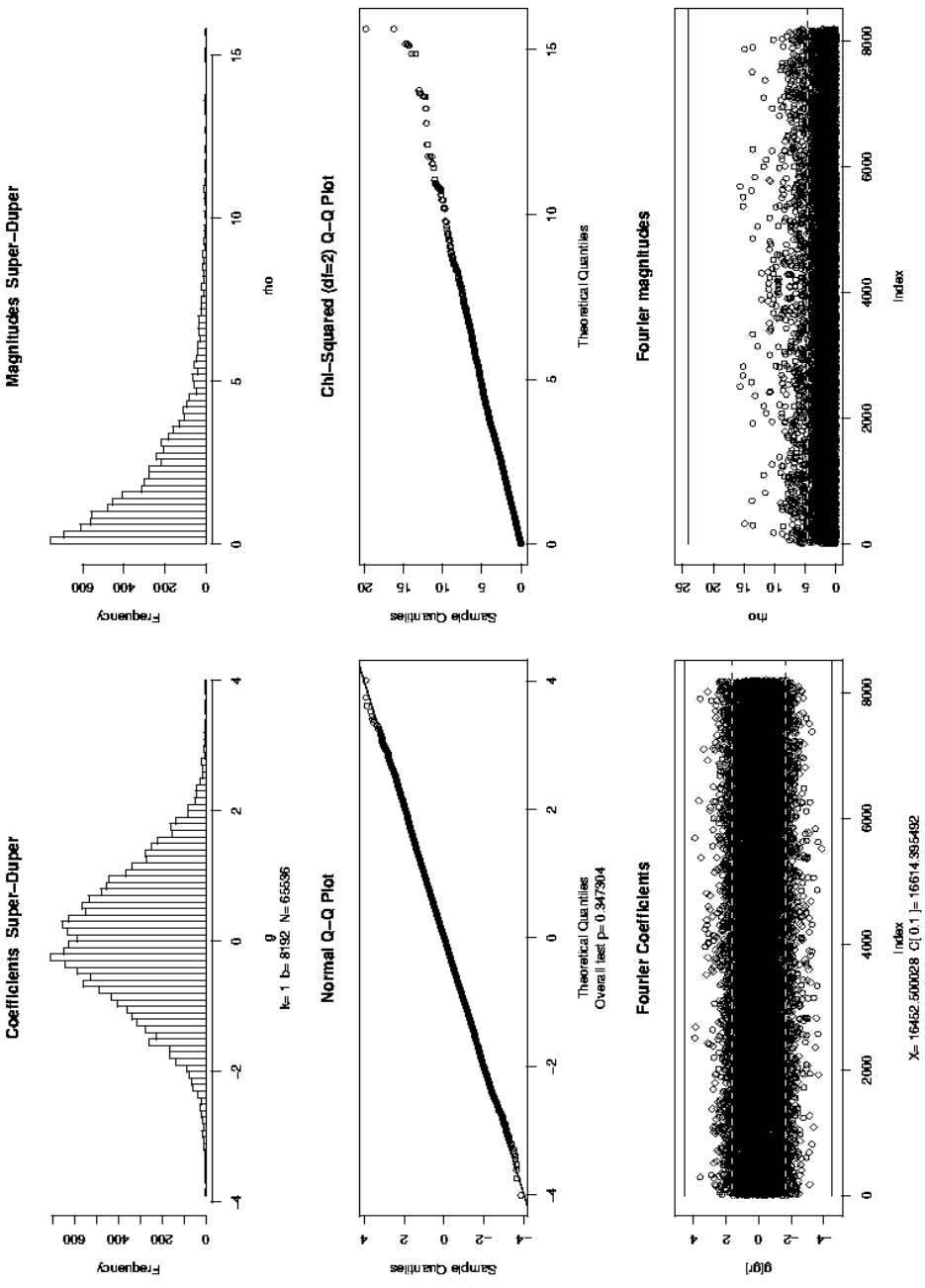
Marsaglia Multicarry k=3



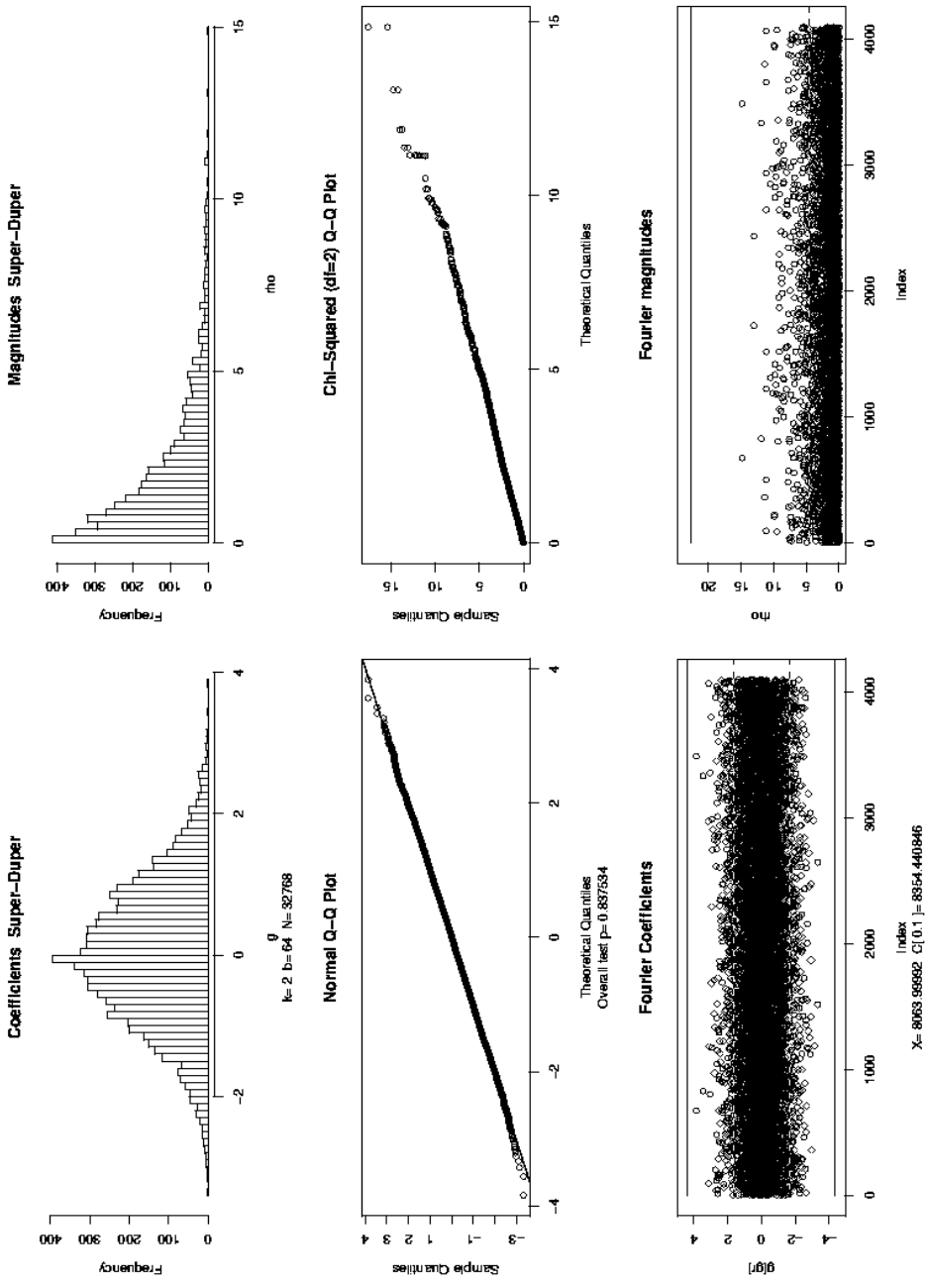
Marsaglia Multicarry k=4



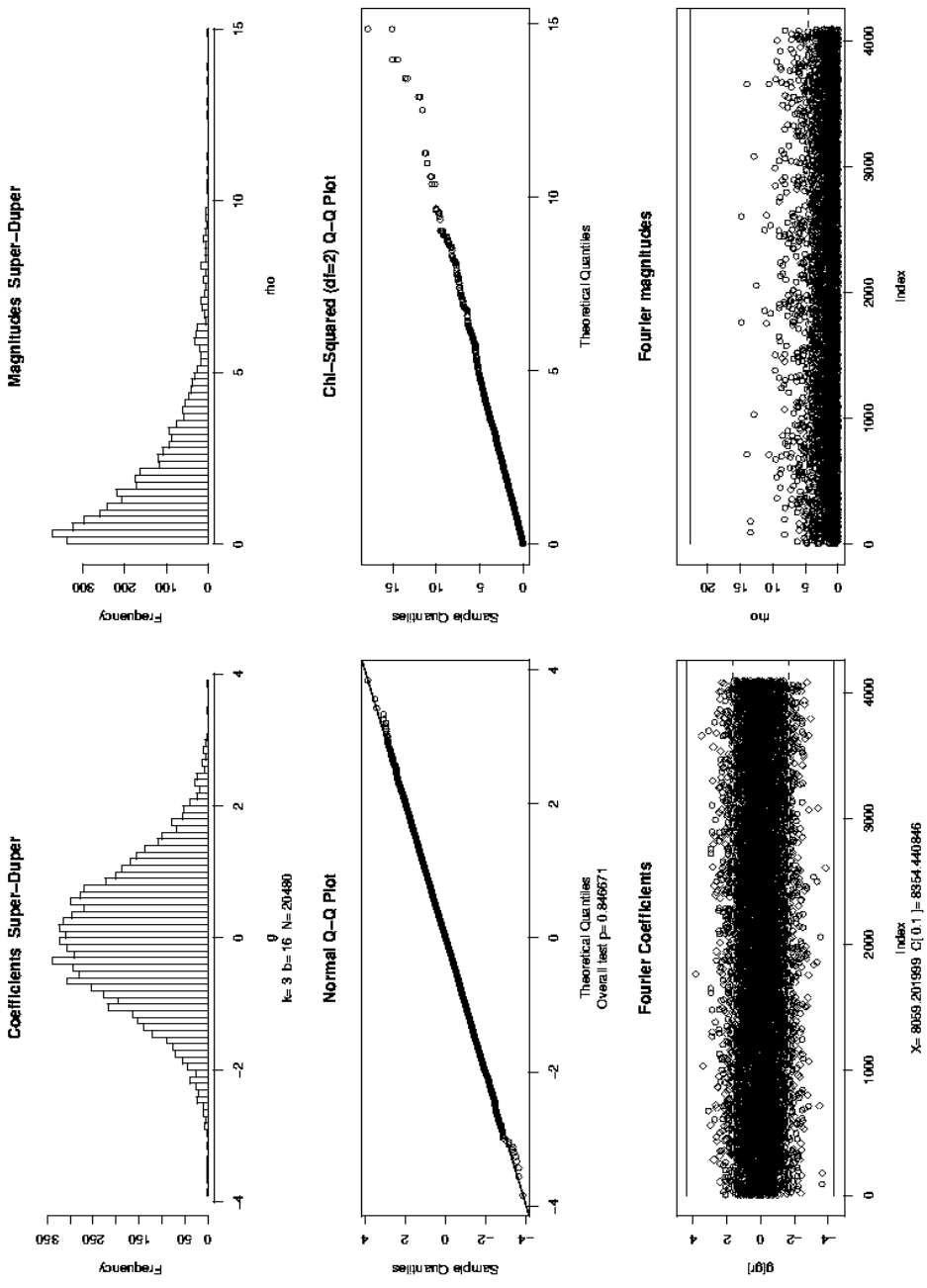
Marsaglia Multicarry k=5



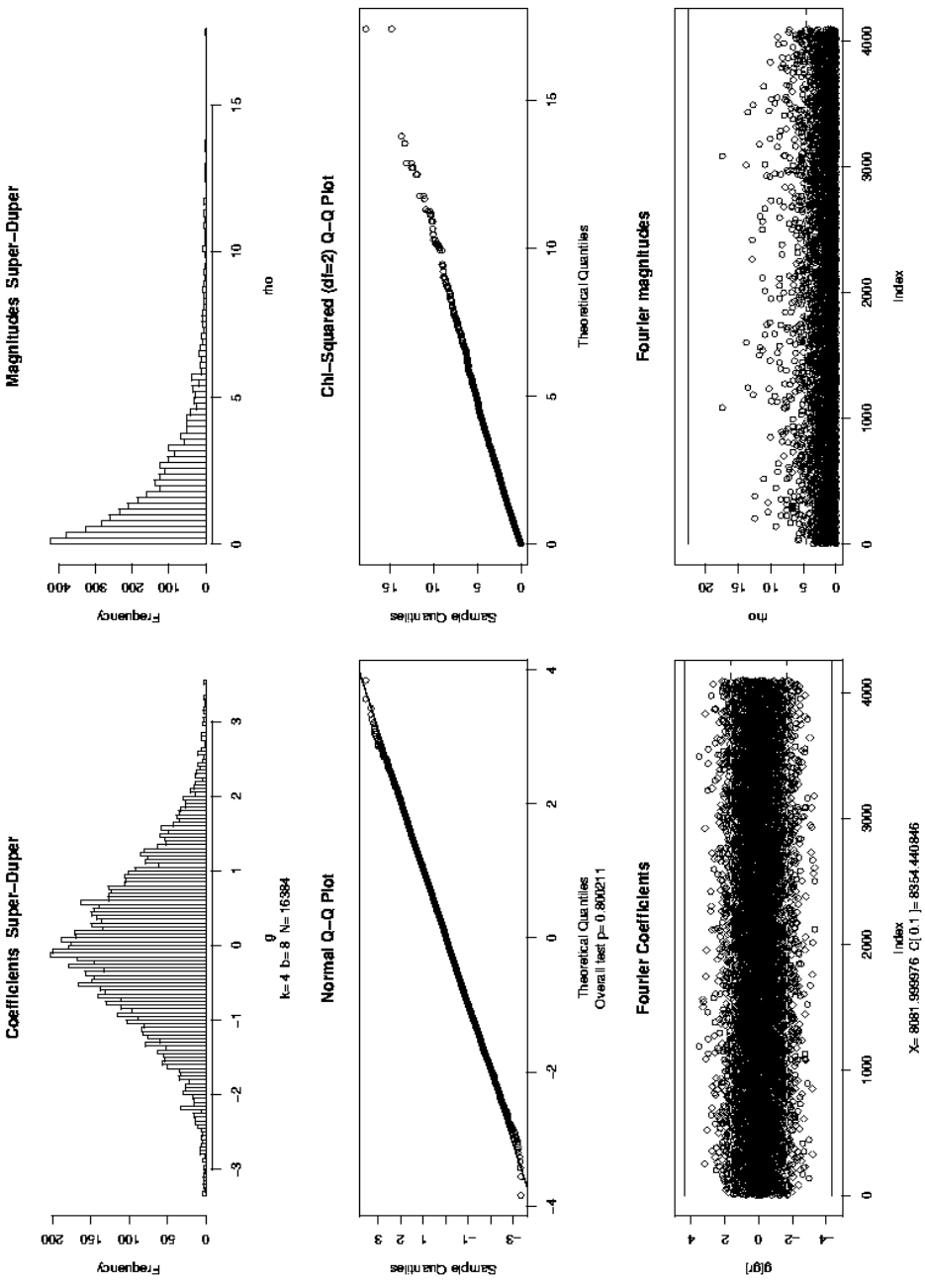
Super-Duper k=1



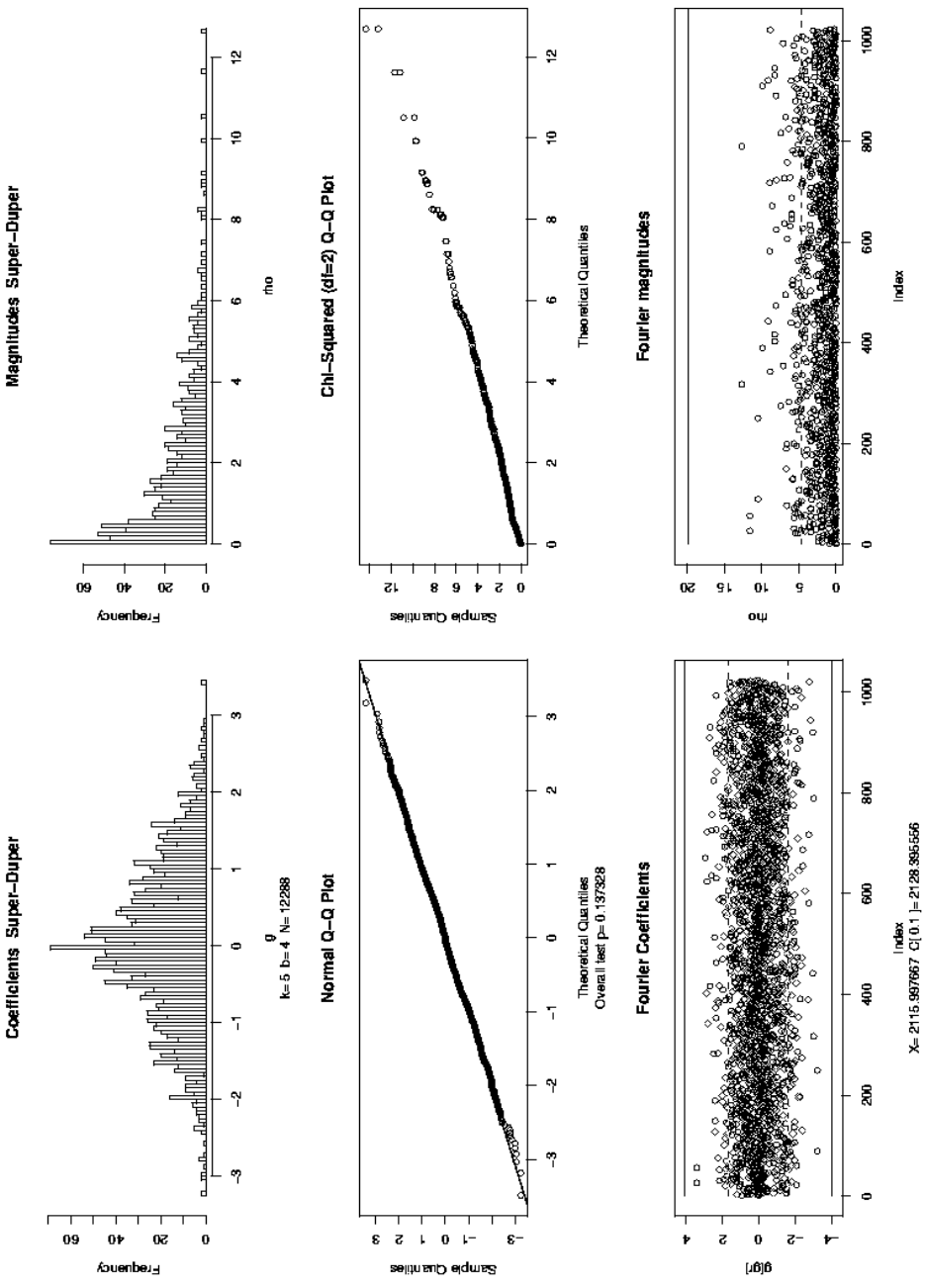
Super-Duper k=2



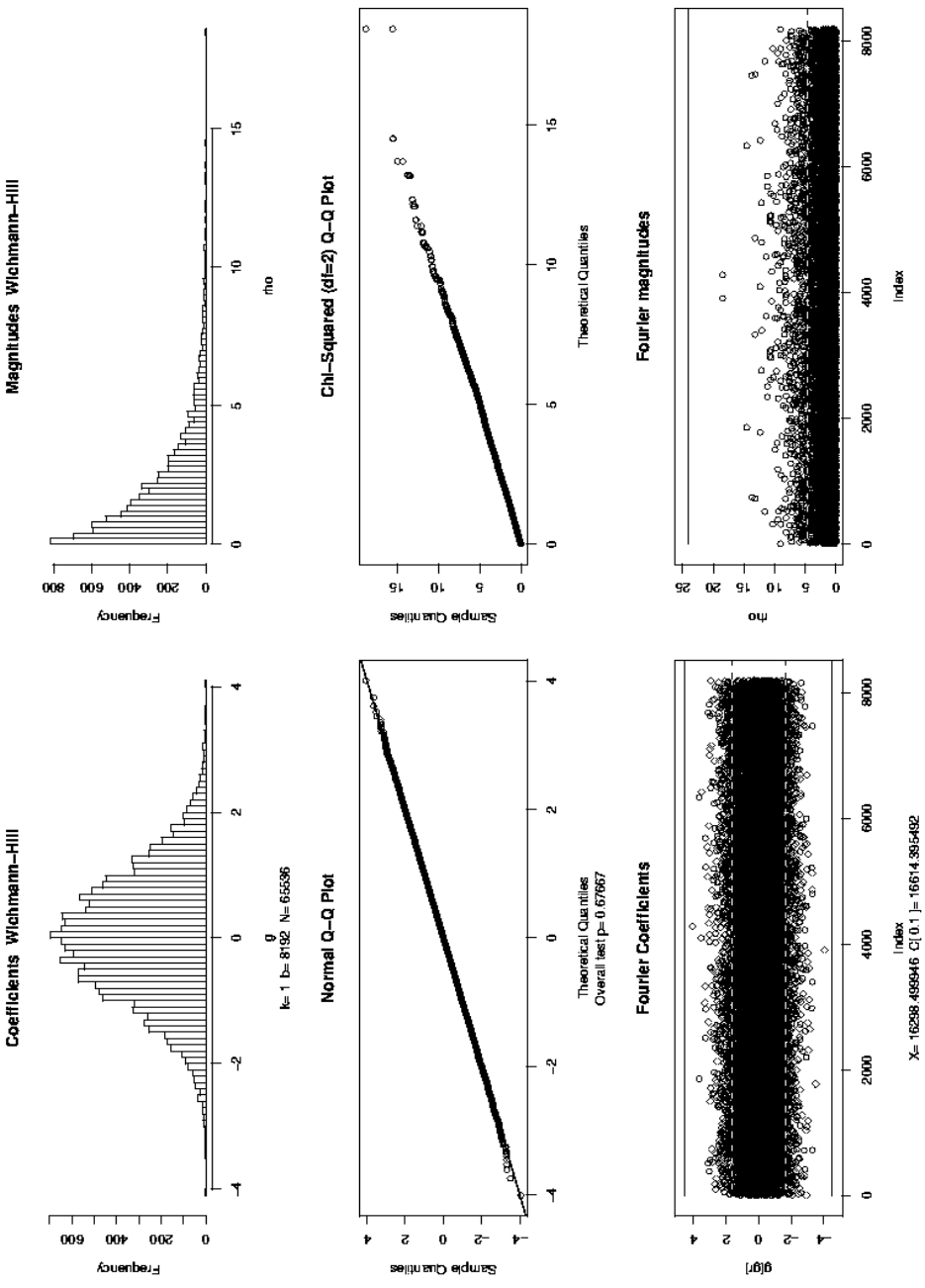
Super-Duper k=3



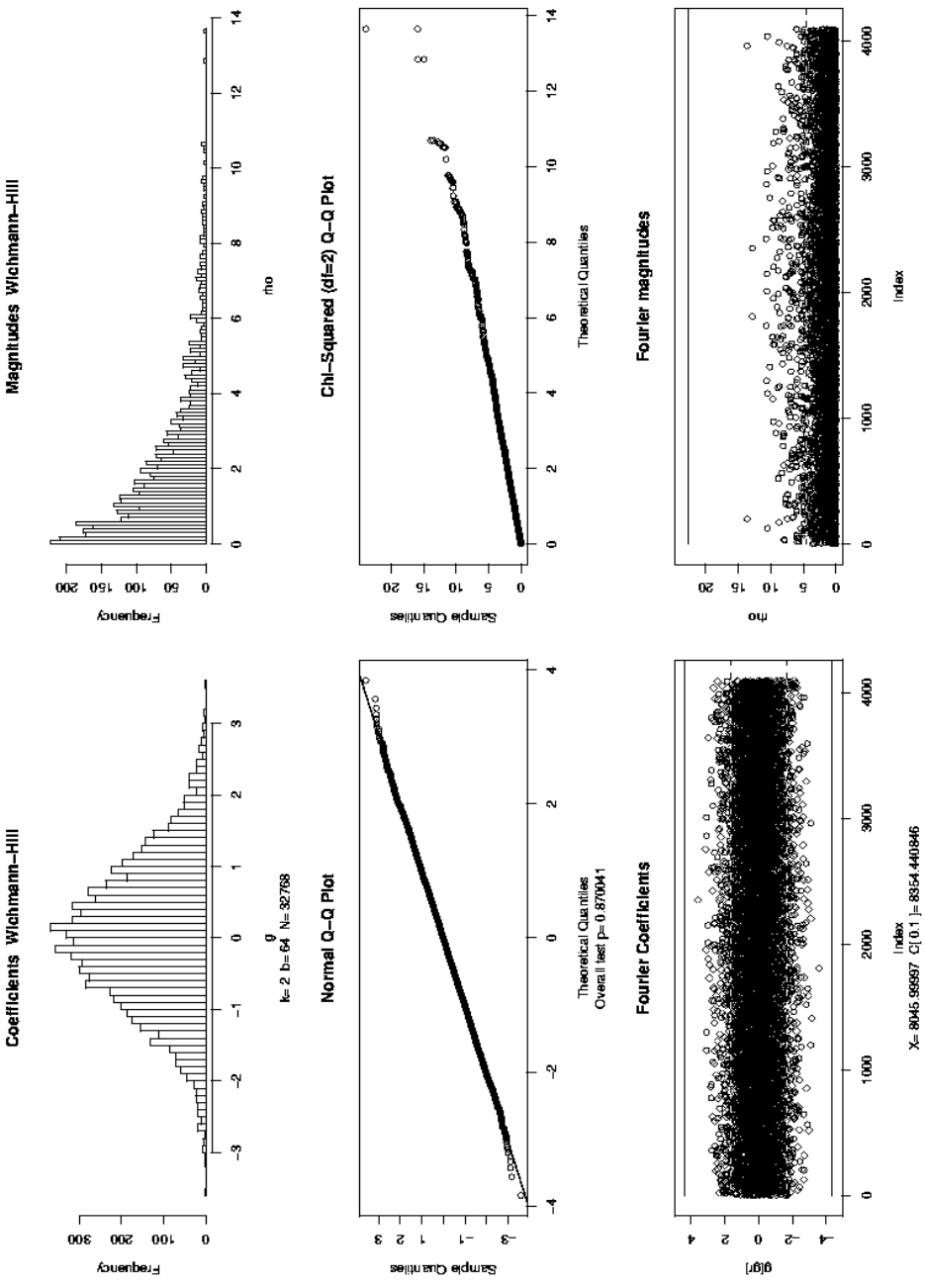
Super-Duper k=4



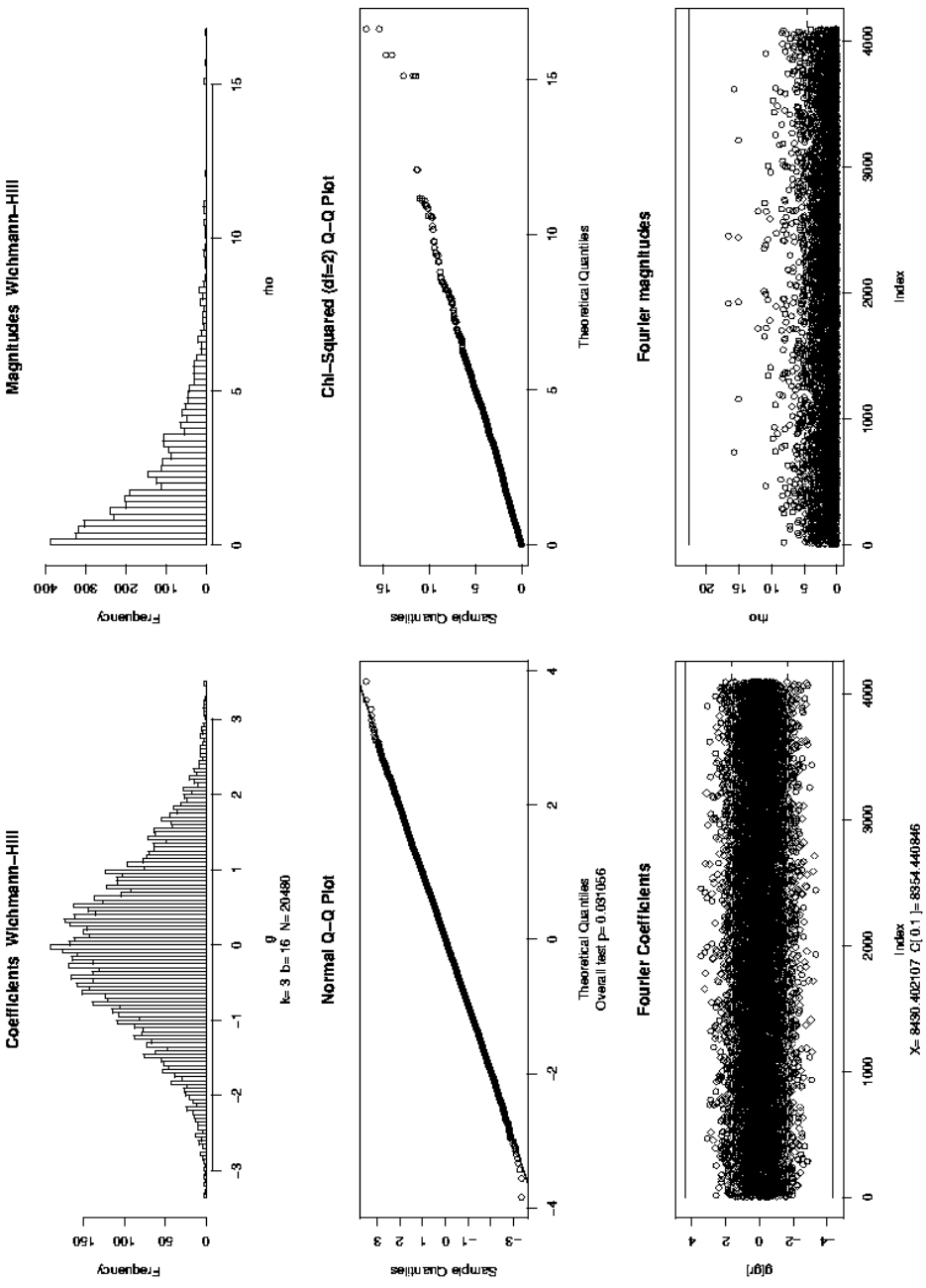
Super-Duper k=5



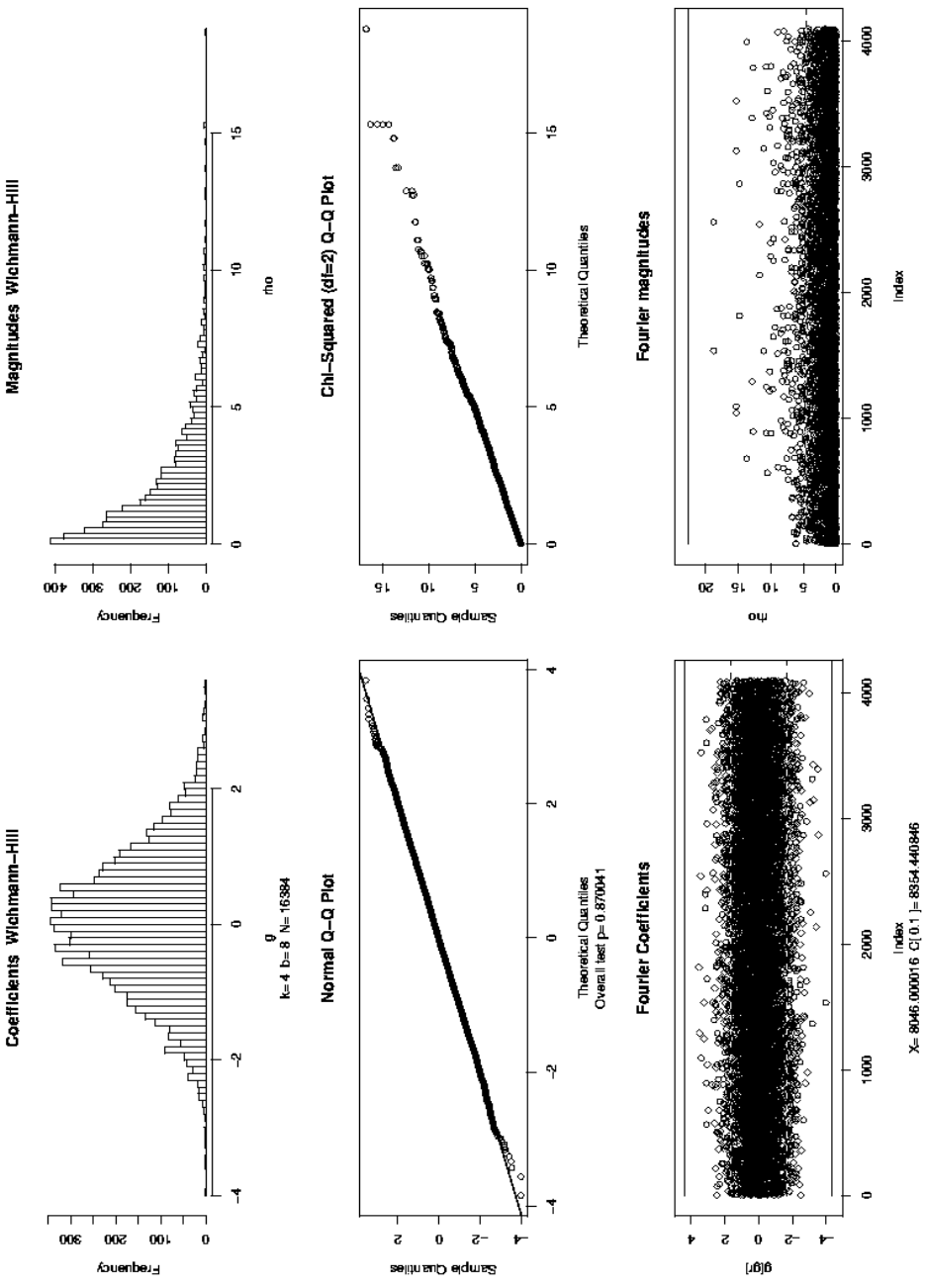
Wichmann Hill k=1



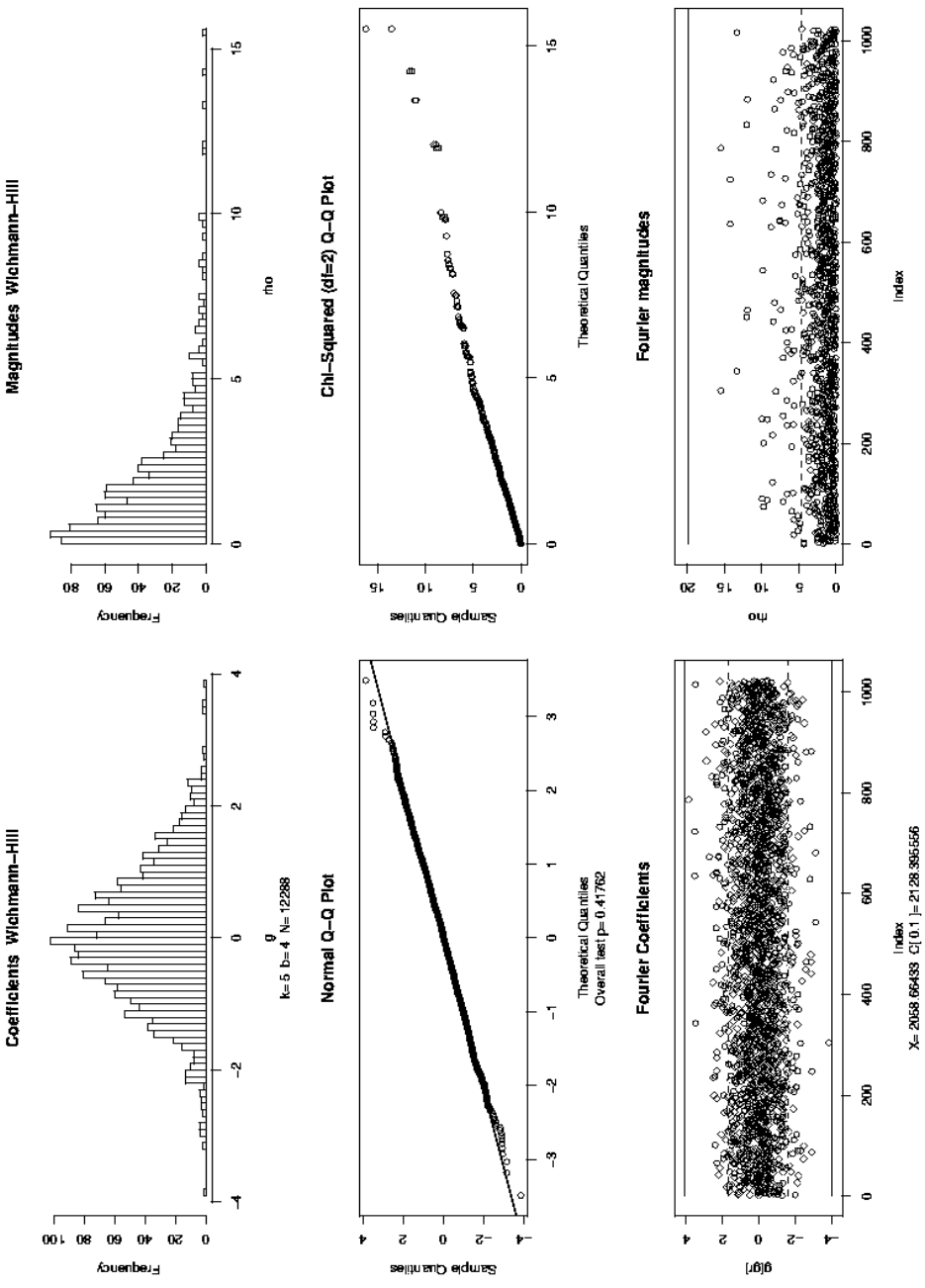
Wichmann Hill k=2



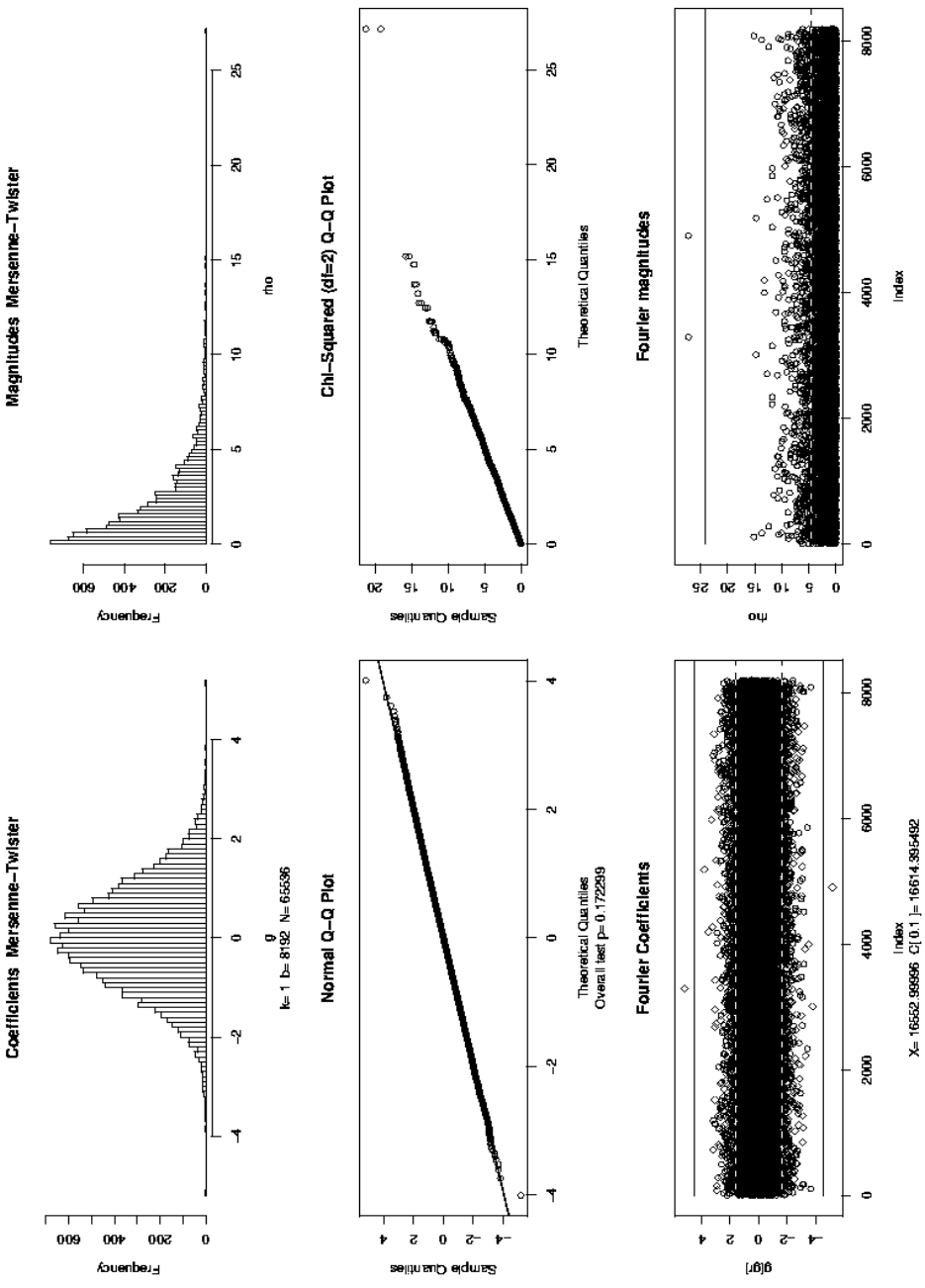
Wichmann Hill k=3



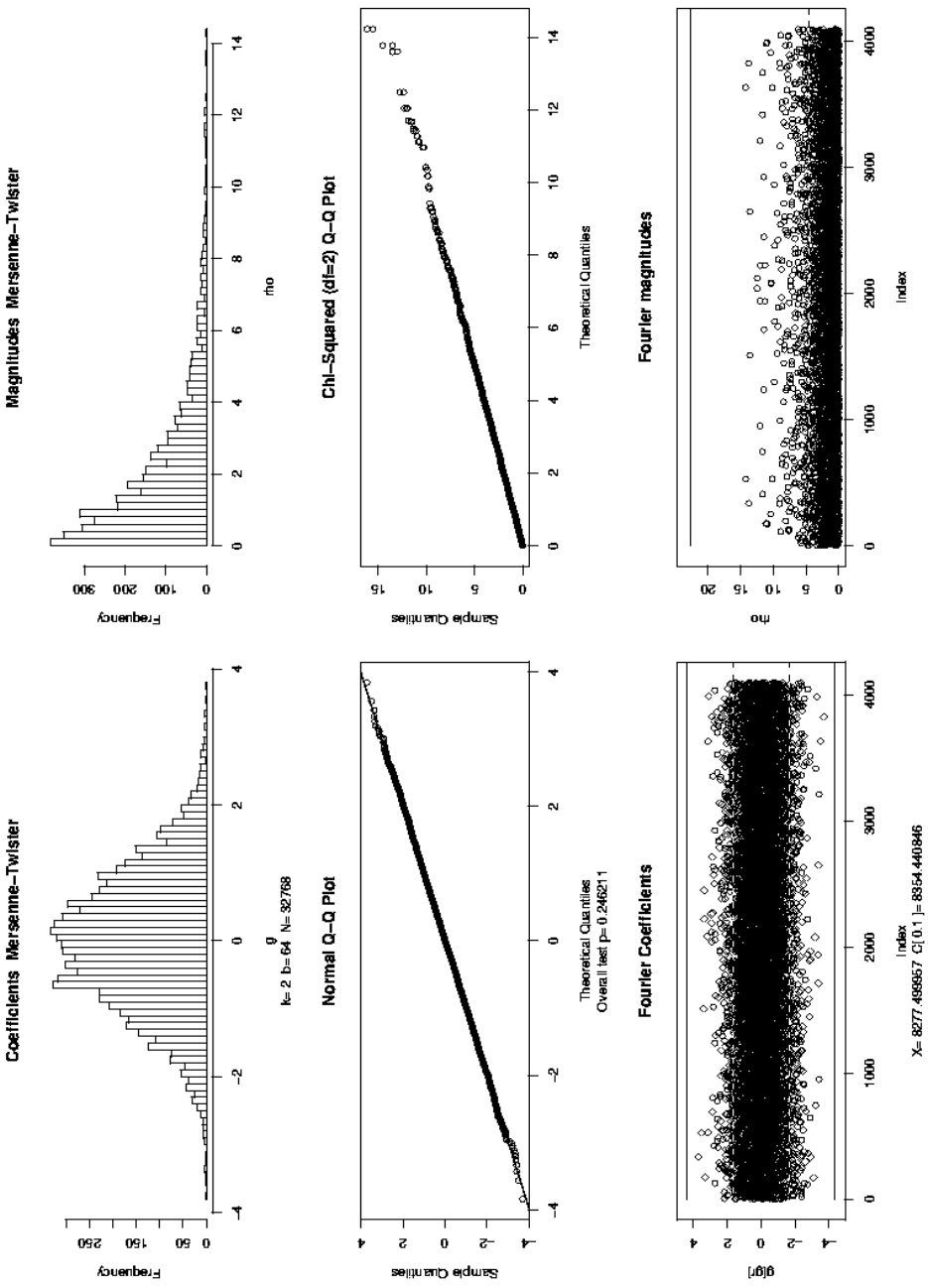
Wichmann Hill k=4



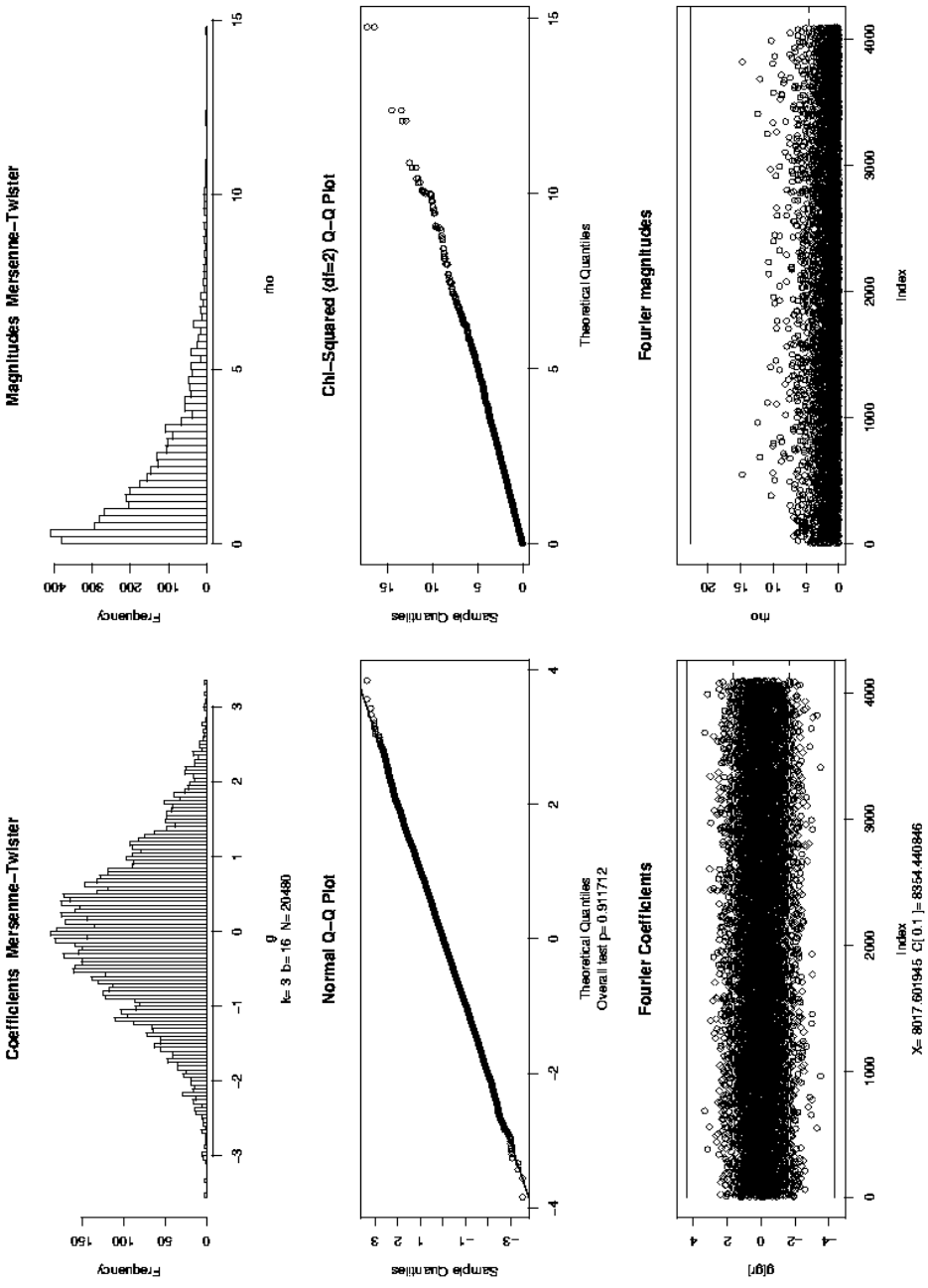
Wichmann Hill k=5



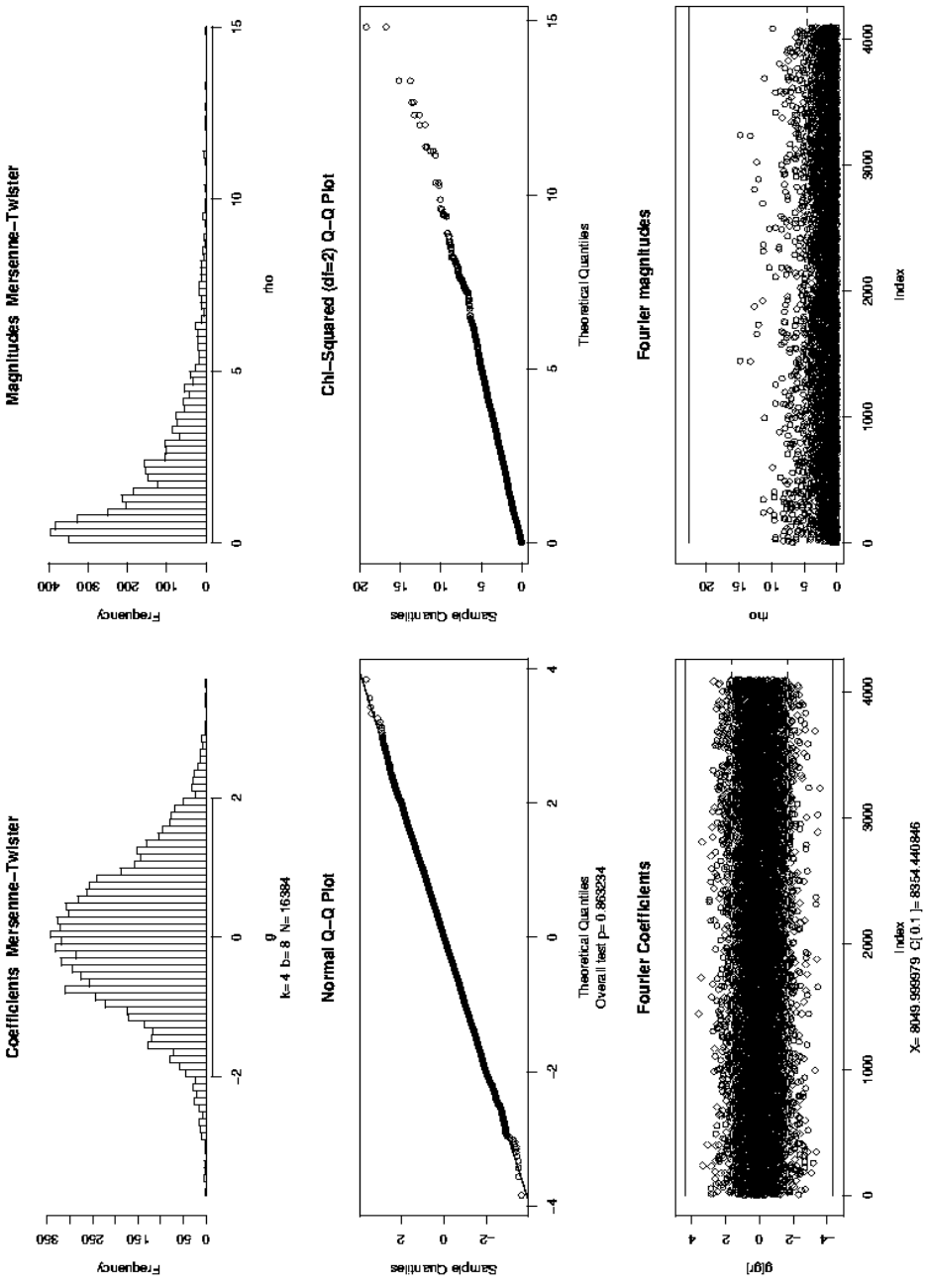
