Testing Independence of Parallel Pseudorandom Number Streams Incorporating the Data's Multivariate Nature

by

Chester Ismay

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved June 2013 by the
Graduate Supervisory Committee:

Randall Eubank, Chair
Dennis Young
Ming-Hung Kao
Nicolas Lanchier
Mark Reiser

ARIZONA STATE UNIVERSITY

August 2013

ABSTRACT

Parallel Monte Carlo applications require the pseudorandom numbers used on each processor to be independent in a probabilistic sense. The `TestU01` software package is the standard testing suite for detecting stream dependence and other properties that make certain pseudorandom generators ineffective in parallel (as well as serial) settings.

`TestU01` employs two basic schemes for testing parallel generated streams. The first applies serial tests to the individual streams and then tests the resulting P-values for uniformity. The second turns all the parallel generated streams into one long vector and then applies serial tests to the resulting concatenated stream. Various forms of stream dependence can be missed by each approach because neither one fully addresses the multivariate nature of the accumulated data when generators are run in parallel.

This dissertation identifies these potential faults in the parallel testing methodologies of `TestU01` and investigates two different methods to better detect inter-stream dependencies: correlation motivated multivariate tests and vector time series based tests. These methods have been implemented in an extension to `TestU01` built in C++ and the unique aspects of this extension are discussed. A variety of different generation scenarios are then examined using the `TestU01` suite in concert with the extension. This enhanced software package is found to better detect certain forms of inter-stream dependencies than the original `TestU01` suites of tests.

DEDICATION

To my lovely new bride, Karolyn. You are a dream come true.

ACKNOWLEDGEMENTS

I am especially appreciative to my advisor, Dr. Randy Eubank. His patience, ability to explain difficult concepts well, incredible work ethic, and willingness to show respect and compassion have meant so much to me during our time working together. He also was vital in helping me to land my first tenure-track job by providing advice, a great reference, and constant support. Dr. Eubank has the qualities in a man that I hope everyone can someday have in their lives. Randy really is the best, both in terms of academic prowess, mentorship, and also as a human being. I am lucky to have worked with him and am honored that he accepted my offer to be his doctoral student.

I am also thankful to my other committee members: Dr. Ming-Hung Kao, Dr. Nicolas Lanchier, Dr. Mark Reiser, and Dr. Dennis Young. Their feedback has been extremely valuable and thought-provoking. I am grateful to have had them as my professors with coursework and also to have had the chance to work with them over the last eighteen months of dissertation work. I am particularly thankful to Dennis, who came out of retirement a bit to serve on my committee.

I also want to thank Debbie Olson, the Graduate Program Coordinator for the School of Mathematical and Statistical Sciences (SoMSS) at Arizona State University. She responds to requests in what seems to be within seconds every time and always works to make sure the graduate students have exactly what they need. SoMSS just wouldn't run without you, Debbie.

My wife Karolyn deserves more praise than I could ever give in a few words here. She has remained by my side through all of the challenging times as a doctoral student and provides strength and comfort that make me a better man every day. She has encouraged and pushed me at just the right level and I am so very grateful to have her in my life.

A special thanks also goes to all of my friends at Arizona State University who took the time to read through many versions of this document looking for typos, asked me to explain the details further, and also listened to me explain my ideas to them over and over again. They are too many to list and I'm sure I will miss a couple but here goes a partial list in no particular order: Michael Tallman, Rebecca Everett, Arturo Valdivia, Arthur Mitrano, Eric DeMarco, Genevieve

Toutain, Jon Adler, and Jason Bradshaw. You all are great and I am so fortunate to have been able to spend this time with you.

My father Norm Ismay deserves special credit for introducing me to mathematics at such a young age and fostering a love of numbers in me. I must also thank Dr. Roger Johnson (South Dakota School of Mines and Technology) for first instilling an interest in statistics in me so many years ago and for serving as my undergraduate advisor. Also deserving of great recognition is Jen Pazour (University of Central Florida) who told me years ago that one day I'd be a professor just like her.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Random number generation has long been of interest to statisticians, computer programmers, and scientists. Most often it provides the means for events in the real world to be modeled using a generator algorithm that produces pseudorandom numbers. These pseudorandom numbers can never truly be “random" in the sense that they must be programmed via a deterministic process or numerical algorithm on a computer with finite memory. Each pseudorandom number produced by this computer process will necessarily have some sort of deterministic relationship to a given initial value (or seed) or to another pseudorandom number (or numbers) generated by the process. In practice this absence of true “randomness" has been overlooked provided the generated sequence of pseudorandom numbers “appears random": i.e., the numbers exhibited the qualities of stochastic variables that are independent and identically distributed from some specified distribution (usually a uniform distribution) as assessed by standard statistical measures [16].

Other important properties are also needed for a given pseudorandom number generator (PRNG) to be considered “good." Including the two properties from above (that the generated numbers are uniformly distributed and independent), an ideal generator should also produce a stream of pseudorandom numbers that (1) is reproducible (on another computer or on the same computer), (2) can be changed by inputting a different initial seed, (3) can be split into many independent subsequences, and (4) can be quickly generated with little computer memory [5]. There have been many different attempts at producing pseudorandom number generators (PRNGs) that have these properties. Since it is impossible for correlations to not exist in data produced by PRNGs due to the algorithmic nature of the generation process, we hope to meet as many of these requirements as possible while understanding that meeting all of them perfectly is not an attainable goal.

Chapter 2 begins with a discussion of many of the PRNGs that have been created and commonly used over the last sixty years. One of the first generators to emerge in the literature was the linear congruential generator (LCG). This generator is defined by a recurrence relation starting on a given seed that is simple and fast to implement. The next value in a given stream is based on the previous value only and, thus, this generator is of order one. Unfortunately, its simplicity also leads to many problems in regards to the randomness properties of the numbers it produces. Given this and the current speed of today's processors and improvements made in generators since the introduction of LCGs, the use of the family of linear congruential generators is not recommended. We consider it here for historical completeness and the connections it has with modern generation methods.

An extension of the LCG is the family of multiple recursive generators (MRGs). MRGs also use a linear recurrence but, in contrast to LCGs, the recurrence is of order greater than or equal to two: i.e, two or more previously generated values are required to advance the current state of the generator. In particular, this has the consequence that if one wants an order k MRG, one must specify k seeds. A popular extension of the MRG is termed a combined multiple recursive generator (CMRG). As the name suggests, CMRGs are created by combining (usually by averaging) the output of two or more MRGs together. There have been CMRGs with particular parameter choices (such as the popular `MRG32k3a` generator developed by L'Ecuyer, et al. [21]) that have performed well in meeting the requirements of a "good" PRNG [17].

A special group of MRGs called linear feedback shift register generators (LFSRs) was proposed by Tausworthe in 1965. It is special in that it produces pseudorandom bits by a linear recurrence modulo two. These generators work at the bitwise level thereby creating numbers that are represented in binary. Similar to LCGs they produce numbers quickly but have been shown to have many faults. In 1973, Lewis and Payne extended LFSRs to what are called generalized feedback shift register generators. These usually involve working with binary vectors and component-wise exclusive-or operations to produce integers. A further variation on Lewis and Payne's generator is the Mersenne Twister developed by Matsumoto and Takuji [32]. Their particular variation of the LFSR has been known to pass many of the most stringent of randomness tests and is one of today's most used PRNGs.

The other standard class of PRNGs that will be mentioned here is the lagged-Fibonnaci generators (LFGs). Their corresponding numerical recursion depends on two lag indices (s and q with $s > q$) and a binary arithmetic operation such as addition, subtraction, or multiplication modulo M for some large integer value M. Thus, the method requires the storage of s previous values in what is typically called the lag table. The choice of the lags is important to reduce correlations among the generated values and lags of greater than 1000 are often recommended. LFGs require more storage than other generators with their need for a lag table but are usually quite quick to compute and some LFGs have proven to have “good" randomness properties [4].

In addition to advancements in single processor speeds, we have also seen an increase in the number of processors used on a single computer. Further, the availability of processor clusters is now commonplace (such as the Saguaro cluster maintained by the Advanced Computing Center at Arizona State University). One way to effectively exploit these types of computing resources for Monte Carlo studies is by computing pseudorandom numbers in parallel. By doing so, large amounts of data can be produced with nearly linear speedup as compared to the same amount produced in the traditional serial generation process. This continuation to a parallel computing environment has brought about many new challenges. For example, it has been found that pseudorandom number streams generated in parallel often exhibit inter-stream dependencies [5]. In particular, long-range correlations are known to be present in pseudorandom numbers produced by LCGs using either of the two most common methods for parallelization of generators [7].

The second part of Chapter 2 discusses the two main ways to produce statistically satisfactory parallel pseudorandom number streams. Both of the popular generation methods employ sophisticated ideas from number theory to create multiple pseudorandom number streams. Both ideas can, in fact, be used to parallelize many of the popular PRNGs discussed in Section 2.1. The first approach involves an idea outlined in [38] called parameterization. The goal of parameterization is to determine a parameter in the recursion of the PRNG that can be varied to create a unique, full-period stream of pseudorandom numbers. The second approach involves taking substreams from one long-period PRNG. This is called the splitting method and is further broken into two techniques of regular spacing and leapfrogging.

There has been much research done in the testing of serial generated pseudorandom numbers to meet the “appears random" criteria described earlier. The `TestU01` suite developed by L'Ecuyer and Simard has become the standard in assessing the "goodness" of a particular PRNG [19]. It employs many of the commonly used tests for this genre including all of those described by Knuth [16] and in the classic DIEHARD suite [25] as well as nearly all of those in the further developments for the SPRNG packages of Mascagni [28]. Some of these tests include the gap, runs, collisions, monkey, poker, birthday spacings, and coupon collector tests. These tests and many others are described in Chapter 3.

Chapter 3 also discusses the use of the "parallel filter" option and two-level tests of the `TestU01` suite to test parallel pseudorandom number generators (PPRNGs). These allow for p different substreams from the same underlying generator started with different seeds to be tested. The parallel generated streams are treated as one long vector and the "parallel filter" runs the standard serial pseudorandom generator testing procedures on this new concatenated vector. The two-level tests do the single-level serial tests for each stream (or smaller substream) and then check the resulting test statistics against their corresponding theoretical distributions using goodness-of-fit tests.

In Chapter 4, we demonstrate that both of the parallel filter and two-level test methods ignore the multivariate structure of the data in a way that can cause some dependencies to be overlooked. Specifically, we were able to develop a generator using a vector moving average of order one (VMA(1)) process with built-in stream dependence that passed the `SmallCrushFile` battery of tests in `TestU01` as well as a modified version of the `Crush` battery that was created to work with a given inputted file of deviates. In addition, we developed two more generators based on time series moving across the vectors. These two generators also passed all of the tests in both `SmallCrushFile` and the much more stringent `CrushFile`, which was mentioned earlier as a modified version of `TestU01`'s `Crush`.

In order to detect the type of dependence exhibited by our problematic example generators, we employed tools that derive from the area of multivariate statistical analysis. This is the goal of the research summarized in this dissertation; rather than creating new parallel generation techniques, our aim is the creation of new statistical testing methodology that can be used to evaluate the performance of existing generators. In this regard, we developed an extension to the `TestU01` suite to test for the independence of parallel pseudorandom number streams using tests based on (1) correlation analysis and (2) vector time series. The correlation analysis approach includes checks for pairwise correlations among the streams in addition to the use of a likelihood ratio statistic for testing that the resulting between stream correlation matrix differs from the identity. Our vector time series tests correspond to portmanteau tests for white noise (e.g., the Hosking, the Li-McLeod, and the Mahdi-McLeod tests). Chapters 5 and 6 provide more information on the specifics of our multivariate extension of `TestU01` that includes some highlights of the novel aspects of the code as well as details on its implementation with examples. In Chapter 7 we outline some ideas of how our extension and the base `TestU01` package could be implemented in a parallel computing environment. We conclude in Chapter 8 with a discussion of our findings, a summary of the extension to the `TestU01` suite, and ideas that could be explored in future research.

Chapter 2

BACKGROUND INFORMATION ON GENERATING PSEUDORANDOM NUMBERS

In this chapter, we will discuss four of the most popular generators that have been used to create pseudorandom numbers since the advent of computers. We will focus initially on serial generation of numbers and will discuss methodologies for creating parallel generators in Section 2.2.

2.1 Serial Generators

The serial generators discussed in this section are 1) linear congruential generators, 2) combined multiple recursive generators, 3) shift-register generators, and 4) lagged-Fibonnaci generators. We will see that each of these generators are similar in that they produce a stream of numbers in which the numbers are defined by some form of recursive process.

Linear Congruential Generators

The first and most basic generator discussed here is the linear congruential generator (LCG). Each of the other generators we consider can be thought of as an extension of this generator family. The LCG is defined by a recurrence relation starting on a given integer seed. We will denote this initial value as x_0. An integer sequence is then specified by the linear congruence of order one

$$x_i = (ax_{i-1} + c) \mod m. \tag{2.1}$$

The integer a is called the multiplier with $0 \leq a < m$, the integer c is called the increment with $0 \leq c < m$, and the integer $m > 0$ is called the modulus. The “mod m" notation here refers to the integer remainder after the integer division of ($ax_{i-1}+c$) by m. Thus, each integer produced by (2.1) will fall between 0 and $m-1$, inclusive. The length of the stream of integers that derives from this recurrence relation before an integer is repeated is called the period of the LCG. The maximal possible period for an LCG occurs when each of the integers 0 to $m-1$ appear exactly once in the sequence and is therefore at most m.

Not every choice of the parameter triplet (a, c, m) produces a maximal period. Knuth described guidelines for “good" parameter choices that produce this largest period [16]. These results are based on the notions of prime numbers and relatively prime numbers. Recall that a prime number is a positive integer greater than one that has only one and itself as divisors. Two positive integers are then relatively prime if their only common divisor is one.

Eubank and Kupresanin [8] summarize results from Knuth [16]:

Theorem. *If $c \neq 0$, the period of the generator in* (2.1) *is equal to m if and only if the following conditions hold: a) c is relatively prime to m, b) $a - 1$ is a multiple of every prime number that divides m, and c) $a - 1$ is a multiple of 4 if m is a multiple of 4.*

Knuth [16] also gives results for the more commonly used multiplicative case where $c = 0$. Also, the binary nature of computer memory has made it common to have m chosen to be a power of two. For example, matching up with a 32 bit computer architecture, choices for m of 2^{31} or 2^{32} seem appropriate. However, choice of a prime modulus is often recommended over a power-of-two modulus [5] and in this particular instance it can be shown that $m = 2^{31} - 1$ actually produces better results than 2^{31} or 2^{32} in the sense that the resulting number streams appear more random. This is due, in part, to $2^{31} - 1$ being an extremely rare (there are currently less than 50 discovered) type of number known as a Mersenne prime. As one can guess, a Mersenne prime is a prime number that is one less than a power of two. It is noteworthy that this particular Mersenne prime was discovered by Leonhard Euler in 1772.

It is important to understand that a long period does not guarantee a generator with “good" properties. As noted in [8], taking $a = 1$ and $c = 1$ provides a maximal period stream; but, this stream is simply the numbers 0 to $m - 1$ in sequence which certainly does not “appear random." In Chapter 3, we will discuss many of the tests used to assess whether a generator provides sufficient randomness properties.

The spectral test, one of those tests noted in Chapter 3, also detects a problem with linear congruential generators given by (2.1). George Marsaglia (in 1968) is noted as the first to recognize this problem in that the numbers generated by linear congruential generators fall on parallel hyperplanes. That is to say that these generated numbers do not “appear random" if plotted in space but rather fall on a predictable number of multidimensional planes that run equidistant to each other.

We will also see that the other three types of generators discussed subsequently have the potential for much longer periods than those of LCGs. (One may ask here how it is possible for a generator to have a period longer than m. We will revise our definition of *period* in the next section to explain this distinction.) As computers have increased in power and storage capabilities, LCGs have become essentially obsolete relative to the other three types we will consider. Nonetheless, they are important both for historical and foundational reasons.

Frequently, the integers x_i generated by an LCG (and other integer-type generators) are transformed to pseudorandom uniform numbers u_i on the unit interval $(0, 1)$ by taking $u_i = x_i/m$. These values can then be used by the inversion method which depends on the probability integral transform to produce pseudorandom deviates from other non-uniform distributions [8]. It follows by construction that if the LCG is full period the generated uniform deviates will all differ numerically.