Mersenne Twister k=1



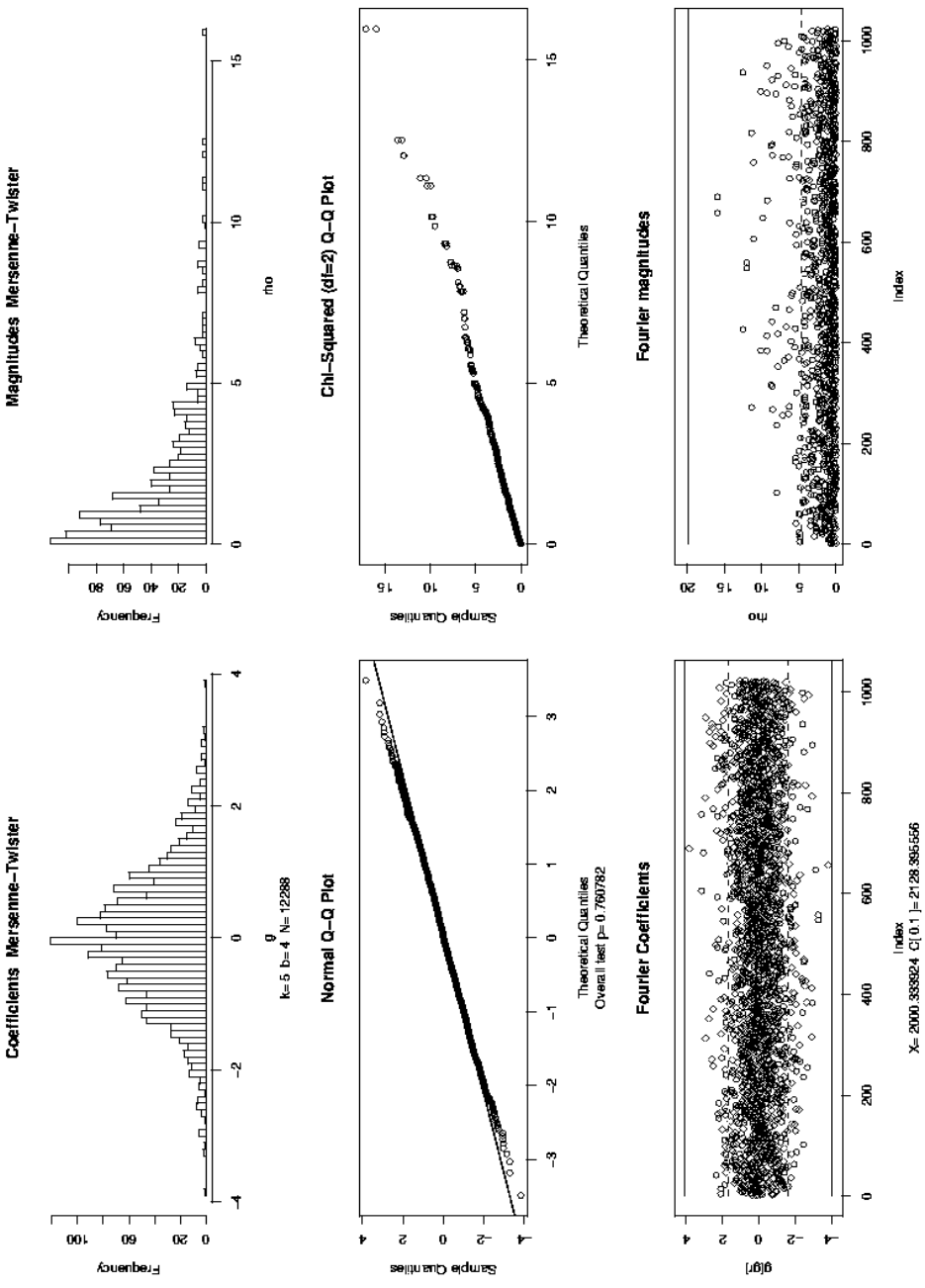
Mersenne Twister k=2



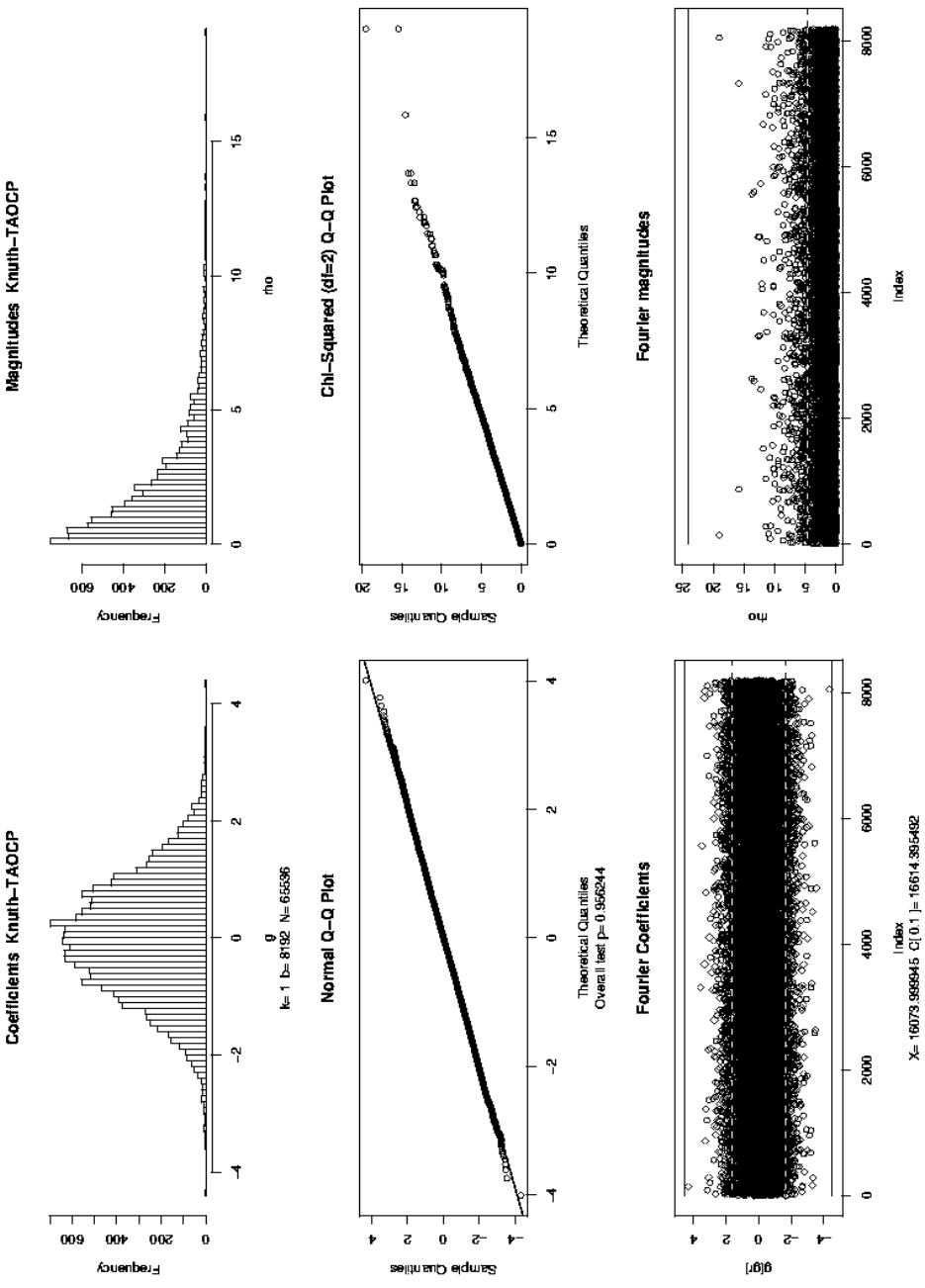
Mersenne Twister k=3



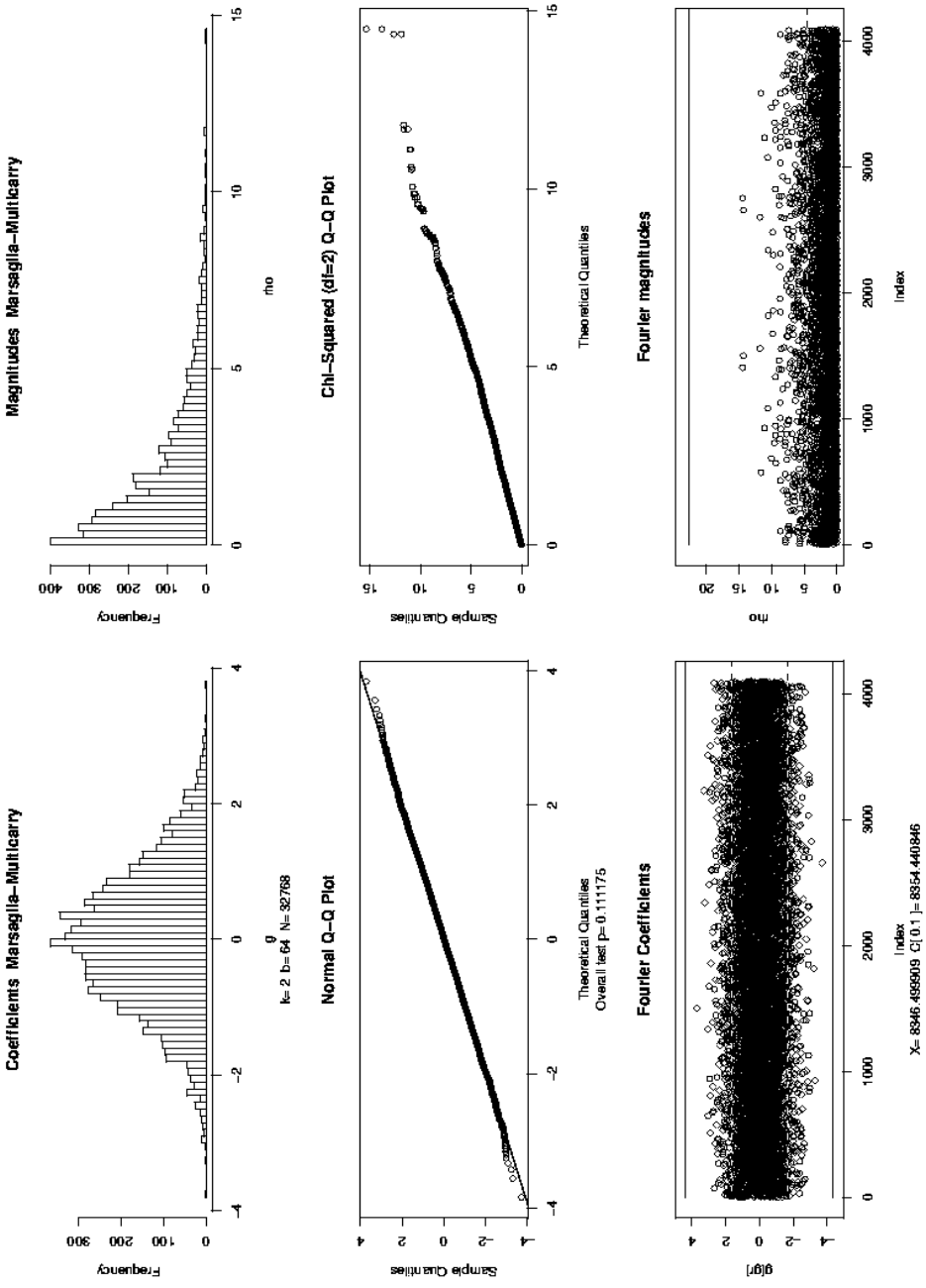
Mersenne Twister k=4



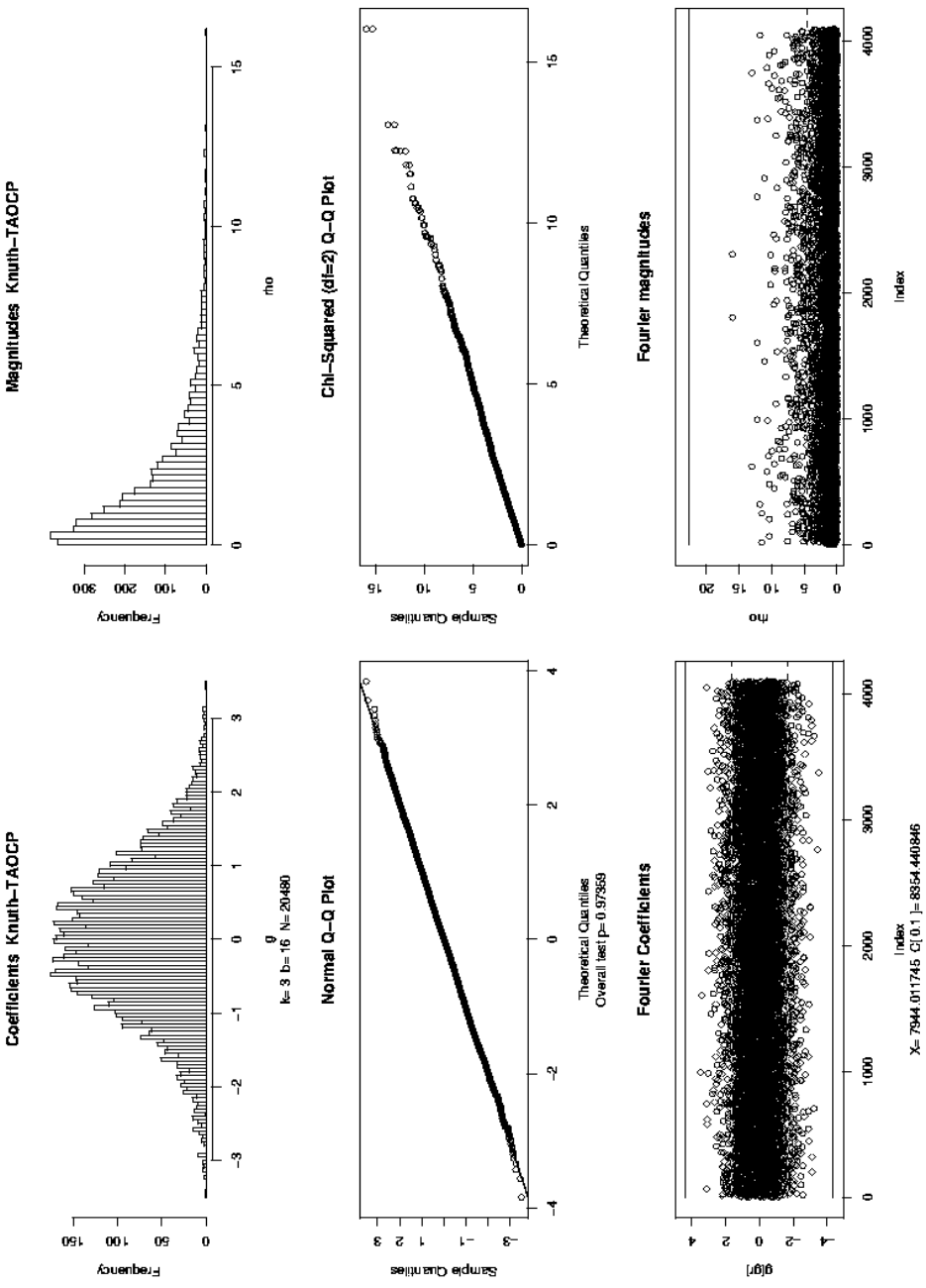
Mersenne Twister k=5



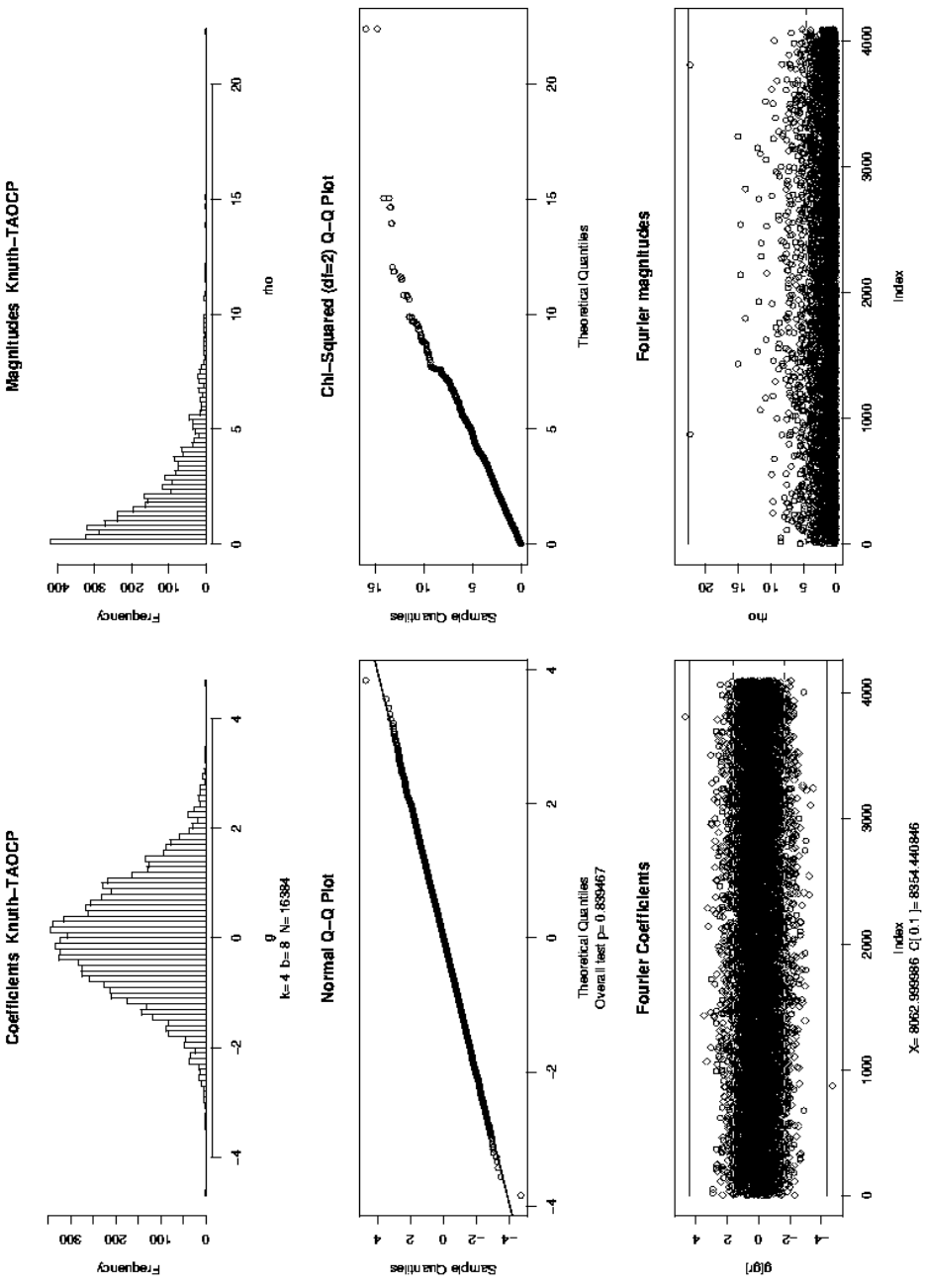
Knuth TAOCP k=1



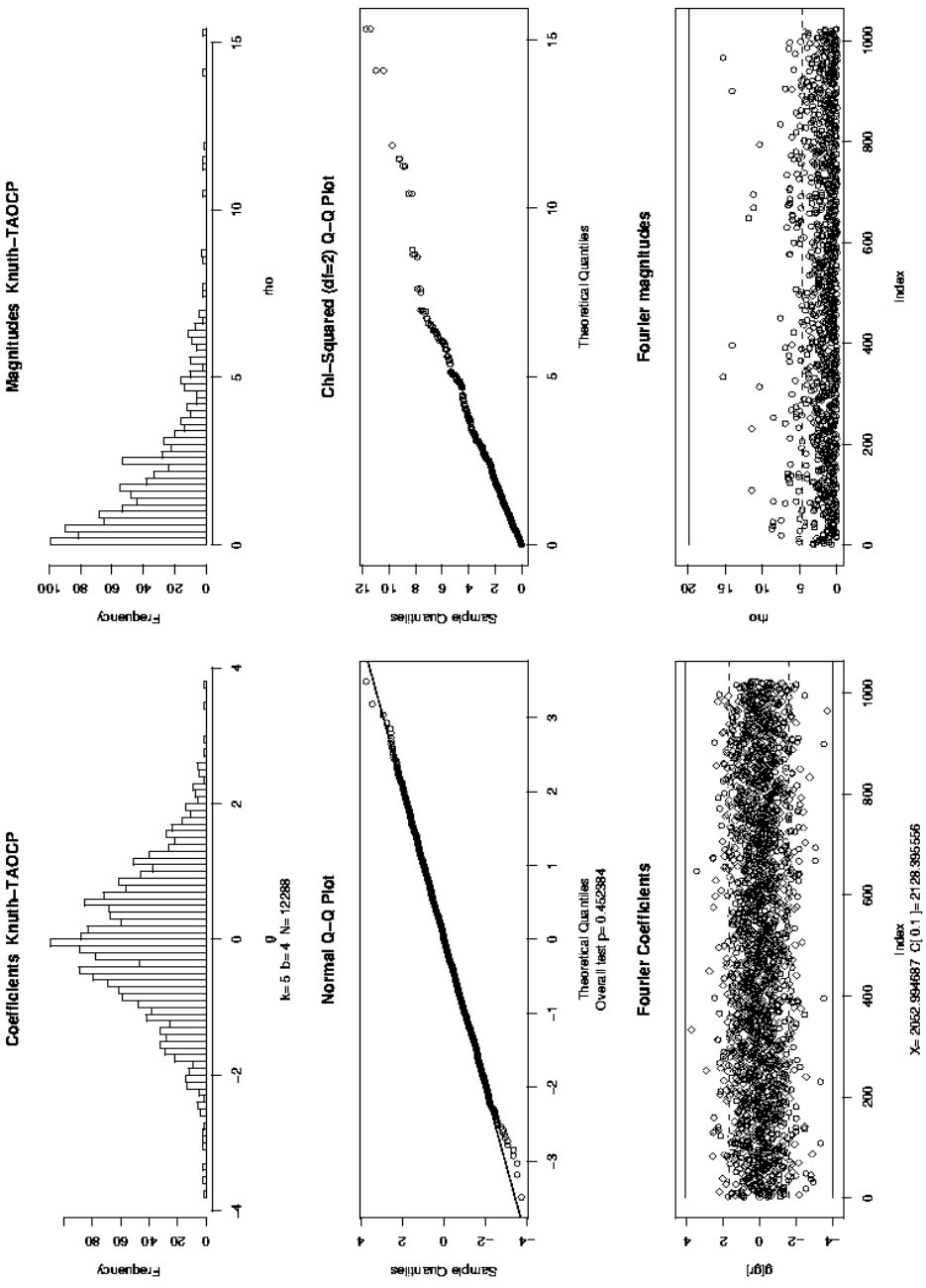
Knuth TAOCP k=2



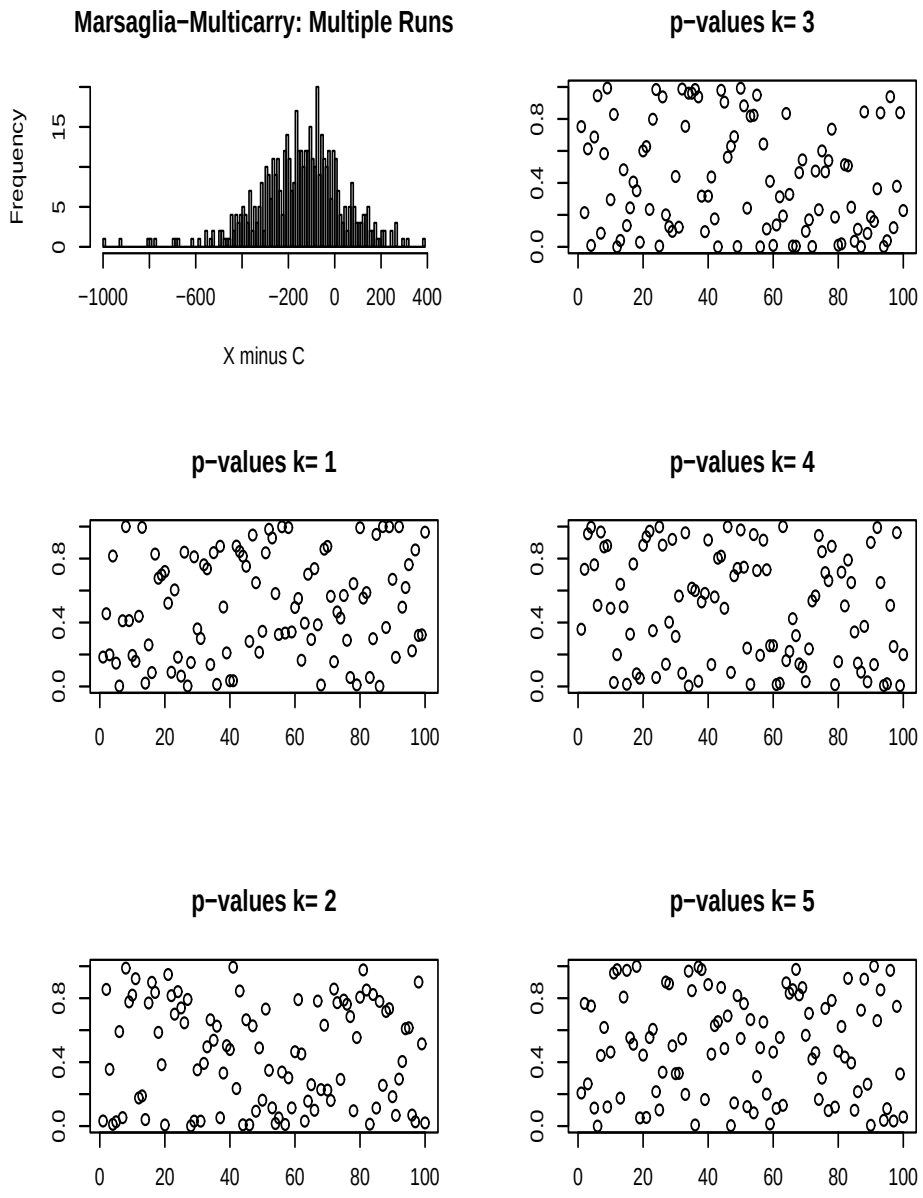
Knuth TAOCP k=3



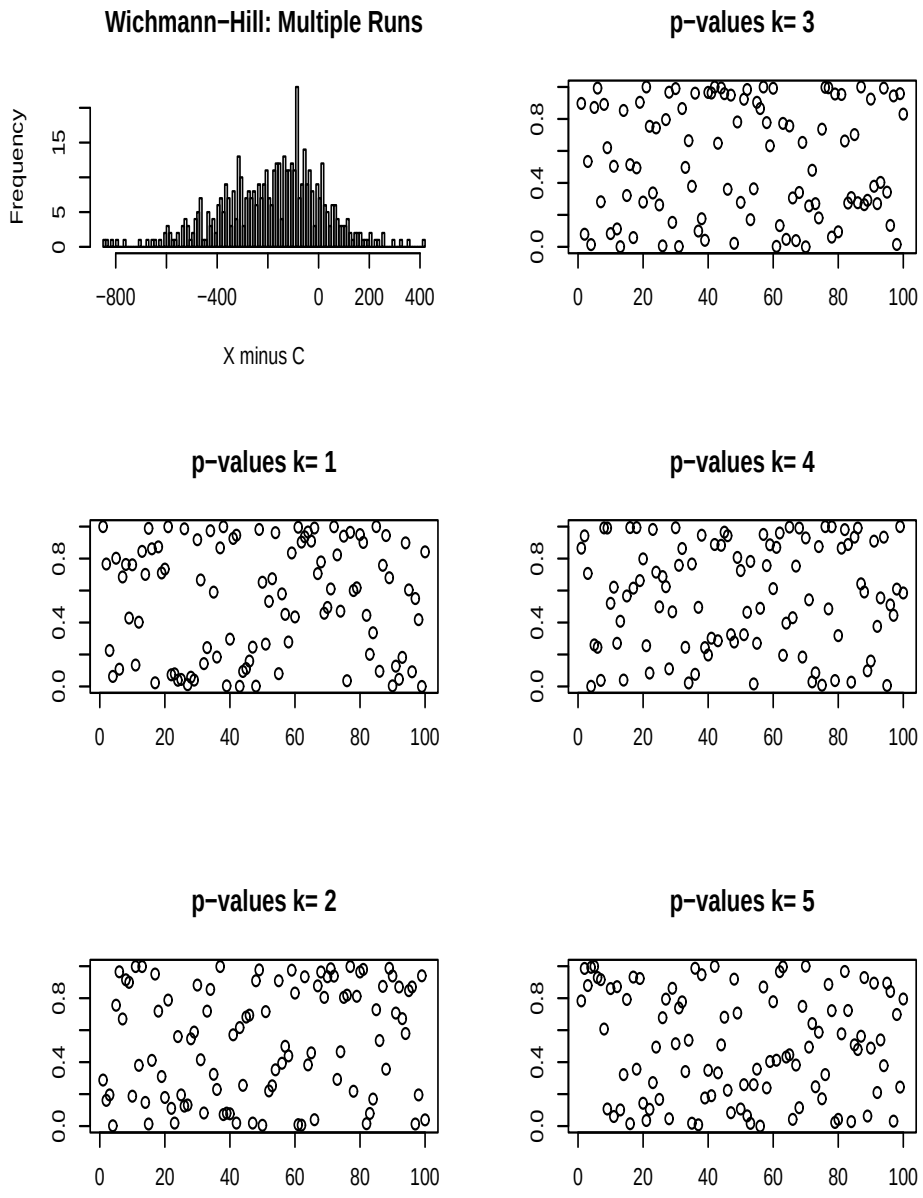
Knuth TAOCP k=4



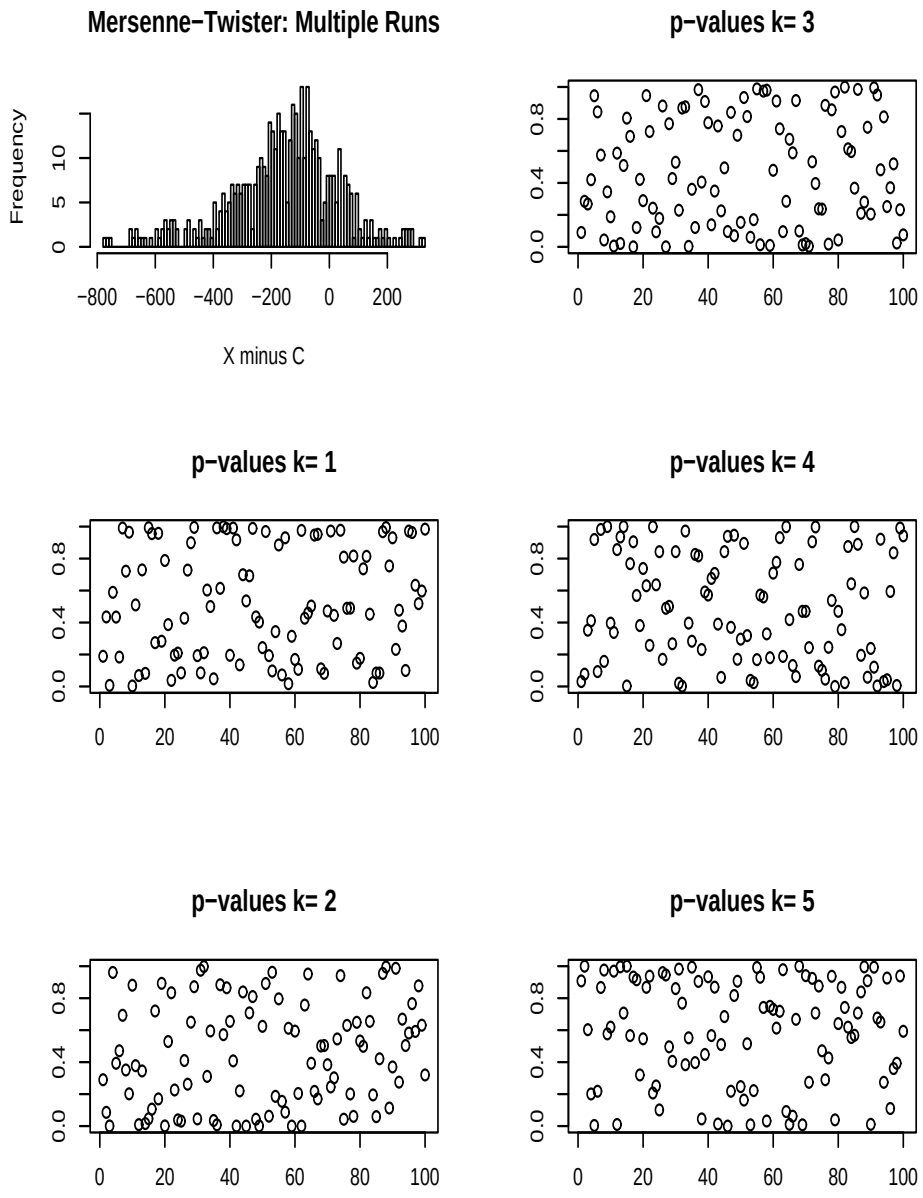
Knuth TAOCP k=5



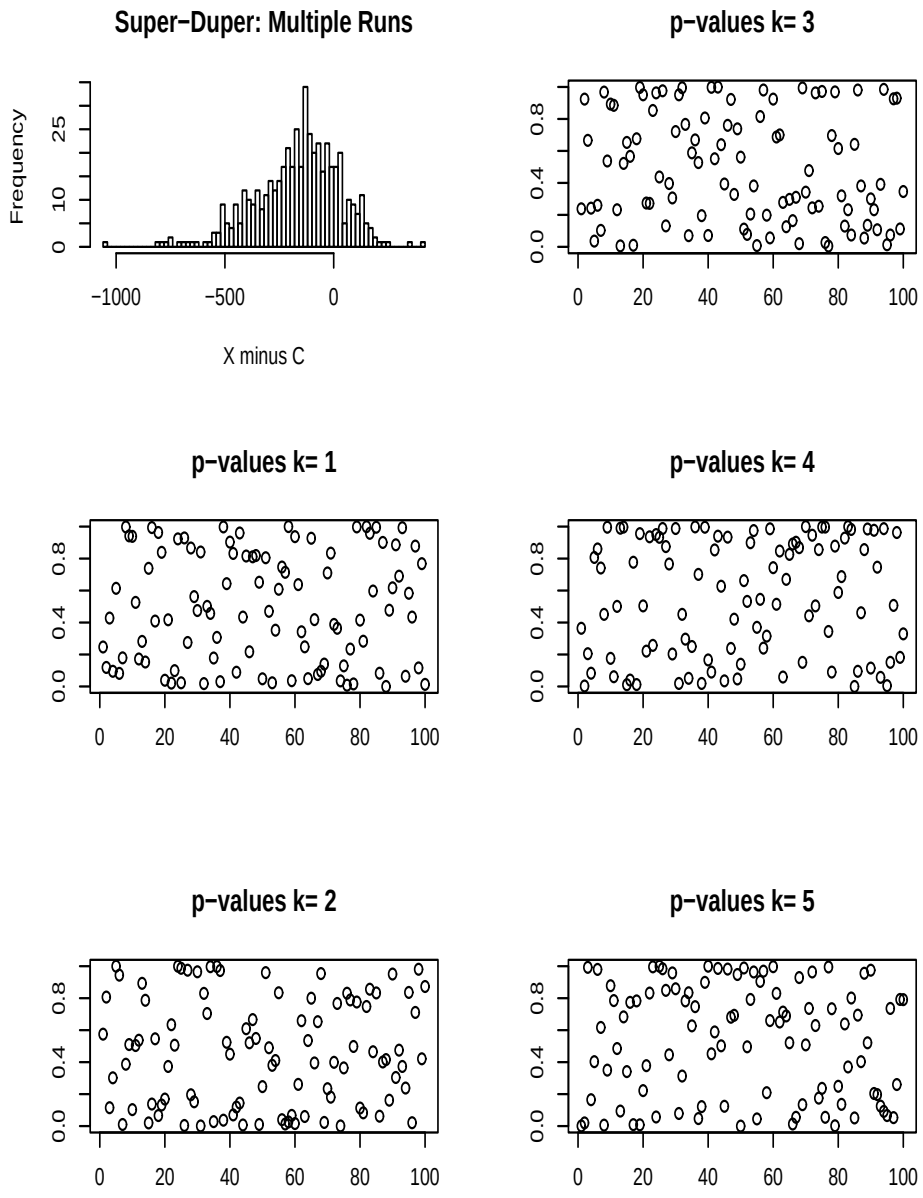
Marsaglia Multicarry 100 Repeated Runs



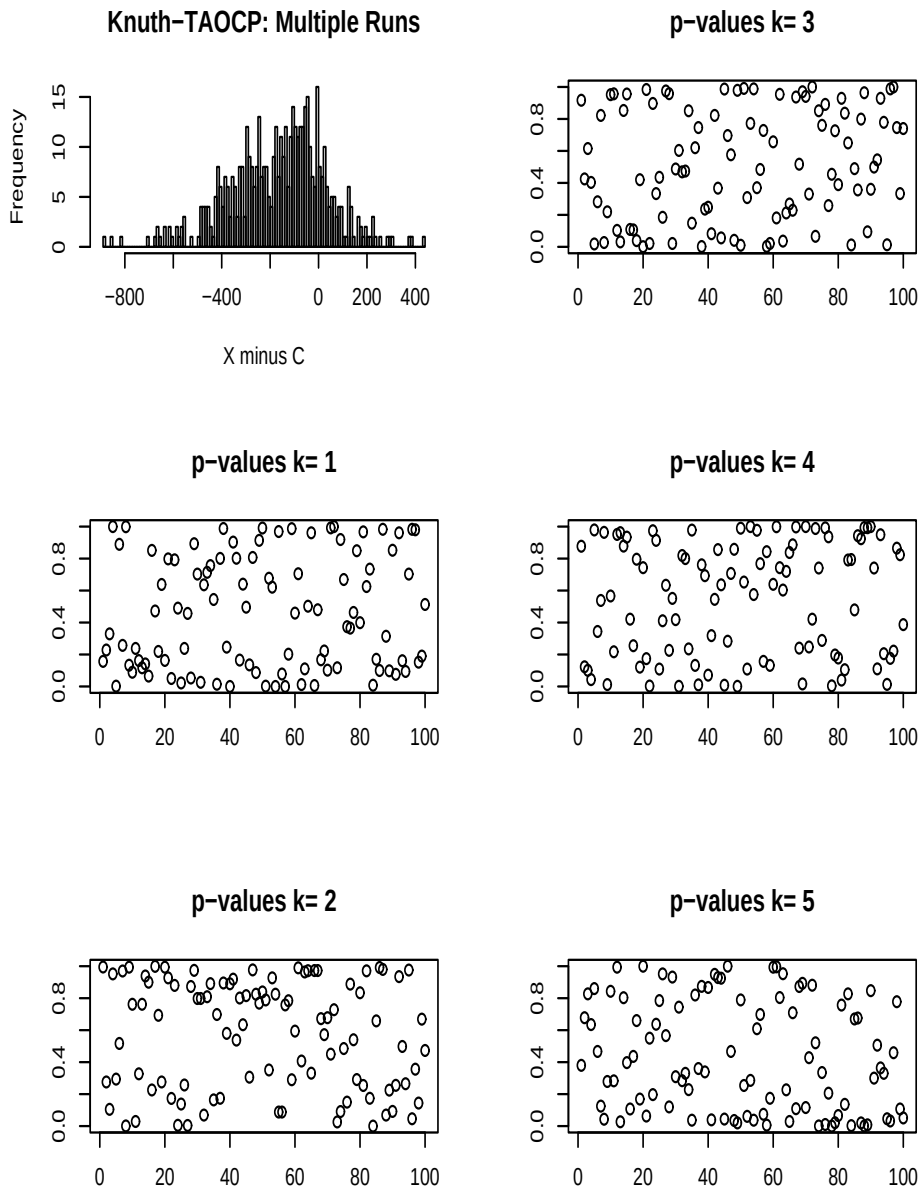
Wichmann Hill 100 Repeated Runs



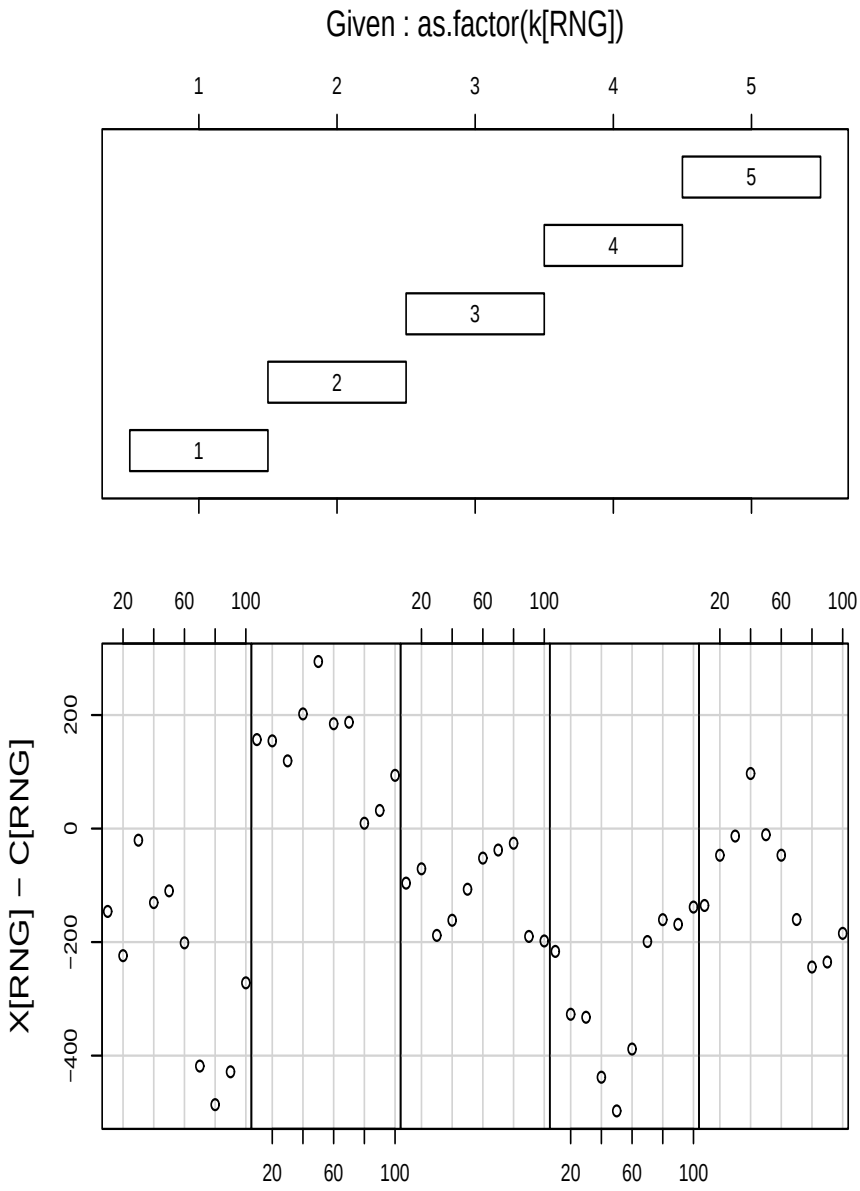
Mersenne Twister 100 Repeated Runs



Super-Duper 100 Repeated Runs

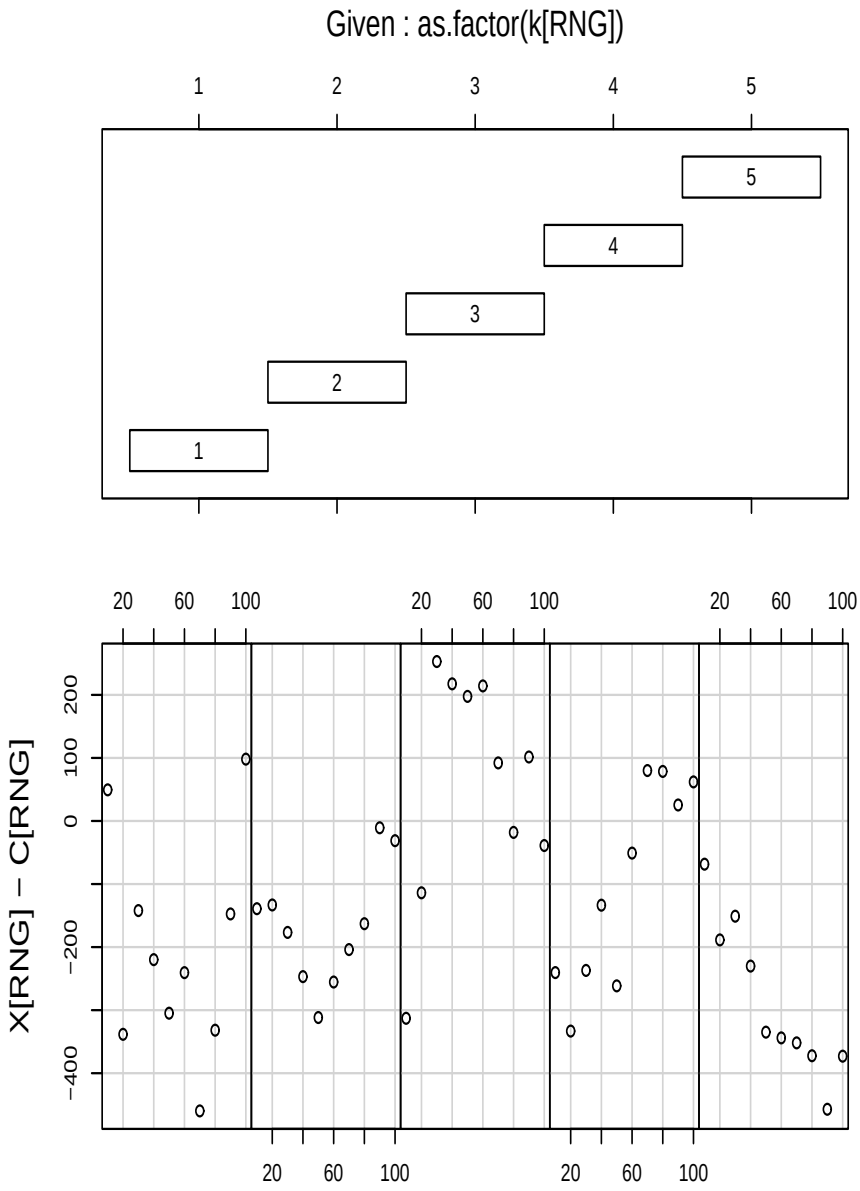


Knuth TAOCP 100 Repeated Runs



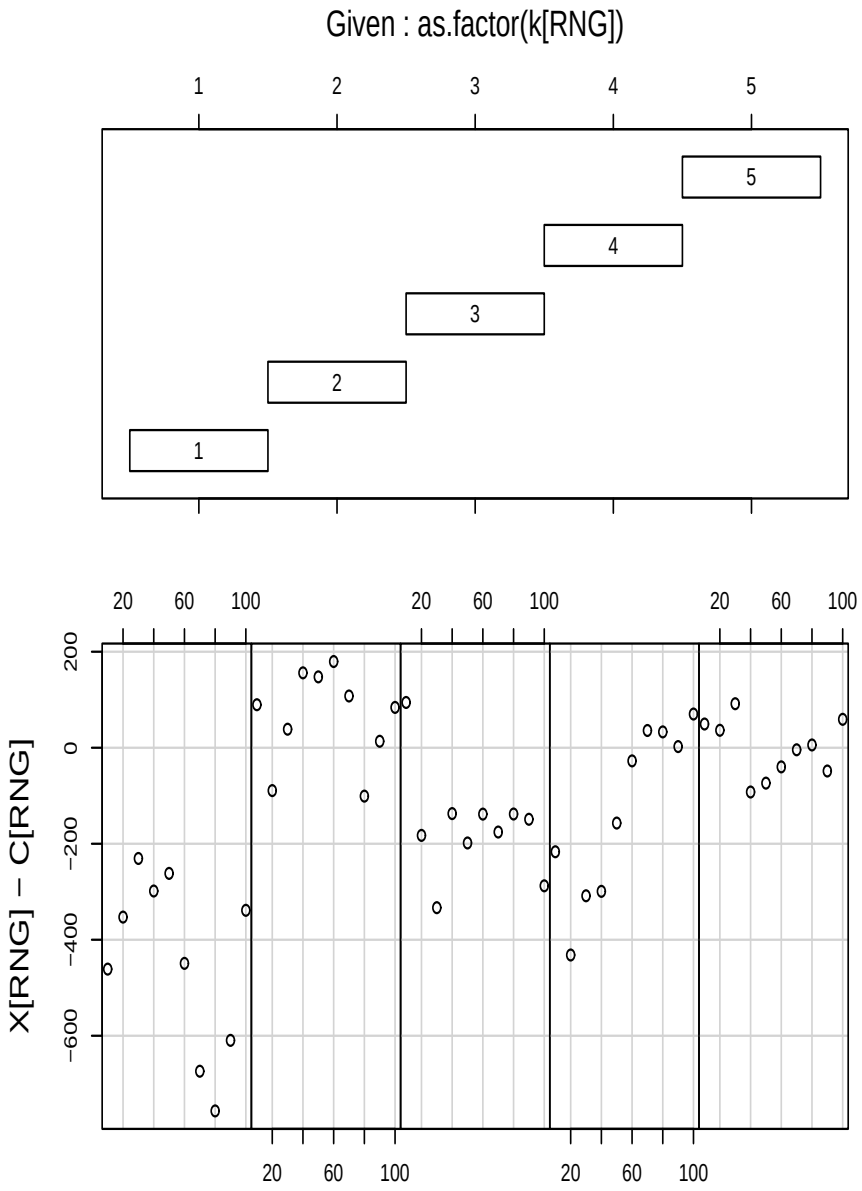
Wichmann−Hill: Multiple N

Wichmann Hill Increasing N



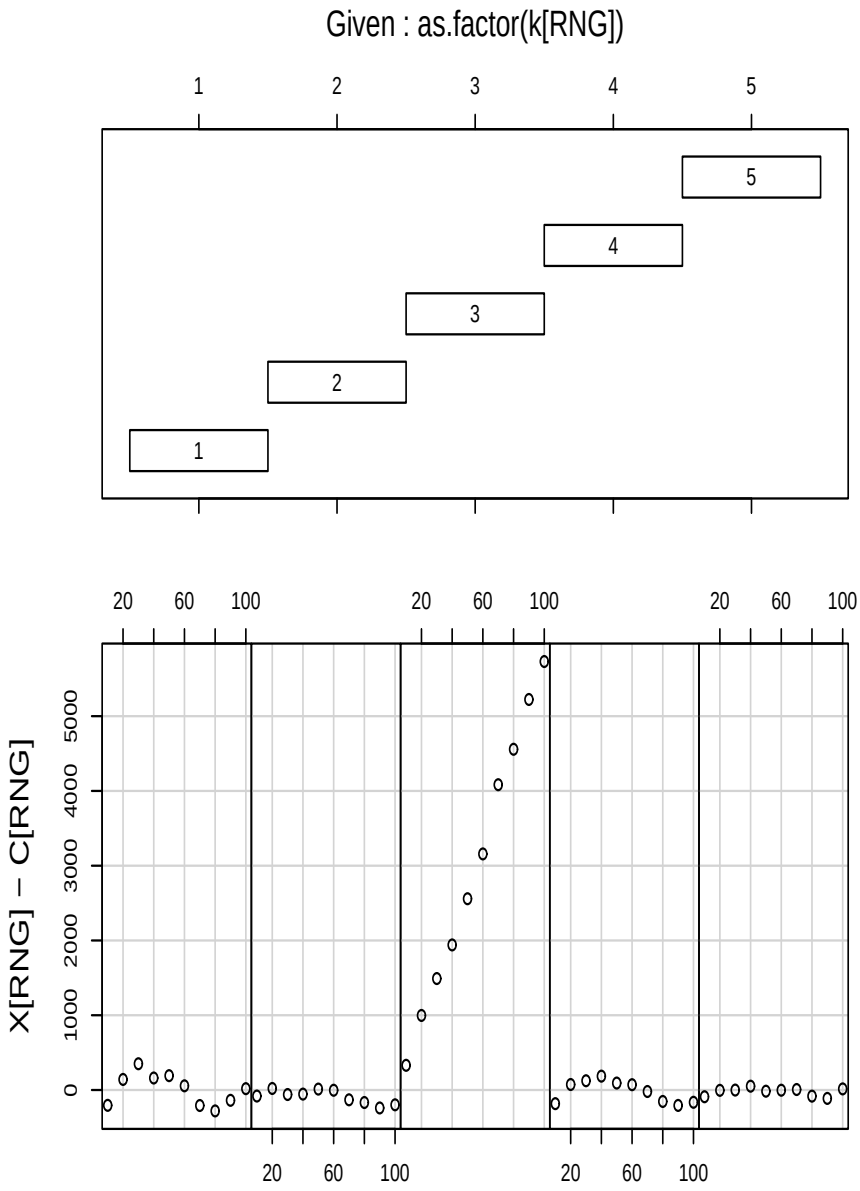
Super−Duper: Multiple N

Super-Duper Increasing N



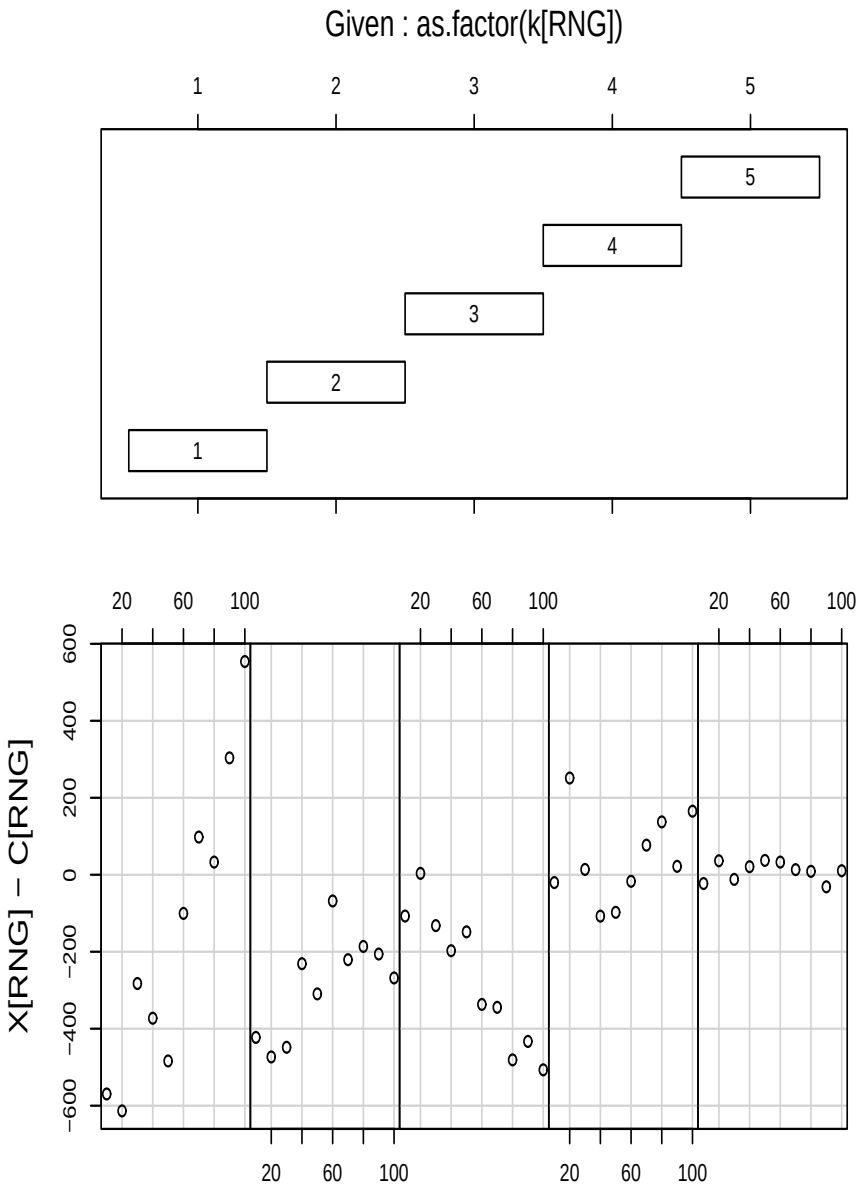
Mersenne−Twister: Multiple N

Mersenne Twister Increasing N



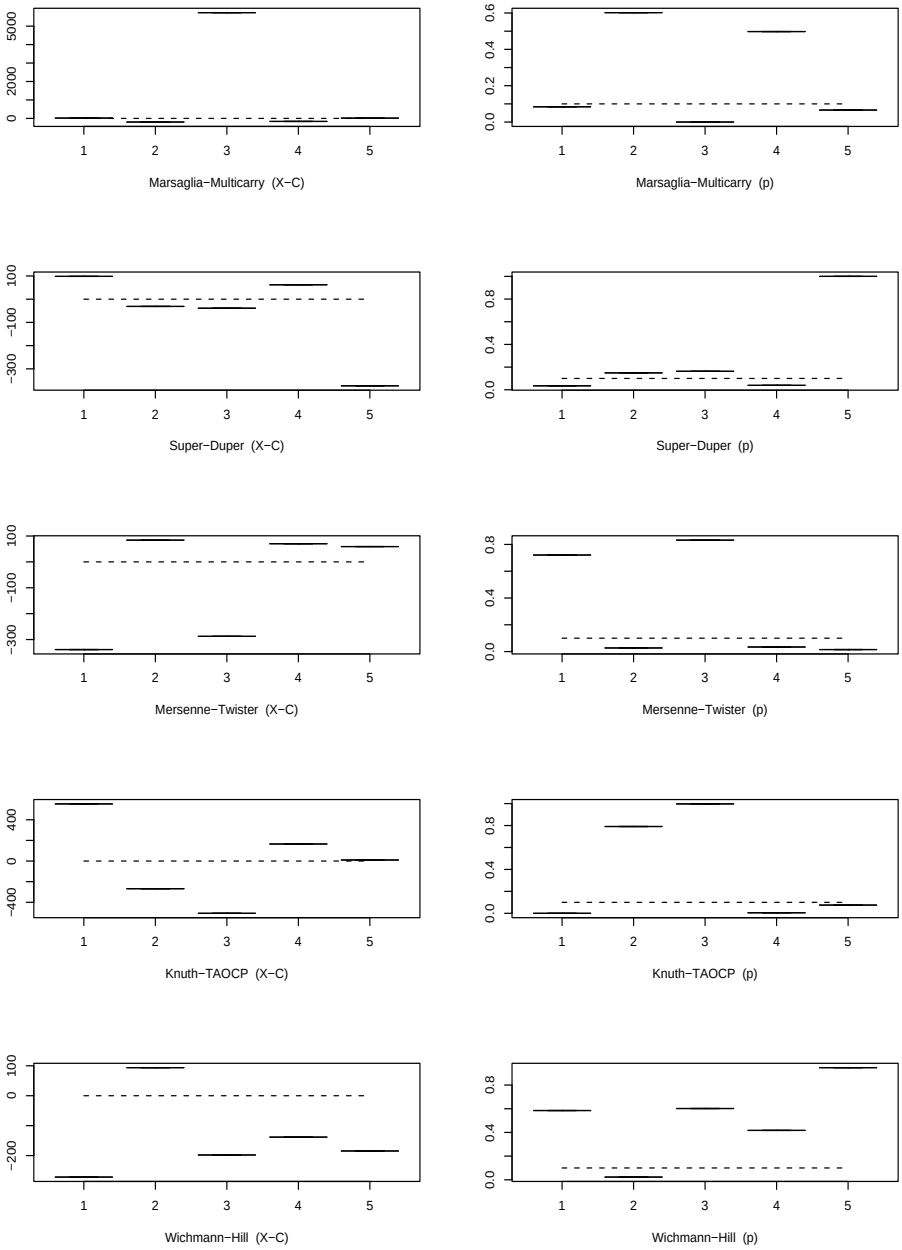
Marsaglia−Multicarry: Multiple N

Marsaglia Multicarry Increasing N

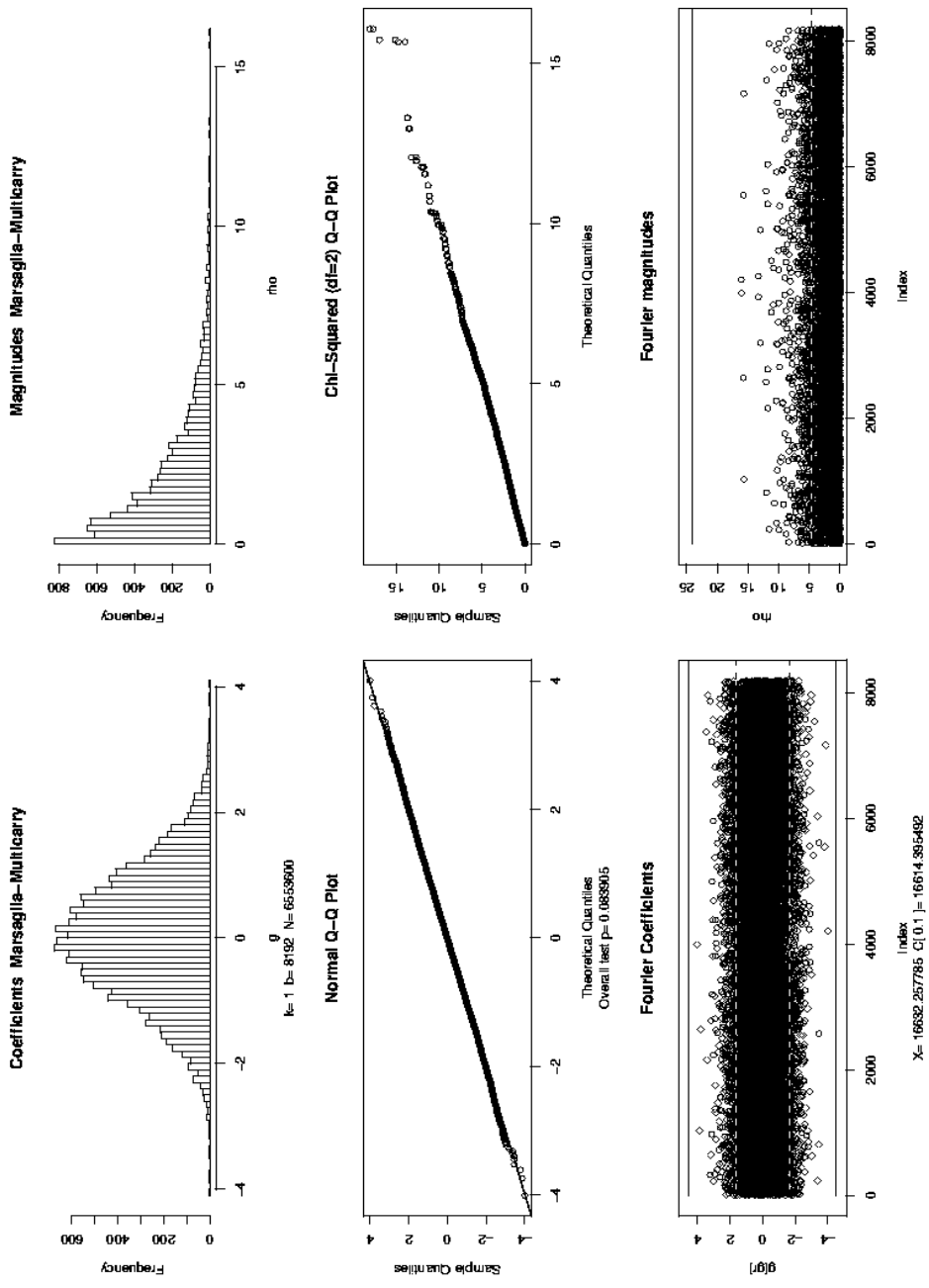


Knuth−TAOCP: Multiple N

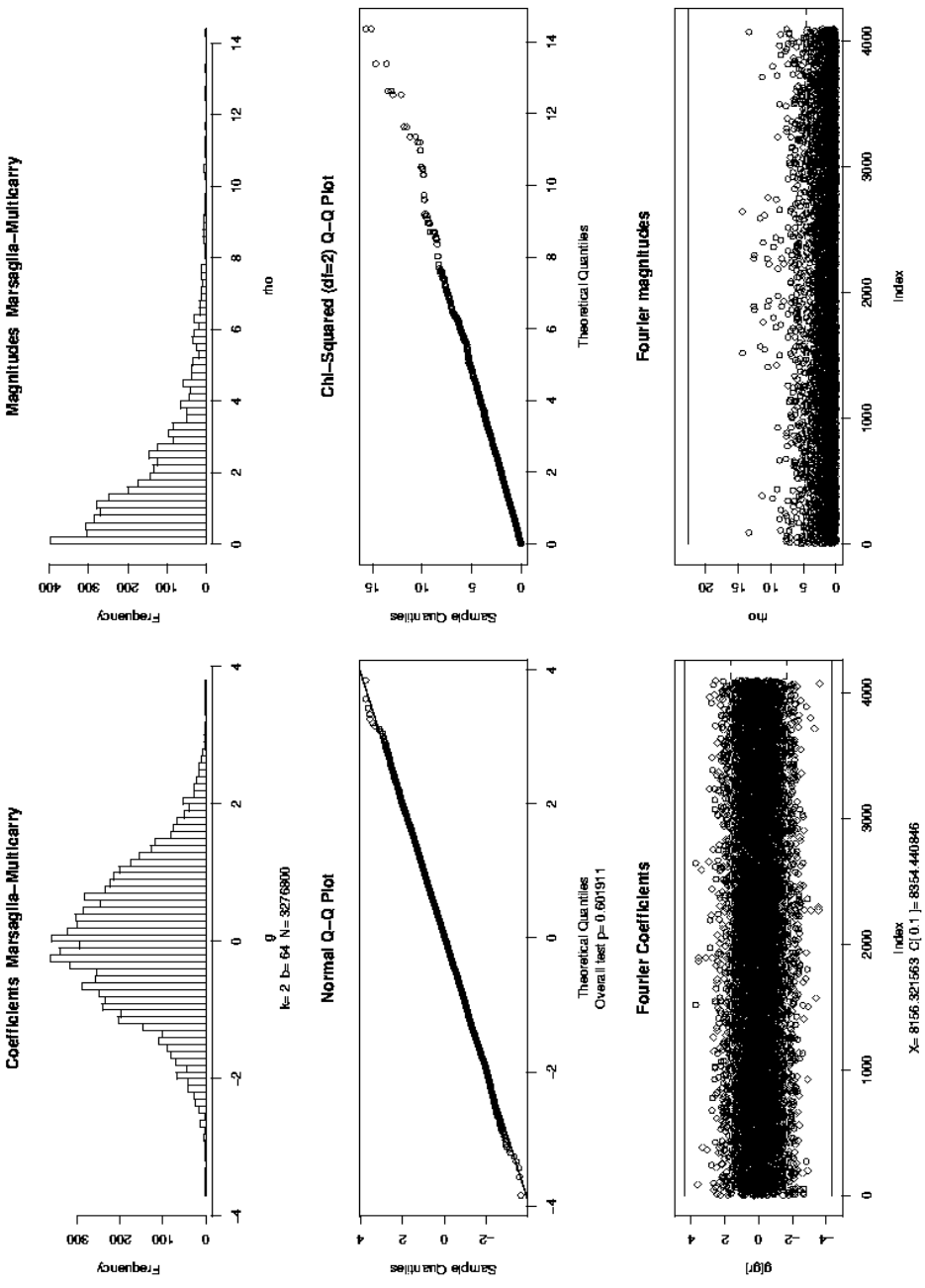
Knuth TAOCP Increasing N


Marsaglia−Multicarry (X−C)
Marsaglia−Multicarry (p)
Super−Duper (X−C)
Super−Duper (p)
Mersenne−Twister (X−C)
Mersenne−Twister (p)
Knuth−TAOCP (X−C)
Knuth−TAOCP (p)
Wichmann−Hill (X−C)
Wichmann−Hill (p)