Combined Multiple Recursive Generators

A common incorrect assumption with pseudorandom number generation is that a more mathematically complex algorithm will produce a better result. To the contrary, as more complexity is introduced, generators often break down and do not improve on what can be accomplished with much simpler schemes. Marsaglia's KISS (Keep It Simple Stupid) generator provides a testament of sorts to this realization [26].

This section discusses one simple attempt at improving on LCGs known as multiple recursive generators (MRGs). As the name implies, basic MRGs take the form

$$x_i = (a_1\, x_{i-1} + a_2\, x_{i-2} + \cdots + a_k\, x_{i-k}) \mod m. \tag{2.2}$$

Here the integers $a_1, \ldots, a_k$ are between 0 and $m - 1$ for some integer k and the modulus m plays the same role as with LCGs. We now require k initial values (seeds) in order to begin the generation of pseudorandom numbers with an MRG. Note that an LCG is a special case of an MRG with $k = 1$. Knuth [16] states that the maximum period length for a multiple recursive generator is $m^k - 1$. We will now explain how this period length is possible with a revision of the definition for the term *period*.

We noted in the last subsection that with a modulus of m, the longest possible period was m, which corresponded to each of the values 0 to $m-1$ appearing exactly once in the generated stream. To obtain a period larger than m, we must allow for integers to be repeated at another time in the stream. Therefore, we focus on the position of each generated integer in relation to the others in the stream instead of the values themselves. With this modification, the period of the generator now corresponds to the length of the sequence up until the entire sequence begins to repeat, albeit with duplications of integers in the stream. This refinement of the *period* concept encompasses our previous notion for LCG since the sequence of a full period LCG will begin to repeat again after it has exhausted each value from 0 to $m-1$.

The parameters of MRGs cannot be chosen arbitrarily since it has been shown that MRGs produce values with the same sort of parallel hyperplane structure exhibited with LCGs. Extensive searches for MRGs that pass the spectral test for having appropriate lattice structure have been carried out by Kao and Tang [15] and L'Ecuyer et al.[18]. Tables of parameter values that give these “good" MRGs are given in both of these articles.

A more general recursive generator allows for the modulus of MRGs to be varied and then for the results of several MRGs to be combined. These types of generators are called combined multiple recursive generators (CMRGs). The set-up here allows for J multiple recursive generators of the form (2.2) with the jth generator producing the pseudorandom stream defined by the nth step of the recursion

$$x_{j,n} = [a_{j1}\, x_{j,n-1} + a_{j2}\, x_{j,n-2} + \cdots + a_{jk}\, x_{j,n-k}] \mod m_j. \tag{2.3}$$

To produce pseudorandom uniform numbers on $(0,1)$ we choose integers $c_1, \ldots, c_J$ (with each $c_j < m_j$) and use modulo-1 arithmetic on real numbers to obtain

$$u_n = \left(c_1 \frac{x_{1,n}}{m_1} + c_2 \frac{x_{2,n}}{m_2} + \cdots + c_J \frac{x_{J,n}}{m_J} \right) \mod 1.$$

Specific CMRGs have period lengths as large as the product of the periods of each of the individual J generators. Good parameter choices for combined multiple recursive generators have been discovered via extensive computer searches with tables provided by L'Ecuyer [17].

We will introduce one popular combined multiple recursive generator here. It is known as `MRG32k3a` and provides the framework for the software package `RngStreams` created by

L'Ecuyer et al. [21]. Using the notation in (2.3) we define the states

$$x_{1,n} =(a_{11}\, x_{1,n-1} + a_{12}\, x_{1,n-2} + a_{13}\, x_{1,n-3}) \mod m_1,$$
$$x_{2,n} =(a_{21}\, x_{2,n-1} + a_{22}\, x_{2,n-2} + a_{23}\, x_{2,n-3}) \mod m_2$$

with $J = 2$, $k = 3$, $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$, $a_{21} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$, $m_1 = 2^{32} - 209$, and $m_2 = 2^{32} - 22853$. Here, the given seeds are $x_{1,-2}$, $x_{1,-1}$, $x_{1,0}$, $x_{2,-2}$, $x_{2,-1}$, and $x_{2,0}$. `MRG32k3a` then produces a pseudorandom uniform deviate u_n via

$$z_n =(x_{1,n} - x_{2,n}) \mod (2^{32} - 209),$$
$$u_n = \begin{cases} z_n/(2^{32} - 208), & \text{if } z_n > 0, \\ (2^{32} - 209)/(2^{32} - 208), & \text{if } z_n = 0. \end{cases}$$

The corresponding period is approximately 2^{191} provided that the seeds $x_{1,0}$, $x_{1,-1}$, $x_{1,-2}$ are not all equal to 0 and are all less than $m_1 = (2^{32} - 209)$ and the seeds $x_{2,0}$, $x_{2,-1}$, $x_{2,-2}$ are all less than $m_2 = (2^{32} - 22853)$ and not all equal to 0 [8]. The generator is also known to pass even the most stringent of serial statistical tests discussed in Chapter 3.

Shift-Register Generators

One instance of a multiple recursive generator is the class of linear feedback shift-register generators (LFSRs) introduced by Tausworthe in 1965. For specified integers s and q with $s > q$, they are defined by the linear recurrence

$$b_i = b_{i-s} + b_{i-(s-q)} \mod 2. \tag{2.4}$$

We use the letter b here to stand for "bit" since LFSRs work on a bitwise level to produce numbers in binary. Since addition modulo-2 is equivalent to bitwise exclusive-OR ($\oplus$), this recurrence is often denoted as $b_i = b_{i-s} \oplus b_{i-s+q}$.

The seed required for (2.4) requires a bit more explanation than the seeds from the previous sections. For this purpose let us assume that the seed is given in the binary form $b_1 \cdots b_s$ for some s with all b_i either zero or one. We then generate subsequent bits as in (2.4) for $s > q$. To generate r-bit integers with $2 \leq r \leq s$, consecutive bits from (2.4) are grouped as needed into disjoint r-sized blocks. This process can be iterated in a circular fashion by changing the initial bit in the process and wrapping around to the beginning of the bit sequence. In total, it is possible for LFSRs that use this multi-step approach to have a maximum period of $(2^s - 1)/gcd(r, 2^s - 1)$, where gcd represents the greatest common divisor. Therefore, as one could guess from the relationship between LFSRs and MRGs, the maximum period for these Tausworthe generators is $2^s - 1$. This relationship also leads to the space distribution problem seen in LCGs and MRGs. Simple LFSRs are no longer recommended; but, more complex combinations outlined below have been shown to perform well.

In 1973, Lewis and Payne [22] extended the idea behind Tausworthe generators by thinking of the r-bit integers as binary vectors v_j of length r consisting of 0 or 1 entries. This group of generators is often called generalized linear feedback shift-register generators (GFSRs). Notationally, the generation algorithm is similar to that of the Tausworthe generator in (2.4) but with $\oplus$ representing the component-wise application of the exclusive-OR operation: i.e., the vector v_i is produced by

$$v_i = v_{i-s} \oplus v_{i-(s-q)}. \tag{2.5}$$

As with LFSRs, GFSRs have maximum period $2^s - 1$.

Matsumoto and Kurita [30] pointed out drawbacks of the Lewis-Payne generators that include

1) initial seed selection is very influential on the “randomness" of the generated values and is rather time-consuming,

2) the algorithm requires a large amount of memory, and

3) the period is much smaller than the anticipated upper bound of $2^{pr} - 1$.

They described a generator called the twisted GFSR which eliminates each of these drawbacks [30]. Further developments and improvements on this idea have led to the popular Mersenne Twister [33] that has the massive period of $2^{19937} - 1$. It has also performed very well in statistical testing.

Lagged-Fibonacci Generators

In the 1950s, one idea suggested for increasing the period for LCGs was to use the Fibonacci sequence $x_i = (x_{i-1} + x_{i-2}) \mod m$. Perhaps unsurprisingly the numbers produced by this recurrence do not pass even basic “randomness" tests. However, revisions of this basic premise have fared much better. For example, Mitchell and Moore in 1958 proposed using

$$x_i = x_{i-24} + x_{i-55} \mod m$$

where $i \geq 55$. Here m must be even and $x_0, \ldots, x_{54}$ are arbitrary integers that are not all even. The values of 24 and 55 are called lags. They are not chosen arbitrarily and are among those given in a table of values in Knuth [16] for the lag pairs (s, q) of the generic recursion $x_{i-q} + x_{i-s}$ that produce long periods. These types of generators are called lagged-Fibonnaci generators (LFGs) and often use a modulus that is a power-of-two.

The general form of additive LFGs (assuming $s > q$ and e an integer often related to the computer's architecture) is given by

$$x_i = x_{i-s} + x_{i-q} \mod 2^e. \tag{2.6}$$

The periods of all such generators given in the table in [16] are $2^{e-1}(2^q - 1)$.

Additive lagged-Fibonacci generators (ALFGs) consistently fail the birthday spacings test discussed in Chapter 3 [16]. Modifications to the generated stream such as discarding contiguous batches of numbers is one way to deal with this problem.

Replacing the binary operation defined in (2.6) with something different than addition has also been proposed. Marsaglia revised Mitchell and Moore's generator as

$$x_i = x_{i-24} \cdot x_{i-55} \mod m$$

with m a multiple of 4 and the seeds $x_0, \ldots, x_{54}$ not all congruent to 1 (modulo 4) [28].

Coddington [4] observed that overflow problems can arise when computing multiplications for multiplicative LFGs (MLFGs) like those proposed by Marsaglia. However, current programming languages and increasing memory availability have made computers much better at handling this problem. Also, the choice of larger lags used to be a problem due to the small amount of available memory and the need to store the seed values in a lag table. It is now commonly recommended to use lags greater than 1000 and far apart to reduce correlations in the data. As long as the necessary modifications are incorporated as noted, lagged-Fibonacci generators are effective serial pseudorandom generators.

2.2 Parallel Pseudorandom Number Streams

In Section 2.1, we discussed four common ways to generate pseudorandom numbers in serial. With serial processes, only one task is executed at a time. Sometimes with large simulation studies, this type of operation is much too computationally intense, often requiring long waiting times. One way to alleviate this problem is to use a system of multiple processors working in parallel to break the single large task into multiple sub-tasks assigned to each of the different processors. This section focuses on using parallelization techniques on serial pseudorandom number generators to break a large generation exercise into smaller parts that can be executed simultaneously.

One would hope that parallelizing a number generation scheme would make it possible to obtain linear speedup. In other words, if there are p processors, ideally the parallel approach would take $1/p$ the amount of time that the original serial process would take. However, in general, there will be some inherently serial aspects of parallel code as well as delays due to inter-processor communication that require us to settle for less than optimal performance improvements.

Unfortunately, parallelization of even thoroughly tested, efficient serial pseudorandom number generators has produced many problems. Coddington even went so far as to say, “The main recommendation we would give to someone who needs to use a (pseudo)random number generator on a parallel computer is very simple - never trust a parallel (pseudo)random number generator" [5].

While many would argue that improvements have been made with parallel generators since the time of Coddington’s assertion in 1996, theoretical results like those of serial generators are still largely not available and this area remains one of current interest for computer scientists and statisticians alike. A focus of the remainder of this thesis will be on addressing one of the common problems with parallel generation: namely, that parallel streams may not behave as if they were probabilistically independent of each other.

For completeness, we note here the following additional requirements for a parallel pseudorandom number generator (PPRNG) to be viewed as “good" [8]:

1. The PPRNG must make intra-processor streams of high statistical quality while showing small dependence among the processor streams.

2. The PPRNG must be adaptable for use in different systems with multiple processors.

3. The PPRNG must produce no data movement among the different processors.

In this section, we will discuss the two common approaches to producing “good" parallel pseudorandom number streams and we will give a brief overview of how the two approaches can be applied to the four generator classes of the previous section. A mention of the impact that lingering serial generator problems has on their parallel counterparts will be made as well.

The two different parallelization methodologies are known as parameterization and splitting. Parameterization involves the generation of multiple (hopefully) independent streams by appropriate choices of the generator's seeds/parameters. Splitting relies on partitioning of one long sequence of pseudorandom deviates into disjoint streams that can then used on the different processors. Two subcategories of splitting exist and we will see examples of both subsequently. The first is known as *regular spacing*. Its principle is to divide the sequence of pseudorandom numbers into disjoint contiguous blocks. Notationally, when splitting the sequence $\{x_i, i = 0, 1, \ldots\}$ into p streams (where p denotes the number of parallel processors) each of length m, the stream assigned to processor j will be $\{x_{k+(j-1)m}, k = 0, \ldots, m-1\}$ for $j = 1, 2, \ldots, p$ [12].

The second splitting technique is called *leapfrogging*. The idea is analogous to a standard card game in which cards are dealt cyclically among the players of the game until no cards remain to be dealt. With this scheme the stream given to the jth processor is $\{x_j, x_{j+p}, x_{j+2p}, x_{j+3p}, \ldots\}$.

The parameterization technique uses different parameters on the same family of generator to produce different independent streams for each processor. Mascagni [38] describes this process for some of our families of generators from Section 2.1 with Matsumoto and Nishimura [33] showing how it can be accomplished for the Mersenne Twister generator. These results as well as those for splitting are discussed below.

Parallelized Linear Congruential Generators

We mentioned in Section 2.1 that the modulus for linear congruential generators is often chosen to be prime (often Mersenne prime) instead of a power-of-two. We will now briefly discuss how to generate pseudorandom numbers in parallel using such an LCG based on either parameterization or leapfrogging.

If m is a prime modulus, the goal is to parameterize the multiplier a. By parameterizing the multiplier instead of the modulus, one can ensure that the modular multiplication remains optimal and does not vary throughout the generation [27]. To increase the speed in computation, c is often set to zero as well; so, parameterizing a is the best choice.

A result from number theory is needed to give an explicit parameterization for the multiplier. If a is primitive modulo m, then any number relatively prime to m is congruent to a^k modulo m for some k. Also, if a and b are primitive modulo m, then $b = a^i \mod m$ for some i relatively prime to $m-1$ [28]. Mascagni [27] gives an efficient algorithm for finding the integers relatively prime to $m-1$. Given the primitive modulo m element a, we can parameterize a for the jth primitive element a_j as $a_j = a^{\ell_j} \mod m$ with ℓ_j being the jth integer relatively prime to $m-1$.

A somewhat simpler calculation based on the generator's recurrence formula can be used to produce a leapfrogging formula. Specifically Knuth [16] gives a generalization of the defining equation of an LCG for $p \geq 0$ and $j \geq 0$ as

$$x_{j+p} = \left(a^p x_j + \frac{a^p - 1}{a-1} c\right) \mod m. \tag{2.7}$$

Thus, processor j generates the subsequence $\{x_j, x_{j+p}, x_{j+2p}, x_{j+3p}, \ldots\}$ from (2.7).

The same dependence problems that persist with serial LCGs provide for correlation problems with parallel LCGs. These long-range correlations have been well-documented [7] and extreme caution should be used with LCGs in both serial and parallel settings.

Parallelized Combined Multiple Recursive Generators

From Section 2.1, we know that the nth state of the jth generator for a CMRG generator is

$$x_{j,n} = [a_{j1}\, x_{j,n-1} + a_{j2}\, x_{j,n-2} + \cdots + a_{jk}\, x_{j,n-k}] \mod m_j.$$

This can be written as the vectors

$$X_{j,n} = \begin{pmatrix} x_{j,n} \\ x_{j,n-1} \\ \vdots \\ x_{j,n-k} \end{pmatrix}$$

for $j = 1, 2, ..., k$. The recurrences in (2.3) can then be compactly expressed as

$$X_{j,n+1} = (A_j\, X_{j,n}) \mod m_j,$$

where A_j is the $k \times k$ matrix

$$A_j = \begin{pmatrix} a_{j1} & a_{j2} & \cdots & a_{j(k-1)} & a_{jk} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.$$

Based on this and similar to the work done with LCGs in the previous subsection, we determine the leapfrog algorithm

$$X_{j,n+\nu} = (A_j^{\nu}\, X_{j,n}) \mod m_j \tag{2.8}$$

for any non-negative integer ν. Computing A_j^{ν} for a large value of ν may appear to be a daunting task but it can be done efficiently by squaring the A_j matrix iteratively using the “divide and conquer" strategy [16] that we now explain.