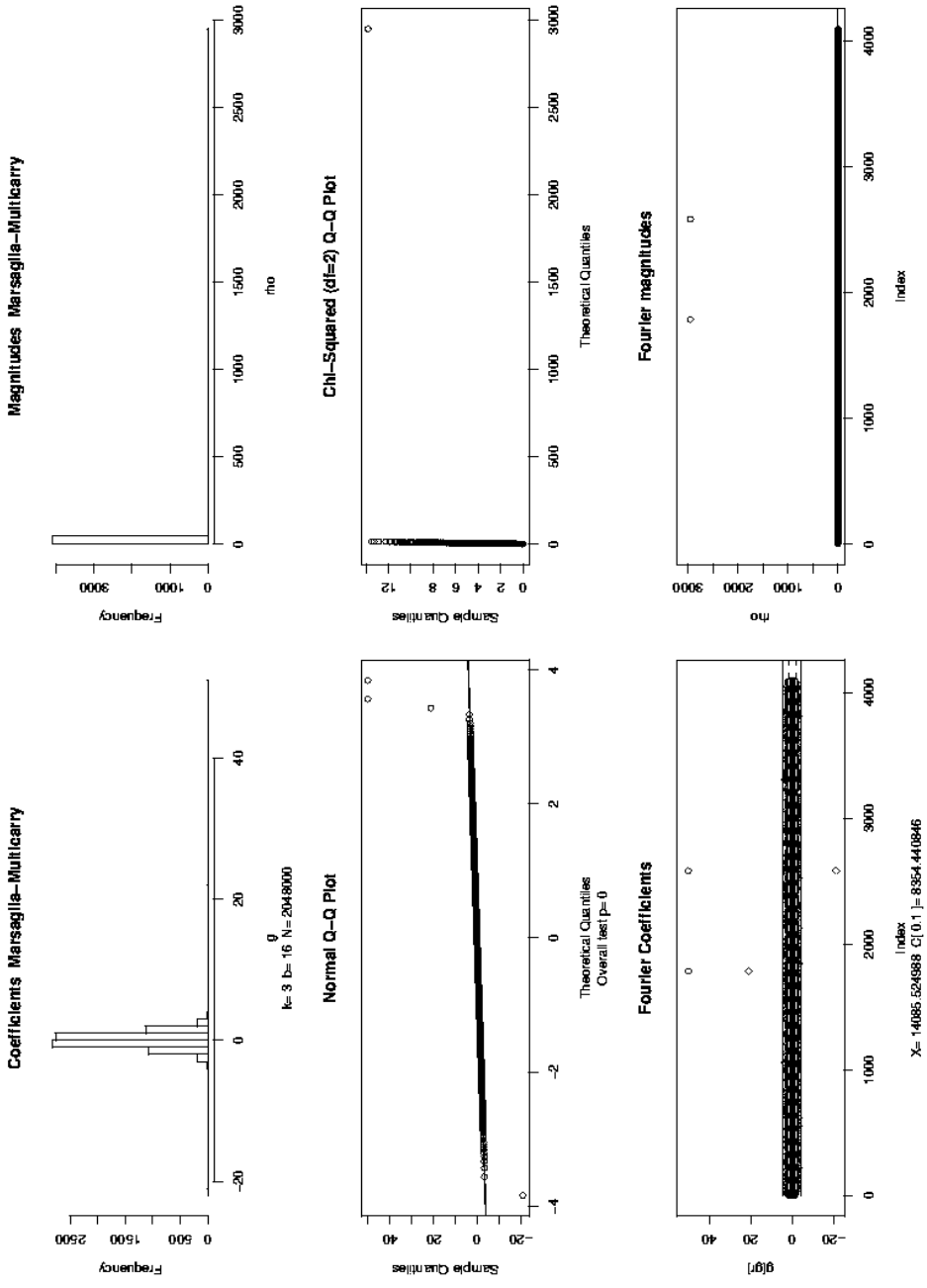
Large N



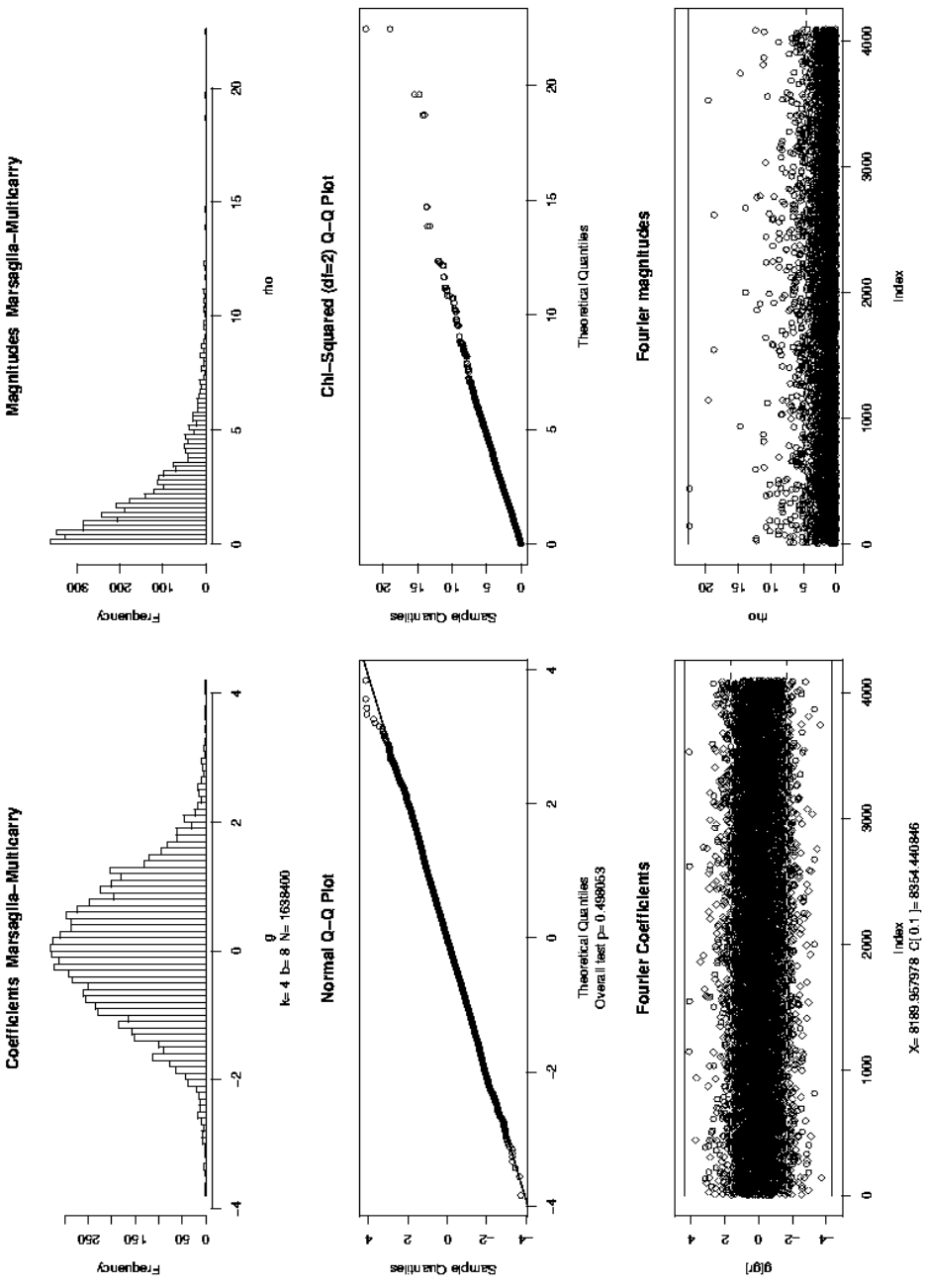
Large N: Marsaglia Multicarry k=1



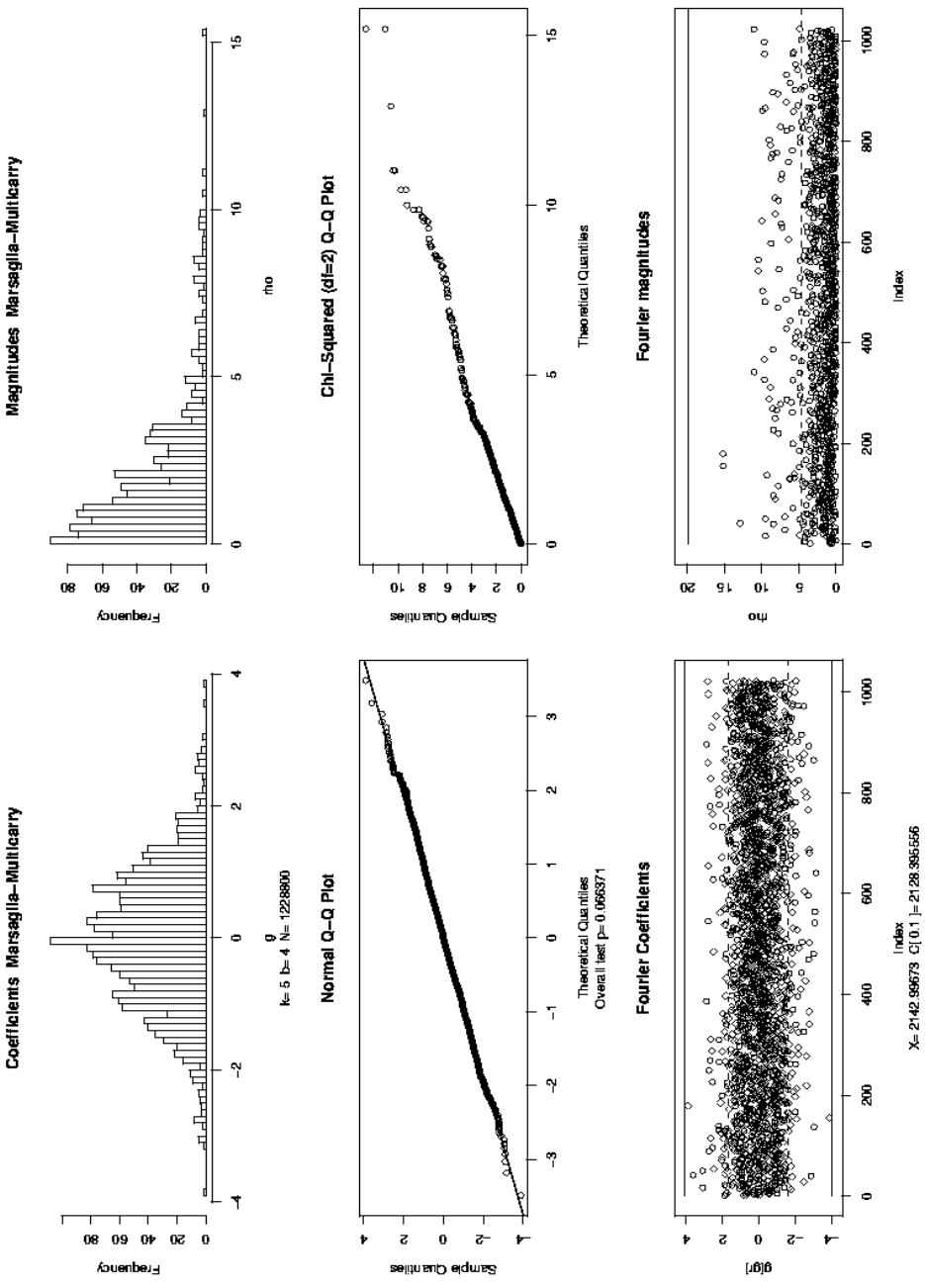
Large N: Marsaglia Multicarry k=2



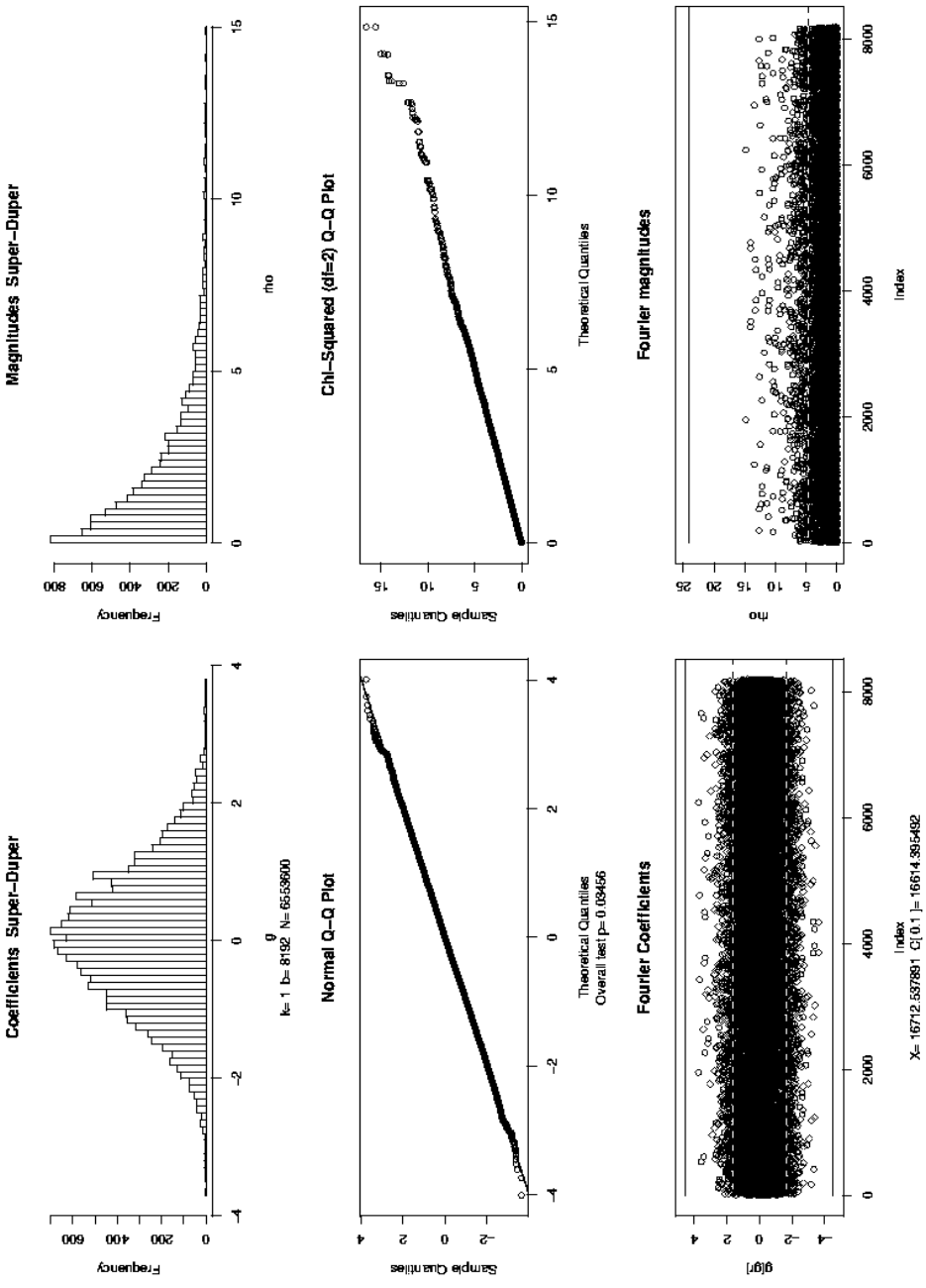
Large N: Marsaglia Multicarry k=3



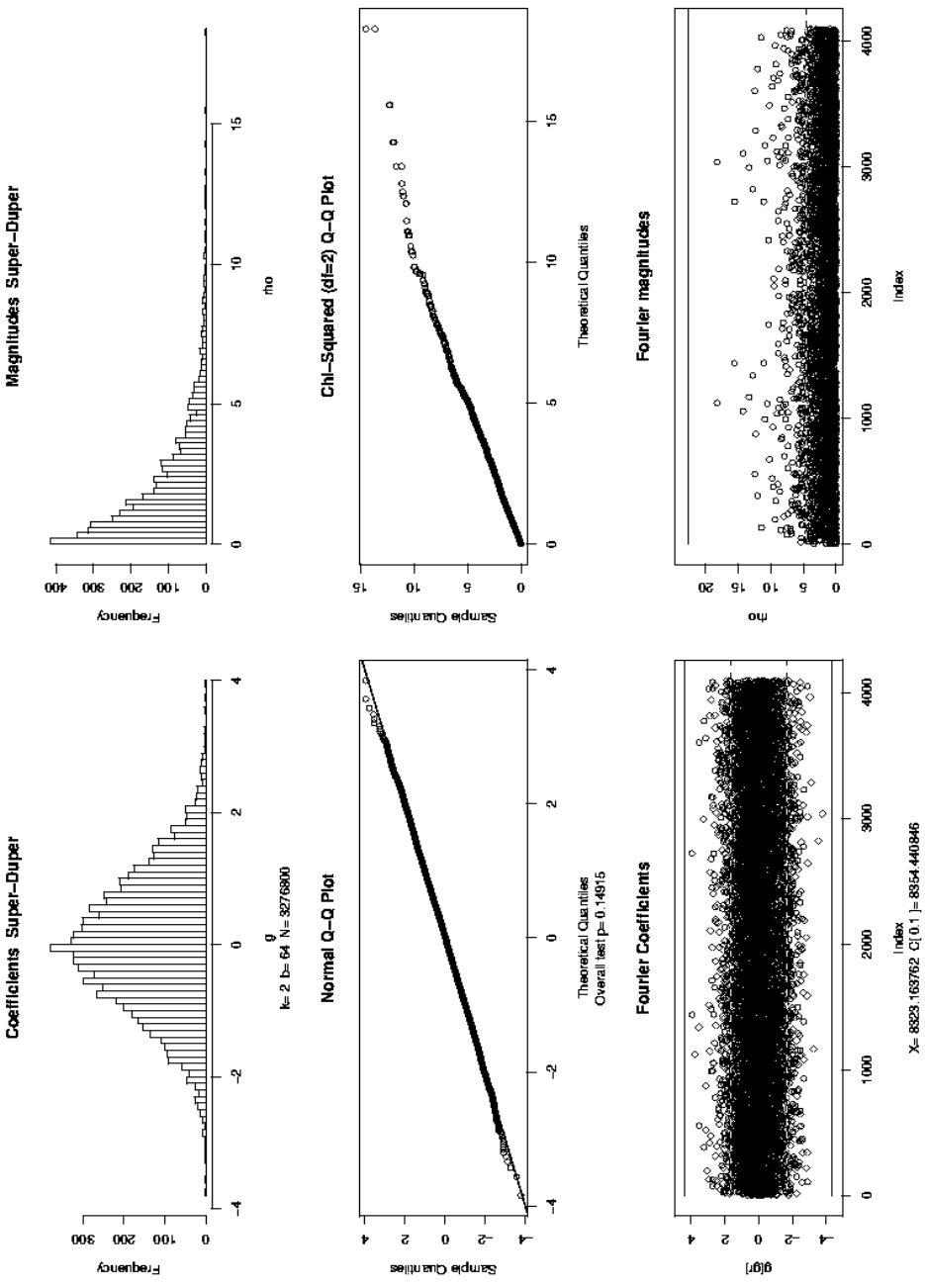
Large N: Marsaglia Multicarry k=4



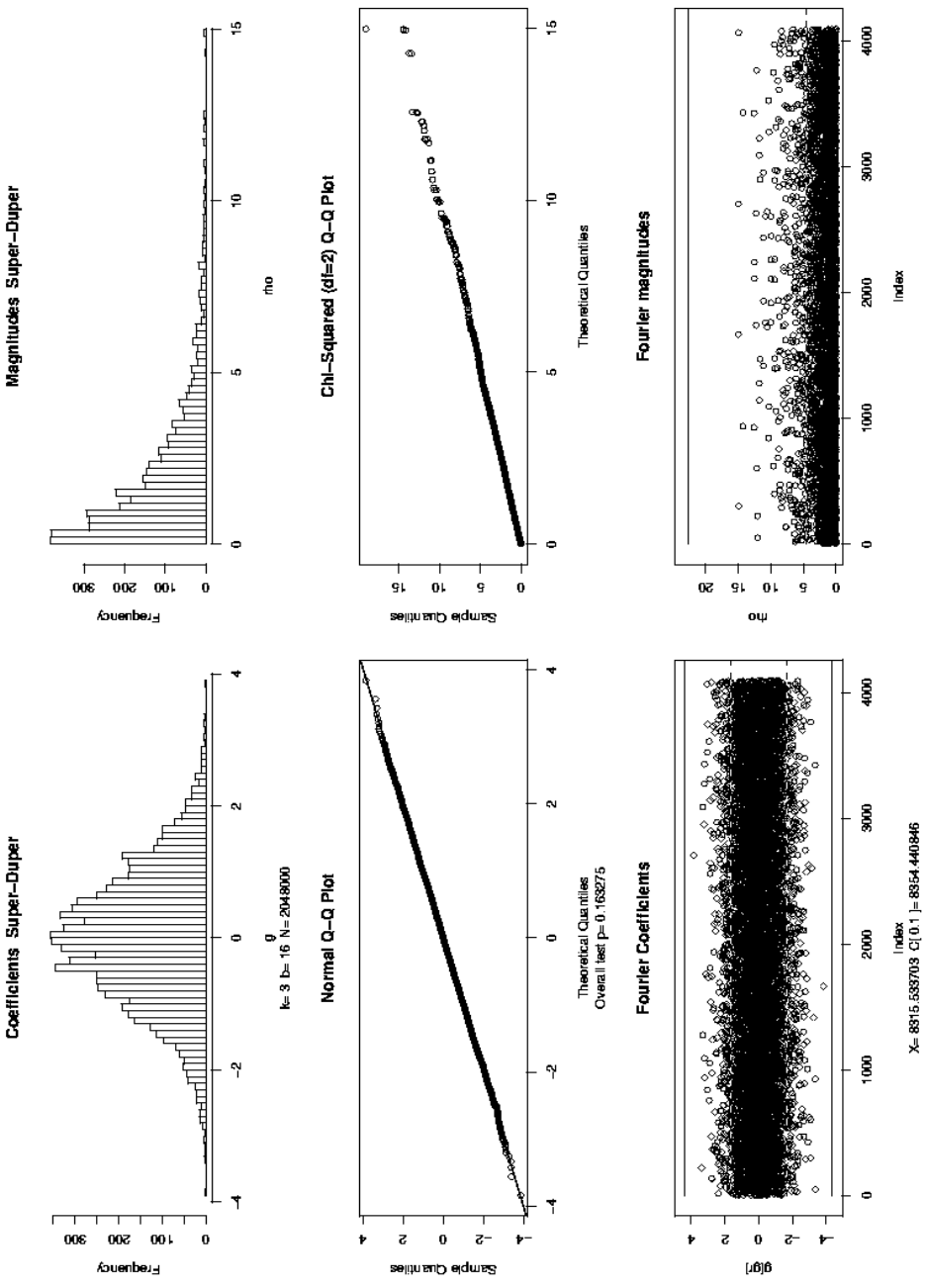
Large N: Marsaglia Multicarry k=5



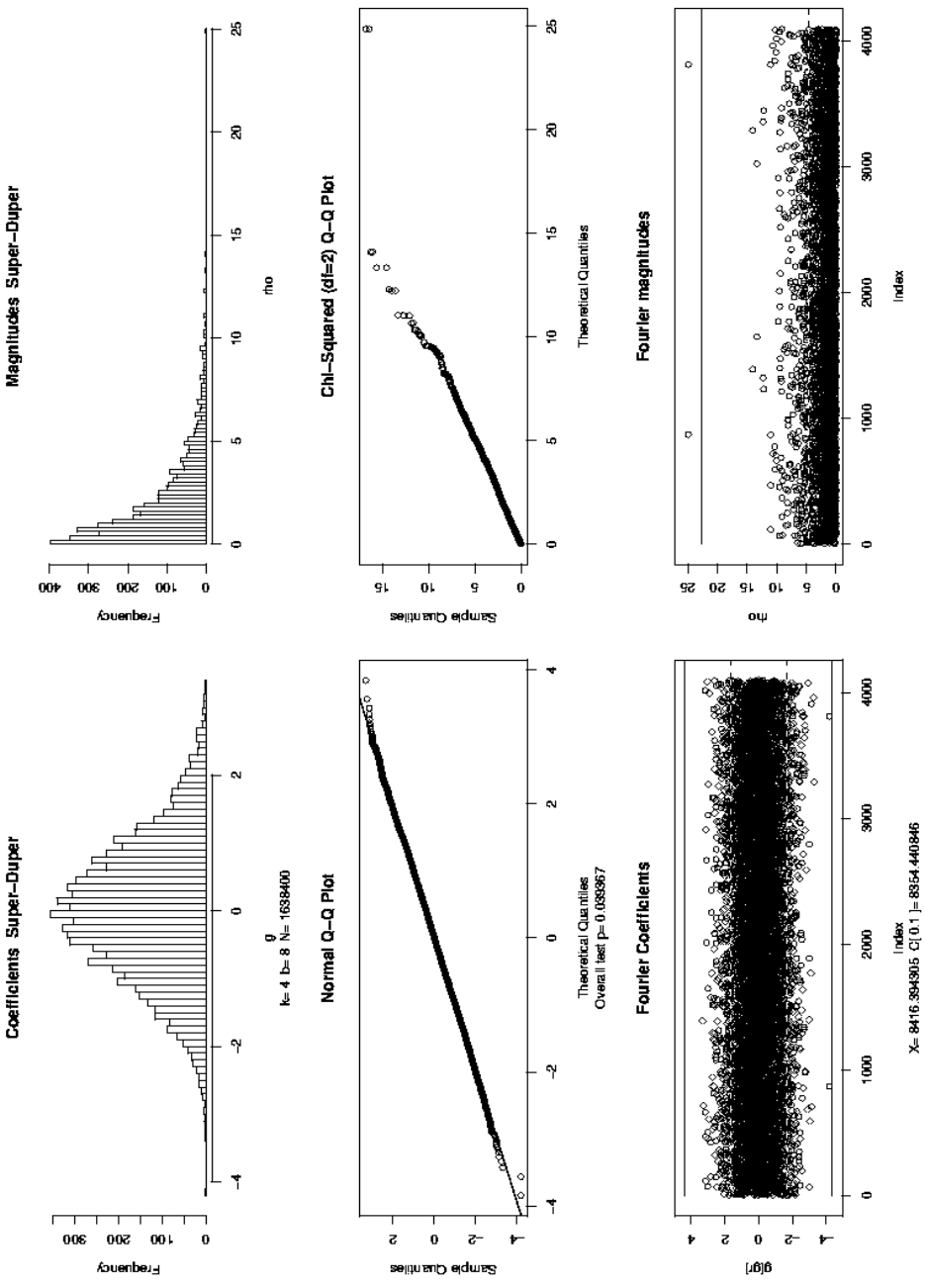
Large N: Super-Duper k=1



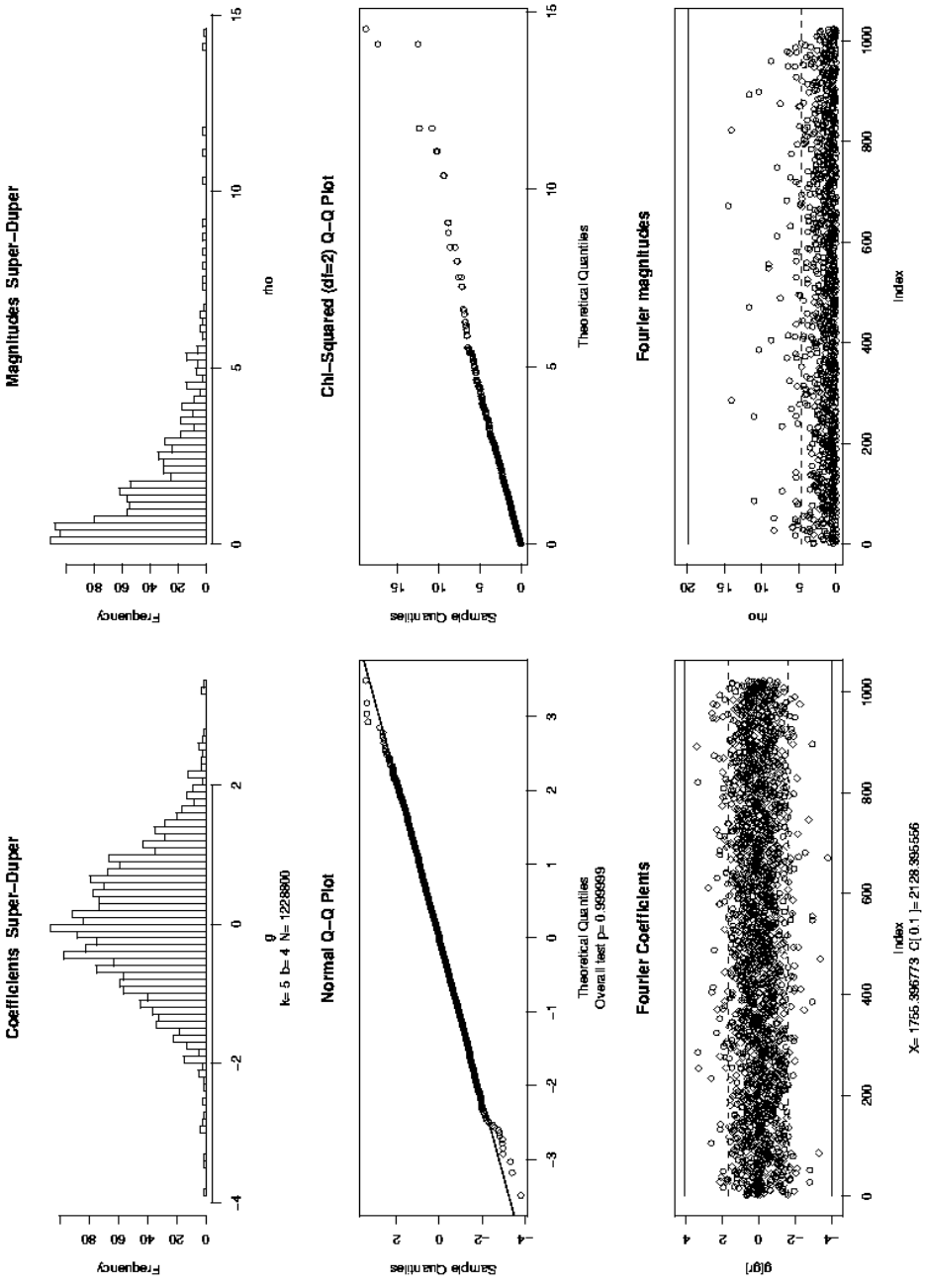
Large N: Super-Duper k=2



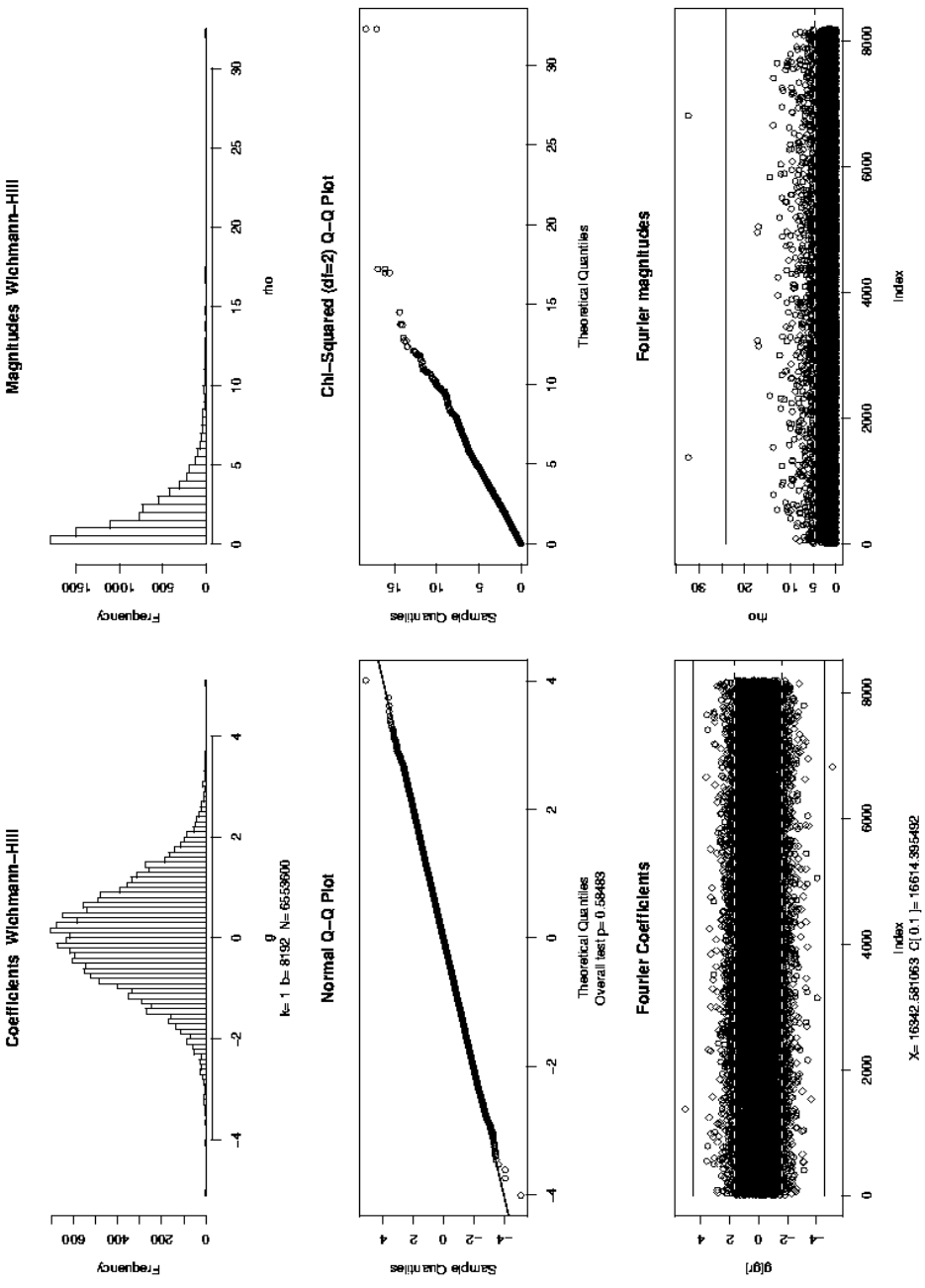
Large N: Super-Duper k=3



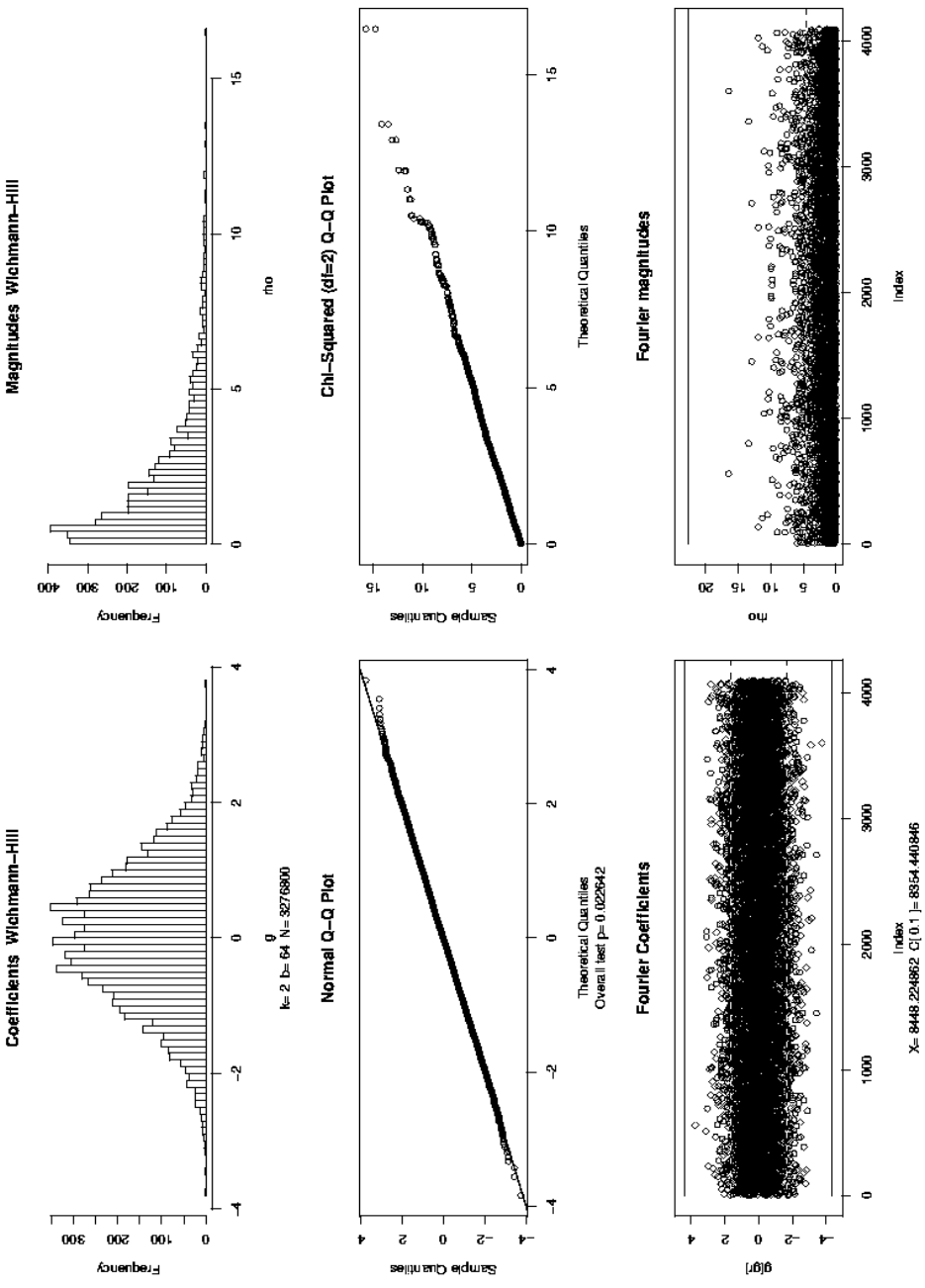
Large N: Super-Duper k=4



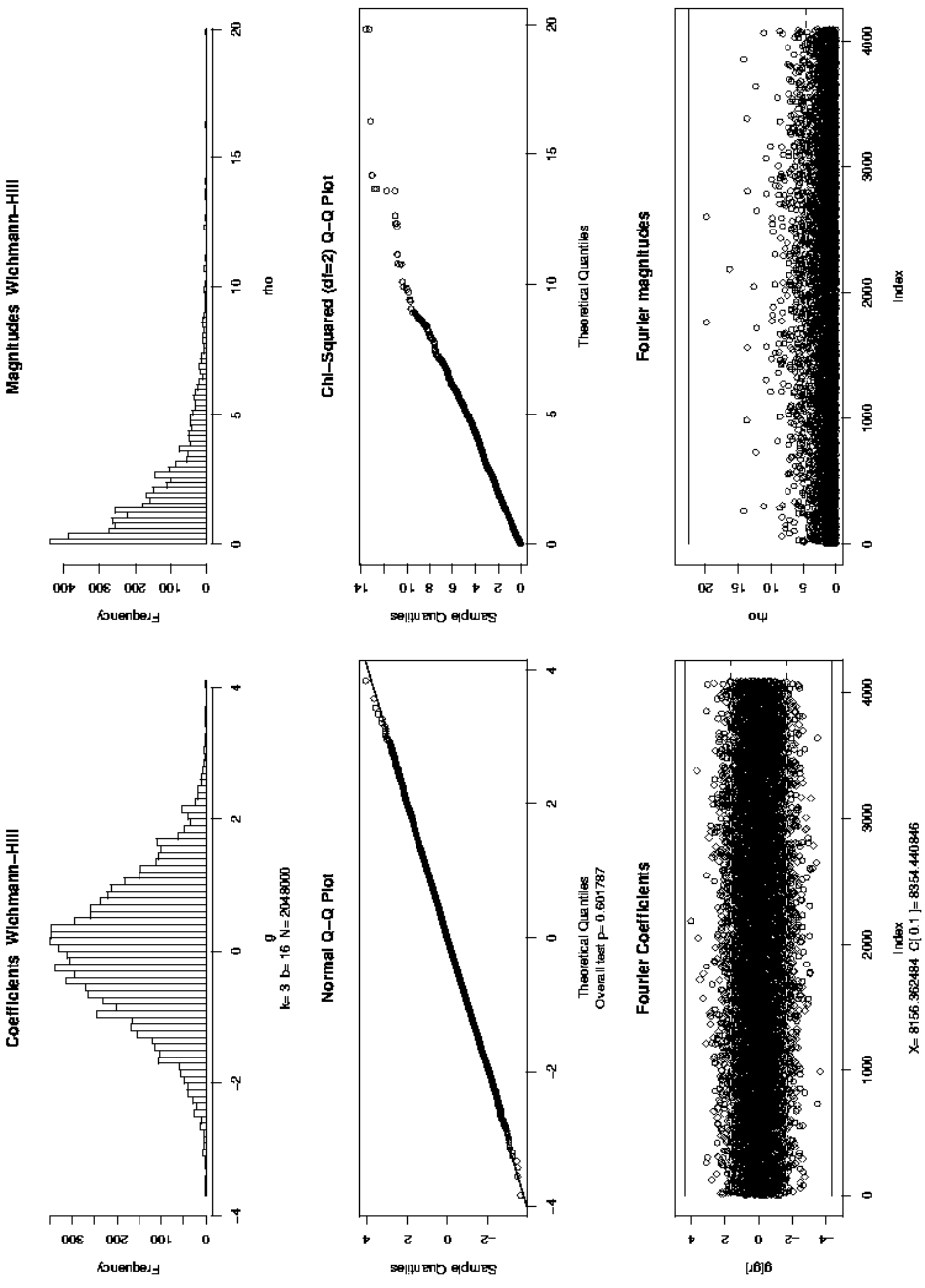
Large N: Super-Duper k=5



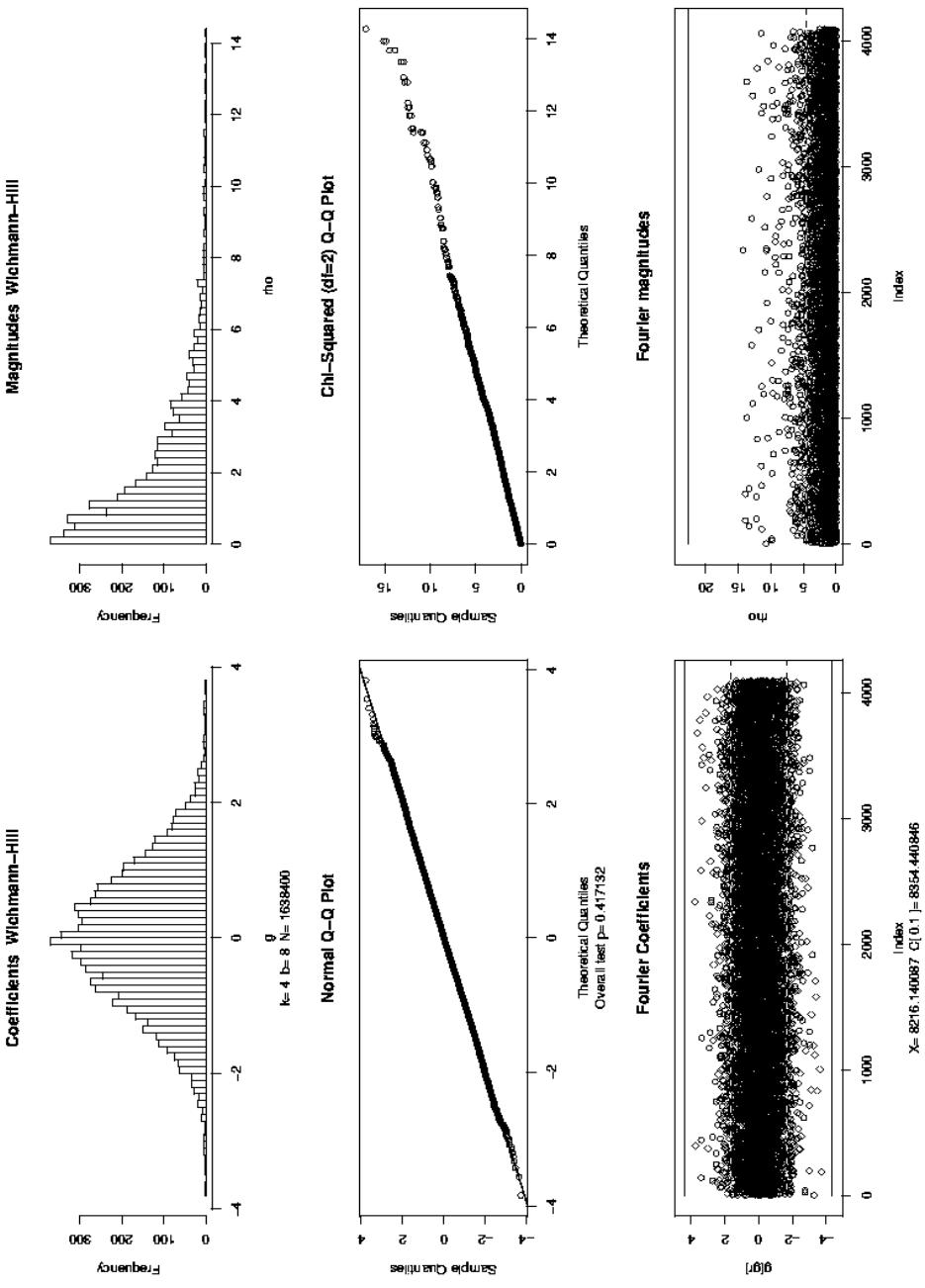
Large N: Wichmann Hill k=1



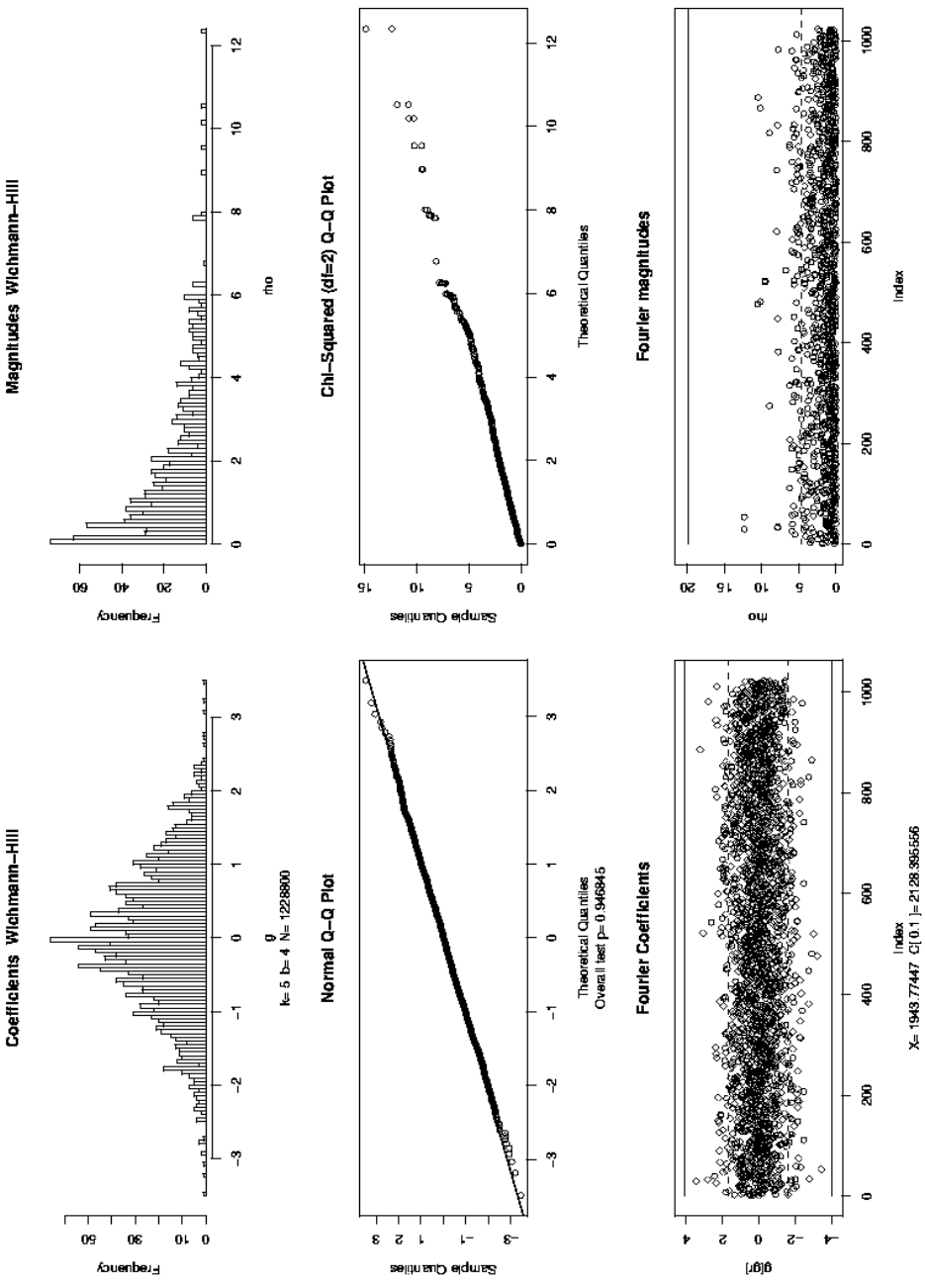
Large N: Wichmann Hill k=2



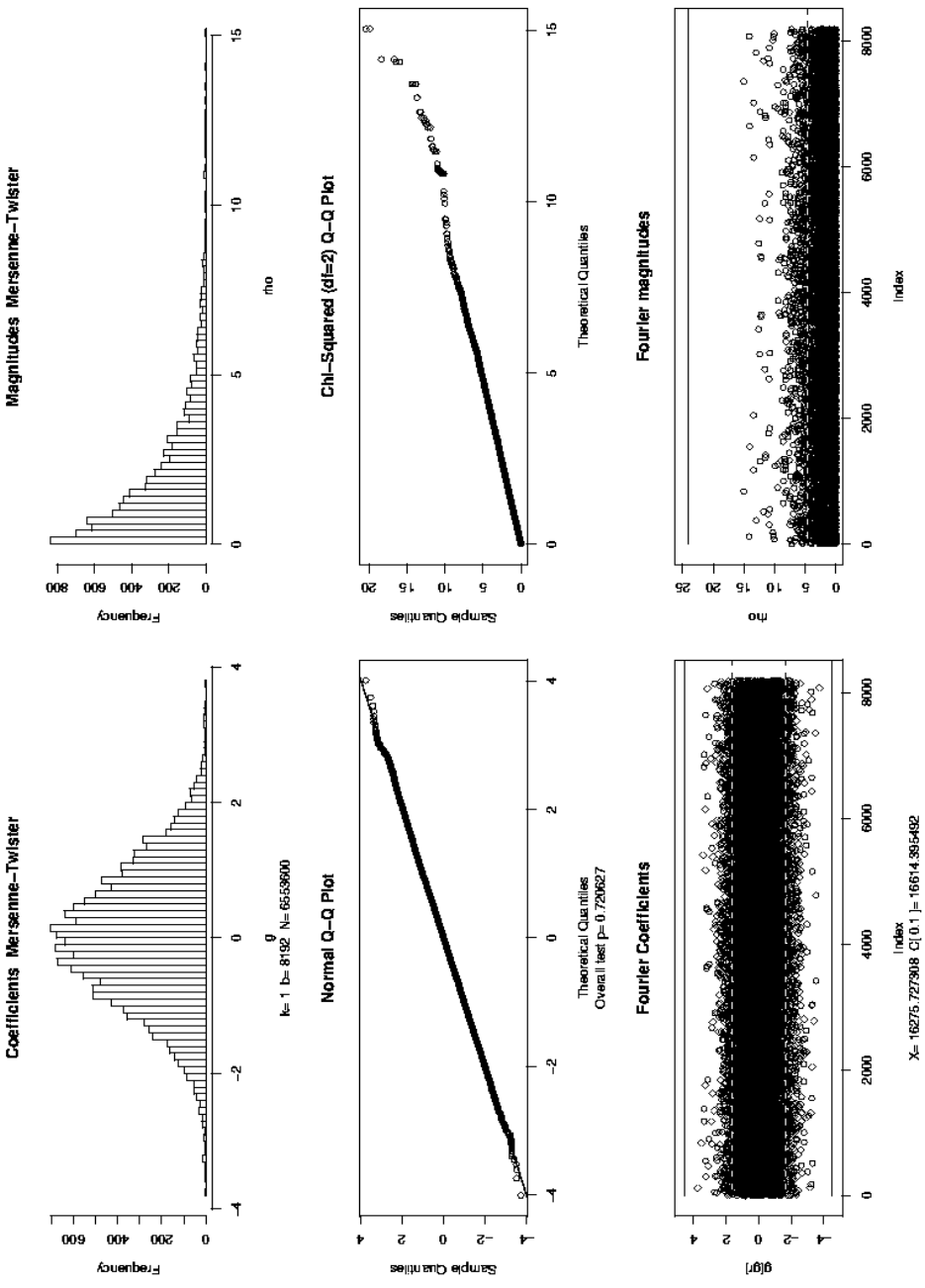
Large N: Wichmann Hill k=3