Define ν as $\nu = \sum_{i=0}^{h} g_i b^i$ for some $b > 2$ and $g_i \in \{0, 1, \ldots, b-1\}$. Then, we compute the following sequence: $A_j, A_j^b, A_j^{b^2}, \ldots, A_j^{b^h} \mod m_j$. Computation of $X_{j,n+\nu}$ in (2.2) then boils down to

$$X_{j,n+\nu} = (A_j^{\nu}\, X_{j,n}) \mod m_j = \left(\prod_{i=0}^{h} A^{g_i b^i}\, X_j \right) \mod m_j$$

for $j = 1, \ldots, k$.

This leapfrog procedure often performs quite well and is commonly thought to be the best way to parallelize CMRGs. Mascagni [28] gives ways to parameterize simple MRGs but not the more frequently used CMRGs. We will conclude this section by explaining the parallelization of the `MRG32k3a` generator using the matrix multiplication techniques above.

Recall from Section 2.1 that `MRG32k3a` is a CMRG defined by the states

$$\begin{aligned} x_{1,n} =& (a_{11}\, x_{1,n-1} + a_{12}\, x_{1,n-2} + a_{13}\, x_{1,n-3}) \mod m_1, \\ x_{2,n} =& (a_{21}\, x_{2,n-1} + a_{22}\, x_{2,n-2} + a_{23}\, x_{2,n-3}) \mod m_2 \end{aligned}$$

with $J = 2, k = 3, m_1 = 2^{32} - 209, a_{11} = 0, a_{12} = 1403580, a_{13} = -810728, m_2 = 2^{32} - 22853,$ $a_{21} = 527612$, $a_{22} = 0$ and $a_{23} = -1370589$. If we express the states of the generator as

$$X_{1,n} = \begin{pmatrix} x_{1,n} \\ x_{1,n-1} \\ x_{1,n-2} \end{pmatrix}, \qquad X_{2,n} = \begin{pmatrix} x_{2,n} \\ x_{2,n-1} \\ x_{2,n-2} \end{pmatrix},$$

the two recurrences above become

$$X_{1,n+1} = (A_1 X_{1,n}) \mod m_1, \qquad X_{2,n+1} = (A_2 X_{2,n}) \mod m_2,$$

where A_1 and A_2 are 3×3 matrices given by

$$A_1 = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} a_{21} & a_{22} & a_{23} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

We then use the “divide and conquer" algorithm on the leapfrog equation to produce a parallel `MRG32k3a` generator.

Parallelized Shift-Register Generators

Linear congruential generators and shift-register generators were the most popular options for generating pseudorandom numbers up until the 1990s. As parallel computing began to become commonplace, algorithms for creating parallel versions of both of these families of generators were also introduced. Aluru et al. [1] and Mascagni [28] gave a leapfrogging algorithm and parameterization algorithm, respectively, for parallelizing generalized feedback shift register generators. Unfortunately, as with LCGs, GFSRs have many limitations and these limitations frequently become more pronounced under parallelization. In this subsection, we will focus on the modified GFSR mentioned in Section 2.1 known as the Mersenne Twister.

Attempts have been made to use leapfrogging to create a parallel version of the Mersenne Twister similar to what was done with `MRG32k3a` in the previous subsection. However here the strategy of working with a $k \times k$ matrix carries with it the need for large amounts of memory. This is particularly relevant for the Mersenne Twister that requires a $19,937 \times 19,937$ matrix. A different kind of algorithm to reduce the amount of storage was obtained through a joint effort by L'Ecuyer, Matsumoto, and others [11]. This algorithm employs number theory results based on polynomial evaluation to produce an efficient jump ahead method for the Mersenne Twister. This more efficient approach uses the polynomial representation of the Mersenne Twister recurrence. We write the characteristic polynomial of the general matrix A (this was each of our A_j's in the previous subsection) as

$$p(z) = det(zI + A) = z^k + \alpha_1 z^{k-1} + \cdots + \alpha_{k-1} z + \alpha_k,$$

where I is the identity matrix and $\alpha_j \in \{0, 1\}$. The fundamental property of a characteristic polynomial is $p(A) = A^k + \alpha_1 A^{k-1} + \cdots + \alpha_{k-1} A + \alpha_k I = 0$. Defining

$$g(z) = z^\nu \mod p(z) = a_1 z^{k-1} + \cdots + a_{k-1} z + a_k,$$

we note that, for some polynomial $q(z)$,

$$g(z) = z^\nu + q(z)p(z).$$

This result and $p(A) = 0$ gives that $g(A) = A^\nu = a_1 A^{k-1} + \cdots + a_{k-1} A + a_k I$.

We now have a computational formula for computing $A^\nu X$ for some vector X using

$$A^\nu X = A(\cdots A(A(Aa_1X + a_2X) + a_3X) + \cdots + a_{k-1}X) + a_kX.$$

This entails advancing the state of the Mersenne Twister generator by $k-1$ steps from state X. Then, we add the states obtained at the steps with nonzero a_i's. An algorithm that reduces the number of these additions while somewhat increasing storage is given in [11].

The Mersenne Twister has also been parallelized using a parameterization technique called dynamic creation, which wass developed and implemented by Matsumoto and Nishimura in [33]. This technique creates a Mersenne Twister based on parameters such as a unique processor ID, word size, and a Mersenne prime. As with the jump ahead algorithm presented in this subsection, the characteristic polynomial of the Mersenne Twister is employed. The unique ID is encoded into the characteristic polynomial to ensure relatively prime characteristic polynomials. If we are working with r-sized words, the implementation given in [33] allows for $2^{r/2}$ parallel Mersenne Twisters with large Mersenne prime periods of up to $2^{44497}-1$.

Parallelized Lagged-Fibonacci Generators

A method similar to the leapfrog matrix algorithm from (2.2) can be used to develop a parallelization scheme for additive LFGs. Remembering the general form of the ALFG given in (2.6) as $x_i = x_{i-s} + x_{i-q} \mod 2^e$, the states of the generator can be expressed as the vector $X_i = (x_i, x_{i-1}, \ldots, x_{i-s-1})^T$. We then define the recurrence by

$$X_{i+1} = AX_i \mod 2^e,$$

where the $s \times s$ matrix A is defined by

$$A = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{pmatrix}.$$

The ones in the first row correspond to the lags of q and s. A similar pattern is exhibited in the remaining rows as was seen with the CMRG parallelization. Thus we can now use the same “divide and conquer" strategy to produce a leapfrog additive lagged-Fibonacci parallel pseudorandom generator.

Suppose that we force all seeds of MLFGs to be odd integers. Any odd integer x modulo 2^e can be expressed as

$$x = [(-1)^y 3^z] \mod 2^e,$$

where $y \in \{0, 1\}$ and $z \in \{0, 1, \ldots, 2^{e-2} - 1\}$ [29]. Substituting this relationship into (2.6) gives

$$x_i = [(-1)^{y_i} 3^{z_i}] \mod 2^e. \tag{2.9}$$

Here, y_i and z_i are from the recurrences

$$y_i = (y_{i-s} + y_{i-q}) \mod 2, \qquad z_i = (z_{i-s} + z_{i-q}) \mod 2^{e-2}$$

and are recognized as ALFGs with periods of 2^{q-1} and $2^{e-3}(2^q - 1)$, respectively. Initializing these two processes thereby gives us the appropriate seeds for use in (2.9).

Mascagni and Srinivasan [29] also describe a parallelization of the ALFG by a process known as seed parameterization. This involves placing a different cycle defined by its seed on each different processor. Defining different seeds to different processors does not guarantee that there will not be overlap between the streams of two different processors. The authors have, however, developed an algorithm that shows that seeds can be bit-wise initialized so that each unique assignment gives a provably full-period cycle. They also extend these parameterization results to MLFGs using (2.9) above.

Chapter 3

CURRENT METHODS FOR TESTING PSEUDORANDOM NUMBERS

3.1 Empirical Tests of Serial Pseudorandom Number Generators

The fundamental goal of serial pseudorandom number generation is to produce numbers that “appear random." This notion of randomness translates into having the output of the generator, say $u_0, u_1, \ldots$, effectively imitate a random sample from a uniform distribution on the interval $[0, 1]$ or simply $U[0, 1]$ subsequently. In the case of a generator that produces bits instead of uniform deviates, we will say that randomness occurs if the bits take the values 0 or 1 independently with equal probabilities. A perfect imitation in either case is our null hypothesis H_0 whose validity must be assessed through the application of statistical methods. Each of the test statistics described below differs in terms of the way they look for departure from the null model to data produced by a generator but each one represents an attempt to find contradictions to the claim that the output stream “appears random." Each has a known (or approximately known) distribution under H_0 so that departures from randomness manifest as unusual (or unlikely) values for the statistic.

In this section we will focus on what are called single-level tests in `TestU01`. By this we mean a test that computes the observed value y of the test statistic Y. We will define the P-value as $p = \mathbb{P}(Y \geq y|H_0)$, where $\mathbb{P}$ denotes probability and “$|H_0$” corresponds to the condition that H_0 is true. The decision rule will be to reject H_0 if the P-value is too close to either 0 or 1.

If the distribution of a test statistic Y is (approximately) continuous under H_0, its corresponding P-values are (approximately) a $U[0, 1]$ random variable under H_0. A P-value too close to 1 can be thought of as producing a stream that is overly uniform and a value too close to 0 can be thought of as not uniform enough. If the null distribution of Y is discrete, we need to use two different P-values to account for the jumps in the statistic’s probability mass function. The right P-value is $p_R = \mathbb{P}(Y \leq y|H_0)$ and the left P-value is $p_L = \mathbb{P}(Y \geq y|H_0)$. Then, the discrete P-value p is given in [20] by the piecewise function

$$p = \begin{cases} p_R, & \text{if } p_R < p_L \\ 1 - p_L, & \text{if } p_R \geq p_L \text{ and } p_L < 0.5 \\ 0.5 & \text{otherwise.} \end{cases}$$

In the next two subsections we will give details on many of the statistical methods that are implemented in `TestU01`. Our presentation splits them into two categories: (1) those that test H_0 for a sequence of real numbers in $(0, 1)$ and (2) those that test H_0 for a sequence of bits. Eleven common tests from group (1) and four tests from group (2) are described. Other tests and further details can be found in [19] and [20].

Tests for a Sequence of Real Numbers in $(0, 1)$

Gap and runs tests

The gap test [16] looks for patterns in a sequence of numbers that may occur locally. One first defines perimeters α and β with $0 \leq \alpha < \beta \leq 1$. We then count the number of steps (also known as the gap size) between any pair of successive visits into $[\alpha, \beta]$ by numbers in the generated stream. If Y_j denotes the number of gaps of size j for $j \geq 0$ the realized values of $Y_0, Y_1, \ldots$ are compared to their mean values under H_0 via a chi-square test.

The runs test also looks for local patterns. After generating n numbers, it counts the different lengths of increasing (or decreasing) sequences (runs). After merging all run lengths greater than or equal to six together, for large values of n, it produces a test statistic that is approximately chi-square distributed.

Poker and coupon collector tests

The poker test [16] begins by generating k integers between 0 and $d - 1$ for $d, k < 128$. Let Y denote the number of distinct integers that occur. This process is repeated n times to produce observations $Y = y_1, \ldots, Y = y_n$ whose empirical frequencies are compared to the expected frequencies under H_0 via a chi-square test.

In a similar manner, the coupon collector test generates a sequence of integers in $\{0,\ldots,d-1\}$. Denote by Y the count corresponding to how many different sequences must be generated before each of the d possible integers appears exactly once. This process is repeated n times. We then count how many times each of the different values of Y were observed and use a chi-square test to compare these to their expected frequencies.

Knuth serial and related tests

Given a number of dimensions t, the Knuth serial test [16] divides the interval $[0,1)$ into d equal segments which divides the corresponding t-dimensional hypercube $[0,1)^t$ into d^t hypercubes. (We refer to this as the Knuth serial test to avoid confusion with the general term “serial tests" that was used to describe the tests that are treated in this section.) A sample of n t-dimensional vectors in $[0,1)^t$ is created by grouping every t successive values from a generator into a vector with no overlap. These vectors are viewed as points in t-dimensional space and we then count the number that fall in each of the d^t hypercubes. Again, these values are compared to the theoretical counts via a chi-square test.

The collision test is a frequently used variant of the Knuth serial test. Instead of counting the number of points in each of the smaller hypercubes, one counts how many times a point falls in a hypercube that already has one or more points in it. The resulting statistic is approximately Poisson distributed when H_0 holds [19]. Both the Knuth serial and collision tests furnish measures of how clustered the data are that are produced by the generator.

A further modification of the Knuth serial and collision tests leads to the birthday spacings test proposed by Marsaglia in 1985. One again generates n points in t-dimensional space and divides the larger hypercube into $k = d^t$ smaller hypercubes (cells) while numbering them from 0 to $k-1$. The name of the test comes from the n points being viewed as n birth dates in a single year consisting of k days. Denote the cell numbers where the n points fall as $I_1, I_2, \ldots, I_n$ and sort them in increasing order. Next, we determine the spacings $I_{j+1} - I_j$ for $1 \leq j < n$. If we denote the number of collisions/recurrences between these spacings as Y, assuming H_0 is true, Y is approximately Poisson distributed with mean $n^3/(4k)$. A corresponding discrete P-value can be computed from this observed statistic.

Overlapping versions of the Knuth serial, collision, and birthday spacings tests were also proposed by Marsaglia. He penned them as “monkey" tests since they treat the generator as a monkey typing “random" characters from an alphabet containing d-letters. For example, the overlapping collision test counts how many times each t-letter word appears in the sequence typed by the monkey. For large sample sizes each test statistic is approximately chi-square distributed.

Others

Another test concerning t-dimensional uniformity, called the spectral test, is described in detail in [16]. It measures the maximal distance between adjacent parallel hyperplanes that are created by the generation of n points in t-space. These n points are also created in an overlapping way. Transformations of discrete Fourier coefficients are computed in the determination of this largest distance and for large enough n the corresponding test statistic is approximately normal.

The maximum-of-t test first generates t values in $[0,1)$. It then computes the maximum value Y of these t values. This process is repeated n times. The empirical distribution of the n values of Y is compared to the theoretical cumulative distribution function of the maximum, $F(y) = y^t$, via a chi-square test and an Anderson-Darling test.

The creators of the Mersenne Twister generator, Matsumoto and Kurita, have also made a contribution to the testing of PRNGs that we mention here [31]. Their test generates k uniform deviates $u_1, \ldots, u_k$. Then, it computes the value

$$Y = \sum_{j=1}^{k} \mathbb{I}[\alpha \leq u_j < \beta]$$

with $\mathbb{I}$ corresponding to the indicator function. Therefore, this quantity represents the number of u_j's in the interval $[\alpha, \beta)$ for some $0 \leq \alpha < \beta < 1$. $Y \sim Binomial(k, \beta - \alpha)$ under H_0. If we repeat this process n times, the outcomes can be compared to the theoretical binomial distribution with a chi-square test.

Tests for a Sequence of Bits

The first test discussed here is the autocorrelations test. The sample autocorrelation of lag d for a sequence $b_1, \ldots, b_n$ of n bits is given by

$$Y = \sum_{i=1}^{n-d} b_i \oplus b_{i+d},$$

where $\oplus$ is the exclusive-or operation. It can be shown that, when H_0 holds, Y is binomially distributed with parameters $n-d$ and $1/2$. So for large $n-d$, Y is approximately normal. This test is an attempt to quantify the clustering of the generated bits.

Gap, runs, and Knuth serial tests also exist for testing a string of bits. For the gap and runs tests, we collect the lengths of all runs of 1's and runs of 0's and proceed similarly to the corresponding tests described in the previous subsection. For the Knuth serial and similar tests, we look at the number of occurrences of a given string in the n strings and compare that to a theoretical expected value.

We can also test a sequence of bits by constructing a binary matrix. We fill up an $L \times k$ matrix row by row using the generated bits in succession, compute the rank of the matrix, and repeat this process n times while keeping track of the number of occurrences of each rank. As one could guess, we compare the resulting frequencies to their null model expectations via a chi-square test.

A Hamming weight test examines the proportion of 1's in a segment of the generated stream. More specifically, the test generates n disjoint blocks of L bits. Under H_0, the number of 1's in each block are independent and binomially distributed. The observed Hamming weights are the number of blocks out of n having j 1's, for $0 \leq j \leq n$, and their values can be compared to the binomial model using a chi-square statistic.