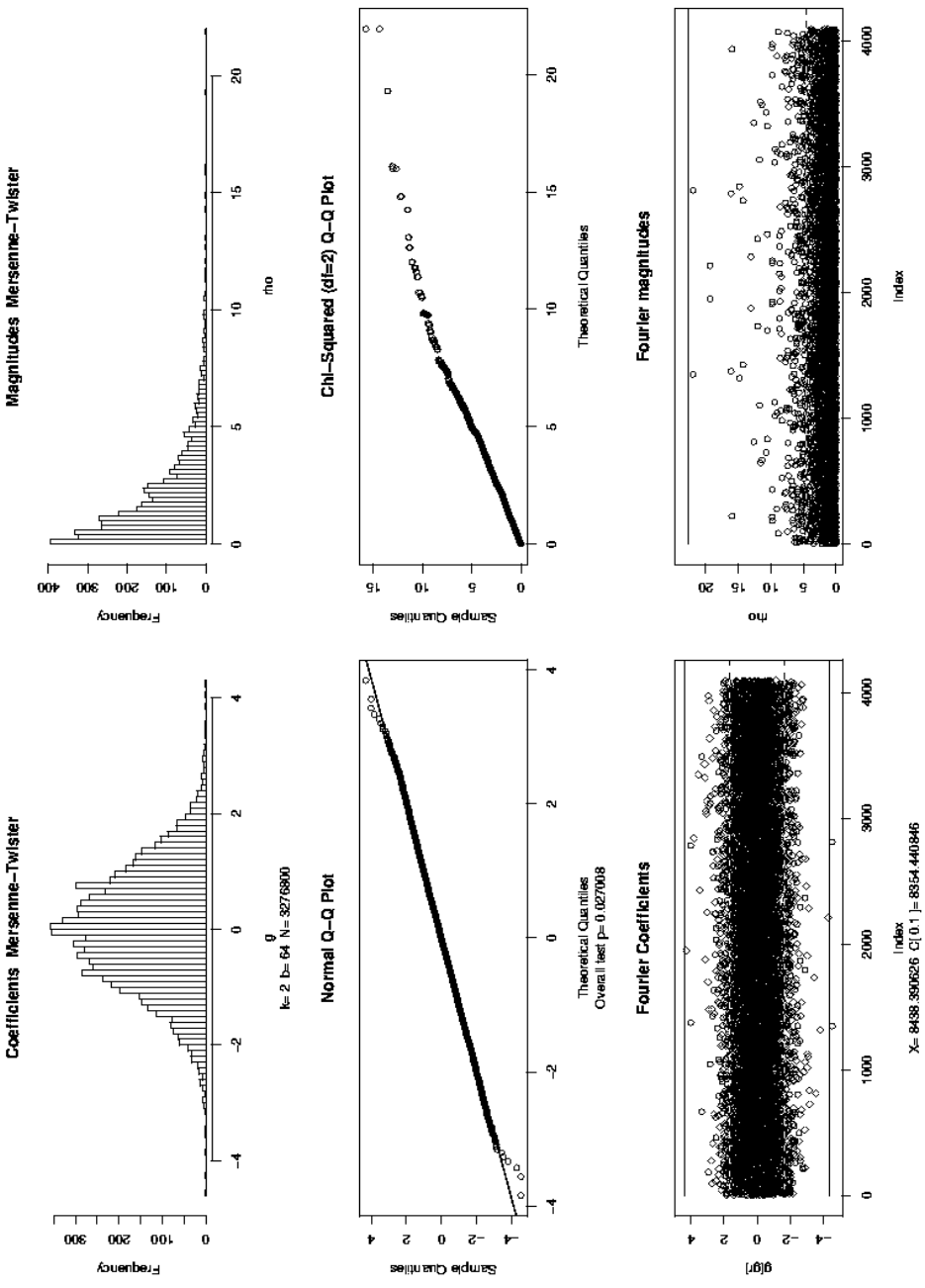
Large N: Wichmann Hill k=4



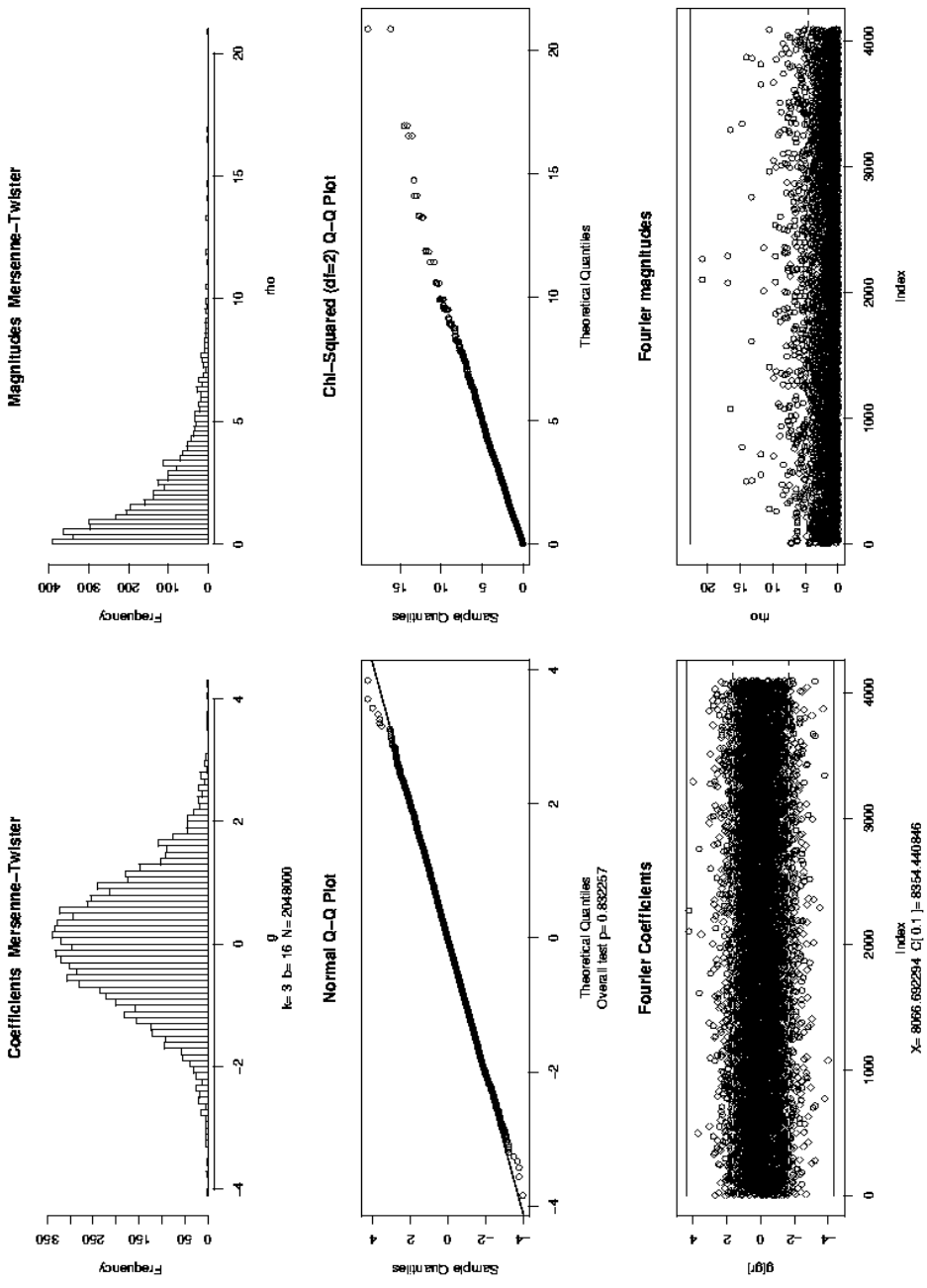
Large N: Wichmann Hill k=5



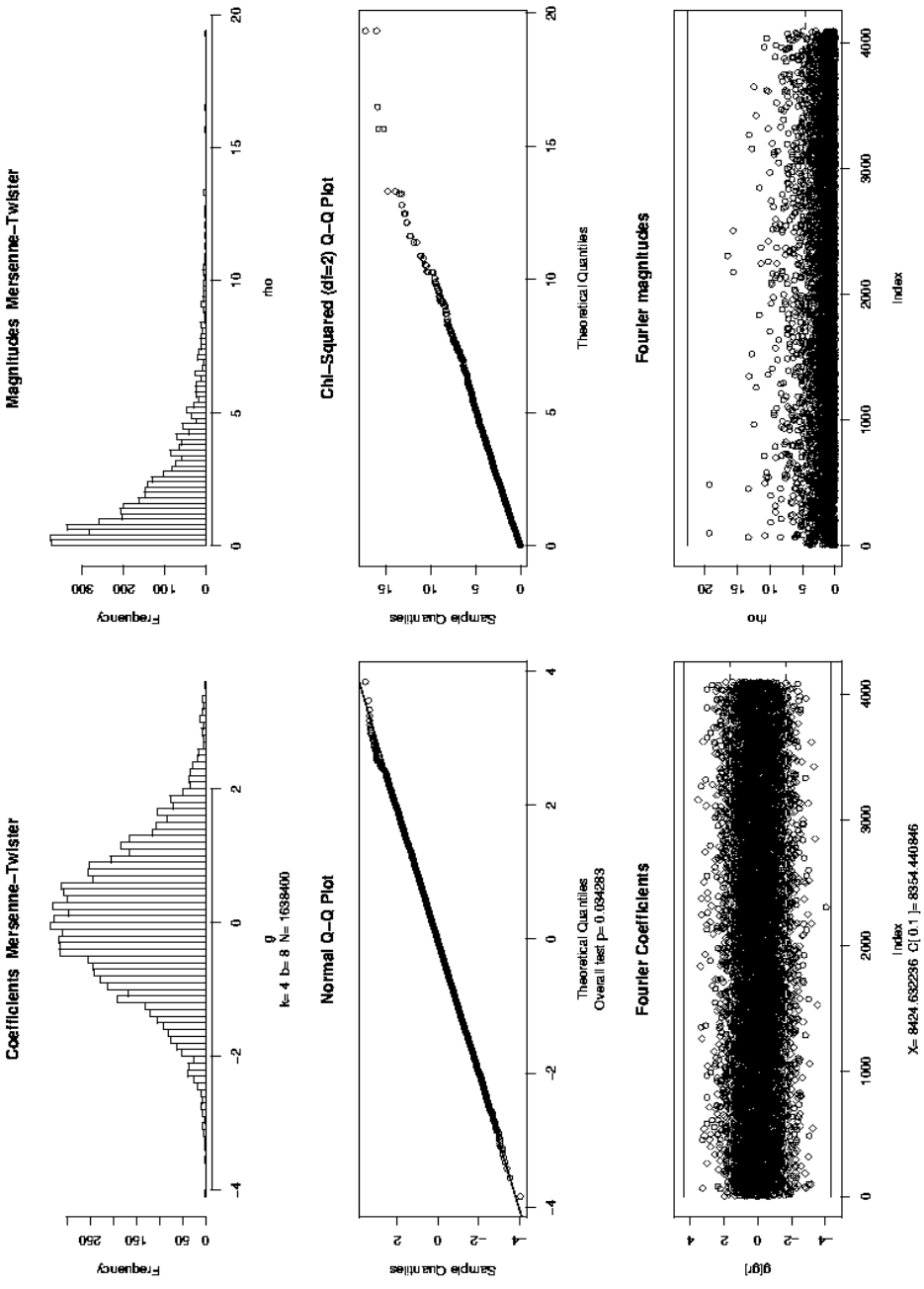
Large N: Mersenne Twister k=1



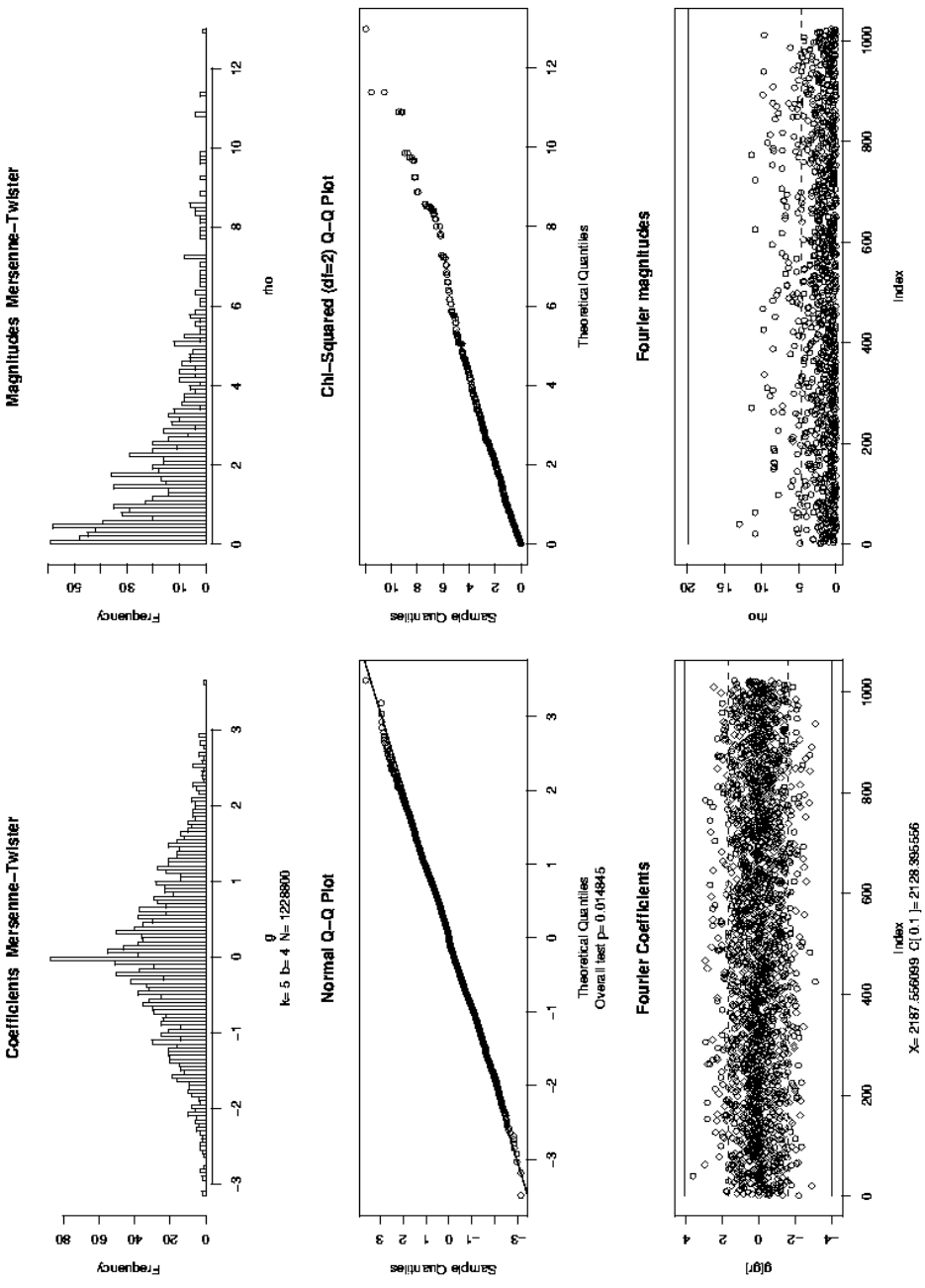
Large N: Mersenne Twister k=2



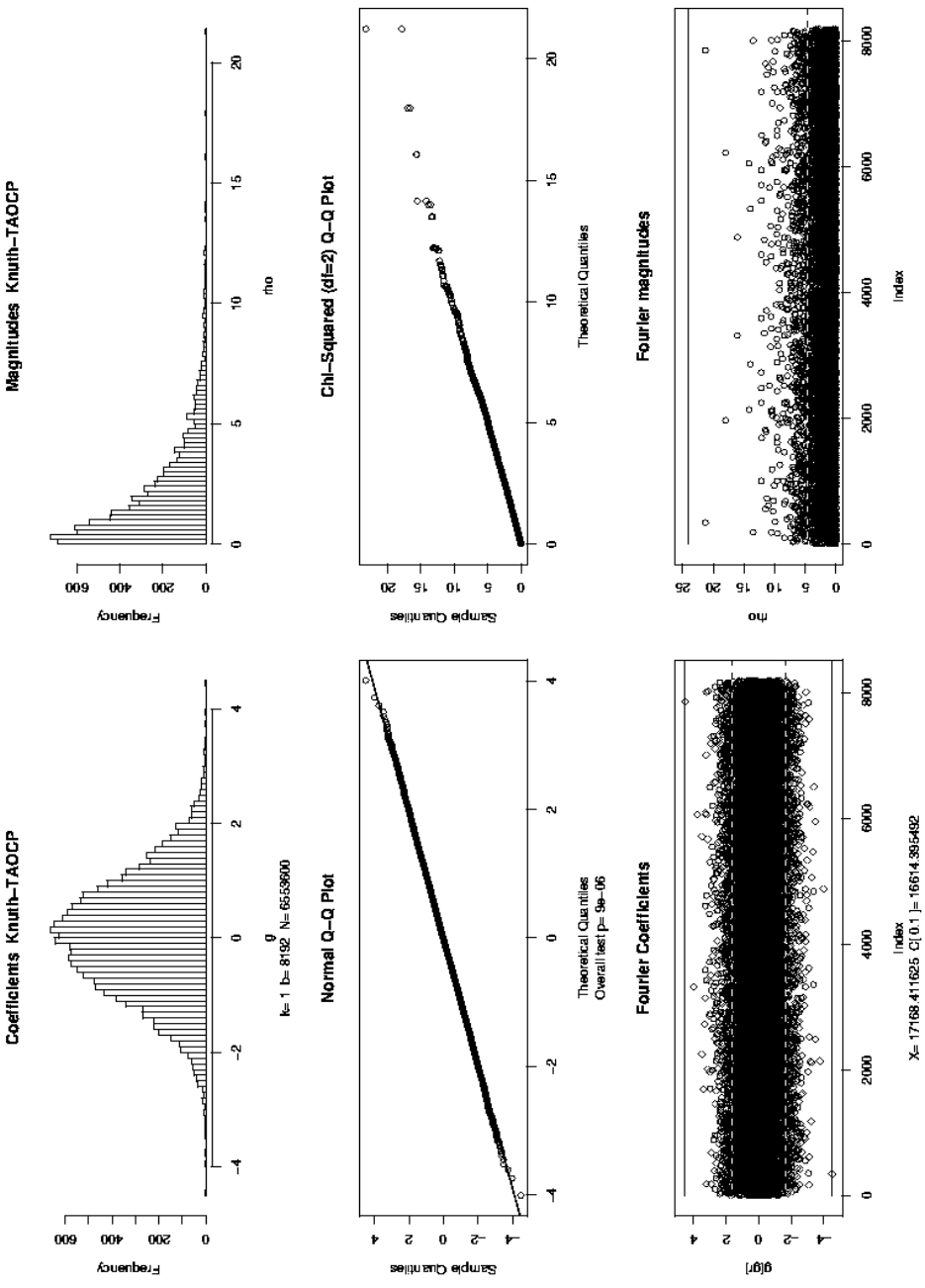
Large N: Mersenne Twister k=3



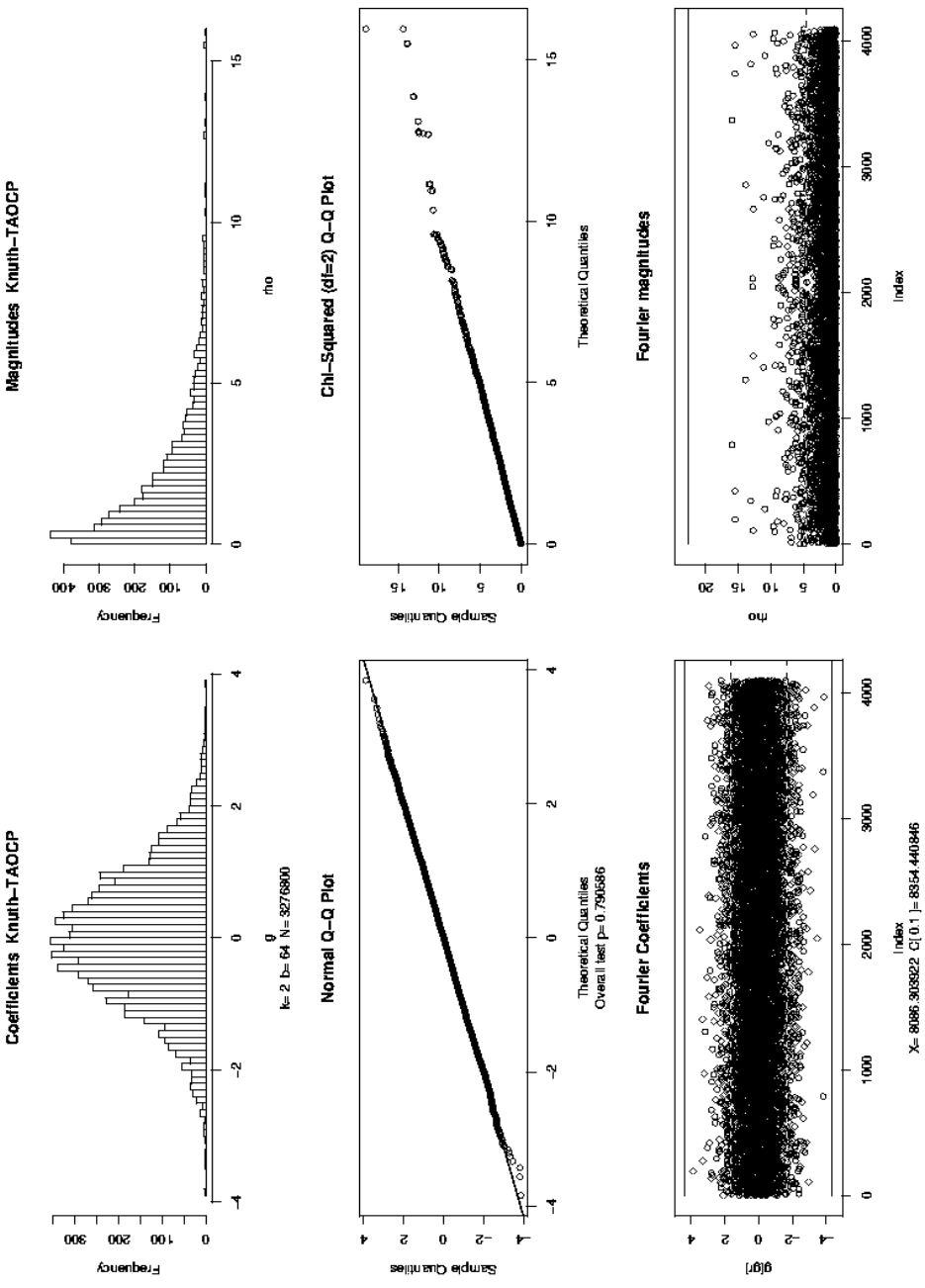
Large N: Mersenne Twister k=4



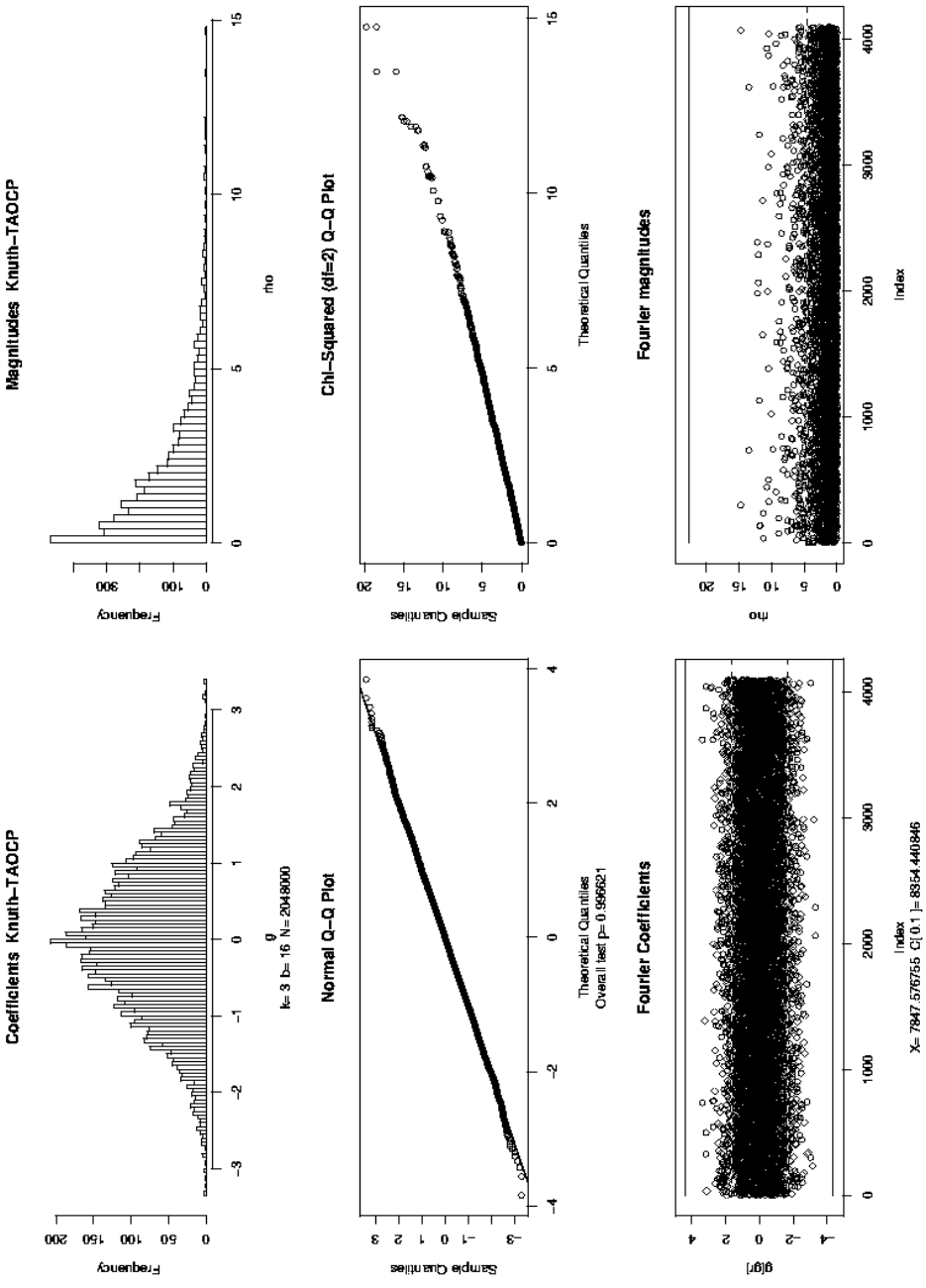
Large N: Mersenne Twister k=5



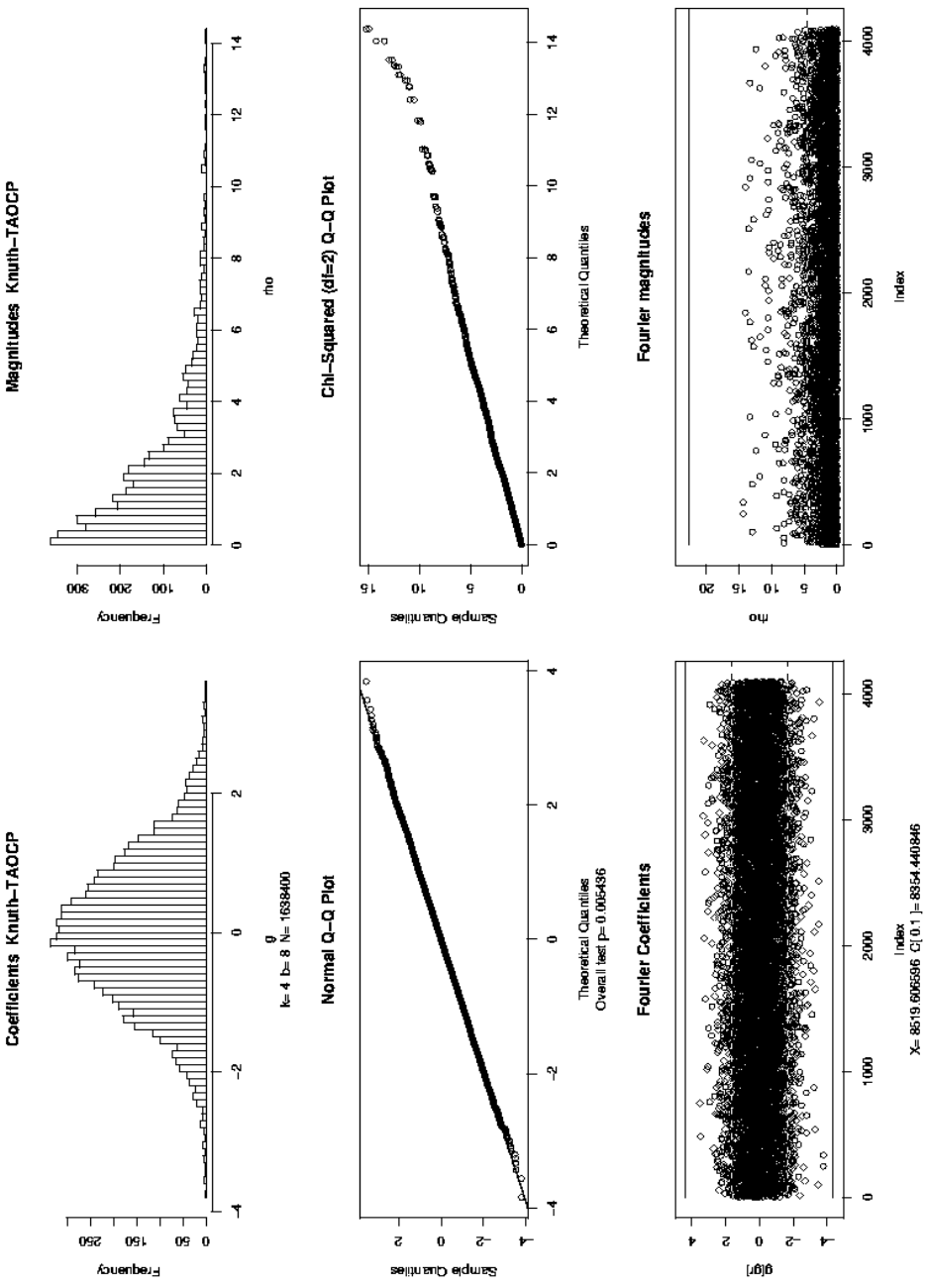
Large N: Knuth TAOCP k=1



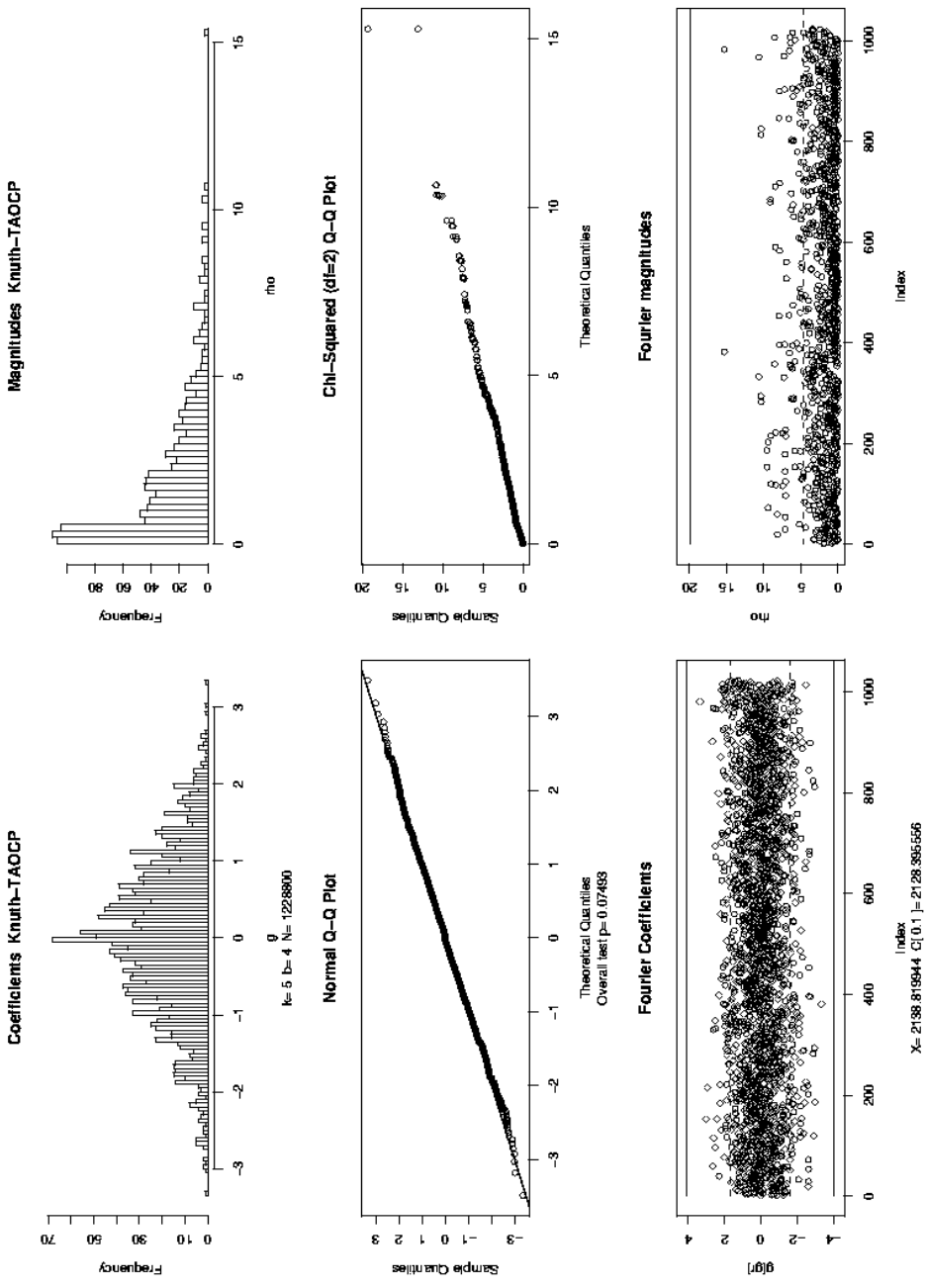
Large N: Knuth TAOCP k=2



Large N: Knuth TAOCP k=3



Large N: Knuth TAOCP k=4



Large N: Knuth TAOCP k=5

APPENDIX B

SOURCE CODE

2.1. Example Code for R

R is a free open source statistical package modeled after the S language developed at AT&T. We use it here to both to provide good graphics output and to demonstrate the mdftt algorithm in a form which should be easier to understand that C code.