Lastly we will discuss the random walk test which creates a random integer walk using l bits $b_1, \ldots, b_l$. The walk begins at 0 and on the jth step moves one unit to the left if $b_j = 0$ or one unit to the right if $b_j = 1$. Define $S_0 = 0$ and $S_k = \sum_{j=1}^{k}(2b_j - 1)$ for $k > 0$. Under H_0, the process $\{S_k, k \geq 0\}$ is a random walk and using the binomial distribution, we have

$$p_{k,y} \equiv \begin{cases} \mathbb{P}[S_k = y] = 2^{-k}\binom{k}{(k+y)/2}, & \text{if } k+y \text{ is even,} \\ 0, & \text{otherwise.} \end{cases}$$

Assuming l is even, we define the test statistics

$$H = l/2 + S_l/2 \text{ (the number of steps to the right),}$$

$$M = \max\{S_k, 0 \leq k \leq l\} \text{ (the maximum value reached by the walk),}$$

$$J = 2\sum_{k=1}^{l/2} \mathbb{I}[S_{2k-1} > 0] \text{ (the fraction of time spent to the right of the origin),}$$

$$P_y = \min\{k : S_k = y\} \text{ for } y > 0 \text{ (the first passage time at } y\text{),}$$

$$R = \sum_{k=1}^{l} \mathbb{I}[S_k = 0] \text{ (the number of returns to 0), and}$$

$$C = \sum_{k=3}^{l} \mathbb{I}[S_{k-2}S_k < 0] \text{ (the number of sign changes).}$$

In 1968, Feller gave the null distributions for these statistics as

$$\begin{aligned}
\mathbb{P}[H = k] &= \mathbb{P}[S_l = 2k - l] = p_{l,2k-l} = 2^{-l}\binom{l}{k}, && 0 \leq k \leq l, \\
\mathbb{P}[M = y] &= p_{l,y} + p_{l,y+1}, && 0 \leq y \leq l, \\
\mathbb{P}[J = k] &= p_{k,0}p_{l-k,0}, && 0 \leq k \leq l, k \text{ even}, \\
\mathbb{P}[P_y = k] &= (y/k)p_{k,y}, \\
\mathbb{P}[R = y] &= p_{l-y,y} && 0 \leq y \leq l/2, \\
\mathbb{P}[C = y] &= 2p_{l-1,2y+1}, && 0 \leq y \leq (l-1)/2.
\end{aligned}$$

The random walk test implemented in `TestU01` proceeds in the following manner. Take two even integers $0 < m_0 < m$ as parameters and generate n random walks of length m. For each $l \in \{m_0, m_0+2, ..., m\}$, compute the n values of the six statistics. Then one compares the empirical distributions of these statistics with the corresponding theoretical ones via a chi-square test.

Usage in TestU01

As mentioned in the introduction to this section, `TestU01` enables users to choose any of the above tests and run them on a particular built-in generator, an external generator built in C, or on a given file of uniform deviates or bits. `TestU01` also includes three batteries of tests for a sequence of uniform deviates that encompass many of the most common tests. The smallest and fastest battery is defined as `SmallCrush` and contains ten of the tests described above: namely, the birthday spacings, collision, gap, poker, coupon collector, maximum-of-t, Matsumoto and Kurita's weight distribution, matrix rank, Hamming number, and random walk tests. It usually requires only a few minutes of computation time. `Crush` contains 96 different variations of the tests explained above as well as other tests. It requires about an hour of computation time and uses approximately 2^{35} pseudorandom numbers [20]. The most stringent of the batteries is `BigCrush` which is made up of 31 different tests and a total of 106 variations. It uses close to 2^{38} pseudop-random numbers and usually takes around eight or more hours to complete. Batteries also exist for testing generated bits and are called `Rabbit` and `Alphabit` in `TestU01`.

It is worthy of mention here that both `MRG32k3a` and the Mersenne Twister that we have focused on in Chapter 2 pass `BigCrush`. The problems that are present for linear congruential and shift-register generators materialize in their failures to pass even `SmallCrush` or `Rabbit` for many chosen parameters.

3.2 Testing Techniques for Parallel Pseudorandom Number Generators

`TestU01` also has the capability to test parallel generated streams (or quasi-parallel, in that parallel processors may not be used but different vectors are generated by the same processor) in two different ways. The first is an extension of the single-level tests discussed in the previous section. The second is what is known as a “parallel filter" in `TestU01`. This allows for the output of several generators or different streams from the same generator to be combined into a single stream of pseudorandom numbers and then tested using the techniques in the previous section. Both approaches are described below.

Two-level Tests

In a two-level test, one replicates the single-level test p times. That is to say, one generates p (ideally) independent copies of the test statistic Y denoted $Y_1, Y_2, \ldots, Y_p$. We define F to be the theoretical cumulative distribution function of Y under H_0. For the continuous case, the transformed variables $U_1 = F(Y_1), \ldots, U_p = F(Y_p)$ should imitate independent and identically distributed $U[0, 1]$ random variables. The second level of the test comes from taking these p uniform deviates and comparing them against the theoretical uniform distribution via a goodness-of-fit test. `TestU01` includes a module that calculates the value of common goodness-of-fit measures such as the Kolmogorov-Smirnov, Anderson-Darling, and Crámer-von Mises statistics. These two-level tests can also be performed by comparing the untransformed observed test statistics $Y_1, \ldots, Y_p$ to the actual distribution of the test statistic for that particular test via any of the goodness-of-fit tests.

One can see how this two-level test could work in testing parallel generated streams of pseudorandom numbers. In that case, p corresponds to the number of processors used (or possibly multiple smaller substreams on each processor could be tested). Then each of the generated streams (or substreams) could be tested using one of the serial tests in `TestU01` to produce an observed test statistic (and a P-value/transformed variable). These results can then be tested at the second level against their theoretical counterparts via a goodness-of-fit test. If the two-level test does not give evidence against the null hypothesis, we stick with the original null model and proceed as if the streams are independent and identically distributed.

Since nearly all of the tests used in `TestU01` produce test statistics that are (at least approximately) distributed as chi-square, normal, or Poisson, we can also make use of the fact that the sum of multiple test statistics from these distributions follow the same type of distribution. For example, if Y is Poisson with mean λ, then $S_p = Y_1 + \ldots + Y_p$ is also Poisson with parameter/mean $p\lambda$. `TestU01` reports the results based on S_p for the case where $p > 1$ as well as the results of the other two-level methods described earlier.

Parallel Filter

By using the `unif01_CreateParallelGen` function in `TestU01`, one is able to specify p different generators (or p generators from the same family with identical or different parameters) each with stream length L. The procedure then outputs these p streams one after another for a total single stream length of pL. One can send a stream generated using this function to any of the batteries of tests or each of the individual tests. Therefore, the “parallel filter" essentially turns a matrix of generated values into a single serial stream of values and then tests that single stream accordingly.

Chapter 4

PROBLEMS WITH EXISTING PARALLEL TESTING METHODS

Both of the parallel testing procedures in `TestU01` fail to recognize the multivariate nature of the generated data. In the two-level testing option, observed test statistics (and corresponding P-values) from a particular serial test are generated for each vector of data. These statistics are then combined into a new data set that is compared to a null distribution using a goodness-of-fit test. This process loses information in the data by reducing each vector to a single univariate summary measure, i.e., an observed test statistic or a P-value. In particular, the correlation between streams is now marginalized to whatever correlation remains between the summary measures and may be difficult to detect with goodness-of-fit methodology. In the parallel filter option, the array or matrix of data from the different streams is transformed into a single vector by stacking the processors' outputs on top of each other. This vector is tested for randomness using the standard serial testing procedures. Correlations between the different vectors are likely rendered undetectable through this process especially if the correlated vectors reside in locations that are far apart from each other in the flattened one-dimensional data array. In this chapter we demonstrate that these parallel testing methods are in many ways unable to effectively detect dependence of parallel pseudorandom generated streams.

The chapter begins with a description of the `CrushFile` addition that we created for the `TestU01` package. `CrushFile` is a modification of the stringent `Crush` battery of 96 tests that is designed to test a file of deviates created by generators that have been implemented in the R programming language. We will then use `CrushFile` to demonstrate that correlated vectors of data can still go undetected in the batteries of tests in `TestU01`.

Three generators, that we will hereafter call the “Smoking Guns", have been designed using time series ideas for the purpose of illustrating the flaws in the parallelization schemes of `TestU01`. The motivation behind how these correlated vectors of data were created as well as the different implementations to match the input requirements of `SmallCrushFile` and `CrushFile` are discussed below. We will see that all three of our Smoking Guns were successful in passing all of the tests in `SmallCrushFile` and `CrushFile`. This provides the prima facie evidence we need to assert that TestU01 is unable to detect basic kinds of correlations that could be anticipated with pseudorandom numbers generated in parallel. This poses a potential problem for scientists conducting Monte Carlo studies since it violates the first of the additional requirements for a PPRNG to be considered “good" by allowing strong dependencies to exist among the processor streams.

4.1 Development of the CrushFile Addition

The SmallCrushFile test battery provides an adequate check of the ability of TestU01 to detect problems with a given generator. However, it only computes 15 test statistics. While it gives us some intuition about TestU01's ability to detect correlations in data from a parallel generation scheme, we believe it is more relevant to run a more demanding set of tests in an attempt to detect shortcomings of the TestU01 parallel testing algorithms. The Crush battery provides a suite of tests that is better suited to our purposes.

Crush is designed to work on a given external generator that has code written in C or is a built-in generator in the TestU01 suite. In contrast our Smoking Gun generators in the next section have all been written in R in order to employ the speed of its vector computations.

The SmallCrushFile test battery is a modification of the SmallCrush battery that is included in the basic install of TestU01. The SmallCrush battery also requires either an external generator or a pre-defined TestU01 generator for its testing purposes. The TestU01 authors created this to allow users to run tests on an inputted file of deviates. They did not however create a CrushFile battery that could run the near 100 tests in Crush on an inputted file of uniform deviates.

The complete listing of the modifications to the Crush battery that read data from an inputted file of deviates is included in the appendix. The sizes of the files required to run this CrushFile addition are very large. Since some of the tests in Crush require more than a billion deviates, the file size is usually between 20 and 30 gigabytes. It then takes between 6 and 8 hours to fully run all of the tests in CrushFile on a modern machine.

In the next section we will describe the Smoking Gun generators that we mentioned above. The results and details of running SmallCrushFile and this new CrushFile extension on their outputted deviates will then be reported. We will see that even seemingly large correlations between vectors and easily noticeable dependencies go undetected in `TestU01`'s parallel testing techniques.

4.2 Three Problematic Generators

Normally Transformed VMA(1) process

To illustrate the shortcomings of the two parallel testing methods in `TestU01`, we designed a vector autoregressive moving average (VARMA) model based on standard normal pseudorandom deviates. This model has built-in correlations among the generated vectors but still passed `TestU01`'s `SmallCrushFile` and the `CrushFile` addition. In what follows we will describe this model as well as provide code for its implementation in R.

We implemented the vector moving average of order one model which will be referred to as VMA(1) going forward. A VMA(1) model is written as $X_t = e_t + \theta e_{t-1}$, where X_t is a vector of length n, θ is an $n \times n$ matrix, and e_t is an n-dimensional normal random vector with mean vector 0 and covariance matrix $\sigma_e^2 I_n$ with $t = 1, \ldots, p$ and I_n the n-dimensional identity matrix. Here we are thinking of X_t as the stream that will be used by the tth process.

For this first Smoking Gun generator, we chose $\theta = 0.9 I_{5{,}500{,}000}$ and $p = 10$. Each e_t is generated from a $N_{5{,}500{,}000}(0, I_{5{,}500{,}000})$ distribution using `rnorm` in R, which is based on the Mersenne Twister generator by default. An initial seed vector e_0 is required and was also chosen from the same distribution as a vector of length $5{,}500{,}000$ using `rnorm`. The values of $p = 10$ and $n = 5{,}500{,}000$ were chosen since `SmallCrushFile` in `TestU01` requires around 55 million deviates.

We also modified n, p, and θ to match the sample size needed for `CrushFile` and created a file of deviates that was used as the input file for `CrushFile`. With `Crush` being a much more rigorous series of tests than `SmallCrushFile`, the values for n and p were greatly increased to accommodate the much larger number of deviates. (We determined, for example, that some of the tests in `Crush` require around 1.3 billion deviates.) The choice of p was increased from 10 to 100 and n was chosen to be 13 million instead of 5.5 million.

The built-in correlations are apparent for this Smoking Gun generator by how it is constructed. This is summarized in the steps taken for its creation:

1. A very long stream e_i of iid $N(0,1)$ random variables is generated.
2. For $j = 1, \ldots, p$, the components of the matrix X are defined as

$$X_{ij} = .9e_{(i-1)p+j} + e_{(i-1)p+j-1}.$$

3. Thus, $Cov(X_{ij}, X_{i,j+1}) = .9$ and X_{ij} are iid $N(0, (1 + .9^2))$ for each j.
4. Therefore, $\Phi(X_{ij}/\sqrt{1 + .9^2})$ are iid $U(0,1)$ for each j, where Φ represents the cumulative distribution function of the standard normal distribution.

As a result of Item 3 we are assured that correlation has been built into the consecutive vectors that are produced by this Smoking Gun generator even if the precise form of the dependence is masked by the transformation to uniform deviates. The correlation that manifests in the data will be revealed empirically in the following chapters that show how our `TestU01` multivariate extension detects these dependencies. An explanation of how these steps are implemented in R follows.

In order to produce pseudorandom $U[0,1]$ deviates we used the `pnorm` function in R after converting the entries in the matrix of data (denoted as `Xmat` in the code below) to standard normal random variables. The function `pnorm` returns the cumulative distribution function for each of our 55 million transformed $N(0,1)$ deviates. This transformation gives the desired $U[0,1]$ deviates. The following R code shows this process as well as the output to a TXT file for the `CrushFile` case. (The `SmallCrushFile` code is very similar with the only changes being the specification of p and n along with a change in the outputted filename. The splitting of the `u` vector prior to output is also not necessary in the code for the `SmallCrushFile` file of deviates.)

Listing 4.1: vma1_large.R

```
set.seed(123)

theta <- .9
```

```
p <- 100
n <- 13000000
e0 <- rnorm(n)
e <- rnorm(n*p)
Xmat <- matrix(0, n, p)
Xmat[, 1] <- theta*e0 + e[1:n]
for(t in 2:p){
  Xmat[, t] <- e[(n*(t - 1) + 1):(n*t)]
        + theta*e[(n*(t - 2) + 1):(n*(t - 1))]
}
sigma <- sqrt(1 + theta^2)
Xmat <- Xmat/sigma
Umat <- pnorm(Xmat)
X <- as.vector(Xmat)
u <- pnorm(X)
a <- u[1:500000000]
b <- u[500000001:1000000000]
c <- u[1000000001:1300000000]
write.table(a, file = "vma1_1.3bl.txt",
        col.names = FALSE, row.names = FALSE)
write.table(b, file = "vma1_1.3bl.txt",
        append = TRUE, col.names = FALSE, row.names = FALSE)
write.table(c, file = "vma1_1.3bl.txt",
        append = TRUE, col.names = FALSE, row.names = FALSE)
```

The generated file `vma1_55ml.txt` was tested using the `SmallCrushFile` battery in `TestU01` via the following C++ code.

Listing 4.2: vma1_smallCrush.cpp

```
extern "C"{
#include "swrite.h"
#include "bbattery.h"
}
```

```
int main (void) {
   swrite_Basic = FALSE;
   bbattery_SmallCrushFile ("vma1_55ml.txt");
   return 0;
}
```

The output below are the results of running this `SmallCrushFile` code.