example1d.R: one dimensional example

LISTING B.1

```
# define problem
k <- 1            # number of dimensions
b <- 4096              # bins per dimensions
T <- b^k          # number of cells
N <- T*E          # set sample size
P <- 1/N

x <- rep(0,T)     # preallocate and zero the cell counts
dim(x) <- c(rep(b,k))

# generate the cell counts from the bin indices
for (i in 1:N) {

  # get random numbers
  u <- runif(k)   # generate the random numbers

  # create cell counts in x
  c <- u*b        # conv [0,1] values to bin indices
  c <- trunc(c)   # by multiply and truncate
  u1 <- c[1]+1
  x[u1] <- x[u1]+P

}

# calculate the fft
f <- fft(x) # transform it

# build up a frequency map for the data
Theta1 <- rep(0,b^k)
dim(Theta1) <- rep(b,k)
for (i in 1:b) {
   Theta1[i] <- ((i-1)/b)*pi;
}
dim(Theta1) <- NULL

# gather mdftt data frame
Re <- c(Re(f))
```

```
Im <- c(Im(f))
Rho <- c(sqrt(Re*Re+Im*Im))
Phi <- c(asin(Re/Rho))

# Build data frame matching mdftt.c output
TestData <- data.frame(Re,Im,Rho,Phi,Theta1);

# Cleanout intermediate data
rm(f,Re,Im,Rho,Phi,x,u1);

TestResult <- mdftt(TestData,N,0.1)
TestReport <- tview(TestData,TestResult,N/T,b,k,0.1,
   RNGkind()[1]);
```

example2d.R: two dimensional example

LISTING B.2

```
# define problem
k <- 2            # number of dimensions
b <- 64           # bins per dimensions
T <- b^k          # number of cells
N <- T*E          # set sample size
P <- 1/N

x <- rep(0,T)     # preallocate and zero the cell counts
dim(x) <- c(rep(b,k))

# generate the cell counts from the bin indices
for (i in 1:N) {

  # get random numbers
  u <- runif(k)   # generate the random numbers

  # create cell counts in x
  c <- u*b        # conv [0,1] values to bin indices
  c <- trunc(c)   # by multiply and truncate
  u1 <- c[1]+1
  u2 <- c[2]+1
  x[u1,u2] <- x[u1,u2]+P

}

# do fft's of the rows first, then columns
v <- rep(0,b) # allocate work vector
for (i in 1:b) {
  v <- fft(x[i,]) # transform it
  # row bind into a new array
  if(i==1)
    y <- v
  else
    y <- rbind(y,v)
```

```
}

for (i in 1:b) {
  v <- fft(y[,i])
  if(i==1)
    f <- v
  else
    f <- cbind(f,v)
}

# build up a frequency map for the data
Theta1 <- rep(0,b^k)
dim(Theta1) <- rep(b,k)
Theta2 <- rep(0,b^k)
dim(Theta2) <- rep(b,k)
for (i in 1:b) {
  for (j in 1:b) {
    Theta1[i,j] <- ((i-1)/b)*pi;
    Theta2[i,j] <- ((j-1)/b)*pi;
  }
}
dim(Theta1) <- NULL
dim(Theta2) <- NULL

# mdftt data
Re <- c(Re(f))
Im <- c(Im(f))
Rho <- c(sqrt(Re*Re+Im*Im))
Phi <- c(asin(Re/Rho))

# Build data frame matching mdftt.c output
TestData <- data.frame(Re,Im,Rho,Phi,Theta1,Theta2);

# Cleanout intermediate data
rm(f,Re,Im,Rho,Phi,x,y,v,u1,u2);

TestResult <- mdftt(TestData,N,0.1)
TestReport <- tview(TestData,TestResult,N/T,b,k,0.1,
   RNGkind()[1]);
```

mdftt.R: R version of the EST

LISTING B.3

```
# mdftt.R
#
# Calculate the mdftt test on the fourier transform
   coefficients in the
# data frame f.
#
# Inputs:
```

```
#     f: data frame input from mdftt.c Fourier
    coefficient output
#           f$Re,f$Im,f$Rho,f$Phi
#
#     N: number of data points used to generate 'f'
mdftt <- function(f,N,a) {

  T <- length(f$Re)

  # Overall Test Statistic z'S^{-1}z = 2Nz'z
  z <- c(f$Re,f$Im)
  X <- 2*N * ((t(z)%*%z) - 1)
  p <- 1 - pchisq(X,2*(T-1))

  # Overall test critical value
  C <- qchisq(1-a,2*(T-1))

  # Fisher PLSD critical values
  K1 <- qnorm(1-a/2)
  K2 <- qchisq(1-a,2)

  # Bonferonni critical values
  L1 <- qnorm(1-a/(4*(T-1)))
  L2 <- qchisq(1-a/(2*(T-1)),2)

  return(data.frame(N,T,a,X,p,C,K1,K2,L1,L2))
}
```

tview.R: Graphics display and summary statistics in R

LISTING B.4

```
# tview.R
#
# # Calculate the mdftt test on the fourier transform
    coefficients in the
# data frame f and display stem and qq plots for the
    coefficents.
#
# Inputs:
#     f: data frame input from mdftt.c Fourier
    coefficient output
#           f$Re,f$Im,f$Rho,f$Phi
#
#     e: number of data points per cell used to
    determine N
#       N<-length(f$Re)*e
#
tview <- function(f,tst,e,b,k,a,rngkind) {

  par(mfcol=c(3,2))
```

```
T <- length(f$Re)
N <- T*e
N

g <- c(f$Re[2:T],f$Im[2:T]) # coefficients
gr <- 2:T;            # Real elements
gi <- (T+1):(2*T-2);  # Imaginary elements

# standardize the coefficients
g <- sqrt(2*N)*g

# normalized magnitudes squared
rho <- 2*N*f$Rho[2:T]^2

# find extents of coefficients & magnitudes
gmax <- 1.05*max(c(tst$K1,tst$L1,c(g)))
gmin <- 1.05*min(c(-tst$K1,-tst$L1,c(g)))
rhomax <- 1.05*max(c(rho,tst$K2,tst$L2));

# formatted output of results
report <- paste("Test of generator: ",rngkind);
report <- c(report,paste("using ",b," bins in ",k,"
   dimensions."));
report <- c(report,paste("Number of total data points (
   N) =",N))
report <- c(report,paste("Number of cells (T) =",T));
report <- c(report,paste("Test is level alpha = ",tst$a
   ));
report <- c(report,paste("Overall (X) =",tst$X, " p =",
   tst$p," C =",tst$C));
if(tst$X>tst$C) {
  report <- c(report,paste("Reject H0 @ ",tst$a));
  report <- c(report,paste("Max absolute g = ",max(abs(
     g))));
  report <- c(report,paste("Coefficient CI = [",-tst$K1
     ,",",tst$K1,"]"));
  report <- c(report,paste("Max rho = ",max(rho)));
  report <- c(report,paste("Magnitude CI = [0,",tst$L1
     ,"]"));
} else {
  report <- c(report,paste("Accept H0 @ ",tst$a));
  report <- c(report,paste("Use Bonferonni CI"));
  report <- c(report,paste("Max absolute g = ",max(abs(
     g))));
  report <- c(report,paste("Coefficient Bonferonni CI
      = [",-tst$K2,",",tst$K2,"]"));
  report <- c(report,paste("Max rho = ",max(rho)));
  report <- c(report,paste("Magnitude Bonferonni CI
      = [0,",tst$L2,"]"));
}
```

```
    # histogram of coefficients
    hist(g,breaks=100,main=paste("Coefficients ", rngkind))
    title(sub=paste("k=",k," b=",b," N=",N))

    # qq plot with regression line
    qqnorm(g); qqline(g)
    title(sub=paste("Overall test p=",tst$p));

    # Plot coefficients
    plot(g[gr],pch=21,ylim=c(gmin,gmax),main="Fourier
       Coefficients");
    points(g[gi],pch=23);

    # Add confidence limits:
    # paint a white line to clear the points and then
    lines(rep(tst$K1,2*(T-1)),lty="solid",col='white');
    lines(rep(-tst$K1,2*(T-1)),lty="solid",col='white');
    # then put in a dashed line
    lines(rep(tst$K1,2*(T-1)),lty="dashed",col='black');
    lines(rep(-tst$K1,2*(T-1)),lty="dashed",col='black');
    #
    title(sub=paste("X=",tst$X," C[",tst$a,"]=",tst$C));

    # Add bonferonni limits
    lines(rep(tst$L1,2*(T-1)));
    lines(rep(-tst$L1,2*(T-1)));

    # histogram of magnitudes
    hist(rho,breaks=100„main=paste("Magnitudes ", rngkind))

    # qq plot of magnitudes against
    # quantiles of a chisq with df=2
    qqplot(rho,rchisq(rho,2),main="Chi-Squared (df=2) Q-Q
       Plot",xlab="Theoretical Quantiles",ylab="Sample
       Quantiles")

    # plot magnitudes
    plot(rho,main="Fourier magnitudes",ylim=c(0,rhomax))

    # Add confidence limits (use white to stand out)
    lines(rep(tst$K2,2*(T-1)),lty="solid",col='white');
    lines(rep(tst$K2,2*(T-1)),lty="dashed",col='black');

    # Add bonferonni limits
    lines(rep(tst$L2,2*(T-1)));

    # return results to caller
    list(Report=report,Data=tst)
  }
```

testRNG.R: Test R generators in 1 and 2 dimensions

LISTING B.5

```
# Pull in the test functions
source('mdftt.R')
source('tview.R')

logfile <- 'testRNG.log';
#logfile <- NULL

sink(logfile)
ps.options(onefile=FALSE, paper="special", print.it=
   FALSE)

E <- 1000

postscript('Marsaglia-Multicarry.1.ps');
RNGkind("Marsaglia-Multicarry")
MarsagliaMulticarry.seed <- .Random.seed;
source("example1d.R",echo=TRUE)
dev.off()
sink('Marsaglia-Multicarry.1.summary')
TestResult$Report
sink(logfile)
system('gv Marsaglia-Multicarry.1.ps &')

postscript('Super-Duper.1.ps')
RNGkind("Super-Duper")
SuperDuper.seed <- .Random.seed;
source("example1d.R",echo=TRUE)
dev.off()
sink('Super-Duper.1.summary')
TestResult$Report
sink(logfile)
system('gv Super-Duper.1.ps &')

postscript('Wichmann-Hill.1.ps')
RNGkind("Wichmann-Hill")
WichmannHill.seed <- .Random.seed;
source("example1d.R",echo=TRUE)
dev.off()
sink('Wichmann-Hill.1.summary')
TestResult$Report
sink(logfile)
system('gv Wichmann-Hill.1.ps &')

postscript('Mersenne-Twister.1.ps')
RNGkind("Mersenne-Twister")
MersenneTwister.seed <- .Random.seed;
source("example1d.R",echo=TRUE)
dev.off()
sink('Mersenne-Twister.1.summary')
TestResult$Report
```

```
sink(logfile)
system('gv Mersenne-Twister.1.ps &')

postscript('Knuth-TAOCP.1.ps')
RNGkind("Knuth-TAOCP")
KnuthTAOCP.seed <- .Random.seed;
source("example1d.R",echo=TRUE)
dev.off()
sink('Knuth-TAOCP.1.summary')
TestResult$Report
sink(logfile)
system('gv Knuth-TAOCP.1.ps &')

postscript('Marsaglia-Multicarry.2.ps')
.Random.seed <- MarsagliaMulticarry.seed;
source("example2d.R",echo=TRUE)
dev.off()
sink('Marsaglia-Multicarry.2.summary')
TestResult$Report
sink(logfile)
system('gv Marsaglia-Multicarry.2.ps &')

postscript('Super-Duper.2.ps')
.Random.seed <- SuperDuper.seed;
source("example2d.R",echo=TRUE)
dev.off()
sink('Super-Duper.2.summary')
TestResult$Report
sink(logfile)
system('gv Super-Duper.2.ps &')

postscript('Wichmann-Hill.2.ps')
.Random.seed <- WichmannHill.seed;
source("example2d.R",echo=TRUE)
dev.off()
sink('Wichmann-Hill.2.summary')
TestResult$Report
sink(logfile)
system('gv Wichmann-Hill.2.ps &')

postscript('Mersenne-Twister.2.ps')
.Random.seed <- MersenneTwister.seed;
source("example2d.R",echo=TRUE)
dev.off()
sink('Mersenne-Twister.2.summary')
TestResult$Report
sink(logfile)
system('gv Mersenne-Twister.2.ps &')

postscript('Knuth-TAOCP.2.ps')
.Random.seed <- KnuthTAOCP.seed;
source("example2d.R",echo=TRUE)
dev.off()
```

```
sink('Knuth-TAOCP.2.summary')
TestResult$Report
sink(logfile)
system('gv Knuth-TAOCP.2.ps &')
```

testRgen.R: Run test sequences against R generaotrs

LISTING B.6

```
logfile <- 'testRgen.log';
#logfile <- NULL

r <- 100; # Number of loops to run
NMax <- 100;

sink(logfile)
ps.options(onefile=FALSE, paper="special", width=6,
   height=6, print.it=FALSE, horizontal=FALSE)

Ebase <- c( 8, 8, 5,4,12)
B   <- c(8192,64,16,8,4)

runtest <- function(RNG,fn,r,NMax) {

 # Generate enough random numbers and write them to a
    file
 # to send them to the C program for analysis.
 RNGkind(RNG);
 rn <- RNGkind()[1]
 rs <- runif((2^16)*NMax);
 rnf <- paste(fn,"runif",sep=".");
 write(rs,rnf,ncolumns=1);

 # Do multiple runs
 E <- Ebase;
 for( K in 1:5 ){
  bfn <- paste("multiRun//",paste(fn,K,sep="."),sep="")
     ;
  log <- paste(bfn,".log",sep="");
  mdftt.cmd <- paste("mdftt 1 -1",rn,E[K],K,B[K],10,r
     ,1,bfn,">",log,"<",rnf,sep=" ");
  mdftt.cmd
  system(mdftt.cmd);
  system(paste('mv mdfttreport.Rout ',bfn,'.Rout',sep=
     ""));
 }

 # Do large N run
 E <- Ebase*NMax;
 for( K in 1:5 ){
  bfn <- paste("largeN//",paste(fn,K,sep="."),sep="");
```

```
    log <- paste(bfn,".log",sep="");
    mdftt.cmd <- paste("mdftt 1 -1",rn,E[K],K,B[K
        ],10,1,1,bfn,">",log,"<",rnf,sep=" ");
    mdftt.cmd
    system(mdftt.cmd);
    system(paste('mv mdfttreport.Rout ',bfn,'.Rout',sep=
        ""));
  }

  # Multiple N runs increasing from 10 to NMax by 10
  for( NM in seq(10,NMax,by=10) ){
    E <- Ebase*NM;
    for( K in 1:5 ){
      bfn <- paste("multiN//",paste(fn,K,NM,sep="."),sep
          ="");
      log <- paste(bfn,"log",sep=".");
      mdftt.cmd <- paste("mdftt 1 -1",rn,E[K],K,B[K
          ],10,1,1,bfn,">",log,"<",rnf,sep=" ");
      mdftt.cmd
      system(mdftt.cmd);
      system(paste('mv mdfttreport.Rout ',bfn,'.Rout',sep=
          ""));
    }
  }
}

runtest("Marsaglia-Multicarry","MM",r,NMax);
runtest("Super-Duper","SD",r,NMax);
runtest("Wichmann-Hill","WH",r,NMax);
runtest("Mersenne-Twister","MT",r,NMax);
runtest("Knuth-TAOCP","KT",r,NMax);
```

multiRun-analysis.R: multi-run graphics

LISTING B.7

```
multiRun <- read.table('multiRun.txt',header=TRUE,sep=
    ",");

sumview <- function(t,RNGName,RNGTitle) {

  attach(t);

  # select the desired generator
  RNG <- RNG==RNGName;

  ps.options(onefile=FALSE, paper="special", width=6,
      height=6,
       print.it=FALSE, horizontal=FALSE);
  GraphicFileName <- paste(RNGName,".multiRun.ps",sep="")
      ;
```

```
  # Open the plot window
  postscript(GraphicFileName);
  par(mfcol=c(3,2))

  # Overall histogram
  hist((X[RNG]-C[RNG]),breaks=100,main=RNGTitle, xlab="X
     minus C")

  # plots
  for(kd in 1:5) {
    # quantiles of a chisq with df=2
    sDF <- DF[k==kd][1];
    RNGk <- RNG & (k==kd);
    plot(p[RNGk],ylim=c(0,1),
       main=paste("p-values k=",kd),
       xlab="",ylab="");
  }

  dev.off();
  system(paste('gv ',GraphicFileName,'&'));

  detach(t);
}

sumview(multiRun,"MM","Marsaglia-Multicarry: Multiple
   Runs");

sumview(multiRun,"WH","Wichmann-Hill: Multiple Runs");

sumview(multiRun,"MT","Mersenne-Twister: Multiple Runs")
   ;

sumview(multiRun,"SD","Super-Duper: Multiple Runs");

sumview(multiRun,"KT","Knuth-TAOCP: Multiple Runs");
```

multiN-analysis.R: multiple sample size (N) graphics

LISTING B.8

```
multiN <- read.table('multiN.txt',header=TRUE,sep=",");

sumview <- function(t,RNGName,RNGTitle) {

  attach(t);

  # select the desired generator
  RNG <- RNG==RNGName;
```

```
  ps.options(onefile=FALSE, paper="special", width=6,
     height=6,
      print.it=FALSE, horizontal=FALSE);
  GraphicFileName <- paste(RNGName,".multiN.ps",sep="");

  # Open the plot window
  postscript(GraphicFileName);

  # coplot of X - C by dimension and by N
  coplot((X[RNG]-C[RNG]) ~ N[RNG] | as.factor(k[RNG]),
      rows=1, xlab=RNGTitle )

  dev.off();
  system(paste('gv ',GraphicFileName,'&'));

  detach(t);
}

sumview(multiN,"MM","Marsaglia-Multicarry: Multiple N");

sumview(multiN,"WH","Wichmann-Hill: Multiple N");

sumview(multiN,"MT","Mersenne-Twister: Multiple N");

sumview(multiN,"SD","Super-Duper: Multiple N");

sumview(multiN,"KT","Knuth-TAOCP: Multiple N");
```

largeN-anlaysis.R: Display large N graphics

LISTING B.9

```
largeN <- read.table('largeN.txt',header=TRUE,sep=",");

sumview <- function(t,RNGName,RNGTitle) {

  attach(t);

  # select the desired generator
  RNG <- RNG==RNGName;

  plot((X[RNG]-C[RNG]) ~ as.factor(k[RNG]), xlab=paste(
     RNGTitle," (X-C)"), title="", ylab="" )
  lines(rep(0,length(X[RNG])),lty="dashed")
  plot(p[RNG] ~ as.factor(k[RNG]), xlab=paste(RNGTitle,
     " (p)"), title="", ylab="" )
  lines(rep(0.10,length(p[RNG])),lty="dashed")
  detach(t);
}
```

```
  ps.options(onefile=FALSE, paper="special", width=8.5,
     height=11,
      print.it=FALSE, horizontal=FALSE);
  GraphicFileName <- "largeN.ps";

  # Open the plot window
  postscript(GraphicFileName);
  par(mfrow=c(5,2))


sumview(largeN,"MM","Marsaglia-Multicarry");

sumview(largeN,"SD","Super-Duper");

sumview(largeN,"MT","Mersenne-Twister");

sumview(largeN,"KT","Knuth-TAOCP");

sumview(largeN,"WH","Wichmann-Hill");

  dev.off();
  system(paste('gv ',GraphicFileName,'&'));
```

2.2. EST Source Code

NOTE. The early versions of the EST were called mdftt for MultiDimensional FFT.Source code has not been updated to reflect the new name. Also, Numerical Recipes(NR) is not free software and therefore cannot be distributed here.We would suggest replacing the NR routines with FFTW.

Makefile: mdftt build control

LISTING B.10

```
CC = gcc
CCC = gcc
CFLAGS = -g
LFLAGS = -L./
LLIBS = -lg++

ROBJS = pgfsr.o pcrand.o pcomb.o ranlib.o \
      super.o randu.o random.o lprand.o

NROBJS = nrutil.o complex.o fourn.o fourfs.o fourew.o

CDFLIBOBJS = ipmpar.o dcdflib.o

mdftt: mdftt.o cell.o incr.o uniform.o \
            libunif.o libnr.a libcdf.a
```

```
	$(CCC) $(LFLAGS) -o mdftt mdftt.o cell.o incr.o
	   uniform.o \
	      dcdflib.o ipmpar.o -lunif -lnr -lm -lcdf

clean:
	rm *.o
	rm *.a
	rm mdftt

libunif.a: $(ROBJS)
	rm -f libunif.a
	ar cr libunif.a $(ROBJS)
	ranlib libunif.a

libnr.a: $(NROBJS)
	rm -f libnr.a
	ar cr libnr.a $(NROBJS)
	ranlib libnr.a

libcdf.a: $(CDFLIBOBJS)
	rm -f libcdf.a
	ar cr libcdf.a $(CDFLIBOBJS)
	ranlib libcdf.a

mdftt.o: mdftt.c uniform.h nrutil.h complex.h

testcell.o: testcell.c cell.c ipow.c


dcdflib.o: dcdflib.c cdflib.h

ipmpar.o: ipmpar.c

nrutil.o: nrutil.c
complex.o: complex.c
fourn.o: fourn.c
fourfs.o: fourfs.c
fourew.o: fourew.c

gammln.o: gammln.c
gammp.o: gammp.c
gammq.o: gammq.c
gcf.o: gcf.c
gser.o: gser.c

cell.o: cell.c
incr.o: incr.c
Kcell.o: Kcell.c
mdftt.o: mdftt.c

RNGHEADERS=pgfsr.h randu.h pcrand.h super.h pcomb.h
   ranlib.h lprand.h
```

```
uniform.o: uniform.c uniform.h $(RNGHEADERS)
random.o: random.c random.h
randu.o: randu.c randu.h
pgfsr.o: pgfsr.c pgfsr.h
pcrand.o: pcrand.c pcrand.h
super.o: super.c super.h
pcomb.o: pcomb.c pcomb.h pgfsr.h
ranlib.o: ranlib.c ranlib.h
```

mdftt.c: Main mdftt program

LISTING B.11

```
/* Multi Dimensional Fourier Transform Test */
/* ftt test of pdf of random number stream */

#define TRUE -1
#define FALSE 0

//#define CS 3.0        // Initial Standard deviations
   at cutoff
//#define DCS 0.01      // Delta Cutoff Stepsize for
   search

#define np 8
int dp[np] = {0,31,98,250,521,607,1279,1279};
int dq[np] = {0, 6,27,103, 32,273, 418,1063};

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "uniform.h"
#include "nrutil.h"
#include "complex.h"

int K;      // dimensions to be used
int B;      // number of bins in each dimension
long T;     // total number of bins = B^K
long N;     // number of points to examine
float E;    // Expected cell frequency
double A;   // Alpha level
int NL;     // Number of loops (tests) to perform
int rng;    // rng identifier
char rng_name_local[80];
int subrng; // pgfsr parameter set
long delay; // pgfsr column-column delay

int printlev; // control output
            // 0: No output files, just summarize
            // 1: Output fourier coefficients
            // 2: Output level 1 + random number file
```

```
            // 3: Output level 2 + bin file

int overlap; // overlapping k-tuples indicator
int ram;    // ram or file indicator

float *bin; // frequencies storage
rstream **rs; // rng stream pointers

char basefn[80];

char runlabel[80]="mdftt"; // label to update status
   file

FILE *sf = NULL; // Summary statistics file

#define min(x,y)  ( ((x)<(y)) ? (x) : (y) )
#define max(x,y)  ( ((x)>(y)) ? (x) : (y) )

//double ncdf(double x);

// cell count and identification functions
long cell(float *x,int K,int B);
int incr(unsigned long *nn,int K,int B);

// command line input functions
void ArgumentHelp(void);
void ProcessArguments(char **argv,int first);

// Numerical Recipes FFT functions
void fourfs(FILE *file[5], unsigned long nn[], int ndim
   , int isign);
void fourn(float data[],long nn[],int ndim,int isign);

// Get parameters interactively
void GetParameters(int first);


void GetParameters(int first)
{ int i;
  char inbuf[80];

  printf("Storage type 0 - FILE, 1 - RAM :");
  ram = atoi(gets(inbuf)); // rng indentifier

  printf("Random number generator type\n");
  printf(" 1 PGFSR 2 LPRAND 3 RANDU\n");
  printf(" 4 SUPER 5 PCRAND 6 RANLIB\n");
  printf(" 7 PCOMB\n");
  printf(" :");
  rng = atoi(gets(inbuf)); // rng indentifier
  if(abs(rng)>NRNG) ArgumentHelp();
  if(rng<0) {
    overlap = TRUE;
```

```
  rng = -rng;
} else
  overlap = FALSE;
delay = 0L;
if(rng<3&&rng>0) {
  printf(" 1 2  3  4  5   6   7\n");
  printf("-----------------------------\n");
  printf(" 31 98 250 521 607 1279 1279\n");
  printf(" 6 27 103 32 273 418 1063\n");
  printf(" :");
  subrng = atoi(gets(inbuf));
  if(subrng>np) {
    printf("Allowed subranges are 1-%d.\n",np);
    ArgumentHelp();
  }
  if(rng==2) { // lprand uses a delay value
     printf("Delay (%ld):",100*dp[subrng]);
     delay = atol(gets(inbuf));
  }
}
else subrng = 0;

printf("expected cell frequency :");
E = atof(gets(inbuf));

printf("dimensions to be used :");
K = atoi(gets(inbuf));

printf("number of bins in each dimension :");
B = atoi(gets(inbuf));

printf("cutoff probability :");
A = atof(gets(inbuf))/100.0;

printf("number of loops [tests] to perform :");
NL= atoi(gets(inbuf));

printf("printlevel :");
printlev = atoi(gets(inbuf));

if(printlev>0) {
  printf("base output file name :");
  strncpy(basefn,gets(inbuf),80);
}

// force the number of bins/dimension to be a power of
    2
for(i=0;i<sizeof(int);i++) {
   if( 1<<i >= B ){
     B = 1<<i;
     break;
   }
}
```

```
  T = 1L;
  for(i=0;i<K;i++)T *= B;

  N = (long)( (float)T*E );

  printf("Running %d test(s) at level %f of %ld points in
      %d dim \n",NL,A,N,K);

  if(rng>0) {
    printf("from %s ",rng_name(rng));
  }
  else {
    printf("from -stdin- ");
  }

  printf("using %d bins/dim, %ld total bins.\n",B,T);

  if(subrng!=0) {
    printf("PGFSR parameters p = %d, q = %d\n",dp[subrng
        ],dq[subrng]);
  }

  if(delay!=0) printf("PGFSR delay = %ld\n",delay);

  printf("Resolution of test is %f in each dimension.\n"
     ,1.0/(float)B);

  if(overlap) printf("Overlapping k-tuples being used.\n"
     );

}

void ProcessArguments(char **argv,int first)
{ int argp;
  int i;

  argp = 1;
  ram = atoi(argv[argp++]); // ram/file indicator
  rng = atoi(argv[argp++]); // rng indentifier
  if(abs(rng)>NRNG) ArgumentHelp();
  // if(rng<0) {
  //   overlap = TRUE;
  //   rng = -rng;
  // } else
  //   overlap = FALSE;
  delay = 0L;
  if(rng>0&&rng<3) {
    subrng = atoi(argv[argp++]);
    if(subrng>np) {
       printf("Allowed subranges are 1-%d.\n",np);
       ArgumentHelp();
```

```
  }
  if(rng==2) delay = atol(argv[argp++]);
} else
  if(rng<0) {
    // get name from commmand line
    strncpy(rng_name_local,argv[argp++],80);
  }
else subrng = 0;
E = atof(argv[argp++]); // expected cell frequency (to
    calc N)
K = atoi(argv[argp++]); // dimensions to be used
B = atoi(argv[argp++]); // number of bins in each
    dimension
A = atof(argv[argp++])/100.0; // cutoff probability for
     individual idents
NL= atoi(argv[argp++]); // number of loops [tests] to
    perform
printlev = atoi(argv[argp++]);

// force the number of bins/dimension to be a power of
     2
// to maintain the efficiency of the FFT algorithm
for(i=0;i<sizeof(int);i++) {
   if( 1<<i >= B ){
     B = 1<<i;
     break;
   }
}

T = 1L;
for(i=0;i<K;i++)T *= B;

N = (long)( (float)T*E );

if(first)
  fprintf(sf,"
     ---------------------------------------------\n")
     ;

printf("Running %d test(s) at level %f of %ld points in
     %d dim \n",NL,A,N,K);
if(first) fprintf(sf,"Running %d scan(s) and test(s) at
     level %f ",NL,A);
if(rng>0) {
  printf("from %s ",rng_name(rng));
  if(first) fprintf(sf,"on %s \n",rng_name(rng));
} else {
  printf("from -stdin- [%s]",rng_name_local);
  if(first) fprintf(sf,"from -stdin- [%s]\n",
     rng_name_local);
}
```

```
  printf("using %d bins/dim, %ld total bins.\n",B,T);

  if(subrng!=0) {
   printf("PGFSR parameters p = %d, q = %d\n",dp[subrng
      ],dq[subrng]);
   if(first) fprintf(sf,"PGFSR parameters p = %d, q = %d
      \n",dp[subrng],dq[subrng]);
  }

  if(delay!=0) {
   printf("PGFSR delay = %ld\n",delay);
   if(first) fprintf(sf,"PGFSR delay = %ld\n",delay);
  }

  printf("Resolution of test is %f in each dimension.\n"
     ,1.0/(float)B);

  if(overlap) {
   printf("Overlapping k-tuples being used.\n");
   if(first) fprintf(sf,"Overlapping k-tuples being used
      .\n");
  }

}

void ArgumentHelp()
{ int i,j;
  printf("Multidimension Fourier transform test of random
      number generator\n");
   printf(" rng: generator number. subrng: pgfsr subtype
      \n");
   printf(" E: exp freq. K: # of dim. B: partitions p:
      alpha level.\n");
   printf(" Output levels:\n");
   printf(" 0 => basic         1 => stats for failures
      only.\n");
   printf(" 2 => save random numbers 3 => stats for all
      .\n");
   printf(" Valid generators:\n");
   printf(" 0: Read random stream from stdin\n");
   for(i=1;i<NRNG;i++) printf(" %d: %s\n",i,rng_name(i))
      ;
   printf(" GFSR sub generators:\n");
   for(j=1;j<np;j++) printf(" %d => %d, %d\n",j,dp[j],dq
      [j]);
   printf("usage: mdftt <ram> <rng> [subrng [delay]] <E
      > <K> <B> <A> <NL> <out> [filename]\n");
   exit(1);
}

FILE *OpenFTFile(char *fn,FILE *ftf,int n)
{
```

```
  char ftfile[80]; // output fourier transform data file
     name
  int i;
  FILE *ftfdesc;

  sprintf(ftfile,"%s.%04d.ft",fn,n);

  if(ftf)
    ftf = freopen(ftfile,"w",ftf);
  else
    ftf = fopen(ftfile,"w");

  if(ftf) {
    printf("Writing fourier data to file [%s].\n",ftfile)
       ;
    fprintf(ftf,"Re Im Rho Phi\n");
     return(ftf);
  }
  else {
    printf("Failed to open transform file %f\n",ftfile);
    return(NULL);
  }
}

void WriteRFiles(char *fn,int n,long N,long T,double A,
   double X,double P,double C,double K1,double K2,double
    L1,double L2)
{
  char ftfile[80];

  sprintf(ftfile,"%s.%04d.desc",fn,n);
  { // Description file output
    FILE *ftfdesc = fopen(ftfile,"w");
    fprintf(ftfdesc,"N K B T A RNG\n");
    fprintf(ftfdesc,"%ld %d %d %ld %lf \"%s\"\n",N,K,B,T,
       A,rng_name_local);
    fclose(ftfdesc);
  }
  sprintf(ftfile,"%s.%04d.result",fn,n);
  { // results file
    FILE *sf = fopen(ftfile,"w");
    fprintf(sf,"N T a X p C K1 K2 L1 L2\n");
    fprintf(sf,"%ld %ld %lf %lf %lf %lf %lf %lf %lf %lf\n
       ",N,T,A,X,P,C,K1,K2,L1,L2);
    fclose(sf);
  }
  { // data load script
    FILE *sf = fopen("loaddat.R","w");
    fprintf(sf,"postscript('%s.%04d.ps')\n",fn,n);
    fprintf(sf,"ftdata <- read.table(\"%s.%04d.ft\",head=
       TRUE);\n",fn,n);
```

```
    fprintf(sf,"ftdesc <- read.table(\"%s.%04d.desc\",
       head=TRUE);\n",fn,n);
    fprintf(sf,"ftrslt <- read.table(\"%s.%04d.result\",
       head=TRUE);\n",fn,n);
    fclose(sf);
  }
}

FILE *OpenRNFile(char *fn,FILE *rnf,int n)
{
 char rnfile[80]; // output random number file
  int i;
  sprintf(rnfile,"%s.%04d.rn",fn,n);
  if(rnf)
    rnf = freopen(rnfile,"w",rnf);
  else
    rnf = fopen(rnfile,"w");
  if(!rnf) {
    printf("Failed to open random number file %f\n",
       rnfile);
  }
  return(rnf);
}

FILE *OpenBinFile(char *fn,FILE *bf,int n)
{
 char bfile[80];  // output random number file
  int i;
  sprintf(bfile,"%s.%04d.bn",fn,n);
  if(bf)
    bf = freopen(bfile,"w",bf);
  else
    bf = fopen(bfile,"w");
  if(!bf) {
    printf("Failed to open cell frequency file %f\n",
       bfile);
  }
  return(bf);
}

void OpenScratch(FILE **f,char *name)
{ *f = fopen(name,"w+");
  if(*f==NULL) {
    printf("Failed open on scratch file %s\n",name);
    exit(1);
  }
}

int WriteScratch(float *buf, int wordsize, int words,
   FILE *f)
{ int cc;
```

```
  cc = fwrite(buf,wordsize,words,f);
  if( cc != words ){
   printf("Failed write\n");
   return(FALSE);
  }
  return(TRUE);
}

int ReadScratch(float *buf, int wordsize, int words,
    FILE *f)
{ int cc;
  cc=fread(buf,wordsize,words,f);
  if( cc != words ){
   printf("Failed read\n");
   return(FALSE);
  }
  return(TRUE);
}

main(int argc,char **argv)
{
  long i,j,k;      // loop indices
  long l;          // long loop index
  long c;          // cell pointer
  float *y;        // individual point in d-space

  long nulls;      // Number of null cells found

  FILE *ftf=NULL,*rnf=NULL,*bf=NULL; // output file
     pointers

  int DCDFLIB_WHICH; // calc control parameter for
     DCDFLIB
  double DCDFQ;    // 1-A
  double DCDFP;    // 1-A
  double DF;       // degrees of freedom for chi-square
  int DCDFLIB_STATUS; // status from DCDFLIB
  double DCDFLIB_BOUND; // DCDFLIB BOUND

  double C;        // Chisquare critical value for
     overall test
  double K1,K2;    // Fisher PLSD critical values
  double L1,L2;    // Bonferonni critical values

  double NK1,NK2;        // Fisher PLSD critical values
  double NL1,NL2;        // Bonferonni critical values

  float rho,phi;   // magnitude squared & phase angle
  double Fm,Fs;    // Sum of Rho for test statistic
  double X;        // overall test statistic
  double smf,sfsq,sfxy; // summary statistics
     intermediates
```

```
    double Mu,Sigma;
410 float CellP;
    size_t NT;

    int *mfail;      // number of suspects found
    int *rfail;
415 int *ifail;
    int *ofail;

    float re,im;
    int il;
420 int o;
    FILE *file[5];
    unsigned long *nn,*nrnn;
    float zero=0.0;
    int cc;
425 float *nrbin;
    int FirstSummary=TRUE;

    printf("
       ------------------------------------------------\n
       ");

430 sf = fopen("mdftt-summary.txt","r");
    if(sf) {
      // close it to reopen with append
      fclose(sf);
      FirstSummary=FALSE;
435 }
    sf = fopen("mdftt-summary.txt","a+");

    if(argc==2) {
        ArgumentHelp();
440 }
    else {
      if(argc>5) {
        ProcessArguments(argv,FirstSummary);
        // save the base file name from last input parameter
445     strncpy(basefn,argv[argc-1],80);
        if(rng>0)strncpy(rng_name_local,rng_name(rng),80);
      } else {
        GetParameters(FirstSummary);
      }
450 }

    if(ram) printf("Using fourn memory based algorithm\n");
    else    {
      printf("Using fourfs disk based algorithm\n");
455   OpenScratch(&file[1],"mdftt.1");
      OpenScratch(&file[2],"mdftt.2");
      OpenScratch(&file[3],"mdftt.3");
      OpenScratch(&file[4],"mdftt.4");
    }
```

```
460
     // Initialize the bin structure
     NT=2*T;
     if(!(bin = malloc(NT*sizeof(float)))) {
       printf("Allocation failure in mdftt: bin.\n");
465    printf("Attempted to allocate space for %ld floats.\n
          ",NT);
       return(2);
     }

     // Initialize the failure structures used to flag
470  // potentially high coefficients based on standard
     // or bonferonni intervals.
     if(!(rfail = malloc(2*NL*sizeof(int)))) {
       printf("Allocation failure in mdftt: rfail.\n");
       return(2);
475  }
     if(!(ifail = malloc(2*NL*sizeof(int)))) {
       printf("Allocation failure in mdftt: ifail.\n");
       return(2);
     }
480  if(!(mfail = malloc(2*NL*sizeof(int)))) {
       printf("Allocation failure in mdftt: mfail.\n");
       return(2);
     }
     if(!(ofail = malloc(2*NL*sizeof(int)))) {
485    printf("Allocation failure in mdftt: ofail.\n");
       return(2);
     }
     for(il=0;il<NL*2;il++) {
       rfail[il] = 0;
490    ifail[il] = 0;
       mfail[il] = 0;
       ofail[il] = 0;
     }

495  if(!(y = malloc(K*sizeof(float)))) {
       printf("Allocation failure in mdftt: y.\n");
       return(2);
     }

500  if(rng>0) {
        printf("Initializing the random number generator(s)\
           n");
        if(overlap) rs = (rstream **) malloc(sizeof(rstream)
           );
        else      rs = (rstream **) malloc(K*sizeof(rstream
           ));
        if (!rs) {
505       printf("Allocation failure in mdftt: rs.");
          exit(2);
        }
```

```
    // initialize the random number generators
    if(overlap) rs[0] = setunif(rng,0,1,-1L,dp[subrng],
       dq[subrng],delay);
    else
      for(i=0;i<K;i++)
        rs[i] = setunif(rng,i,K,-1L,dp[subrng],dq[subrng
           ],delay);

    if(overlap) {
      for(o=0;o<K;o++)
        y[o] = unif(rs[0]);
      o = 0;
    }
}

nn = (unsigned long *)malloc(K*sizeof(unsigned long));

// create pointers for the numerical recipes routines
nrbin = bin;
--nrbin;
nrnn = nn;
--nrnn;

// use cdflib to calculate the critical values
DCDFLIB_WHICH = 2; // calculate X

//
//  # Overall test critical value
//  C <- qchisq(1-a,2*(T-1))

DF = 2.0*((double)T-1.0); // use 2(T-1) for df
DCDFP = 1.0 - A;
DCDFQ = 1.0 - DCDFP;
cdfchi( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &C, &DF, &
   DCDFLIB_STATUS, &DCDFLIB_BOUND);

switch(DCDFLIB_STATUS) {
case 0:
  printf("Overall cutoff at C = %.3f, DF = %.0f.\n",(
     float)C,(float)DF);
  break;
case 1:
  printf("Answer appears to be lower than lowest search
      bound.\n");
  break;
case 2:
  printf("Answer appears to be higher than greatest
     search bound.\n");
  break;
case 3:
  printf("P + Q != 1.\n");
```

```
  break;
case 10:
  printf("Error returned from cumgam.\n");
  break;
default:
  if(DCDFLIB_STATUS<0) {
    printf("Paramerter %d is out of range.\n",-
       DCDFLIB_STATUS);
  }
  else printf("Unexpected status returned [%d].\n",
     DCDFLIB_STATUS);
  break;
}

// # Fisher PLSD critical values

// K1 <- qnorm(1-a/2)
Mu = 0.0;
Sigma = 1.0; //sqrt(1.0/2.0/(float)N);
DCDFP = 1.0 - A/2.0;
DCDFQ = 1.0 - DCDFP;
cdfnor( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &K1, &Mu, &
   Sigma, &DCDFLIB_STATUS, &DCDFLIB_BOUND);

switch(DCDFLIB_STATUS) {
case 0:
  printf("Individual PLSD cutoff at K1 = %.3f.\n",(
     float)K1);
  break;
case 1:
  printf("Answer appears to be lower than lowest search
      bound.\n");
  break;
case 2:
  printf("Answer appears to be higher than greatest
     search bound.\n");
  break;
case 3:
  printf("P + Q != 1.\n");
  break;
case 10:
  printf("Error returned from cumgam.\n");
  break;
default:
  if(DCDFLIB_STATUS<0) {
    printf("Paramerter %d is out of range.\n",-
       DCDFLIB_STATUS);
  }
  else printf("Unexpected status returned [%d].\n",
     DCDFLIB_STATUS);
  break;
```

```
}

// K2 <- qchisq(1-a,2)
DF = 2;
DCDFP = 1.0 - A;
DCDFQ = 1.0 - DCDFP;
cdfchi( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &K2, &DF, &
   DCDFLIB_STATUS, &DCDFLIB_BOUND);

switch(DCDFLIB_STATUS) {
case 0:
printf("Magnitude Squared PLSD cutoff at K2 = %.3f.\n"
   ,(float)K2);
  break;
case 1:
  printf("Answer appears to be lower than lowest search
      bound.\n");
  break;
case 2:
  printf("Answer appears to be higher than greatest
     search bound.\n");
  break;
case 3:
  printf("P + Q != 1.\n");
  break;
case 10:
  printf("Error returned from cumgam.\n");
  break;
default:
  if(DCDFLIB_STATUS<0) {
    printf("Paramerter %d is out of range.\n",-
       DCDFLIB_STATUS);
  }
  else printf("Unexpected status returned [%d].\n",
     DCDFLIB_STATUS);
  break;
}

// # Bonferonni critical values
// L1 <- qnorm(1-a/(4*(T-1)))
Mu = 0.0;
Sigma = 1.0;
DCDFP = 1.0 - A/(4*((double)T-1));
DCDFQ = 1.0 - DCDFP;
cdfnor( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &L1, &Mu, &
   Sigma, &DCDFLIB_STATUS, &DCDFLIB_BOUND);
switch(DCDFLIB_STATUS) {
case 0:
  printf("Individual Bonferonni cutoff at L1 = %.3f.\n"
     ,(float)L1);
```

```
      break;
    case 1:
      printf("Answer appears to be lower than lowest search
          bound.\n");
      break;
    case 2:
      printf("Answer appears to be higher than greatest
         search bound.\n");
      break;
    case 3:
      printf("P + Q != 1.\n");
      break;
    case 10:
      printf("Error returned from cumgam.\n");
      break;
    default:
      if(DCDFLIB_STATUS<0) {
        printf("Paramerter %d is out of range.\n",-
            DCDFLIB_STATUS);
      }
      else printf("Unexpected status returned [%d].\n",
         DCDFLIB_STATUS);
      break;
    }

    // L2 <- qchisq(1-a/(2*(T-1)),2)
    DF = 2;
    DCDFP = 1.0 - A/(2*((double)T-1));;
    DCDFQ = 1.0 - DCDFP;
    cdfchi( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &L2, &DF, &
       DCDFLIB_STATUS, &DCDFLIB_BOUND);
    switch(DCDFLIB_STATUS) {
    case 0:
      printf("Magnitude Squared Bonferonni cutoff at L2
          = %.3f.\n",(float)L2);
      break;
    case 1:
      printf("Answer appears to be lower than lowest search
          bound.\n");
      break;
    case 2:
      printf("Answer appears to be higher than greatest
         search bound.\n");
      break;
    case 3:
      printf("P + Q != 1.\n");
      break;
    case 10:
      printf("Error returned from cumgam.\n");
      break;
    default:
      if(DCDFLIB_STATUS<0) {
```

```
      printf("Paramerter %d is out of range.\n",-
          DCDFLIB_STATUS);
    }
    else printf("Unexpected status returned [%d].\n",
        DCDFLIB_STATUS);
    break;
690 }

  // adjust the CI values to be applied directly to the
     coefficients before normalization
  NK1 = K1/sqrt(2.0*(float)N);
  NK2 = K2/sqrt(2.0*(float)N);
695 NL1 = L1/(2.0*(float)N);
  NL2 = L2/(2.0*(float)N);

  CellP = 1.0/(float)N;
  for(il=0;il<NL;il++) { // main testing loop
700
    printf("\nLoop %d\n",il+1);

    // printlev
    // 0: No output files, just summarize
705 // 1: Output fourier coefficients
    // 2: Output level 1 + random number file
    // 3: Output level 2 + bin file

    if( printlev>0 ){ ftf = OpenFTFile(basefn,ftf,il+1);
        }
710 if( printlev>1 ){ rnf = OpenRNFile(basefn,rnf,il+1);
        }
    if( printlev>2 ){ bf = OpenBinFile(basefn,bf,il+1); }

    for(l=0;l<T;l++) bin[l] = 0.0;

715 for(l=0;l<N;l++) {
      // generate a point in K-space
      //if(rnf) fprintf(rnf,"%ld, ",l);
        if(overlap) {
          o = (++o) % K;
720       if(rng>0) {
            y[o] = unif(rs[0]);
          }
          else {
            if( scanf("%f\n",&y[o]) < 1 ){
725           printf("Out of input on stdin at %ld:%d\n",l,o
                  );
              exit(3);
            }
          }
        }
730     else {
          for(j=0;j<K;j++) {
```

```
        if(rng>0) y[j] = unif(rs[j]);
        else {
          char buf[80];
          if( gets(buf)==NULL ){
            printf("Out of input on stdin at %ld:%d\n",l
                ,j);
            exit(3);
          } else
            y[j] = atof(buf);

        }
        if(rnf) fprintf(rnf," %f",y[j]);
      }
    }
    c = cell(y,K,B);  // find which cell it's in
    bin[c] += CellP;  // update frequence counter
    if(rnf) //fprintf(rnf," %ld, %f\n",c,bin[c]);
      fprintf(rnf,"\n");
  }

  if(bf) {
    for(i=1;i<=K;i++) nn[i]=0L;
    for(l=0;l<T;l++) {
      //fprintf(bf,"%ld,",l);
      //for(i=0;i<K;i++) fprintf(bf,"%ld,",nn[i]);
      fprintf(bf,"%.0f\n",bin[l]);
      //incr(nn,K,B);
    }
  }

  // printf("Setup for doing the Fourier transform\n");

  for(i=1;i<=K;i++) nn[i]=B;
  nulls = 0;
  zero = 0.0;

  if(ram) {
    for(l=T;l>0;l--) {
      if(bin[l]==0.0) nulls++;
      bin[2*l-2] = bin[l-1];
      bin[2*l-1] = 0.0;
    }
  } else {
    for(l=0;l<T/2;l++) {
      if(bin[l]==0.0) nulls++;
      if( !WriteScratch(&bin[l],sizeof(float),1,file[1])
          ||
         !WriteScratch(&zero,sizeof(float),1,file[1]) ){
        printf("File 1, element %ld\n",l);
        exit(1);
      }

      if(bin[l+T/2]==0.0) nulls++;
```

```
      if( !WriteScratch(&bin[l+T/2],sizeof(float),1,file
          [2]) ||
          !WriteScratch(&zero,sizeof(float),1,file[2]) ){
        printf("File 2, element %ld\n",l+T/2);
        exit(1);
      }

    }
    rewind(file[1]);
    rewind(file[2]);
  }

  printf("Generated %ld null cells out of %ld, %lf%%
      empty.\n",nulls,T,
            (double)nulls/(double)T*100.0);

  printf("Perform the transform.\n");

  if(ram) fourn(nrbin,nn,K,1);
  else {
     fourfs(file,nn,K,1);
     if( !ReadScratch(&bin[0],sizeof(float),T,file[3])
         ||
        !ReadScratch(&bin[T],sizeof(float),T,file[4]) )
           exit(1);
  }

  printf("Scanning for supsect coefficients & gathering
      statistics\n");
  for(i=0;i<K;i++) nn[i] = 0L;
  sfsq = 0.0;
  sfxy = 0.0;
  smf = 0.0;
  for(l=0;l<T;l++) {
    re = bin[l*2];
    im = bin[l*2+1];
    rho = sqrt(re*re+im*im);
    phi = asin(re/rho);
    if(ftf) fprintf(ftf,"%f %f %f %f\n",re,im,rho,phi);
    //    if( l!=0 ){   // this one is different
    //      re -= 1.0;  // subtract off E[f(0)]
    //    }
    sfsq += (re*re + im*im);
    sfxy += 2.0*(re*im);
    smf += (re + im);

    // scan for plsd hits
    if( fabs(re) > NK1 )rfail[il]++;
    if( fabs(im) > NK1 )ifail[il]++;
    if( rho*rho > NL1 )mfail[il]++;
    // scan for bonferroni hits
    if( fabs(re) > NK2 )rfail[il+NL]++;
```

```
      if( fabs(im) > NK2 )ifail[il+NL]++;
      if( rho*rho > NL2 )mfail[il+NL]++;

      // next
      incr(nn,K,B);
    }

    X = (sfsq-1.0) * 2.0*(float)N;
    Fm = smf / ( 2.0 * (float)T - 2.0 );
    Fs = sqrt( sfsq - 2.0*sfxy + Fm*Fm );

    if(X>C) {
      printf("-- Reject H0: generator is not uniform --\n"
          );
      ofail[il] = 1;
    }
    else
      printf("-- Accept HO: generator is uniform --\n");


    // use cdflib to calculate the test significance
    DCDFLIB_WHICH = 1; // calculate P, Q

    // p <- 1 - pchisq(X,2*(T-1))

    DF = 2.0*((double)T-1.0); // use 2(T-1) for df
    cdfchi( &DCDFLIB_WHICH, &DCDFP, &DCDFQ, &X, &DF, &
        DCDFLIB_STATUS, &DCDFLIB_BOUND);
    switch(DCDFLIB_STATUS) {
    case 0:
      printf("Test Significance: X = %.1f, C = %.1f P
          = %.3f, DF = %.0f.\n",
           (float)X, (float)C, (float)DCDFQ,(float)DF);
      break;
    case 1:
      printf("Answer appears to be lower than lowest
          search bound.\n");
      break;
    case 2:
      printf("Answer appears to be higher than greatest
          search bound.\n");
      break;
    case 3:
      printf("P + Q != 1.\n");
      break;
    case 10:
      printf("Error returned from cumgam.\n");
      break;
    default:
      if(DCDFLIB_STATUS<0) {
        printf("Paramerter %d is out of range.\n",-
            DCDFLIB_STATUS);
```

```
  }
  else printf("Unexpected status returned [%d].\n",
     DCDFLIB_STATUS);
  break;
 }

printf("-- mean = %f stddev = %f \n",Fm,Fs);

if(ofail[il]!=0) {
  printf("Fishers PLSD cutoff tests:\n");
  if( rfail[il] != 0 )printf("-- Suspect Real -- (%d)\
     n",rfail[il]);
  else           printf("-- No Suspect Real -- \n");

  if( ifail[il] != 0 )printf("-- Suspect Imaginary
      -- (%d)\n",ifail[il]);
  else           printf("-- No Suspect Imaginary -- \
     n");

  if( mfail[il] != 0 )printf("-- Suspect Magnitude
     Squared -- (%d)\n",mfail[il]);
  else           printf("-- No Suspect Magnitude
     Squared -- \n");
 } else {

  printf("Bonferroni cutoff tests:\n");
  if( rfail[il+NL] != 0 )printf("-- Suspect Real -- (%
     d)\n",rfail[il+NL]);
  else           printf("-- No Suspect Real -- \n");

  if( ifail[il+NL] != 0 )printf("-- Suspect Imaginary
      -- (%d)\n",ifail[il+NL]);
  else           printf("-- No Suspect Imaginary -- \
     n");

  if( mfail[il+NL] != 0 )printf("-- Suspect Magnitude
     Squared -- (%d)\n",mfail[il+NL]);
  else           printf("-- No Suspect Magnitude
     Squared -- \n");
 }

if(ftf) {
  fflush(ftf);
 }
if(rnf) {
  fflush(rnf);
 }
if(bf) {
  fflush(bf);
 }

// use R to generate plots
```

```
  WriteRFiles(basefn,il+1,N,T,A,X,DCDFQ,C,K1,K2,L1,L2);
  system("R BATCH --no-save --no-restore mdfttreport.R"
     );
}

printf("\n
   ================================================\n
   ");
printf("Testing Summary\n");
printf("i\tChiSq\tRe\tIm\tMag\n");
for(il=0;il<NL;il++) {
  if(ofail[il]>0)
    printf("%d\t%d\t%d\t%d\n",il+1,ofail[il],rfail[il],
       ifail[il],mfail[il]);
  else
    printf("%d\t%d\t%d\t%d\n",il+1,ofail[il],rfail[il+NL
       ],ifail[il+NL],mfail[il+NL]);
}

fprintf(sf,"
   ================================================\n
   ");
fprintf(sf,"Test Summary\n%ld points in %d dim using %d
    bins/dim, %ld total bins.\n",N,K,B,T);
fprintf(sf,"Resolution of test is %f in each dimension
   .\n",1.0/(float)B);
fprintf(sf,"i\tChiSq\tRe\tIm\tMag\n");
for(il=0;il<NL;il++) {
  if(ofail[il]>0)
    fprintf(sf,"%d\t%d\t%d\t%d\n",il+1,ofail[il],rfail[
       il],ifail[il],mfail[il]);
  else
    fprintf(sf,"%d\t%d\t%d\t%d\n",il+1,ofail[il],rfail[
       il+NL],ifail[il+NL],mfail[il+NL]);
}


if(!ram) {
  fclose(file[1]); unlink("mdftt.1");
  fclose(file[2]); unlink("mdftt.2");
  fclose(file[3]); unlink("mdftt.3");
  fclose(file[4]); unlink("mdftt.4");
}

  return(0);
}
```