Listing 4.3: vma1_smallCrush_out.txt

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 File:             /home/ismay/VMA1/vma1_55ml.txt
 Number of statistics:  15
 Total CPU time:   00:02:08.68

 All tests were passed
```

Similarly, the `vma1_1.3bl.txt` file was tested using the `CrushFile` addition via the following C++ code. The main function from `CrushFile` is given below. (Recall that the full code from the addition is in the appendix.)

Listing 4.4: vma1_crush.cpp

```
int main(){

   bbattery_CrushFile ("vma1_1.3bl.txt");

  return 0;
}
```

The output below are the results of running this `CrushFile` code.

Listing 4.5: vma1_Crush_out.txt

```
========= Summary results of Crush =========
Version:          TestU01 1.2.3
Generator:        ufile_CreateReadText
Number of statistics:  144
Total CPU time:   07:12:01.68

All tests were passed
```

Univariate Time Series Moving Across the Processors

Let us view the data from a parallel generation scheme as a matrix with columns that correspond to the different streams/processors. Then, the first Smoking Gun creates a column-wise dependence that has the same form for every row. As another alternative we investigated how `TestU01` would perform with a time series that evolves more locally by moving down rows and moving across streams/processors. The univariate case of this will be discussed here and the bivariate case will be discussed in the next subsection. We again used R for its ability to easily simulate time series data to create a file of deviates for assessing both `SmallCrushFile`'s and `CrushFile`'s ability to detect faulty parallel generators.

For this univariate case, we begin by simulating a long vector x of length $n \times p$ following a moving average of order one process (MA(1)). This is similar to the VMA(1) process discussed in the last subsection except the process is now working down the individual elements instead of across the vectors. For further specificity, the vector x can be written component-wise as $x_t = e_t + \theta e_{t-1}$, where x_t represents the time series process at time t, $\theta = 0.3$, and e_t is the white noise error term at time t with $t \in \{1, \ldots, np\}$. To standardize these data, we then divide this vector x by $\sqrt{1 + \theta^2}$. Similar to the first Smoking Gun, we then transform to $U[0, 1]$ deviates by using the Φ function. The next step involves placing this long vector of deviates into a matrix representing a time series moving across the vectors. Lastly, to prepare for the parallel testing procedures in `TestU01`, we flatten this matrix into one long vector again and output it to a file. The R code below shows this procedure for creating the file used as input for `CrushFile`. (The `SmallCrushFile` code is similar apart from changes in the specification of p, n, and the outputted filename. Again, the splitting of the Xvec vector prior to output is not necessary for code corresponding to the input file for SmallCrushFile.)

Listing 4.6: mvaTS_1_huge.R

```
theta <- 0.3
p <- 100
n <- 13000000
set.seed(123)
x <- arima.sim(list(ma = theta), n*p)
sigma <- sqrt(1 + theta^2)
x <- x/sigma
u <- pnorm(x)
Xmat <- matrix(0, n, p)
for(i in 1:n)
  Xmat[i,] <- u[((i-1)*p + 1):(i*p)]
Xvec <- as.vector(Xmat)
a <- Xvec[1:500000000]
b <- Xvec[500000001:1000000000]
c <- Xvec[1000000001:1300000000]
write.table(a, file = "mvaTS_1_1.3bl.txt",
        col.names = FALSE, row.names = FALSE)
write.table(b, file = "mvaTS_1_1.3bl.txt", append = TRUE,
        col.names = FALSE, row.names = FALSE)
write.table(c, file = "mvaTS_1_1.3bl.txt", append = TRUE,
        col.names = FALSE, row.names = FALSE)
```

The generated file mvaTS_1_55ml.txt was tested using the SmallCrushFile battery in TestU01 via the following C++ program. The test results appear below the listing.

Listing 4.7: mvaTS_1_smallCrush.cpp

```
extern "C"{
#include "swrite.h"
#include "bbattery.h"
}
```

```
int main (void) {
   swrite_Basic = FALSE;
   bbattery_SmallCrushFile ("mvaTS_1_55ml.txt");
   return 0;
}
```

The results of running this `SmallCrushFile` code are given in the output below.

Listing 4.8: mvaTS_1_smallCrush_out.txt

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 File:             /home/ismay/MVATS/mvaTS_1_55ml.txt
 Number of statistics:  15
 Total CPU time:   00:02:06.77

 All tests were passed
```

The `mvaTS_1_1.3bl.txt` file was analyzed similarly using the `CrushFile` addition as indicated in the subsequent listing. The main function from `CrushFile` is given below with the full set of code from the addition available in the appendix.

Listing 4.9: mvaTS_1_crush.cpp

```
int main(){

   bbattery_CrushFile ("mvaTS_1_1.3bl.txt");

  return 0;
}
```

The results of running this `CrushFile` code are given in the output below.

Listing 4.10: mvaTS_1_Crush_out.txt

```
========= Summary results of Crush =========

 Version:          TestU01 1.2.3
 Generator:        ufile_CreateReadText
 Number of statistics:  144
 Total CPU time:   06:03:29.84

 All tests were passed
```

Bivariate Time Series Moving Across the Processors

For the bivariate case, we begin by simulating two long vectors x_1 and x_2 each of length $(np)/2$ and each following a moving average of order one process (MA(1)). We then stack x_1 on top of x_2, divide this stacked matrix X by $\sqrt{1+\theta^2}$, and transform to $U[0,1]$ deviates as before. Then an array of data from the “processors" is filled in a similar fashion to that in the previous subsection except in pairs instead of one at a time. Lastly, to prepare for the parallel testing procedures in `TestU01`, we flatten this array into a long vector again and output the result to a file. The R code below implements this procedure for creating the file used as input for `CrushFile`. (The `SmallCrushFile` code requires only changes in p, n, and the outputted filename. The `Xvec2` vector need not be split in this case.)

Listing 4.11: mvaTS_2_huge.R

```
theta <- 0.3
p <- 1000
n <- 1300000
set.seed(123)
x1 <- arima.sim(list(ma = theta), n*p/2)
x2 <- arima.sim(list(ma = theta), n*p/2)
X <- rbind(x1, x2)
sigma <- sqrt(1 + theta^2)
X <- X/sigma
U <- pnorm(X)
Xmat <- matrix(0, n, p)
for(i in 1:n)
```

```
  Xmat[i,] <- as.vector(U[1:2, ((i-1)*p/2 + 1):(i*p/2)])
Xvec2 <- as.vector(Xmat)
a <- Xvec2[1:500000000]
b <- Xvec2[500000001:1000000000]
c <- Xvec2[1000000001:1300000000]
write.table(a, file = "mvaTS_2_1.3bl.txt",
        col.names = FALSE, row.names = FALSE)
write.table(b, file = "mvaTS_2_1.3bl.txt", append = TRUE,
        col.names = FALSE, row.names = FALSE)
write.table(c, file = "mvaTS_2_1.3bl.txt", append = TRUE,
        col.names = FALSE, row.names = FALSE)
```

The C++ code below tests `mvaTS_2_55ml.txt` using the `SmallCrushFile` test battery.

Listing 4.12: mvaTS_2_smallCrush.cpp

```
extern "C"{
#include "swrite.h"
#include "bbattery.h"
}

int main (void) {
   swrite_Basic = FALSE;
   bbattery_SmallCrushFile ("mvaTS_2_55ml.txt");
   return 0;
}
```

The `SmallCrushFile` results are now given.

Listing 4.13: mvaTS_2_smallCrush_out.txt

```
========= Summary results of SmallCrush =========

 Version:          TestU01 1.2.3
 File:             /home/ismay/MVATS/mvaTS_2_55ml.txt
 Number of statistics:  15
 Total CPU time:   00:02:09.37

 All tests were passed
```

The `mvaTS_2_1.3bl.txt` file was tested using the `CrushFile` addition for which the main function is given below followed by the test results.

Listing 4.14: mvaTS_2_crush.cpp

```
int main(){

   bbattery_CrushFile ("mvaTS_2_1.3bl.txt");

  return 0;
}
```

The `CrushFile` results are now given.

Listing 4.15: mvaTS_2_Crush_out.txt

```
========= Summary results of Crush =========

 Version:          TestU01 1.2.3
 Generator:        ufile_CreateReadText
 Number of statistics:  144
 Total CPU time:   06:37:43.22

 All tests were passed
```

4.3 Discussion

This chapter has provided three examples of generators with built-in dependencies among streams/processors that pass the `SmallCrushFile` and `CrushFile` batteries of tests. These support our contention that the flattening processes in `TestU01` can weaken inter-stream dependence to the extent that it becomes undetectable by the tests in the package.

The first of our Smoking Gun generators focused on building in pairwise correlations between successive vectors. Since `TestU01` requires a single serial stream, it was found that these pairwise correlations go undetected because the correlated elements fall far apart in the flattened vector of deviates.

The second and third Smoking Gun generators demonstrate that the flattening processes mask dependence that is created by a time series progressing across the streams. In concert with the first generator, they illustrate how simple it is to fool the `TestU01` flattening approach with cases where the dependence between streams is actually quite obvious.

The goal of the following chapters is to provide methodology for detecting dependencies such as those found in our Smoking Gun generators and to show the importance of analyzing the generation of parallel deviates from a multivariate perspective. The chapters will describe an extension of `TestU01` built in C++ that aims to add to the current strong serial methodologies in the package by enhanced parallel testing techniques.

Chapter 5

TESTU01 MULTIVARIATE EXTENSION - CORRELATION MOTIVATED MULTIVARIATE TESTS

In the next two chapters, we will describe the intuition behind, the development of, and the results of tests housed in our multivariate extension to `TestU01`. This extension was built in C++ and one of its purposes is to better detect types of inter-stream dependence that tend to be overlooked by `TestU01`'s parallel testing schemes. In the following chapter, we will explore the methods based on testing for correlations between the generated output of the different vectors/processors. Chapter 6 will focus on tests deriving from time series analysis methodology.

The first section of this chapter is focused on testing for pairwise correlations between the vectors of generated data. As was noticed in the previous chapter, `TestU01`'s flattening techniques allow for reasonably simple consecutive pairwise correlations to pass through its suites of tests undetected. A few of the novel aspects of this portion of the C++ implementation will also be discussed including the use of the Benjamini/Hochberg/Yekutieli algorithm for controlling the experiment-wise error rate for these simultaneous tests of significant pairwise stream correlations that are an aspect of our extension. The section will conclude with the results of testing a variety of different generators including our Smoking Guns from Chapter 4 with our pairwise correlation tests that are referred to as `mcorr` in the extension.

The second section of this chapter is related to the first section in that the focus is on sample correlations. The difference is that we avoid the need for experiment-wise error control by employing a single likelihood ratio statistic to test that the matrix of pairwise inter-stream correlations is the identity. The details behind the derivation of this test statistic and its rejection region are provided for completeness of exposition. Our discussion will highlight a few interesting aspects of the code development and, similar to Section 5.1, results from a variety of different generators will be analyzed via this `mmult` part of our extension.

All of the source code from the extension described in this chapter can be found in Appendix B at the end of the thesis.

5.1 Pairwise Correlations

A typical data set in statistics would consist of n observations on p variables that is stored in an $n \times p$ matrix. A natural choice for detecting dependencies between the variables is to test for pairwise correlations among the columns of the data. In our setting the variables of interest $X_1, \ldots, X_p$ correspond to the pseudorandom numbers produced by the different streams or processors. Our extension implemented three different traditional types of correlation coefficients: Pearson's product-moment coefficient, Spearman's rank correlation coefficient, and Kendall's rank correlation coefficient. Transformation of these coefficients to (approximate) normality was performed in each case. With the sample sizes being used in our setting, the normal approximations can be relied on to give accurate P-values to help in the decision making process.

Aspects of the Pairwise Correlations Part of the Extension

With matrices and vectors the primary storage unit in our code, we used the C++ version of the Template Numerical Toolkit (TNT) developed by the National Institute of Standards and Technology [35] throughout our `TestU01` extension. This package is particularly valuable in that it is open source and, as a result, successfully addresses many of the portability and maintenance problems of creating and using multidimensional arrays in C++.

The constructor for our `mcorr` class requires an integer `N` corresponding to the number of rows in the matrix of generated values to be tested, an integer `P` for the number of columns, a TNT Array2D object `Mat` to store the inputted matrix of deviates, and a specified significance level `Alpha` for the hypothesis tests that check for pairwise vector dependence. Given this matrix of inputted deviates of size `NP`, we create three different correlation matrices containing each of the three correlation coefficients for determining the correlations between the `P` vectors. Since the correlation matrices are symmetric with unit diagonal entries, we save computation time and space by computing only the lower triangular part of the matrices. This implies that $\binom{\mathtt{P}}{2}$ correlations are determined and stored in a TNT Array2D object `corrData` with $\binom{\mathtt{P}}{2}$ rows and three columns with the second and third columns specifying the locations for its entries in the original correlation matrix.

To evaluate the binomial coefficient necessary to compute the size of $\binom{\mathtt{P}}{2}$, we used the recursive identity

$$\binom{N}{K} = \frac{N}{K}\binom{N-1}{K-1}.$$

This algorithm is much faster and more efficient than the traditional factorial recursive algorithm that is often used to compute binomial coefficients. An implementation of this recursion is provided in the C++ code listed below from the `mcorr.cpp` source file.

Listing 5.1: mcorr_binomCoef

```
unsigned int mcorr::mcorr_binomCoef(unsigned int N,
        unsigned int K){
  if(K == 0 || K == N)
    return 1;
  else
    return (N*mcorr_binomCoef(N - 1, K - 1))/K;
}
```

Note that we employ the C++ facility for recursion (i.e., a function may call itself) in carrying out the calculation.

Pearson correlation

The `mcorr.cpp` file uses the standard two-pass algorithm for computing the lower triangle of the Pearson correlation coefficient matrix. This entails computing the means of each column, the resulting $\binom{\mathtt{P}}{2}$ covariance matrix entries, and then the division necessary to create the correlation values. Next, the Pearson correlation values are transformed into a z-score $Z(r)$ based on the Fisher transformation $F(r)$

$$F(r) = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right)$$

as

$$Z(r) = \sqrt{\mathtt{N}-3}\,F(r),$$

where r represents the Pearson correlation and `N` is the number of rows/observations as before [10]. These transformed values are stored in a TNT Array2D object `Z` keeping the three column structure of the `corrData` object. Each of the $Z(r)$ values follow approximately a standard normal distribution under the null hypothesis of statistical independence. More precisely, we are testing the hypothesis that $Corr(X_t, X_{t'}) = 0$ versus $Corr(X_t, X_{t'}) \neq 0$ for each of the $m = \binom{\mathrm{P}}{2}$ choices of t and t'. If this hypothesis is rejected for any (t, t') it will signify rejection of the overall inter-stream independence model.

Spearman correlation

The `mcorr.cpp` file includes code that also computes the lower half of the Spearman correlation matrix by first ranking each column in ascending order and accounting for any possible ties (duplicate values in a column). This new rank matrix $Ranks$ is then passed into the Pearson correlation function described above.

To convert the Spearman correlation values to z-scores, the Fisher transform is slightly modified as

$$Z_{spear}(r) = \frac{1}{2}\sqrt{\frac{N-3}{1.06}}\ln\left(\frac{1+r}{1-r}\right),$$

where r now represents the Spearman correlation [9]. The Z_{spear} values are again stored in a `Z` object as before. Similar tests for each of the m correlation values are then performed analogous to the Pearson case.

Kendall correlation

Lastly, the extension provides the ability to compute Kendall pairwise correlation values. The direct computation of the Kendall's tau correlation is $O(n^2)$ in complexity which becomes extremely slow for any reasonably large value of n. An improved $O(n \log n)$ algorithm was implemented in C by Simcha [37] and that is used in our extension to compute the $\binom{P}{2}$ Kendall correlations. These Kendall tau values are converted to approximate standard normal deviates by dividing by $\sqrt{\frac{2(2N+5)}{9N(N-1)}}$ [36]. The resulting z-scores are then tested as in the Pearson and Spearman cases.

Computation of P-values

Since all the test statistics are approximately standard normal, P-values can be determined by looking at the tail probabilities of the $N(0, 1)$ distribution. This can be done for each of the m values stored in the variable `numCorrs` below using a linear transformation of the complementary error function `erfc` that is built into C++ in conjunction with `fabs`, which denotes the absolute value of a floating point argument. Here, column 0 corresponds to the P-values with the other two columns of `Z` denoting the corresponding column and row of data.

Listing 5.2: mcorr_getPVals

```
void mcorr::mcorr_getPVals(){
  for(int i = 0; i < mcorr::numCorrs; i++){
    //Two tailed P-value from Z test
    mcorr::pVals[i][0] = erfc(fabs(mcorr::Z[i][0]) / sqrt(2));
    mcorr::pVals[i][1] = mcorr::Z[i][1];
    mcorr::pVals[i][2] = mcorr::Z[i][2];
  }
}
```

The tests returned corresponding P-values $p_1, \ldots, p_m$. To control the experiment-wise error rate, we used the Benjamini/Hochberg/Yekutieli (BHY) algorithm [2] as described in Algorithm 1 below with the significance level, α, chosen accordingly by the user.

Algorithm 1 Benjamini/Hochberg/Yekutieli FDR control method

```
Arrange $p_1, \ldots, p_m$ in numerically ascending order as $p_{(1)} \leq \cdots \leq p_{(m)}$
$q = \alpha / \sum_{j=1}^{m} \frac{1}{j}$
$k = \max\left\{1 \leq i \leq m : p_{(i)} \leq q * (i/m)\right\}$
**if** $k$ exists **then**
    Reject the null hypotheses corresponding to $p_{(1)}, \ldots, p_{(k)}$
**else**
    Reject nothing
**end if**
```

Results

In this section, we will outline the results of testing all three of our Smoking Gun generators in addition to a Mersenne Twister generator and a `MRG32k3a` generator using the three pairwise correlation methods and the BHY algorithm. The output from the extension will be given and then will be followed by a brief summary. This output was designed to match up with the output given by the `TestU01` suite.

Performance with the problematic generators

Each of the three Smoking Gun generators produced this same output.

Listing 5.3: Smoking Gun Pairwise Correlation output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx


-----------------------------------------------
PairCorr test for Pearson Correlations:
-----------------------------------------------
   n =  10000,  p = 10


-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Pearson Correlations:
Alpha = 0.01

Reject the null hypotheses of nonzero correlation corresponding to
P-value_(1):  Vector 6 and Vector 7
P-value_(2):  Vector 3 and Vector 4
P-value_(3):  Vector 4 and Vector 5
P-value_(4):  Vector 5 and Vector 6
P-value_(5):  Vector 7 and Vector 8
P-value_(6):  Vector 9 and Vector 10
P-value_(7):  Vector 1 and Vector 2
P-value_(8):  Vector 2 and Vector 3
P-value_(9):  Vector 8 and Vector 9

-----------------------------------------------
```

```
PairCorr test for Spearman Correlations:
-----------------------------------------------
   n =  10000,  p = 10

-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Spearman Correlations:
Alpha = 0.01

Reject the null hypotheses of nonzero correlation corresponding to
P-value_(1):  Vector 6 and Vector 7
P-value_(2):  Vector 3 and Vector 4
P-value_(3):  Vector 4 and Vector 5
P-value_(4):  Vector 5 and Vector 6
P-value_(5):  Vector 7 and Vector 8
P-value_(6):  Vector 9 and Vector 10
P-value_(7):  Vector 1 and Vector 2
P-value_(8):  Vector 2 and Vector 3
P-value_(9):  Vector 8 and Vector 9

-----------------------------------------------
PairCorr test for Kendall Correlations:
-----------------------------------------------
   n =  10000,  p = 10

-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Kendall Correlations:
Alpha = 0.01

Reject the null hypotheses of nonzero correlation corresponding to
P-value_(1):  Vector 6 and Vector 7
P-value_(2):  Vector 3 and Vector 4
```

```
P-value_(3):  Vector 4 and Vector 5
P-value_(4):  Vector 5 and Vector 6
P-value_(5):  Vector 7 and Vector 8
P-value_(6):  Vector 9 and Vector 10
P-value_(7):  Vector 1 and Vector 2
P-value_(8):  Vector 2 and Vector 3
P-value_(9):  Vector 8 and Vector 9
```

Each of the three correlation based test procedures resulted in rejection of H_0 for each of $p_{(1)}, \ldots, p_{(9)}$ corresponding to the nine tests for nonzero correlation between vector 1 and vector 2, vector 2 and vector 3, . . . , vector 9 and vector 10. This agrees with our intuition about the generated data for the first Smoking Gun generator since the largest correlation (about 0.5) occurs between successive pairs of the original normally distributed data. For the transformed data that provides the actual streams, each of the corresponding P-values for testing nonzero correlation was essentially 0 and the sample correlations are close to 0.48. The built-in correlations in the second and third Smoking Guns are also easily detected by the pairwise methods here.

Mersenne Twister and MRG32k3a

As a further check on our algorithms, we tested two known “good" generators that we discussed earlier in this document. For both the Mersenne Twister and `MRG32k3a` we obtained

Listing 5.4: Mersenne Twister and MRG32k3a Pairwise Correlation output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                Starting Multivariate Extension
                Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



---------------------------------------------
PairCorr test for Pearson Correlations:
---------------------------------------------
   n =  10000,  p = 10
```

```
-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Pearson Correlations:
Alpha = 0.01

Reject none of the null hypotheses



-----------------------------------------------
PairCorr test for Spearman Correlations:
-----------------------------------------------
   n =  10000,  p = 10



-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Spearman Correlations:
Alpha = 0.01

Reject none of the null hypotheses



-----------------------------------------------
PairCorr test for Kendall Correlations:
-----------------------------------------------
   n =  10000,  p = 10



-----------------------------------------------
Test results using Benjamini/Hochberg/Yekutieli
for Kendall Correlations:
```

```
Alpha = 0.01

Reject none of the null hypotheses.
```

5.2 Testing for an Identity Correlation Matrix

We now describe an alternative test statistic that can be used to test for pairwise dependence in streams of uniform pseudorandom numbers that have been generated in parallel. We begin by converting the uniform deviates to standard normal deviates by using the inverse of the cumulative distribution function of the $N(0,1)$ distribution [3]. We then assume $X_1, \ldots, X_n$ are independent $N_p(\mathbf{0}, \Sigma)$ random vectors with joint density $f(\cdot, \theta)$ for θ, a parameter vector in the parameter set Θ. In this case, Θ is the $\frac{p(p+1)}{2}$-dimensional space of variances and covariances such that $\Sigma = ((\sigma_{ij}))$ is positive definite and $\theta' = (\sigma_{11}, \ldots, \sigma_{1p}, \sigma_{21}, \ldots, \sigma_{2p}, \ldots, \sigma_{p-1,p}, \sigma_{pp})$.

We want to test that $\theta \equiv vec(\Sigma) = vec(I_p)$ versus the alternative $\theta \neq vec(I_p)$, where $vec(\cdot)$ produces a column vector created by stacking the columns of its matrix argument on top of one another. The first step is to derive the appropriate likelihood function $L(\theta)$. For this purpose, suppose we observe $X_1 = x_1, \ldots, X_n = x_n$ and let $A = \sum_{i=1}^{n} x_i x_i^T$.

The sample likelihood now takes the form

$$\begin{aligned}
L(\theta) &= f(x_1, \ldots, x_n; \theta) \\
&= \prod_{i=1}^{n} f(x_i; \theta) \\
&= \prod_{i=1}^{n} \left[(2\pi)^{-p/2} (\det \Sigma)^{-1/2} \exp\left\{ -\frac{1}{2} x_i^T \Sigma^{-1} x_i \right\} \right] \\
&= (2\pi)^{-np/2} (\det \Sigma)^{-n/2} \exp\left\{ \sum_{i=1}^{n} -\frac{1}{2} x_i^T \Sigma^{-1} x_i \right\} \\
&= (2\pi)^{-np/2} (\det \Sigma)^{-n/2} \exp\left\{ \sum_{i=1}^{n} -\frac{1}{2} \operatorname{tr}\left[x_i^T \Sigma^{-1} x_i \right] \right\} \\
&\qquad \left[\text{since } \left(x_i^T \Sigma^{-1} x_i \right) \text{ is a } 1 \times 1\right] \\
&= (2\pi)^{-np/2} (\det \Sigma)^{-n/2} \exp\left\{ \sum_{i=1}^{n} \operatorname{tr}\left[-\frac{1}{2} x_i^T \Sigma^{-1} x_i \right] \right\} \\
&\qquad [\text{since } \operatorname{tr}(cA) = c \cdot \operatorname{tr}(A)]
\end{aligned}$$

Continuing to simplify we have

$$\begin{aligned}
L(\theta) &= (2\pi)^{-np/2}(\det\Sigma)^{-n/2}\exp\left\{\sum_{i=1}^{n}\operatorname{tr}\left[-\frac{1}{2}\Sigma^{-1}x_ix_i^T\right]\right\} \\
&\qquad [\text{since } \operatorname{tr}(ABC)=\operatorname{tr}(BCA)] \\
&= (2\pi)^{-np/2}(\det\Sigma)^{-n/2}\exp\left\{\operatorname{tr}\left(\sum_{i=1}^{n}\left[-\frac{1}{2}\Sigma^{-1}x_ix_i^T\right]\right)\right\} \\
&\qquad [\text{since } \operatorname{tr}(A+B)=\operatorname{tr}(A)+\operatorname{tr}(B)] \\
&= (2\pi)^{-np/2}(\det\Sigma)^{-n/2}\exp\left\{\operatorname{tr}\left(-\frac{1}{2}\Sigma^{-1}\sum_{i=1}^{n}\left[x_i^Tx_i\right]\right)\right\} \\
&\qquad \left[\text{since } -\frac{1}{2}\Sigma^{-1} \text{ does not depend on } i\right] \\
&= (2\pi)^{-np/2}(\det\Sigma)^{-n/2}\exp\left\{\operatorname{tr}\left(-\frac{1}{2}\Sigma^{-1}A\right)\right\} \\
&\qquad [\text{by substitution}] \\
&= (2\pi)^{-np/2}(\det\Sigma)^{-n/2}\operatorname{etr}\left(-\frac{1}{2}\Sigma^{-1}A\right) \\
&\qquad [\text{by defining } \operatorname{etr}(\cdot)=\exp\{\operatorname{tr}(\cdot)\}].
\end{aligned}$$

From this we obtain Λ, the likelihood ratio statistic for testing our hypothesis, as

$$\begin{aligned}
\Lambda &= \frac{\sup\limits_{\theta\in\Theta_0} L(\theta)}{\sup\limits_{\theta\in\Theta} L(\theta)} = \frac{\sup\limits_{\Sigma=I_p} L(\Sigma)}{\sup\limits_{\Sigma} L(\Sigma)} \\
&= \frac{L(I_p)}{L(\hat\Sigma)} \text{ where } \hat\Sigma = n^{-1}A \text{ [By Theorem 3.1.5 of [34]]}
\end{aligned}$$

with

$$\begin{aligned}
L(I_p) &= (2\pi)^{-np/2}(\det I_p)^{-n/2}\operatorname{etr}\left(-\frac{1}{2}I_p^{-1}A\right) \\
&= (2\pi)^{-np/2}(1)^{-n/2}\operatorname{etr}\left(-\frac{1}{2}I_pA\right) \\
&= (2\pi)^{-np/2}\operatorname{etr}\left(-\frac{1}{2}A\right)
\end{aligned}$$

and

$$\begin{aligned}
L(\hat{\Sigma}) &= (2\pi)^{-np/2}(\det\left[\frac{1}{n}A\right])^{-n/2}\operatorname{etr}\left(-\frac{1}{2}\left[\frac{1}{n}A\right]^{-1}A\right) \\
&= (2\pi)^{-np/2}\left(\left[\frac{1}{n}\right]^{p}\det A\right)^{-n/2}\operatorname{etr}\left(-\frac{n}{2}A^{-1}A\right) \\
&\qquad\qquad [\text{since } A \text{ is } p\times p \text{ and } \det(c\cdot A_{p\times p}) = c^{p}\det(A)] \\
&= (2\pi)^{-np/2}\left[\frac{1}{n}\right]^{-np/2}[\det A]^{-n/2}\exp\left(-\frac{n}{2}\operatorname{tr} I_p\right) \\
&= (2\pi)^{-np/2}\left[\frac{1}{n}\right]^{-np/2}e^{-np/2}[\det A]^{-n/2} \\
&= (2\pi)^{-np/2}\left[\frac{e}{n}\right]^{-np/2}[\det A]^{-n/2}.
\end{aligned}$$

After some simplification the expression for Λ reduces to

$$\begin{aligned}
\Lambda &= \frac{(2\pi)^{-np/2}\operatorname{etr}\left(-\frac{1}{2}A\right)}{(2\pi)^{-np/2}\left[\frac{e}{n}\right]^{-np/2}[\det A]^{-n/2}} = \frac{\operatorname{etr}\left(-\frac{1}{2}A\right)}{\left[\frac{e}{n}\right]^{-np/2}[\det A]^{-n/2}} \\
&= \left[\frac{e}{n}\right]^{np/2}\operatorname{etr}\left(-\frac{1}{2}A\right)[\det A]^{n/2}.
\end{aligned}$$

From pages 219-220 of [14], the identity matrix hypothesis is rejected if $\Lambda < c_\alpha$ with α chosen appropriately for a size-α test. In practice, we know that

$$-2\ln\Lambda \xrightarrow{d} \chi^2_{\frac{p(p+1)}{2}},$$

with "$\xrightarrow{d}$" indicating convergence in distribution. Thus, for large n, critical values can be obtained from the chi-square distribution. Some algebra shows that $-2\ln\Lambda$ can be simplified to the expression that was used in our code: namely,

$$\begin{aligned}
\ln\Lambda &= \ln\left[\left(\frac{e}{n}\right)^{np/2}\right] + \ln\left[\exp\left\{\operatorname{tr}\left(-\frac{1}{2}A\right)\right\}\right] + \ln\left[(\det A)^{n/2}\right] \\
&= \frac{np}{2}\ln\left(\frac{e}{n}\right) + \operatorname{tr}\left(-\frac{1}{2}A\right)\ln e + \frac{n}{2}\ln(\det A) \\
&= \frac{np}{2}(\ln e - \ln n) + \operatorname{tr}\left(-\frac{1}{2}A\right) + \frac{n}{2}\ln(\det A)
\end{aligned}$$

Letting $C = \hat{\Sigma} = n^{-1}A$ and, thus, $A = nC$ we have

$$\begin{aligned}
\ln \Lambda &= \frac{np}{2}(1 - \ln n) + \operatorname{tr}\left(-\frac{n}{2}C\right) + \frac{n}{2}\ln(\det\{nC\}) \\
&= \frac{np}{2}(1 - \ln n) - \frac{n}{2}\operatorname{tr}(C) + \frac{n}{2}\ln\left[n^p \det C\right] \\
&= \frac{np}{2}(1 - \ln n) - \frac{n}{2}\operatorname{tr}(C) + \frac{n}{2}\left[p\ln n + \ln(\det C)\right] \\
&= \frac{np}{2} - \frac{np}{2}\ln n - \frac{n}{2}\operatorname{tr}(C) + \frac{np}{2}\ln n + \frac{n}{2}\ln(\det C) \\
&= \frac{np}{2} - \frac{n}{2}\operatorname{tr}(C) + \frac{n}{2}\ln(\det C)
\end{aligned}$$

or

$$-2\ln\Lambda = -np + n\operatorname{tr}(C) - n\ln(\det C) = n[\operatorname{tr}(C) - \ln(\det C) - p].$$

P-values are then calculated using the cumulative distribution function of the χ^2 distribution. A specified significance level of 0.01 was used below in the results.

Results

Similar to the previous section, we now provide the results of the likelihood ratio test on the three Smoking Gun, Mersenne Twister, and `MRG32k3a` generators.

Performance with the problematic generators

The first set of results are for the Smoking Gun based on a normally transformed VMA(1) process.

Listing 5.5: First Smoking Gun LRT output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                Starting Multivariate Extension
                Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
---------------------------------------------
Likelihood Ratio Test for
     Pairwise Correlation Matrix = Identity:
Alpha = 0.01


---------------------------------------------
   n =  10000,  p = 10
```

```
-----------------------------------------------
LR Test Statistic                   :  43779.6
p-value of test                     :  0
```

The next set of results are for the Smoking Gun based on a univariate time series moving across the processors.

Listing 5.6: Second Smoking Gun LRT output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----------------------------------------------
Likelihood Ratio Test for
     Pairwise Correlation Matrix = Identity:
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10

-----------------------------------------------
LR Test Statistic                   :  7636.69
p-value of test                     :  0
```

Finally results for the Smoking Gun based on a bivariate time series moving across the processors are given below.

Listing 5.7: Third Smoking Gun LRT output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----------------------------------------------
```

```
Likelihood Ratio Test for
     Pairwise Correlation Matrix = Identity:
Alpha = 0.01


-----------------------------------------------
   n =  10000,  p = 10


-----------------------------------------------
LR Test Statistic                  :  6651.46
p-value of test                    :  0
```

Note that each of the P-values are essentially zero. Consequently, the test results provide strong evidence that the vectors created by the three Smoking Gun generators are correlated and, thus, that dependence exists among the streams.

Mersenne Twister and MRG32k3a

The results for the Mersenne Twister generator are given below.

Listing 5.8: Mersenne Twister LRT output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----------------------------------------------
Likelihood Ratio Test for
     Pairwise Correlation Matrix = Identity:
Alpha = 0.01


-----------------------------------------------
   n =  10000,  p = 10


-----------------------------------------------
LR Test Statistic                  :  65.2344
p-value of test                    :  0.162607
```

The next results are for the `MRG32k3a` generator.