mdftt.hlp: Help File

LISTING B.12

```
mdftt <ram> <rng> [subrng] <E> <D> <B> <P> <NL> <
   printlev> <base-outfile>

<ram>
 1: run in ram only
 2: use disk based algorithm

<rng>
 1: pgfsr - parallel generalized feedbach shift register
     <subrng>
       1  2  3  4  5   6   7
      ----------------------------
       31 98 250 521 607 1279 1279
        6 27 103 32 273 418 1063
 2: lprand
     <subrng> delay value for generator
 3: randu
 4: super-duper
 5: pcrand
 6: ranlib
 7: pcomb

<E> Expected cell frequency (integer) N=B^D*E
<D> Dimension of grid
<B> Bins per dimension (forced to be a power of 2)
<P> Cutoff probability (x100)
<NL> Number of loops (successive test runs)
<printlev>
 0: No output files, just summarize
 1: Output fourier coefficients
 2: Output level 1 + random number file
 3: Output level 2 + bin file
<base-outfile> base directory and name for output file(s
   )
```

cell.c: Cell count utilities

LISTING B.13

```
long ipow(long x, int y)
{ int i;
  long z;
  z = (long)pow((double)x,(double)y);
  return(z);
}

long bpow(long k)
{ long z=1L;
  if(k==0) return(1L);
  else return(z<<k);
}
```

```
/* compute cell membership from real vectors (x)
 *
 * Store the stats in a one dimensional structure of
    bins
 * stored in to match the requirements of NR's fourn.c
    and fourfs.c routines.
 * Bins are identified by a D-vector d[]=(d0,d1,d2,d3
    ,...,dD-1)
 * where 0<=di<=(B-1).
 *
 * Bin (d0,d1,d2,...,di,...,dD-1) is at location l
    determined by
 *    l = sum(di*B^(D-i))
 *
 * Finding l for x[]=(x0,x1,...,xi,...,xD-1)
 *    0 <= x[i] < 1.0 ==> 0 <= truncate( x[i]*B )<= B-1
 *
 */

long cell(float *x,int D,int B)
{  int i;
   long l;
   int di;
   long BDi;

   l = 0L;
   BDi = 1L;
   for(i=D-1;i>=0;i--) {
    di = (int)(x[i]*B); //
    if(x[i]==1.0) --di; // just in case the generator
        produces 1.0
    l += di*BDi;        //
    BDi *= B;           //
   }
   return(l);
}
```

incr.c: Indexing utility function

LISTING B.14

```
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif

/* Increment a base B number represented by the integer
   array nn */
int incr(unsigned long *nn,int D,int B)
{ int Dm1;
  if(D>0) {
```

```
    Dm1=D-1;
    if(nn[Dm1]<B-1)
      nn[Dm1] += 1;
    else
      if(incr(nn,Dm1,B)) nn[Dm1] = 0;
    return(TRUE);
  } else
    return(FALSE);
}
```

uniform.c: Uniform pseudo random number generator

LISTING B.15

```
/* file: uniform.c
 *
 * uniform manages streams of random numbers for single
    or multi
 * processing environments. Type is (L:Leap-frog,B:
    Blocked,T:Tree).
 *
 * RNG TYP Name Description
 * --- --- ----- ------------
 * 1  L  PGFSR - My code for Aluru, Prabhu & Gustafson's
    parallel GFSR
 * 2  L  LPRAND - Direct port and parallelization of
    Lewis & Payne RNG
 * 3  L  RANDU - Classic bad rng used for comparitive
    testing
 * 5  L  SUPER - Marsaglia's Super-duper combined
    congruential and
 *             fibonacci generator (crude/slow
    parallelization)
 * 5  L  PCRAND - Leap-frog of Portable Combined RNG due
    to
 *             Jesse Chisholm and modified by Clark
    Thomborson
 *             (crude/slow parallelization)
 * 6  B  RANLIB - RANLIB.C generator good for small
    number of streams
 *             or larger numbers of short streams. Uses
    combined
 *             congruential generator(s) to produce
    blocks of rn's
 *             rather than leap-frogging as above rng's
    do.
 * 7  L  PCOMB - Parallel combination generator
 */

#include <stdio.h>
#include <limits.h>
```

```
#include "uniform.h"
#include "randu.h"
#include "pgfsr.h"
#include "pcrand.h"
#include "super.h"
#include "lprand.h"
#include "ranlib.h"
#include "pcomb.h"

/* define malloc here to avoid conflict with rand
   definition in stdlib */
void * malloc(int size);
void exit(int);

char *name[] = {"PGFSR","LPRAND","RANDU","SUPER","PCRAND
   ","RANLIB","PCOMB"};

char *rng_name(rng)
int rng;
{     if( rng>0 && rng<=NRNG )return(name[rng-1]);
      else return("UNKNOWN");
}

int int_bits(int w)
{ int i,b=0;
  for(i=0;i<sizeof(int)*8-1;i++)
    if( ((1<<i) & w) != 0 )b++;
  return(b);
}

int long_bits(long w)
{ int i,b=0;
  for(i=0;i<sizeof(long)*8-1;i++)
    if( ((1L<<i) & w) != 0L )b++;
  return(b);
}

rstream *setunif(rtype,id,ns,seed,c1,c2,c3)
int rtype;
int id,ns;
long seed;
long c1,c2,c3;
{
   int i;
   long l;
   rstream *rst;

   rst = (rstream *)malloc(sizeof(rstream));
   rst -> rtype = rtype;

   switch(rtype) {
```

```
      case 1:  /* PGFSR - my version of the Aluru, Prabhu
          & Gustafson prng */
75              rst -> max_N = UINT_MAX-1;
                rst -> state = pgfsr_init(seed,(int)c1,(int)
                   c2,ns,id);
                break;
      case 2:  /* LPRAND */
                rst -> state = initlprand((int)c1,(int)c2,c3
                   );
80              /* rst -> max_N = (unsigned long)(((lpint)
                   2<<(sizeof(lpint)*8-2))-1)*2+1; */
                rst -> max_N = 0x7fffffff;
                rst -> ns = ns;
                for(i=0;i<id;i++)
                   lprng(rst->state);
85              break;
      case 3:  /* RANDU */
                rst -> state = initrandu(seed,ns,id);
                rst -> max_N = 0x7fffffff;
                break;
90    case 4:  /* SUPER */
                rst -> state = initsuper(seed,ns,id);
                rst -> max_N = UINT_MAX-1;
                break;
      case 5:  /* PCRAND - Leap-frog of Portable Combined
         RNG due to
95                      Jesse Chisholm and modified by
                           Clark Thomborson */
                rst -> state = malloc(2*sizeof(long));
                rst -> max_N = ((long) (PRIME1-2L));
                rst -> ns = ns;
                setran(1L,1L,rst->state);
100             for(i=0;i<id;i++) l = lpcrand(rst->state);
                break;
      case 6:  /* RANLIB */
                if(id>31) { /* ranlib only has 32 blocks or
                   generators */
                   printf("Uniform: Fatal error, Cannot
                      use > 32 generators from RANLIB\n")
                      ;
105                exit(1);
                }
                rst -> max_N = 2147483562L;
                seed = seed & 0x7fffffff;
                /* setall(seed,seed); */
110             setall(1234567890L,123456789L);
                l = id+1;
                gscgn(1L,&l);
                break;
      case 7:  /* PCOMB */
115             rst -> state = initpcomb(seed,ns,id,(int)c1
                   ,(int)c2);
```

```
            rst -> max_N = UINT_MAX-1;
            break;
      default:
            rst = NULL;
            break;
      }
      return(rst);
}

double unif(rst)
rstream *rst;
{  double a;

   a = (double) unifint(rst);
   a = a / (rst -> max_N);
   return(a);
}

double *unifvec(rst,x,n)
rstream *rst;
double *x;
int n;
{  int i;
   if(!x) x = (double *)malloc(n*sizeof(double));
   for(i=0;i<n;i++) x[i] = unif(rst);
   return(x);
}


unsigned long unifint(rst)
rstream *rst;
{  unsigned long a;
   int i;

   switch(rst -> rtype) {
   case 1:  /* PGFSR */
            a = (unsigned long) pgfsr(rst->state);
            break;
   case 2:  /* LPRAND */
            for(i=0;i<rst->ns;i++)
               a = (unsigned long) lprng(rst->state);
            break;
   case 3:  /* RANDU */
            a = (unsigned long) randu(rst->state);
            break;
   case 4:  /* SUPER */
            a = (unsigned long) super(rst->state);
            break;
   case 5:  /* PCRAND */
            for(i=0;i<rst->ns;i++) a = (unsigned long)
               lpcrand(rst->state);
            break;
   case 6:  /* RANLIB */
```

```
            a = ignlgi();
            break;
   case 7:  /* PCOMB */
            a = (long) pcomb(rst->state);
            break;
   default: a = -1L;
            break;
   }
   return(a); /* & 0x7fffffff); */
}

/* interfaces for S/S+/XlispStat */

/* local storage structure for random streams */
typedef struct {
      rstream *rst;
      void *next;
} rstream_list;

rstream_list *root_stream = NULL;
rstream_list *last_stream;
int allocated_streams = 0;

/* initialize a random stream */
/* passing pointers to-from a stat package won't work,
   so we'll
 * build a local table of points and pass an integer
    pointer into
 * the table that we can pass back and forth...
 */
void initunif(rtype,id,ns,seed,c1,c2,c3,nrst)
int rtype;
int id,ns;
long seed;
long c1,c2,c3;
int *nrst;
{
   int i,dim;
   long l;
   double f;

   if(!root_stream) { /* first call, setup the root
      stream */
      root_stream = (rstream_list *)malloc(sizeof(
         rstream_list));
      last_stream = root_stream;
   } else {
      last_stream -> next = (rstream_list *)malloc(
         sizeof(rstream_list));
      last_stream = last_stream -> next;
   }
```

```
    last_stream -> rst = setunif(rtype,id,ns,seed,c1,c2,
       c3);
    last_stream -> next = NULL;

    *nrst = ++allocated_streams;
}

/* return a single random number */
void uniform(nrst,u)
int nrst;
double *u;
{     rstream_list *rsl;
      int i;
      rsl = root_stream;
      for(i=0;i<nrst;i++) rsl = (rstream_list *)rsl ->
          next;
      *u = unif(rsl -> rst);
}
```

uniform.h: Interface specification for uniform package

LISTING B.16

```
/* file: uniform.h
 *
 * uniform random stream definition file
 *
 */

#define NRNG 7

/* names for the generators */
char *rng_name(int rng);

/* rng data structure */
typedef struct {
      int rtype;  /* rng type */
      int ns;     /* scratch integer (rng dependent use)
          */
      void *state; /* state structure (rng dependent
         structure) */
      unsigned long max_N;  /* maximum integer returned
         by rng */
} rstream;

/* initialize a uniform random number generator */
rstream *setunif( int rtype,  /* rng type */
                  int id,     /* stream id */
                  int ns,     /* number of streams */
                  long seed,  /* initial seed */
```

```
                    long c1,    /* configuration parameter
                        1 */
                    long c2,    /* configuration parameter
                        2 */
                    long c3     /* configuration parameter
                        3 */
                    );

/* return a single uniform random number */
double unif(rstream *rst);

/* return a uniform random vector */
double *unifvec(rstream *rst, double *x, int n);

/* return a uniform random integer */
unsigned long unifint(rstream *rst);

/* interfaces for use with S/S+ or XlispStat */

/* initialize a random stream */
void initunif( int rtype, /* rng type */
               int id,     /* stream id */
               int ns,     /* number of streams */
               long seed,  /* initial seed */
               long c1,    /* configuration parameters 1 */
               long c2,    /* configuration parameters 2 */
               long c3,    /* configuration parameters 3 */
               int *rs     /* returned random stream */
               );

/* return a single uniform random number */
void uniform(int nrst, double *u);

/* bit counting routines */
int int_bits(int w);
int long_bits(long w);
```

testunif.c: Test utility for the uniform package

LISTING B.17

```
/* testunif <gen> <n> <ns> <print> */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "uniform.h"

void corr( double d1[], double d2[], int n, double *ans)
   ;
```

```
int main(argc,argv)
int argc;
char **argv;
{  int ns,n,type;
   int prlev;
   int i,j,k;
   rstream **r;
   double **u;
   long **iu;
   double *c1,*c2,ans;
   int integer_rng=0;

   if(argc>3) prlev = atoi(argv[4]);
   else prlev = 0;

   if(argc>2) {
      ns = atoi(argv[3]);
      n = atoi(argv[2]);
      type = atoi(argv[1]);
      if( type < 0 ){
         integer_rng = 1;
         type = -type;
      }
   }
   if(argc<3 || type<1 || type>NRNG) {
      printf("testunif <rng> <N> <ns> [prlev]\n");
      printf("Valid generators are:\n");
      for(i=1;i<=NRNG;i++) printf("%d: %s\n",i,rng_name(
          i));
      return(1);
   }

   /* allocate space to store the random numbers */
   if( prlev != 2 ){
      if( integer_rng ){
         iu = malloc(ns*sizeof(long*));
         for(i=0;i<ns;i++) iu[i] = malloc(n*sizeof(long
             *));
      } else {
         u = malloc(ns*sizeof(float*));
         for(i=0;i<ns;i++) u[i] = malloc(n*sizeof(double
             *));
      }

      c1 = malloc(n*sizeof(double));
      c2 = malloc(n*sizeof(double));
   }

   r = (rstream **) malloc(ns*sizeof(rstream));
   if (!r) {
      printf("allocation failure in testunif()");
      exit(1);
```

```
    }

    for(i=0;i<ns;i++) {
       r[i] = setunif(type,i,ns,-1L,607,273);
    }

    for(i=0;i<n;i++) {
       for(j=0;j<ns;j++) {
          if( integer_rng ){
             if(prlev!=2)iu[j][i] = unifint(r[j]);
             if(prlev==2)printf("%ld ",unifint(r[j]));
          }else{
             if(prlev!=2)u[j][i] = unif(r[j]);
             if(prlev==2)printf("%f ",unif(r[j]));
          }
       }
       if(prlev==2)printf("\n");
    }

    if(prlev==1) {
       printf("Generator %s[%d] correlations, %d from
           each of %d streams.\n",
          rng_name(type),type,n,ns);
       for(i=0;i<ns;i++)
          for(j=i+1;j<ns;j++) {
             for(k=0;k<n;k++) {
                if( integer_rng ){
                   c1[k] = (double)iu[i][k];
                   c2[k] = (double)iu[j][k];
                } else {
                   c1[k] = (double)u[i][k];
                   c2[k] = (double)u[j][k];
                }
             }
             corr(c1,c2,n,&ans);
             printf("%d %d: %g",i,j,ans);
             if(ans>2.0/sqrt((float)n)) printf("<- reject
                ");
             printf("\n");
          }
    }
    return;
 }
```

[Uniform]lprand.c

LISTING B.18

```
 #include <stdio.h>
 #include <stdlib.h>
 #include "lprand.h"
```

```
int setr(lpint *m, int p, long delay, int q);

static int intsiz = sizeof(lpint)*8 - 1;
lpint M;

lprand_state *initlprand(int p,int q,long delay)
{  lprand_state *s;
   s = (lprand_state *)malloc(sizeof(lprand_state));
   s->vec = (lpint *)malloc(p*sizeof(lpint));
   s->p = p;
   s->q = q;
   s->sp = -1;
   printf("lprand: p %d q %d l %ld linear columns %d\n",
      p,q,delay,setr(s->vec,p,delay,q));
   printf("    sizeof int is %d\n",intsiz);
   return(s);
}

lpint lprng(lprand_state *s)
{
 return(ilprand(s->vec,s->p,s->q,&s->sp));
}

lpint ilprand(lpint *m, int p, int q, int *lpj)
{
/*    m(p) = table of p previous random numbers.
 *    p,q=polynomial parametsrs:x**p+x**q+1.
 */

 int k;
 *lpj = (*lpj+1) % p;
 k = (*lpj+q) % p;
 m[*lpj]=m[*lpj]^m[k];
 return(m[*lpj]);
}

float lprand(lpint *m, int p, int q)
{
 lpint M;
 int lpj;
 M=(((lpint)2<<(intsiz-1))-1)*2+1;
 return((float)ilprand(m,p,q,&lpj)/(float)M);
}

int setr(lpint *m,int p,long delay,int q)
{

/*    setr=column number of repeating one pattern. if
   setr <= p,
 *    then an improper shift length has been selected.
 *    m(p)=table of random numbers to be initialized.
 *    p,q,polynomial parameters: x**p+x**q+1.
```

```
 *    delay= relative delay between columns of m(p), in
    bits.
 *    intsiz-integer size (bits) of host machine: e.t.,
 *    ibm 360, 31; cdc6000, 48; sru 1100, 35; hp
     2100, 15.
 */

 int r,i,j,k,kount,lpj=-1;
 lpint one,itemp;

 M=(((lpint)2<<(intsiz-2))-1)*2+1;

 r=p+1;
 one= (lpint)2<<(intsiz-2);
 for(i=0;i<p;i++) m[i] = one;
 for(k=0;k<intsiz;k++) {
   for(j=0;j<delay;j++) ilprand(m,p,q,&lpj);
   kount=0;
   for(i=0;i<p;i++) {
     itemp=one>>k;
     itemp= (m[i]-(m[i]/one)*one)/itemp;
     if(itemp==1) kount++;
     if(k<(intsiz-1)) {
       m[i]=m[i]/2+one;
     }
   }
   if(kount==p) r=k;
 }
 for(i=0;i<5000;i++)
   for(j=0;j<p;j++) ilprand(m,p,q,&lpj);

 return(r);
}
```

[Uniform]lprand.h

LISTING B.19

```
/* file: lprand.h
 *
 */

#ifndef LPRAND_DEF

#define LPRAND_DEF

#define lpint long

/* lprand state memory structure */
typedef struct {
      int   p;    /* state memory length */
      int   q;    /* polynomial offset */
```

```
      int   sp;   /* state pointer */
      lpint *vec; /* state vector pointer */
} lprand_state;

/* random number generator */
lpint ilprand(lpint *m, int p, int q, int *lpj);

/* floating point generator */
float lprand(lpint *m, int p, int q);

/* full initialization */
lprand_state *initlprand(int p,int q,long delay);

/* uniform specific interface */
lpint lprng(lprand_state *s);

#endif
```

[Uniform]pcomb.c

LISTING B.20

```
/* file: pcomb.c
 *
 *    Parallel Generalized Feedback Shift Register (
    PGFSR) rng combined
 *    with a simple congruential.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pcomb.h"

unsigned int pcomb(s)
pcomb_state *s;
{  /* compute congruential rn */
   s->cseed *= s->cp;

   /* combine with the difference of the generators */
   return(pgfsr(s->ps)-s->cseed);
}

pcomb_state *initpcomb(seed,ns,id,p,q)
unsigned int seed;
int ns,id,p,q;
{     int i;
      pcomb_state *s;
      s = (pcomb_state *)malloc(sizeof(pcomb_state));
      s->ps = pgfsr_init(seed,p,q,ns,id);
      s->cseed = seed;
```

```
30        for(i=0;i<id;i++) s->cseed *= s->cp;
          return(s);
    }
```

[Uniform]pcomb.h

LISTING B.21

```
    /* file: pcomb.h
     *
     */

 5  #include "pgfsr.h"

    typedef struct {
          pgfsr_state *ps;
          unsigned int cseed;
10        unsigned int cp;
    } pcomb_state;

    /* random number generator */
    unsigned int pcomb(pcomb_state *state);
15
    /* full initialization */
    pcomb_state *initpcomb(unsigned int seed,int ns,int id,
       int p,int q);
```

[Uniform]pcrand.c

LISTING B.22

```
    /* pcrand.c 1.2 5/12/92 */
    /* ************************************************

    Purpose:    This file has routines to implement a
 5              Combined Prime Multiplicative Congruential
                Psuedo-Random Number Generator.

                The original algorithms and selection of
                prime numbers comes from two articles in the
10              Communications of the ACM: June 1988 (v31
                    #6)
                the article by Pierre L'Ecuyer called,
                "Efficient and Portable Combined Random
                   Number Generators"
                and the October (v31 #10), Stephen Park and
                   Keith Miller
                "Random Number Generators: Good Ones are
                   Hard to Find."
15
```

```
            This implementation of these algorithms has
               been released
            to the Public Domain. 91.03.10; 91.09.25.

Author:     Jesse Chisholm
Modified by: Clark Thomborson

Auditor:

Creation Date: 91.03.04

Modifications by Jesse Chisholm:

      0.0.0 91.03.04    Initial creation from the
         articles
                        mentioned.

      0.0.1 91.03.04    cleaned up some, and added test
         main(s).

      0.0.2 91.03.06    added support routines for
         retaining
                        continuation of sequence across
                           program
                        invokations.

      0.0.3 91.03.10    ported to UNIX

      1.0.0 91.03.10    Released to the Public Domain.

      1.0.1 91.09.25    Re-Released to the Public Domain
         .

Modifications by Clark Thomborson:

      1.0.2 92.01.01    Added SUN compatibility; allowed
          RNGHOME
                        to differ from $HOME; wrote Unix
                            Makefile

      2.0.0 92.01.02    Rewrote _rand to comply with L'
         Ecuyer's
                        recommendations. New function is
                            called
                        pcrand(), of type double.
                           Discarded
                        nrand(), since it was inaccurate
                           .
                        Added RNGFILENAME. Reformatted
                           statefile
```

```
                        to include pedigree of current
                           state.

55      2.0.1 92.01.06  Rewrote urand() to generate full
           -range
                        (31-bit) integers in an unbiased
                            fashion.

        2.0.2 92.01.15  Removed a tiny residual bias in
           urand().

60      3.0.0 92.05.06  Interfaced to mrandom().
           Discarded
                        urand() and non-Sun code.

***************************************************** */

65  /* include directives */

    #include <stdio.h>
    #include "pcrand.h"

70  /*
    **    this is the current state of the generator
    */
    #define curr_seed1 rngstate[0]
    #define curr_seed2 rngstate[1]
75
    /* These constants define the generator */

    #define ROOT1 40014L
    #define ROOT2 40692L
80
    /*
    **    If you should choose to change primes or roots,
    **    please be sure that these conditions hold:
    **          (QUO > REM)
85  **          (QUO > 0)
    **          (REM > 0)
    */
    /* QUO = PRIME / ROOT */
    #define QUO1 53668L
90  #define QUO2 52774L

    /* REM = PRIME % ROOT */
    #define REM1 12211L
    #define REM2 3791L
95
    /*
    **    The 10000th seed from 1L for arithmetic checking
    */
    #define PRCHK1 1919456777L
```

```
#define PRCHK2 2006618587L

/* ************************************************

Name:       lpcrand

Purpose:    This is the basic routine for generating
            random numbers. The range is 0..(PRIME1-2).
            The period is humongous. Roughly 2^62.

Sample Call: l = lpcrand(rngstate);

Inputs:     rngstate (array of 2 longs, holding seed1
   and seed2)

Outputs:    none

Returns:    long l; MSBs are random

Algorithm:  seed1 = seed1 * ROOT1 % PRIME1
            seed2 = seed2 * ROOT2 % PRIME2
            U = (seed1 - seed2) mod (PRIME1 - 1)
            return (U==0 ? (PRIME1 - 1): U)

      This implementation used the following numbers:

      PRIME       ROOT        RANGE
      2147483563  40014       1..2147483562
      2147483399  40692       1..2147483398

************************************************ */
/*
**    NOTES ABOUT RND taken from earlier implementation
**
** This is the basic routine.
**
**    new_seed = old_seed * ROOT % PRIME;
**
** The table below shows some convenient values
**
**    PRIME    primitive root alternate root range
**
**    17          5              6           1..16
**    257         19             17          1..256
**    32749       182            128         1..32748
**    65537       32749          32719       1..65536
**    2147483647  16807          39373
   1..2147483646
**    2147483647  48271          69621
   1..2147483646
**    2147483647  397204094      630360016
   1..2147483646
```

```
**
** This is the best single primitive root for 2^31-1
   according to
** Pierre L'Ecuyer; "Random Numbers for Simulation";
   CACM 10/90.
**
**    2147483647  41358
   1..2147483646
**
** He also lists the prime 4611685301167870637
** and the primitive root 1968402271571654650
** as being pretty good. However, it doesn't lend itself
    to
** this algorithm as the quotient is less than the
   remainder in p/r.
**
** As a way of checking the arithmetic in any given
   implementation,
** the table below shows what the 10,000th seed should
   be when the
** generator is started at 1L for various primes and
   roots.
**
**    PRIME       ROOT  1st seed    10,000th seed
**    ----------  ----- -           ----------
**    2147483647L 16807L 1L         1043618065L
**    2147483647L 48271L 1L          399268537L
**    2147483647L 69621L 1L          190055451L
**    2147483563L 40014L 1L         1919456777L
**    2147483399L 40692L 1L         2006618587L
**
** There are many other primitive roots available for
   these
** or other primes. These were chosen as esthetically
   pleasing.
** Also chosen so that 'p / x > p % x' for prime p, and
   root x.
**
** A number (x) is a primitive root of a prime (p)
** iff in the equation:
**
**    t = x^i % p
**
**    when (i == p-1) and (t == 1)
**
**    for all (1 <= i < p-1) then (t >= 2)
**
** The number of primitive roots for a given prime (p)
   is the number
** of numbers less that (p-1) that are relatively prime
   to (p-1).
**
```

```
** Note: that is how many, not which ones.
**
** if x is a primitive root of p, and y is relatively
   prime to p-1
** then (x^y % p) is also a primitive root of p.
** also (x^(p-1-y) % p) is a primitive root of p.
**
*/

long lpcrand(rngstate)
long rngstate[RNGstatesize_3];
{
  long k;
  long U;

   /* the following code is taken from Figure 3 of L'
      Ecuyer's article */
  k = curr_seed1 / QUO1;
  curr_seed1 = ROOT1 * (curr_seed1 - k * QUO1) - k * REM1
     ;
  if (curr_seed1 < 0) {
    curr_seed1 += PRIME1;
  }

  k = curr_seed2 / QUO2;
  curr_seed2 = ROOT2 * (curr_seed2 - k * QUO2) - k * REM2
     ;
  if (curr_seed2 < 0) {
    curr_seed2 += PRIME2;
  }

  U = curr_seed1 - curr_seed2;
  if (U < 1) {
    U += PRIME1 - 1;
  }

  return(U-1);
}

/* ************************************************

Name:       checkran

Purpose:    This routine returns 1 if the RNG state
   looks ok.

Sample Call: if ( !checkran(rngstate) )fprintf(stderr,"
   RNG unitialized!\n");

Inputs:     none

Outputs:    none
```

```
Returns:    none

Algorithm:  Return 1 if 0 < seed1 < PRIME1 and 0 < seed2
    < PRIME2,
            else return 0

************************************************* */
extern int checkran(rngstate)
long rngstate[RNGstatesize_3];
{
 if (curr_seed1 <= 0 || curr_seed1 >= PRIME1 ||
    curr_seed2 <= 0 || curr_seed2 >= PRIME2) {
  return(0);
 } else {
  return(1);
 }
}

/* ************************************************

Name:       setran

Purpose:    This routine sets the random number sequence
   , checking to see
            if the seeds are legal. Prints message to
               stderr if illegal.

Sample Call: setran(1L,1L,rngstate);

Inputs:     long seed1;       starting seed for 1st
   generator
            long seed2;       starting seed for 2nd
               generator
            long rngstate[2]; where seed1 and seed2 are
               to be stored

Outputs:    none

Returns:    none

Algorithm:  copy seed1, seed2 into rngstate[]; call
   checkran() to
            confirm that seeds are legal. Print error
               message to
            stderr if illegal.

************************************************* */
void setran(seed1, seed2, rngstate)
long seed1, seed2, rngstate[RNGstatesize_3];
{
 curr_seed1 = seed1;
```

```
  curr_seed2 = seed2;
  if ( !checkran(rngstate) ){
    fprintf(stderr,"Illegal seed values: %ld, %ld", seed1
        , seed2);
    fflush(stderr);
  }
}

#ifdef TEST

main()
{
int num;
long dd;
int limit;
long rngstate[RNGstatesize_3];

      printf("Testing validity of arithmetic in this
         implementation\n");
      printf("of the EXTENDED STANDARD RANDOM NUMBER
         GENERATOR\n\n");

      printf("Testing error check code.");
      setran(0L,0L,rngstate);

      /* now try a legal seed */
      setran(1L,1L,rngstate);
      limit = 10000;

      for(num=0; num<limit; num++) {
            dd = lpcrand(rngstate);
            if ((num % 100) == 0) {
                  printf("\rIteration %d /10000", num);
            }
      }
      printf("\rSeed should be %ld,%ld\n and is %ld,%ld
         .\n",
            PRCHK1, PRCHK2, curr_seed1, curr_seed2);
      if ((curr_seed1 == PRCHK1) && (curr_seed2 ==
         PRCHK2)) {
            printf ("Arithmetic is performed correctly.\
               n");
      } else {
            printf ("Arithmetic is performed INCORRECTLY
               !\n");
      }

      exit(0);
}

#endif /* TEST */
```

```
/* end of pcrand.c */
```

[Uniform]pcrand.h

LISTING B.23

```
/* pcrand.h 1.3 5/12/92 */
/* ************************************************

Purpose:    This file has routines to implement a
            Combined Prime Multiplicative Congruential
            Psuedo-Random Number Generator.

            The original algorithms and selection of
            prime numbers comes from two articles in the
            Communications of the ACM: June 1988 (v31
                #6)
            the article by Pierre L'Ecuyer called,
            "Efficient and Portable Combined Random
               Number Generators"
            and the October (v31 #10), Stephen Park and
               Keith Miller
            "Random Number Generators: Good Ones are
               Hard to Find."

            This implementation of these algorithms has
               been released
            to the Public Domain. 91.03.10; 91.09.25.

Author:     Jesse Chisholm
Modified by: Clark Thomborson
*/

/* constants defining this multiplicative generator */
#define PRIME1 2147483563L
#define PRIME2 2147483399L

#define RNGstatesize_3 2
#define RNGrangem1_3 (PRIME1-3L)

#define RNGfileLINE0_3 "(Portable Combined
   Multiplicative)\n"


/* ************************************************

Name:       lpcrand

Purpose:    This is the basic routine for generating
            random numbers. The range is 0..(PRIME1-2).
            The period is humongous. Roughly 2^62.
```

```
40
   Sample Call: l = lpcrand(rngstate);

   Inputs:     rngstate (array of 2 longs, holding seed1
      and seed2)

45 Outputs:    none

   Returns:    long l; MSBs are random

   Algorithm:  seed1 = seed1 * ROOT1 % PRIME1
50             seed2 = seed2 * ROOT2 % PRIME2
               U = (seed1 - seed2) mod (PRIME1 - 1)
               return (U==0 ? (PRIME1 - 1): U)

   */
55
   long lpcrand(long rngstate[]);

   /* ************************************************

60 Name:       setran

   Purpose:    This routine sets the random number sequence
      , checking to see
               if the seeds are legal. Prints message to
                  stderr if illegal.

65 Sample Call: setran(1L,1L,rngstate);

   Inputs:     long seed1;       starting seed for 1st
      generator
               long seed2;       starting seed for 2nd
                  generator
               long rngstate[2]; where seed1 and seed2 are
                  stored
70
   Outputs:    none

   Returns:    none

75 Algorithm:  copy seed1, seed2 into rngstate[]; call
      checkran() to
               confirm that seeds are legal. Print error
                  message to
               stderr if illegal.

   ************************************************ */
80 void setran(long seed1, long seed2, long rngstate[] );

   /* ************************************************
```

```
   Name:       checkran
85
   Purpose:    This routine returns 1 if the RNG state
      looks ok.

   Sample Call: if ( !checkran(rngstate) )fprintf(stderr,"
      RNG unitialized!\n");

90 Inputs:     none

   Outputs:    none

   Returns:    none
95
   Algorithm:  Return 1 if 0 < seed1 < PRIME1 and 0 < seed2
       < PRIME2,
               else return 0

   ********************************************** */
100 int checkran( long rngstate[] );
```

[Uniform]pgfsr.c

LISTING B.24

```
   /* file: pgfsr.c
    *
    *    Parallel Generalized Feedback Shift Register (
       PGFSR) rng.
    *
5   */

   /* #define dbg /* debug code flag */

   #include <stdio.h>
10 #include <stdlib.h>
   #include "pgfsr.h"

   #define dp 98
   #define dq 27
15 /* #define dp 607 */
   /* #define dq 273 */
   /* #define dp 1279 */
   /* #define dq 418 */

20 pgfsr_state *alloc_state(int p,int q);
   void pstep_pgfsr(pgfsr_state *s, int p);

   unsigned int pgfsr(s)
   pgfsr_state *s;
25 {  int j;
```

```
   if(!s) s = pgfsr_init(-1,dp,dq,1,0); /* initalize if
      needed */
   s->sp = (s->sp+1) % s->p;        /* update state
      pointer */
   j = (s->sp+s->q) % s->p;         /* compute offset
      pointer */
   s->vec[s->sp] = s->vec[s->sp] ^ s->vec[j];  /*
      generate new value */
   return(s->vec[s->sp]);           /* and send back to
      caller */
}

void dump_pgfsr_state(s,f,m)
pgfsr_state *s;
FILE *f;
char *m;
{  int i;
   if(!f) f = fopen(DUMP_FILE,"w");
   fprintf(f,"PGFSR state memory dump [%s]\n",m);
   fprintf(f,"%d %d %d",s->p,s->q,s->sp);
   for(i=0;i<s->p;i++) {
      if(!(i%8))fprintf(f,"\n");
      fprintf(f,"%x ",s->vec[i]);
   }
   fprintf(f,"\n");
}

void load_pgfsr_state(s,f)
pgfsr_state *s;
FILE *f;
{  int i;
   char h[80];
   if(!f) f = fopen(DUMP_FILE,"r");
   fscanf(f,"%s\n",h);
   fscanf(f,"%d %d %d",&s->p,&s->q,&s->sp);
   for(i=0;i<s->p;i++) {
      if(!(i%8))fscanf(f,"\n");
      fscanf(f,"%x ",&s->vec[i]);
   }
   fclose(f);
}

pgfsr_state *alloc_state(p,q)
int p,q;
{  pgfsr_state *s;
   s = (pgfsr_state *)malloc(sizeof(pgfsr_state));
   if (!s) printf("State memory allocation failure");
   s->vec = (unsigned int *)malloc(p*sizeof(unsigned int
      ));
   if (!s->vec) printf("State memory allocation failure"
      );
   s->p = p;
```

```
   s->q = q;
   return(s);
}

pgfsr_state *pgfsr_qinit(seed,ns,id)
unsigned int seed;
int ns,id;
{     return(pgfsr_init(seed,dp,dq,ns,id));
}

pgfsr_state *pgfsr_init(seed,P,Q,ns,id)
unsigned int seed;
int ns,id;
int P,Q;
{  pgfsr_state *st;
   FILE *f;
   int    i,j,m,n,sum,stride,skip;
   unsigned int pos;
   short  int *seq,*temp;
   int l=8*sizeof(unsigned int); /* setup the word
      length */

   /* take ns to be the greatest power of 2 >= ns */
   pos = 2;
   for(i=1;i<l;i++) {
      if( pos >= ns ){
         ns = pos;
         break;
      }
      pos = pos<<1;
   }

   /* working state memory */
   seq = malloc(2*P*sizeof(short int));
   temp = malloc(2*P*sizeof(short int));

   /* allocate required state memory structure */
   st = alloc_state(P,Q);
   st -> sp = 0;

#ifdef dbg
   f = fopen("pgfsr.dbg","w");
   fprintf(f,"Making state memory for stream %d, %d %d\n
      ",id,P,Q);
#endif

   /* generate the initial column of p bits */
   seq[0] = seed;               /* put seed in first
      word */
   for(i=1;i<l;i++) seq[i] = (seed<<=1); /* spread the
      seed */
```

```
for(i=l;i<P;i++) seq[i] = seq[i%l]; /* replicate the
   pattern */

/* initialize seq with polynomial x^Q+1 */
/* for(i=0;i<P;i++)seq[i] = 0; /* initialize to zero
    */
/* seq[0] = seq[Q] = 1;    /* set two bits on
   initially */

/* iterate 8192P times to make the seed's pattern die
    out */
for(i=0;i<13;i++)
{
 for(m=0;m<P;m++)temp[2*m] = seq[m]; /* square the
    polynomial */
 for(m=1;m<2*P-2;m+=2)temp[m] = 0;
 for(m=2*P-2;m>=P;m--)        /* compute modulo  */
  if(temp[m]!=0)              /* primitive trinomial
      */
   {
    temp[m-P+Q] += temp[m];
    temp[m-P] += temp[m];
   }
 for(m=0;m<P;m++)seq[m] = temp[m] & 01; /*
    coefficients mod 2 */
}
for(m=0;m<P;m++)temp[m]=seq[m];

/* compute first P bits of the sequence */
for(i=0;i<P;i++)
{
 for(m=0,sum=0;m<=P-i-1;m++)sum += temp[m];
 if(i>P-Q)
  for(m=2*P-Q-i;m<=P-1;m++)sum += temp[m];
 seq[i] = sum & 01;
}

/* initialize the table */
for(i=0;i<P;i++) st -> vec[i] = 0;

/* compute the first P bits for this processor */
pos = stride = 1 << ns;
if(stride<128) skip = 128 / stride; else skip = 1;
while ((pos>>=1) > 0)
{
 for(i=P,n=0,j=Q;i<2*P;i++)seq[i] = seq[n++]^seq[j
    ++];
 for(i=id & 01,j=0;i<2*P;i+=2)seq[j++] = seq[i];
 id /= 2;
}

/* compute the first P numbers for this processor */
```

```
    n = -1;
    j = n+Q;
    pos = 1 << (sizeof(int)*8 - 1);
    while (pos > 0)
    {
      for(i=0;i<P;i++)
        if(seq[i]) st -> vec[i] |= pos;
      for(i=0;i<skip*P;i++)
        {
          if((++n)==P)n=0;
          if((++j)==P)j=0;
          seq[n] ^= seq[j];
        }
      pos >>= 1;
    }

    /* free up the workspace */
    free(seq);
    free(temp);

#ifdef dbg
    fclose(f);
#endif

    return(st);
}

#ifdef TEST

#define ns 3
#define n 1024

#define correlations

void corr( double d1[], double d2[], int N, double *ans)
    ;

main()
{  pgfsr_state *r[ns];
    unsigned int u[ns][n];
    double c1[n],c2[n],ans;
    int i,j,k;

    for(i=0;i<ns;i++) {
        r[i] = pgfsr_init(0xffffffffL,98,27,ns,i);
    }

    for(i=0;i<n;i++) {
        for(j=0;j<ns;j++) {
            u[j][i] = pgfsr(r[j]);
        }
    }
```

```
#ifdef correlations
      printf("Generator %s correlations, %d from each of
          %d streams.\n",
            "PGFSR",n,ns);
      for(i=0;i<ns;i++)
         for(j=i+1;j<ns;j++) {
            for(k=0;k<n;k++) {
               c1[k] = (double)u[i][k];
               c2[k] = (double)u[j][k];
            }
            corr(c1,c2,n,&ans);
            printf("%d %d: %g\n",i,j,ans);
         }
#endif

#ifdef xlisp
   printf("(defun source () (format t\n \"Source: ");
   printf("Uniform random numbers from ");
   printf("generator: %s[%d], %d from each of %d streams
      ~%\"))\n",
                  "pgfsr",0,n,ns);
   printf("(def data-names '(");
   for(i=0;i<ns;i++) printf("\"s%d\" ",i);
   printf("))\n(def data (transpose (split-list '(\n");
   for(i=0;i<n;i++) {
      for(j=0;j<ns;j++) {
         printf("%d ",u[j][i]);
      }
      printf("\n");
   }
   printf(") %d)))\n",ns);
   printf("(source)\n");
   printf("(regression-model (cdr data) (car data))\n");
#endif


}
#endif
```

[Uniform]pgfsr.h

LISTING B.25

```
/* file: pgfsr.h
 *
 */

#ifndef PGFSR_DEF

#define PGFSR_DEF
```

```
/* default dump file name */
#define DUMP_FILE "pgfsr.state.dump"

/* pgfsr state memory structure */
typedef struct {
      int   p;    /* state memory length */
      int   q;    /* polynomial offset */
      int   sp;   /* state pointer */
      unsigned int *vec; /* state vector pointer */
} pgfsr_state;

/* random number generator */
unsigned int pgfsr(pgfsr_state *state);

/* full initialization */
pgfsr_state *pgfsr_init(unsigned int seed,int p,int q,
   int ns,int id);

/* quick initialization (uses default p & q values) */
pgfsr_state *pgfsr_qinit(unsigned int seed,int ns,int id
   );

/* dump state memory to a file. opens default file if f
   is null */
void dump_pgfsr_state(pgfsr_state *s,FILE *f,char *m);

/* load state memory from file. opens default file if f
   is null */
void load_pgfsr_state(pgfsr_state *s,FILE *f);

#endif
```

[Uniform]random.c

LISTING B.26

```
/*
 * Title:   random_number
 * Last Mod:  Fri Mar 18 08:52:13 1988
 * Author:  Vincent Broman
 *          <broman@schroeder.nosc.mil>
 */

#define P 179
#define PM1 (P - 1)
#define Q (P - 10)
#define STATE_SIZE 97
#define MANTISSA_SIZE 24
#define RANDOM_REALS 16777216.0
#define INIT_C 362436.0
#define INIT_CD 7654321.0
```

```
#define INIT_CM 16777213.0

static unsigned int ni;
static unsigned int nj;
static double u[STATE_SIZE];
static double c, cd, cm;

static unsigned int collapse (int anyint, unsigned int
   size);

static unsigned int collapse (anyint, size)
int anyint;
unsigned int size;
/*
 * return a value between 0 and size-1 inclusive.
 * this value will be anyint itself if possible,
 * otherwise another value in the required interval.
 */
{
   if (anyint < 0)
      anyint = - (anyint / 2);
   while (anyint >= size)
      anyint /= 2;
   return (anyint);
}


void start_random_number (seed_a, seed_b)
int seed_a;
int seed_b;
/*
 * This procedure initialises the state table u for a
    lagged
 * Fibonacci sequence generator, filling it with random
    bits
 * from a small multiplicative congruential sequence.
 * The auxilliaries c, ni, and nj are also initialized.
 * The seeds are transformed into an initial state in
    such a way that
 * identical results are guaranteed across a wide
    variety of machines.
 */
{
double s, bit;
unsigned int ii, jj, kk, mm;
unsigned int ll;
unsigned int sd;
unsigned int elt, bit_number;

   sd = collapse (seed_a, PM1 * PM1);
   ii = 1 + sd / PM1;
   jj = 1 + sd % PM1;
```

```
      sd = collapse (seed_b, PM1 * Q);
      kk = 1 + sd / PM1;
      ll = sd % Q;
      if (ii == 1 && jj == 1 && kk == 1)
         ii = 2;

      ni = STATE_SIZE - 1;
      nj = STATE_SIZE / 3;
      c = INIT_C;
      c /= RANDOM_REALS;   /* compiler might mung the
          division itself */
      cd = INIT_CD;
      cd /= RANDOM_REALS;
      cm = INIT_CM;
      cm /= RANDOM_REALS;

      for (elt = 0; elt < STATE_SIZE; elt += 1) {
         s = 0.0;
         bit = 1.0 / RANDOM_REALS;
         for (bit_number = 0; bit_number < MANTISSA_SIZE;
             bit_number += 1) {
            mm = (((ii * jj) % P) * kk) % P;
            ii = jj;
            jj = kk;
            kk = mm;
            ll = (53 * ll + 1) % Q;
            if (((ll * mm) % 64) >= 32)
               s += bit;
            bit += bit;
         }
         u[elt] = s;
      }
   }


   double next_random_number()
   /*
    * Return a uniformly distributed pseudo random number
    * in the range 0.0 .. 1.0-2**(-24) inclusive.
    * There are 2**24 possible return values.
    * Side-effects the non-local variables: u, c, ni, nj.
    */
   {
   double uni;

      if (u[ni] < u[nj])
         uni = u[ni] + (1.0 - u[nj]);
      else
         uni = u[ni] - u[nj];
      u[ni] = uni;

      if (ni > 0)
         ni -= 1;
```

```
   else
      ni = STATE_SIZE - 1;

   if (nj > 0)
      nj -= 1;
   else
      nj = STATE_SIZE - 1;

   if (c < cd)
      c = c + (cm - cd);
   else
      c = c - cd;

   if (uni < c)
      return (uni + (1.0 - c));
   else
      return (uni - c);
}
```

[Uniform]random.h

LISTING B.27

```
/*
 * Title:   randomnumber
 * Last Mod:  Fri Mar 18 07:28:57 1988
 * Author:  Vincent Broman
 *          <broman@schroeder.nosc.mil>
 */

extern void start_random_number();

extern double next_random_number();

/*
 * This package makes available Marsaglia's highly
    portable generator
 * of uniformly distributed pseudo-random numbers.
 *
 * The sequence of 24 bit pseudo-random numbers produced
    has a period
 * of about 2**144, and has passed stringent statistical
    tests
 * for randomness and independence.
 *
 * Supplying two seeds to start_random_number is
    required once
 * at program startup before requesting any random
    numbers, like this:
 *    start_random_number(101, 202);
 *    r := next_random_number();
```

```
 * The correspondence between pairs of seeds and
    generated sequences
 * of pseudo-random numbers is many-to-one.
 *
 * This package should compile and run identically on
    any
 * machine/compiler which supports >=16 bit integer
    arithmetic
 * and >=24 bit floating point arithmetic.
 *
 * References:
 *    M G Harmon & T P Baker, “An Ada Implementation of
    Marsaglia’s
 *    "Universal" Random Number Generator”, Ada Letters
    , late 1987.
 *
 *    G Marsaglia, “Toward a universal random number
    generator”,
 *    to appear in the Journal of the American
    Statistical Association.
 *
 * George Marsaglia is at the Supercomputer Computations
     Research Institute
 * at Florida State University.
 */
```

[Uniform]randu.c

LISTING B.28

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "randu.h"

randu_state *initrandu(seed,ns,id)
long seed;
int ns,id;
{  int i;
   randu_state *s;
   s = (randu_state *)malloc(sizeof(randu_state));
   if(!s) {
      printf("RANDU state memory allocation failure\n");
      exit(2);
   }
   s->seed = seed;
   s->ns = ns;
   s->id = id;
   for(i=0;i<id;i++) s->seed = (s->seed*65539) & 0
      x7fffffff;
   return(s);
```

```
}

long olrandu(s)
randu_state *s;
{  s->seed = ( s->seed*65539 )& 0x7fffffff;
   return(s->seed);
}

long randu(s)
randu_state *s;
{  int i;
   for(i=0;i<s->ns;i++) s->seed = ( s->seed*65539 )& 0
      x7fffffff;
   return(s->seed);
}
```

[Uniform]randu.h

LISTING B.29

```
#define RANDU_MAX 2147483648L

typedef struct {
      int ns;
      int id;
      long seed;
} randu_state;

randu_state *initrandu(long seed, int ns, int id);
long randu(randu_state *s);
```

[Uniform]ranlib.c

LISTING B.30

```
#include "ranlib.h"
#include <stdio.h>
#include <stdlib.h>

long mltmod(long a,long s,long m);

void advnst(long k)
/*
**** ******** ******** ******** ******** ******** ********

    void advnst(long k)
           ADV-a-N-ce ST-ate
    Advances the state of the current generator by 2^K
       values and
    resets the initial seed to that value.
```

```
      This is a transcription from Pascal to Fortran of
         routine
15    Advance_State from the paper
      L'Ecuyer, P. and Cote, S. "Implementing a Random
         Number Package
      with Splitting Facilities." ACM Transactions on
         Mathematical
      Software, 17:98-111 (1991)
                          Arguments
20    k -> The generator is advanced by2^K values
**** ******** ******** ******** ******** ******** ********

*/
{
#define numg 32L
25 extern void gsrgs(long getset,long *qvalue);
extern void gscgn(long getset,long *g);
extern long Xm1,Xm2,Xa1,Xa2,Xcg1[],Xcg2[];
static long g,i,ib1,ib2;
static long qrgnin;
30 /*
    Abort unless random number generator initialized
*/
   gsrgs(0L,&qrgnin);
   if(qrgnin) goto S10;
35 puts(" ADVNST called before random generator
      initialized - ABORT");
   exit(1);
S10:
   gscgn(0L,&g);
   ib1 = Xa1;
40 ib2 = Xa2;
   for(i=1; i<=k; i++) {
      ib1 = mltmod(ib1,ib1,Xm1);
      ib2 = mltmod(ib2,ib2,Xm2);
   }
45 setsd(mltmod(ib1,*(Xcg1+g-1),Xm1),mltmod(ib2,*(Xcg2+g
      -1),Xm2));
/*
    NOW, IB1 = A1**K AND IB2 = A2**K
*/
#undef numg
50 }
void getsd(long *iseed1,long *iseed2)
/*
**** ******** ******** ******** ******** ******** ********

    void getsd(long *iseed1,long *iseed2)
55         GET SeeD
    Returns the value of two integer seeds of the
       current generator
```

```
       This is a transcription from Pascal to Fortran of
          routine
       Get_State from the paper
       L'Ecuyer, P. and Cote, S. "Implementing a Random
          Number Package
60     with Splitting Facilities." ACM Transactions on
          Mathematical
       Software, 17:98-111 (1991)
                         Arguments
       iseed1 <- First integer seed of generator G
       iseed2 <- Second integer seed of generator G
65  **** ******** ******** ******** ******** ******** ********

   */
   {
   #define numg 32L
   extern void gsrgs(long getset,long *qvalue);
70 extern void gscgn(long getset,long *g);
   extern long Xcg1[],Xcg2[];
   static long g;
   static long qrgnin;
   /*
75     Abort unless random number generator initialized
   */
      gsrgs(0L,&qrgnin);
      if(qrgnin) goto S10;
      printf(
80      " GETSD called before random number generator
           initialized -- abort!\n");
      exit(0);
   S10:
      gscgn(0L,&g);
      *iseed1 = *(Xcg1+g-1);
85    *iseed2 = *(Xcg2+g-1);
   #undef numg
   }
   long ignlgi(void)
   /*
90 **** ******** ******** ******** ******** ******** ********

       long ignlgi(void)
              GeNerate LarGe Integer
       Returns a random integer following a uniform
          distribution over
       (1, 2147483562) using the current generator.
95     This is a transcription from Pascal to Fortran of
          routine
       Random from the paper
       L'Ecuyer, P. and Cote, S. "Implementing a Random
          Number Package
       with Splitting Facilities." ACM Transactions on
          Mathematical
```

```
    Software, 17:98-111 (1991)
**** ******** ******** ******** ******** ******** ********

*/
{
#define numg 32L
extern void gsrgs(long getset,long *qvalue);
extern void gssst(long getset,long *qset);
extern void gscgn(long getset,long *g);
extern void inrgcm(void);
extern long Xm1,Xm2,Xa1,Xa2,Xcg1[],Xcg2[];
extern long Xqanti[];
static long ignlgi,curntg,k,s1,s2,z;
static long qqssd,qrgnin;
/*
    IF THE RANDOM NUMBER PACKAGE HAS NOT BEEN
       INITIALIZED YET, DO SO.
    IT CAN BE INITIALIZED IN ONE OF TWO WAYS : 1) THE
       FIRST CALL TO
    THIS ROUTINE 2) A CALL TO SETALL.
*/
   gsrgs(0L,&qrgnin);
   if(!qrgnin) inrgcm();
   gssst(0,&qqssd);
   if(!qqssd) setall(1234567890L,123456789L);
/*
    Get Current Generator
*/
   gscgn(0L,&curntg);
   s1 = *(Xcg1+curntg-1);
   s2 = *(Xcg2+curntg-1);
   k = s1/53668L;
   s1 = Xa1*(s1-k*53668L)-k*12211;
   if(s1 < 0) s1 += Xm1;
   k = s2/52774L;
   s2 = Xa2*(s2-k*52774L)-k*3791;
   if(s2 < 0) s2 += Xm2;
   *(Xcg1+curntg-1) = s1;
   *(Xcg2+curntg-1) = s2;
   z = s1-s2;
   if(z < 1) z += (Xm1-1);
   if(*(Xqanti+curntg-1)) z = Xm1-z;
   ignlgi = z;
   return ignlgi;
#undef numg
}
void initgn(long isdtyp)
/*
**** ******** ******** ******** ******** ******** ********

    void initgn(long isdtyp)
        INIT-ialize current G-e-N-erator
```

```
        Reinitializes the state of the current generator
        This is a transcription from Pascal to Fortran of
           routine
        Init_Generator from the paper
150     L'Ecuyer, P. and Cote, S. "Implementing a Random
           Number Package
        with Splitting Facilities." ACM Transactions on
           Mathematical
        Software, 17:98-111 (1991)
                          Arguments
        isdtyp -> The state to which the generator is to be
           set
155         isdtyp = -1 => sets the seeds to their initial
               value
            isdtyp = 0 => sets the seeds to the first value
               of
                        the current block
            isdtyp = 1 => sets the seeds to the first value
               of
                        the next block
160     **** ******** ******** ******** ******** ******** ********

    */
    {
    #define numg 32L
    extern void gsrgs(long getset,long *qvalue);
165 extern void gscgn(long getset,long *g);
    extern long Xm1,Xm2,Xa1w,Xa2w,Xig1[],Xig2[],Xlg1[],Xlg2
       [],Xcg1[],Xcg2[];
    static long g;
    static long qrgnin;
    /*
170     Abort unless random number generator initialized
    */
       gsrgs(0L,&qrgnin);
       if(qrgnin) goto S10;
       printf(
175      " INITGN called before random number generator
             initialized -- abort!\n");
       exit(1);
    S10:
       gscgn(0L,&g);
       if(-1 != isdtyp) goto S20;
180    *(Xlg1+g-1) = *(Xig1+g-1);
       *(Xlg2+g-1) = *(Xig2+g-1);
       goto S50;
    S20:
       if(0 != isdtyp) goto S30;
185    goto S50;
    S30:
    /*
        do nothing
```

```
*/
   if(1 != isdtyp) goto S40;
   *(Xlg1+g-1) = mltmod(Xa1w,*(Xlg1+g-1),Xm1);
   *(Xlg2+g-1) = mltmod(Xa2w,*(Xlg2+g-1),Xm2);
   goto S50;
S40:
   printf("isdtyp not in range in INITGN");
   exit(1);
S50:
   *(Xcg1+g-1) = *(Xlg1+g-1);
   *(Xcg2+g-1) = *(Xlg2+g-1);
#undef numg
}
void inrgcm(void)
/*
**** ******** ******** ******** ******** ******** ********

    void inrgcm(void)
        INitialize Random number Generator CoMmon
                        Function
    Initializes common area for random number generator
        . This saves
    the nuisance of a BLOCK DATA routine and the
       difficulty of
    assuring that the routine is loaded with the other
       routines.
**** ******** ******** ******** ******** ******** ********

*/
{
#define numg 32L
extern void gsrgs(long getset,long *qvalue);
extern long Xm1,Xm2,Xa1,Xa2,Xa1w,Xa2w,Xa1vw,Xa2vw;
extern long Xqanti[];
static long T1;
static long i;
/*
    V=20;                    W=30;
    A1W = MOD(A1**(2**W),M1) A2W = MOD(A2**(2**W),M2)
    A1VW = MOD(A1**(2**(V+W)),M1) A2VW = MOD(A2**(2**(V+
       W)),M2)
  If V or W is changed A1W, A2W, A1VW, and A2VW need to
     be recomputed.
   An efficient way to precompute a**(2*j) MOD m is to
      start with
   a and square it j times modulo m using the function
      MLTMOD.
*/
   Xm1 = 2147483563L;
   Xm2 = 2147483399L;
   Xa1 = 40014L;
   Xa2 = 40692L;
```

```
   Xa1w = 1033780774L;
   Xa2w = 1494757890L;
   Xa1vw = 2082007225L;
   Xa2vw = 784306273L;
   for(i=0; i<numg; i++) *(Xqanti+i) = 0;
   T1 = 1;
/*
    Tell the world that common has been initialized
*/
   gsrgs(1L,&T1);
#undef numg
}
void setall(long iseed1,long iseed2)
/*
**** ******** ******** ******** ******** ******** ********

    void setall(long iseed1,long iseed2)
           SET ALL random number generators
    Sets the initial seed of generator 1 to ISEED1 and
       ISEED2. The
    initial seeds of the other generators are set
       accordingly, and
    all generators states are set to these seeds.
    This is a transcription from Pascal to Fortran of
       routine
    Set_Initial_Seed from the paper
    L'Ecuyer, P. and Cote, S. "Implementing a Random
       Number Package
    with Splitting Facilities." ACM Transactions on
       Mathematical
    Software, 17:98-111 (1991)
                      Arguments
    iseed1 -> First of two integer seeds
    iseed2 -> Second of two integer seeds
**** ******** ******** ******** ******** ******** ********

*/
{
#define numg 32L
extern void gsrgs(long getset,long *qvalue);
extern void gssst(long getset,long *qset);
extern void gscgn(long getset,long *g);
extern long Xm1,Xm2,Xa1vw,Xa2vw,Xig1[],Xig2[];
static long T1;
static long g,ocgn;
static long qrgnin;
   T1 = 1;
/*
    TELL IGNLGI, THE ACTUAL NUMBER GENERATOR, THAT THIS
       ROUTINE
     HAS BEEN CALLED.
*/
```

```
   gssst(1,&T1);
   gscgn(0L,&ocgn);
/*
    Initialize Common Block if Necessary
*/
   gsrgs(0L,&qrgnin);
   if(!qrgnin) inrgcm();
   *Xig1 = iseed1;
   *Xig2 = iseed2;
   initgn(-1L);
   for(g=2; g<=numg; g++) {
      *(Xig1+g-1) = mltmod(Xa1vw,*(Xig1+g-2),Xm1);
      *(Xig2+g-1) = mltmod(Xa2vw,*(Xig2+g-2),Xm2);
      gscgn(1L,&g);
      initgn(-1L);
   }
   gscgn(1L,&ocgn);
#undef numg
}
void setant(long qvalue)
/*
**** ******** ******** ******** ******** ******** ********

    void setant(long qvalue)
           SET ANTithetic
    Sets whether the current generator produces
       antithetic values. If
    X  is the value normally returned from a uniform
        [0,1] random
    number generator then 1 - X is the antithetic value
       . If X is the
    value normally returned from a uniform [0,N] random
       number
    generator then N - 1 - X is the antithetic value.
    All generators are initialized to NOT generate
       antithetic values.
    This is a transcription from Pascal to Fortran of
       routine
    Set_Antithetic from the paper
    L'Ecuyer, P. and Cote, S. "Implementing a Random
       Number Package
    with Splitting Facilities." ACM Transactions on
       Mathematical
    Software, 17:98-111 (1991)
                      Arguments
    qvalue -> nonzero if generator G is to generating
       antithetic
                values, otherwise zero
**** ******** ******** ******** ******** ******** ********

*/
{
```

```
#define numg 32L
extern void gsrgs(long getset,long *qvalue);
extern void gscgn(long getset,long *g);
extern long Xqanti[];
static long g;
static long qrgnin;
/*
    Abort unless random number generator initialized
*/
   gsrgs(0L,&qrgnin);
   if(qrgnin) goto S10;
   printf(
     " SETANT called before random number generator
         initialized -- abort!\n");
   exit(1);
S10:
   gscgn(0L,&g);
   Xqanti[g-1] = qvalue;
#undef numg
}
void setsd(long iseed1,long iseed2)
/*
**** ******** ******** ******** ******** ******** ********

    void setsd(long iseed1,long iseed2)
            SET S-ee-D of current generator
    Resets the initial seed of the current generator to
       ISEED1 and
    ISEED2. The seeds of the other generators remain
       unchanged.
    This is a transcription from Pascal to Fortran of
       routine
    Set_Seed from the paper
    L'Ecuyer, P. and Cote, S. "Implementing a Random
       Number Package
    with Splitting Facilities." ACM Transactions on
       Mathematical
    Software, 17:98-111 (1991)
                        Arguments
    iseed1 -> First integer seed
    iseed2 -> Second integer seed
**** ******** ******** ******** ******** ******** ********

*/
{
#define numg 32L
extern void gsrgs(long getset,long *qvalue);
extern void gscgn(long getset,long *g);
extern long Xig1[],Xig2[];
static long g;
static long qrgnin;
/*
```

```
    Abort unless random number generator initialized
*/
   gsrgs(0L,&qrgnin);
   if(qrgnin) goto S10;
   printf(
     " SETSD called before random number generator
         initialized -- abort!\n");
   exit(1);
S10:
   gscgn(0L,&g);
   *(Xig1+g-1) = iseed1;
   *(Xig2+g-1) = iseed2;
   initgn(-1L);
#undef numg
}
long Xm1,Xm2,Xa1,Xa2,Xcg1[32],Xcg2[32],Xa1w,Xa2w,Xig1
   [32],Xig2[32],Xlg1[32],
   Xlg2[32],Xa1vw,Xa2vw;
long Xqanti[32];

void gsrgs(long getset,long *qvalue)
/*
**** ******** ******** ******** ******** ******** ********

    void gsrgs(long getset,long *qvalue)
           Get/Set Random Generators Set
    Gets or sets whether random generators set (
       initialized).
    Initially (data statement) state is not set
    If getset is 1 state is set to qvalue
    If getset is 0 state returned in qvalue
**** ******** ******** ******** ******** ******** ********

*/
{
static long qinit = 0;

   if(getset == 0) *qvalue = qinit;
   else qinit = *qvalue;
}
void gscgn(long getset,long *g)
/*
**** ******** ******** ******** ******** ******** ********

    void gscgn(long getset,long *g)
                   Get/Set GeNerator
    Gets or returns in G the number of the current
       generator
                        Arguments
    getset --> 0 Get
            1 Set
```

```
405     g <-- Number of the current random number generator
            (1..32)
    **** ******** ******** ******** ******** ******** ********

    */
    {
    #define numg 32L
410 static long curntg = 1;
       if(getset == 0) *g = curntg;
       else {
          if(*g < 0 || *g > numg) {
             puts(" Generator number out of range in GSCGN")
                 ;
415          exit(0);
          }
          curntg = *g;
       }
    #undef numg
420 }
    long mltmod(long a,long s,long m)
    /*
    **** ******** ******** ******** ******** ******** ********

        long mltmod(long a,long s,long m)
425                 Returns (A*S) MOD M
        This is a transcription from Pascal to Fortran of
            routine
        MULtMod_Decompos from the paper
        L'Ecuyer, P. and Cote, S. "Implementing a Random
            Number Package
        with Splitting Facilities." ACM Transactions on
            Mathematical
430     Software, 17:98-111 (1991)
                          Arguments
        a, s, m -->
    **** ******** ******** ******** ******** ******** ********

    */
435 {
    #define h 32768L
    static long mltmod,a0,a1,k,p,q,qh,rh;
    /*
        H = 2**((b-2)/2) where b = 32 because we are using a
             32 bit
440      machine. On a different machine recompute H
    */
       if(!(a <= 0 || a >= m || s <= 0 || s >= m)) goto S10;
       puts(" a, m, s out of order in mltmod - ABORT!");
       printf(" a = %12ld s = %12ld m = %12ld\n",a,s,m);
445    puts(" mltmod requires: 0 < a < m; 0 < s < m");
       exit(1);
    S10:
```

```
      if(!(a < h)) goto S20;
      a0 = a;
      p = 0;
      goto S120;
   S20:
      a1 = a/h;
      a0 = a-h*a1;
      qh = m/h;
      rh = m-h*qh;
      if(!(a1 >= h)) goto S50;
      a1 -= h;
      k = s/qh;
      p = h*(s-k*qh)-k*rh;
   S30:
      if(!(p < 0)) goto S40;
      p += m;
      goto S30;
   S40:
      goto S60;
   S50:
      p = 0;
   S60:
   /*
       P = (A2*S*H)MOD M
   */
      if(!(a1 != 0)) goto S90;
      q = m/a1;
      k = s/q;
      p -= (k*(m-a1*q));
      if(p > 0) p -= m;
      p += (a1*(s-k*q));
   S70:
      if(!(p < 0)) goto S80;
      p += m;
      goto S70;
   S90:
   S80:
      k = p/qh;
   /*
       P = ((A2*H + A1)*S)MOD M
   */
      p = h*(p-k*qh)-k*rh;
   S100:
      if(!(p < 0)) goto S110;
      p += m;
      goto S100;
   S120:
   S110:
      if(!(a0 != 0)) goto S150;
   /*
       P = ((A2*H + A1)*H*S)MOD M
   */
      q = m/a0;
```

```
      k = s/q;
      p -= (k*(m-a0*q));
      if(p > 0) p -= m;
      p += (a0*(s-k*q));
S130:
      if(!(p < 0)) goto S140;
      p += m;
      goto S130;
S150:
S140:
      mltmod = p;
      return mltmod;
#undef h
}
void gssst(long getset,long *qset)
/*
**** ******** ******** ******** ******** ******** ********

    void gssst(long getset,long *qset)
        Get or Set whether Seed is Set
    Initialize to Seed not Set
    If getset is 1 sets state to Seed Set
    If getset is 0 returns T in qset if Seed Set
    Else returns F in qset
**** ******** ******** ******** ******** ******** ********

*/
{
static long qstate = 0;
   if(getset != 0) qstate = 1;
   else *qset = qstate;
}
float ranf(void)
/*
**** ******** ******** ******** ******** ******** ********

    float ranf(void)
            RANDom number generator as a Function
    Returns a random floating point number from a
       uniform distribution
    over 0 - 1 (endpoints of this interval are not
       returned) using the
    current generator
    This is a transcription from Pascal to Fortran of
       routine
    Uniform_01 from the paper
    L'Ecuyer, P. and Cote, S. "Implementing a Random
       Number Package
    with Splitting Facilities." ACM Transactions on
       Mathematical
    Software, 17:98-111 (1991)
```

```
**** ******** ******** ******** ******** ******** ********

*/
{
static float ranf;
/*
    4.656613057E-10 is 1/M1 M1 is set in a data
       statement in IGNLGI
     and is currently 2147483563. If M1 changes, change
        this also.
*/
   ranf = ignlgi()*4.656613057E-10;
   return ranf;
}
```

[Uniform]ranlib.h

LISTING B.31

```
/* Prototypes for all user accessible RANLIB routines */

extern void advnst(long k);
extern void getsd(long *iseed1,long *iseed2);
extern long ignlgi(void);
extern void initgn(long isdtyp);
extern void setall(long iseed1,long iseed2);
extern void setant(long qvalue);
extern void setsd(long iseed1,long iseed2);
extern float ranf(void);
extern void gscgn(long getset,long *g);
```

[Uniform]super.c

LISTING B.32

```
/* Marsaglia's "super-duper" random number generator,
   which is also
 * used in at least two statistical computing
    environments -- S and
 * Berkeley ISP.
 *
 * This generator outputs the exclusive-or of two good
    generators. The
 * output is
 *
 *  tseed ^ cseed
 *
 * (or that divided by 2 to the 32) where cseed a linear
    congruential
 * generator updated by the single C statement
```

```
 *
 *  cseed *= 69069;
 *
 * and tseed is a a Tausworthe (linear feedback shift
    register) generator
 * updated by
 *
 *  tseed ^= ((tseed >> 15) & 0377777); \
 *  tseed ^= (tseed << 17); \
 *
 * this produces an incredibly long period and very good
    randomness properties.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "super.h"

super_state *initsuper(seed,ns,id)
unsigned int seed;
int id,ns;
{     int i;
      super_state *s;
      s = (super_state *)malloc(sizeof(super_state));
      if(!s) {
         printf("SUPER state memory allocation failure\n
            ");
         exit(2);
      }
      s->ns = ns;
      s->id = id;
      s->cseed = seed;
      s->tseed = seed;
      for(i=0;i<id;i++) {
         s->cseed *= 69069;
         s->tseed ^= ((s->tseed >> 15) & 0x1ffff);
         s->tseed ^= (s->tseed << 17);
      }
      return(s);
}

unsigned int super(s)
super_state *s;
{  int i;
   for(i=0;i<s->ns;i++) {
      s->cseed *= 69069;
      s->tseed ^= ((s->tseed >> 15) & 0x1ffff);
      s->tseed ^= (s->tseed << 17);
   }
   return(s->tseed^s->cseed);
```

```
}
```

[Uniform]super.h

LISTING B.33

```
typedef struct {
      int ns;
      int id;
      unsigned long cseed;
      unsigned long tseed;
} super_state;

super_state *initsuper(unsigned int seed, int ns, int id
   );
unsigned int super(super_state *s);
```