Listing 5.9: MRG32k3a LRT output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----------------------------------------------
Likelihood Ratio Test for
     Pairwise Correlation Matrix = Identity:
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10

-----------------------------------------------
LR Test Statistic                    :  49.5653
p-value of test                      :  0.681589
```

Thus, both of the generators “pass" the likelihood ratio test.

Chapter 6

TESTU01 MULTIVARIATE EXTENSION - VECTOR TIME SERIES BASED TESTS

Vector time series techniques provide a rich collection of tools for modeling the evolution of vector-valued random variables as a function of any type of discrete index. Tests for white noise that arise in this context can be used for detection of between stream dependence when they are applied to our particular setting. In this section we will focus on the following multivariate portmanteau tests for white noise: the Hosking, the Li-McLeod, and the Mahdi-McLeod tests. The *portmanteau* label signifies that the tests are all purpose in nature, and, at least in this chapter, consistent against all alternatives.

We define a more general version of the VMA(1) process that was mentioned in Section 4.2. $\{X_t : t = 1, \ldots, n\}$, a p-dimensional autoregressive moving average (VARMA) process of order (γ, ω), is defined as

$$\sum_{\ell=0}^{\gamma} \Phi_\ell X_{t-\ell} = \sum_{\ell=0}^{\omega} \Xi_\ell e_{t-\ell}. \tag{6.1}$$

Here the Φ_ℓ, $\ell = 0, 1, \ldots, \gamma$, are $p \times p$ matrices corresponding to potential autoregressive model terms and the Ξ_ℓ, $\ell = 0, 1, \ldots, \omega$, are $p \times p$ matrices that represent potential moving average components. We adopt the convention that Φ_0 and Ξ_0 are identity matrices and assume that $\Phi_\gamma \neq \mathbf{O}$ and $\Xi_\omega \neq \mathbf{O}$ for $\mathbf{O}$ a matrix of all zeroes. The process $\{e_t\}$ is assumed to be white-noise: i.e., the e_t are independent and identically distributed random vectors of dimension p with $\mathbb{E}(e_t) = 0$, $\mathbb{E}(e_t e_t^T) = \Sigma = ((\sigma_{ij}))$, and $Cov(e_t, e_{t-\ell}) = 0$ for $\ell = 1, \ldots, m$ where m is chosen large enough to cover all lags of interest. Note that our earlier VMA(1) process can also be defined as a VARMA(0, 1) process.

For specificity, we are now thinking of the X_t as representing the vector of pseudorandom numbers produced by the p processes at the tth step of the generation exercise. Given a matrix of $n \times p$ deviates, we first mean-corrected each of the p columns of data so that each could be viewed as the residuals from a white noise model "fit." Then we applied the tests from this chapter to these residuals. The portmanteau nature of the test statistics gives them power against any alternatives that have a VARMA form. At least approximately this includes all covariance stationary processes.

The expression (6.1) can be written as

$$\Phi(B)X_t = \Xi(B)e_t, \tag{6.2}$$

where the backward shift operator B is defined by $BX_t = X_{t-1}$. $\Phi(B) = \sum_{\ell=0}^{\gamma} \Phi_\ell B^\ell$ and $\Xi(B) = \sum_{\ell=0}^{\omega} \Xi_\ell B^\ell$. The residuals $\hat{e}_t$ from a fitted model (e.g., from maximum likelihood) are given by

$$\hat{\Phi}(B)X_t = \hat{\Xi}(B)\hat{e}_t,$$

where $\hat{\Phi}$ and $\hat{\Xi}$ are the least squares (or asymptotically equivalent) estimators of Φ and Ξ, respectively. These estimators are asymptotically normally distributed with variances $O(n^{-1})$ [6].

Define the ℓth white-noise autocovariance matrix C_ℓ as

$$C_\ell = ((c_{ij\ell})) = n^{-1} \sum_t e_t e_{t-\ell}^T$$

and the ℓth residual autocovariance matrix

$$\hat{C}_\ell = ((\hat{c}_{ij\ell})) = n^{-1} \sum_t \hat{e}_t \hat{e}_{t-\ell}^T.$$

The least squares estimator of Σ is $\hat{\Sigma} = \hat{C}_0$ and Dunsmuir and Hannan [6] show that $\hat{\Sigma}$ is asymptotically normally distributed. Also, $\hat{\Sigma}$ is asymptotically uncorrelated with the other least-squares parameter estimators. Using these results, C_ℓ is seen to be asymptotically normally distributed [13] with $\mathbb{E}(C_\ell) = \Sigma\delta_{0\ell}$ and asymptotically

$$Cov(c_{ij\ell}, c_{kl\ell'}) = n^{-1}\sigma_{ik}\sigma_{jl}\delta_{\ell,\ell'}, \tag{6.3}$$

where $\delta_{\ell,\ell'}$ is the Kronecker delta function that is one when $\ell = \ell'$ and zero otherwise. We will use this fact along with an approximate linear relationship between C_ℓ and $\hat{C}_\ell$ given below to deduce the asymptotic distribution of the $\hat{C}_\ell$ and $vec\hat{C}$.

For some integer s, define $C = (C_1, C_2, \ldots, C_s)$, $\hat{C} = (\hat{C}_1, \hat{C}_2, \ldots, \hat{C}_s)$, and let $\Phi = (\Phi_1, \Phi_2, \ldots, \Phi_\gamma)$, $\Xi = (\Xi_1, \Xi_2, \ldots, \Xi_\omega)$, and $\Lambda = (\Phi, \Xi)$. We also use $vecC$ to denote the vector formed by stacking the columns of C one on top of each other (similar to the parallel filter option in `TestU01` given in Section 3.2). Thus,

$$vecC = (c_{111}, c_{211}, \ldots, c_{121}, \ldots, c_{112}, \ldots, c_{pps})^T.$$

We now define the two matrix power series $\Pi(z)$ and $\Psi(z)$ as

$$\Pi(z) = \{\Phi(z)\}^{-1} = \sum_{\ell=0}^{\infty} \Pi_\ell z^\ell$$

and

$$\Psi(z) = \{\Phi(z)\}^{-1}\Xi(z) = \sum_{\ell=0}^{\infty} \Psi_\ell z^\ell$$

with the elements of Π_ℓ and Ψ_ℓ converging exponentially to zero as $\ell \to \infty$. A linear expansion of the residuals $\hat{e}_t$ can be made using Taylor's theorem via

$$\hat{e}_t = e_t + \sum_{\ell=1}^{\gamma}\sum_{u=0}^{\infty} \Pi_u(\hat{\Phi}_\ell - \Phi_\ell)X_{t-\ell-u} - \sum_{\ell=1}^{\omega}\sum_{u=0}^{\infty} \Pi_u(\hat{\Xi}_\ell - \Xi_r)e_{t-\ell-u} + O_p(n^{-1}), \qquad (6.4)$$

with O_p corresponding to the order in probability. This result can be used to show that for $v > 0$,

$$\hat{C}_v = C_v + \sum_{\ell=1}^{\gamma}\sum_{u=0}^{\infty} \Pi_u(\hat{\Phi}_\ell - \Phi_\ell)\Psi_{v-\ell-u}\Sigma - \sum_{\ell=1}^{\omega} \Pi_{v-\ell}(\hat{\Xi}_\ell - \Xi_\ell)\Sigma + O_p(n^{-1}).$$

Using vec notation, (6.4) can be expressed as

$$vec\hat{E} = vecE + L\,vec(\hat{\Lambda} - \Lambda),$$

where E represents the $p \times n$ matrix whose (i,t)th entry is e_{it}. To define L let J_ℓ and K_ℓ be $p\gamma \times n$ and $p\omega \times n$ matrices whose (i,t)th p subvectors are $X_{t-i-\ell}$ and $e_{t-i-\ell}$, respectively. If we now define $M = \sum_{\ell=0}^{\infty}(J_\ell^T \otimes \Pi_\ell)$ and $N = \sum_{\ell=0}^{\infty}(K_\ell^T \otimes \Pi_\ell)$ with $\otimes$ denoting the Kronecker product, L is the $pn \times p^2(\gamma + \omega)$ matrix $(M - N)$.

Since the e_t are assumed independent and identically distributed with mean zero and covariance Σ, $vecE$ has mean zero and covariance $I_n \otimes \Sigma$ [13]. We can also use the $vec\hat{C}_v$ notation to write $\hat{C}_v$ as

$$vec\hat{C}_v = vecC_v + \sum_{\ell=1}^{\gamma} G_{v-\ell}vec(\hat{\Phi}_\ell - \Phi_\ell) - \sum_{\ell=1}^{\omega} D_{v-\ell}vec(\hat{\Xi}_r - \Xi_\ell),$$

where $G_\ell = \sum_{u=0}^{\infty}(\Sigma\Psi_u^T \otimes \Pi_{\ell-u})$, $D_\ell = \Sigma \otimes \Pi_\ell$, and $G_0 = D_0 = \Sigma \otimes I_p$. Thus,

$$vec\hat{C} = vecC + V vec(\hat{\Phi}_\ell - \Phi_\ell) - Y vec(\hat{\Xi}_\ell - \Xi_\ell)$$

where

$$V = \begin{pmatrix} G_0 & \mathbf{O} & \mathbf{O} & \dots & \mathbf{O} \\ G_1 & G_0 & \mathbf{O} & \dots & \mathbf{O} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ G_{s-1} & G_{s-2} & G_{s-3} & \dots & G_{s-\gamma} \end{pmatrix}$$

and

$$Y = \begin{pmatrix} D_0 & \mathbf{O} & \mathbf{O} & \dots & \mathbf{O} \\ D_1 & D_0 & \mathbf{O} & \dots & \mathbf{O} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ D_{s-1} & D_{s-2} & D_{s-3} & \dots & D_{s-\omega} \end{pmatrix}.$$

Define $W = I_s \otimes \Sigma \otimes \Sigma$ and $Q = Z(Z^T W^{-1} Z)^{-1} Z^T W^{-1}$. Then Hosking [13] proves that

$$vec\hat{C} = (I_{p^2 s} - Q) vecC + O_p(n^{-1}). \tag{6.5}$$

From (6.3), asymptotically $vecC \sim N_{p^2 s}(0, n^{-1} W)$. Using these results, asymptotically we have $vec\hat{C} \sim N_{p^2 s}(0, n^{-1}(I_{p^2 s} - Q)W)$.

If a random vector $X \sim N(0, Q\Sigma)$ where Q is idempotent of rank r and Σ is positive-definite, then $X^T \Sigma^{-1} X \sim \chi^2_r$. In our current setting, $I - Q$ is idempotent of rank $p^2(s - \gamma - \omega)$ and W is positive-definite and symmetric. It therefore follows that

$$n(vec\hat{C})^T W^{-1} vec\hat{C} \sim \chi^2_{p^2(s-\gamma-\omega)}. \tag{6.6}$$

The result continues to hold if W^{-1} is replaced by a consistent estimator such as

$$\hat{W}^{-1} = I_s \otimes \hat{\Sigma}^{-1} \otimes \hat{\Sigma}^{-1} = I_s \otimes \hat{C}_0^{-1} \otimes \hat{C}_0^{-1}.$$

Hence, the portmanteau statistic P for a white noise null model (i.e., Φ and Ξ are identity operators) can be expressed approximately as

$$P = n(vec\hat{C})^T \left(I_s \otimes \hat{C}_0^{-1} \otimes \hat{C}_0^{-1} \right) vec\hat{C}.$$

Consequently, P is asympotically chi-square with degrees of freedom $p^2(s - \gamma - \omega)$.

6.1 Hosking Portmanteau Test Statistic

Hosking [13] provides some convenient forms for computation of P including

$$P = n\sum_{\ell=1}^{s} \mathrm{tr}(\hat{C}_\ell^T \hat{C}_0^{-1} \hat{C}_\ell \hat{C}_0^{-1}).$$

We can further simplify this form for P by writing the Cholesky factorization of C_0^{-1} as LL^T for L a lower triangular matrix and by defining

$$\hat{R}_\ell = L^T \hat{C}_\ell L. \tag{6.7}$$

Let $\hat{r}_\ell = vec\hat{R}_\ell^T$ be the $1 \times p^2$ row vector with the rows of $\hat{R}_\ell$ laid successively one after another. Then, Mahdi and McLeod [24] provide a simplified formula

$$P = n\sum_{\ell=1}^{m} \hat{r}_\ell(\hat{C}_0^{-1} \otimes \hat{C}_0^{-1})\hat{r}_\ell^T$$

and give the following modified version of P that is commonly referred to as the Hosking test statistic

$$P_{H(mod)} = n^2 \sum_{\ell=1}^{m} (n-\ell)^{-1} \hat{r}_\ell(\hat{C}_0^{-1} \otimes \hat{C}_0^{-1})\hat{r}_\ell^T. \tag{6.8}$$

$P_{H(mod)}$ has a large sample chi-square distribution with $p^2(m-\gamma-\omega)$ degrees of freedom. This modified portmanteau statistic is expected to have a small-sample distribution more nearly $\chi^2_{p^2(m-\gamma-\omega)}$ than that of P [13]. Recall that m denotes the size of the largest order to be tested and is specified by the user of the extension. The $P_{H(mod)}$ test statistic is the first of the portmanteau statistics that will be used in our extension to `TestU01` called `mport`.

6.2 Li-McLeod Portmanteau Test Statistic

Another modification of P that was suggested by Li and McLeod [23] is the second portmanteau statistic in `mport`. This test statistic and its asymptotic distribution are

$$Q_{LM} = \frac{p^2 m(m+1)}{2n} + n\sum_{\ell=1}^{m} \hat{r}_\ell(\hat{C}_0^{-1} \otimes \hat{C}_0^{-1})\hat{r}_\ell^T \sim \chi^2_{p^2(m-\gamma-\omega)}. \tag{6.9}$$

This modification was done so that the expected value of the statistic Q_{LM} under the null hypothesis was equal to the degrees of freedom $p^2(m-\gamma-\omega) + O_p(1/n)$.

6.3 Mahdi-McLeod Portmanteau Test Statistic

Mahdi and McLeod [24] proposed a multivariate portmanteau test statistic based on Hosking's original results and it is the third portmanteau statistic in our extension. The test statistic depends on a matrix of matrices composed of the matrices $\hat{R}_\ell$, $\ell = 1, \ldots, m$, and is given by

$$\hat{\boldsymbol{\mathcal{T}}}_m = \begin{pmatrix} \hat{R}_0 & \hat{R}_1 & \ldots & \hat{R}_m \\ \hat{R}_1^T & \hat{R}_0 & \ldots & \hat{R}_{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{R}_m^T & \hat{R}_{m-1}^T & \ldots & \hat{R}_0 \end{pmatrix}$$

with $\hat{R}_0 = I_p$. We then define

$$A_m = \frac{-3n}{2m+1} \log |\hat{\boldsymbol{\mathcal{T}}}_m|. \tag{6.10}$$

If there are no significant autocorrelations in the residuals, $\hat{R}_\ell = O_p(n^{-1/2})$ and $\hat{\boldsymbol{\mathcal{T}}}_m$ is approximately block diagonal with $|\hat{\boldsymbol{\mathcal{T}}}_m| \approx 1$. When there is autocorrelation present, $|\hat{\boldsymbol{\mathcal{T}}}_m|$ will be smaller than 1 so that we reject the white noise model for large values of A_m. Under the null model, the Mahdi-McLeod statistic A_m is approximately chi-square distributed with degrees of freedom $p^2 \left(\frac{1.5m(m+1)}{2m+1} - \gamma - \omega \right)$ [24].

6.4 Results

Performance with the problematic generators

Testing of the first Smoking Gun generator with the white noise portmanteau tests produced the following output.

Listing 6.1: First Smoking Gun Portmanteau Tests output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx




---------------------------------------------
Portmanteau Test for White Noise (Hosking):
```

```
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10


-----------------------------------------------
Hosking Test Statistic              :  1001.58
Degrees of Freedom                  :  1000
p-value of test                     :  0.479951


-----------------------------------------------
Portmanteau Test for White Noise (Li-McLeod):
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10


-----------------------------------------------
Li-McLeod Test Statistic            :  1001.59
Degrees of Freedom                  :  1000
p-value of test                     :  0.479892


-----------------------------------------------
Portmanteau Test for White Noise (Mahdi-McLeod):
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10
```

```
----------------------------------------------
Determinant of Toeplitz Block matrix:  0.570518
Mahdi-McLeod Test Statistic         :  801.73
Degrees of Freedom                  :  785.714
p-value of test                     :  0.337942
```

We see that, contrary to our initial expectations, the first Smoking Gun was able to pass the portmanteau tests. The likely reason for this is that the number of rows n or length of the vectors in the series is much larger than the number of columns (processors) p. Thus, the time series here is actually quite short as it evolves over only ten columns.

The second and third Smoking Guns have lengthy built-in time series. These generators were able to pass `SmallCrushFile` and `CrushFile`. The results that follow are for the second Smoking Gun generator based on a univariate time series.

Listing 6.2: Second Smoking Gun Portmanteau Tests output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



----------------------------------------------
Portmanteau Test for White Noise (Hosking):
Alpha = 0.01


----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Hosking Test Statistic              :  1726.93
Degrees of Freedom                  :  1000
```

```
p-value of test                  :  0



-----------------------------------------------
Portmanteau Test for White Noise (Li-McLeod):
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



-----------------------------------------------
Li-McLeod Test Statistic         :  1726.86
Degrees of Freedom               :  1000
p-value of test                  :  0



-----------------------------------------------
Portmanteau Test for White Noise (Mahdi-McLeod):
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



-----------------------------------------------
Determinant of Toeplitz Block matrix:  0.270533
Mahdi-McLeod Test Statistic      :  1867.66
Degrees of Freedom               :  785.714
p-value of test                  :  0
```

The results for the third Smoking Gun follow. They are very similar to those of the second Smoking Gun.

Listing 6.3: Third Smoking Gun Portmanteau Tests output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



----------------------------------------------
Portmanteau Test for White Noise (Hosking):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Hosking Test Statistic              :  2612.72
Degrees of Freedom                  :  1000
p-value of test                     :  0



----------------------------------------------
Portmanteau Test for White Noise (Li-McLeod):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Li-McLeod Test Statistic            :  2612.57
Degrees of Freedom                  :  1000
p-value of test                     :  0
```

```
----------------------------------------------
Portmanteau Test for White Noise (Mahdi-McLeod):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Determinant of Toeplitz Block matrix:  0.104578
Mahdi-McLeod Test Statistic         :  3225.45
Degrees of Freedom                  :  785.714
p-value of test                     :  0
```

Thus, the portmanteau tests easily detect the built-in dependence in the second and third Smoking Gun generators. All of the P-values are near zero which leads us to reject the null hypothesis of white noise.

Mersenne Twister and MRG32k3a

As with the correlation tests in the previous chapter, we also tested the Mersenne Twister and `MRG32k3a` generators with the results reported below. The output immediately following is for the Mersenne Twister generator.

Listing 6.4: Mersenne Twister Portmanteau Tests output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



----------------------------------------------
Portmanteau Test for White Noise (Hosking):
```

```
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Hosking Test Statistic              :  922.114
Degrees of Freedom                  :  1000
p-value of test                     :  0.961965



----------------------------------------------
Portmanteau Test for White Noise (Li-McLeod):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Li-McLeod Test Statistic            :  922.152
Degrees of Freedom                  :  1000
p-value of test                     :  0.961891



----------------------------------------------
Portmanteau Test for White Noise (Mahdi-McLeod):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10
```

```
-----------------------------------------------
Determinant of Toeplitz Block matrix:  0.603593
Mahdi-McLeod Test Statistic         :  721.221
Degrees of Freedom                  :  785.714
p-value of test                     :  0.951215
```

Similar results for the `MRG32k3a` generator follow.

Listing 6.5: MRG32k3a Portmanteau Tests output

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 Starting Multivariate Extension
                 Version: TestU01 1.2.3
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



-----------------------------------------------
Portmanteau Test for White Noise (Hosking):
Alpha = 0.01

-----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



-----------------------------------------------
Hosking Test Statistic              :  961.259
Degrees of Freedom                  :  1000
p-value of test                     :  0.805864



-----------------------------------------------
Portmanteau Test for White Noise (Li-McLeod):
Alpha = 0.01
```

```
----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Li-McLeod Test Statistic            :  961.255
Degrees of Freedom                  :  1000
p-value of test                     :  0.805895



----------------------------------------------
Portmanteau Test for White Noise (Mahdi-McLeod):
Alpha = 0.01

----------------------------------------------
   n =  10000,  p = 10,  Lag order = 10



----------------------------------------------
Determinant of Toeplitz Block matrix:  0.602564
Mahdi-McLeod Test Statistic         :  723.658
Degrees of Freedom                  :  785.714
p-value of test                     :  0.944278
```

Thus, none of our time series based tests detects a departure from the white noise model in the data produced by the Mersenne Twister or `MRG32k3a`.

Chapter 7

DESIGN CONSIDERATIONS FOR PARALLEL PROCESSING

The computations performed by `TestU01` are conducted exclusively in serial mode. This is satisfactory in the sense that parallel number generation schemes can still be evaluated provided that the generators are paired with the package appropriately. However, the pairing mechanism could be made more flexible and processing time could be reduced if the computations were conducted in parallel. In this chapter we provide a simple illustration of how modern C++ code design might be employed to circumvent such shortcomings.

The problem we will focus on is that of computing the correlation matrix corresponding to p streams of uniform pseudorandom numbers. We want to be able to calculate correlations between streams that arise from a collection of possibly unrelated generators. That means, for example, that all the generators need not be from a common class. There are two aspects to consider: formulation of the problem in C++ and parallelization of the calculations. We will address each of these in turn.

7.1 Object-oriented Formulation

Conceptually, we can view the situation as one where a user provides us with some arbitrary conglomeration of pseudorandom number generators. We then reach into this "bucket" of generators and select two generation algorithms that need to be compared in terms of their linear correlation. Some structure must, of course, be imposed on how the conglomeration is packaged in order for this to actually be feasible. A rather minimal standard would be that our generator "bucket" take the form of a class/struct with the number of generators, `nRng`, as a data member and that each generator be associated with a unique integer index in the range from 0 to `nRng - 1`. This index can then be combined with an overloaded () operator to give access to the pseudorandom numbers produced by each generator as shown below.

The generators in the user's bucket should themselves be objects from a class. While all the classes need not be the same, they do need to have a common function that returns the pseudorandom uniform produced by the underlying generator. One way to accomplish this is by overloading the () operator to produce what is then typically called a functor.

For purposes of illustration we will frequently use the `RngStream` class from the RngStreams package that contains the `MRG32k3a` multiple recursive generator from previous chapters. Pseudorandom uniforms for `RngStream` objects are produced with the `RandU01` class member function. Thus, the requisite functor property is obtained in this case by simply adding the inline function

```
double operator()(){return RandU01();}
```

to the class header file `RngStream.h`.

As another illustration we have the standard Wichmann/Hill generator represented in the next listing.

Listing 7.1: Wichmann/Hill generator code

```
//WH.h
#ifndef WH_H
#define WH_H
#include <math.h>

struct WH{
  unsigned long seed[3];
  WH(){};
  WH(unsigned long Seed[3]){
    for(int i = 0; i < 3; i++) seed[i] = Seed[i];
  }

  double operator()(){
    seed[0] = (171*seed[0])%30269;
    seed[1] = (172*seed[1])%30307;
    seed[2] = (173*seed[2])%30323;
    return fmod((double)(seed[0])/30269.0
                + (double)(seed[1])/30307.0
                + (double)(seed[2])/30323.0, 1.0);
  }
};
```

```
#endif
```

This generator needs three integer seeds as class data members. These are initialized in the class constructor. As for the `RngStream` class, we have defined the () operator to return the values produced by the generation algorithm.

The code listing below shows how one might create a generator collection that includes both the Wichmann/Hill and `MRG32k3a` generators.

Listing 7.2: rngBucket.h

```
//rngBucket.h
#ifndef RNGBUCKET_H
#define RNGBUCKET_H

#include "WH.h"
#include "RngStream.h"

struct rngBucket{

  int nRngs;
  WH w;
  RngStream* pRng;
  rngBucket(int nrngs, unsigned long seed1[3],
            unsigned long seed2[6]): nRngs(nrngs) {
    w = WH(seed1);
    RngStream::SetPackageSeed(seed2);
    pRng = new RngStream[nRngs - 1];
  }

  rngBucket(const rngBucket& rng){
    nRngs = rng.nRngs;
    w = rng.w;
    pRng = new RngStream[nRngs - 1];
    for(int i = 0; i < nRngs - 1; i++)
      pRng[i] = rng.pRng[i];
  }

  double operator()(int i){
    if(i == 0) return w();
    else return pRng[i - 1]();
  }
};
#endif
```

The most important feature of our `rngBucket` class is the way we access the generators. This again employs an overloaded () operator except that now the argument is the integer index of the generator that is to be used in the calculation. Once that generator has been located, its associated pseudorandom uniform algorithm is called this time using the generator specific definition of (). In this example we used a single `WH` object along with an array of `RngStream` objects to populate the bucket. However, it is clear that any finite number of generators all of possibly different types can be bundled together in this fashion.

Note that a specific copy constructor has been included in the `rngBucket` class. This is needed to insure that the dynamic memory allocated for the array of `RngStream` objects will be handled correctly when an `rngBucket` object is passed into a function.

With the `rngBucket` class in hand, we can write a simple function to compute the inter-stream correlation such as the following.

Listing 7.3: unifCorr

```
double unifCorr(rngBucket rng, int ind1, int ind2,
                int n){
    double sum = 0.;
    for(int i = 0; i < n; i++)
      sum += (rng(ind1) - .5)*(rng(ind2) - .5);
    return 12.*sum/(double)n;
}
```

Here we have used the fact that under the uniform (null) model, the population mean and variance are 1/2 and 1/12, respectively. Note that what requires storage in the calling program is only an `rngBucket` object. The actual arrays of pseudorandom numbers produced by the generators are not retained. This entails some loss of computational efficiency when conducting pairwise comparisons of generators, for example. However, for applications with very long streams of pseudorandom numbers this is the only practical approach.

7.2 Parallelization

The next step is to speed up the computation of the correlation matrix by parallelizing the computations. One way this can be accomplished is by having each processor perform the calculations on a subset of the array elements and then combine their individual results onto the master node.

Suppose that there are $\texttt{nStream} = 2^r$ streams (i.e., pseudorandom number generators) to be analyzed and $r = b + s$ so that we can group them as $\texttt{nb} = 2^b$ blocks each of which corresponds to $\texttt{bs} = 2^s$ generators. This gives us a total of `nTasks = nb*(nb + 1)/2` correlation submatrices that must be evaluated corresponding to the pairings of all the streams in our `nb` blocks. For the "diagonal" blocks (i.e., where the two stream subsets being compared are the same) only the upper diagonal of the correlation submatrix need be evaluated. The block indexing becomes quite tedious at this point and we have found it most expedient to simply have every process create a (common) index array via the function

Listing 7.4: indexArray

```
void indexArray(int** indices, nb){
  int ind = 0;
  for(int i = 0; i < nb; i++)
    for(int j = i; j < nb; j++){
      indices[ind] = new int[2];
      indices[ind][0] = i;
      indices[ind][1] = j;
      ind += 1;
    }
}
```

This produces a two-dimensional array whose elements give the “row” and “column” indices for all the blocks of the correlation matrix. This array will have `nTasks` rows that can then be distributed to the processors. For simplicity we assume here that `nTasks` and the number of processors coincide. More generally the tasks will need to be portioned out in groups to the processors. This entails some additional code complexity that will be addressed in future work on the package extension.

The `main` function for our MPI code now takes the form that appears in the next listing. The `twoToTheK` function that is used here is a utility function that returns an integer value of two raised to the power of its integer argument.

Listing 7.5: main MPI

```
int main(int argc, char** argv){

  //number of variables
  int nStream = twoToTheK(atoi(argv[1]));
  //number of variables per block
  int bs = twoToTheK(atoi(argv[2]));
  //number of blocks
  int nb = twoToTheK(atoi(argv[3]));

  //WH seed
  unsigned long seed1[3] = {1, 2, 3};
  //RngStream seed
  unsigned long seed2[6] = {1, 2, 3, 4, 5, 6};
  //all processes initialize the same rngBucket object
  rngBucket rng(nStream, seed1, seed2);
  //sample size
  int n = atoi(argv[4]);
```

```
//number of processors = number of blocks
int nTasks = nb*(nb + 1)/2;
//all processes create the block index array
int** indices = new int*[nTasks];
indexArray(indices, nb);

MPI::Init();
int myRank = MPI::COMM_WORLD.Get_rank();

if(myRank == 0){
  //0 node
  cout << MPI::COMM_WORLD.Get_size() << endl;
  //number of correlations to compute
  int dim = nStream*(nStream - 1)/2;
  double* corr = new double[dim];

  //compute my part of the correlation array
  int ind = 0;
  for(int i = 0; i < bs; i++)
    for(int j = i + 1; j < bs; j++){
      corr[ind] = unifCorr(rng, i, j, n);
      ind += 1;
    }
  //myLength <= maxLength
  int maxLength = bs*bs;
  double* tempCorr = new double[bs*bs];
  int myLength = 0;
  //Start with 0 process that has a diagonal block
  int length = bs*(bs - 1)/2;

  for(int i = 1; i < nTasks; i++){
    MPI::COMM_WORLD.Recv(tempCorr, maxLength,
                         MPI::DOUBLE, i, 1);
    MPI::COMM_WORLD.Recv(&myLength, 1,
                         MPI::INT, i, 1);
    for(int k = 0; k < myLength; k++)
      corr[length + k] = tempCorr[k];
    length += myLength;
  }
  //now give the results
  for(int i = 0; i < length; i++)
  cout << corr[i] << endl;
}

else{
  double* myCorr;
  int myLength;
  //diagonal block
  if(indices[myRank][0] == indices[myRank][1]){
    myLength = bs*(bs - 1)/2;
    myCorr = new double[myLength];
  }
  else{
    myLength = bs*bs;
    myCorr = new double[bs*bs];
  }
  int ind = 0;
  for(int i = bs*indices[myRank][0];
      i < bs*(indices[myRank][0] + 1); i++)
    for(int j = bs*indices[myRank][1];
        j < bs*(indices[myRank][1] + 1); j++)
      if(i < j) {
        myCorr[ind] = unifCorr(rng, i, j, n);
        ind += 1;
      }
  MPI::COMM_WORLD.Send(myCorr, myLength,
                        MPI::DOUBLE, 0, 1);
  MPI::COMM_WORLD.Send(&myLength, 1,
                        MPI::INT, 0, 1);
}

MPI::Finalize();
return 0;
}
```

The number of streams, number of blocks, and block size are all determined from command line input. Then, every process creates the same index array and the same `rngBucket` object. Next, each process uses its rank to select an element of the index array `indices` that, in turn, determines the task it will perform: that is, the correlation submatrix it will evaluate. Upon completion of their tasks the processes send their results to the master or rank 0 node that collects them into a vector that can be used, for example, in evaluation of the test statistics of the previous two chapters.

We compiled and ran this program on the ASU Saguaro cluster. To illustrate the possible improvement in run time we then considered the case of $8 = 2^3$ variables with a sample size of $\mathtt{n} = 10^8$. The 8 variables can be arranged as one block of size 8, 2 blocks of size 4 or 4 blocks of size 2. This translates into `nTasks` values of 1, 3 and 10, respectively. We then assigned a processor to every task with the result that our observed speedup was 1.7 when using 2 blocks of size 4 with 3 processors and 5.9 when using 4 blocks of size 2 with 10 processors. This was based on an average of three run times for each of the three scenarios.

7.3 Discussion

The developments in this chapter can be viewed as providing a precursor of how the `TestU01` package might be rewritten in C++ to incorporate parallel processing capabilities. The presentation has been limited to the computation of correlations which is of most relevance to our correlation based testing methods from Chapter 5. However, in principle, all the tests from the `TestU01` package can be adapted to this type of treatment.

One can envision C++ `TestU01` classes that perform the operations that are currently embodied in the procedural C code format. These would then operate on something such as our `rngBucket` objects to compare and evaluate generators. Conceptually, this is fairly straightforward although the actual code development will undoubtedly be quite involved and challenging.

Of more statistical substance is how the `TestU01` test statistics should be handled when working in a true parallel setting. Our use of the `rngBucket` class provides one way to package generators so that they can be easily used in such environments with a minimal memory signature. Each process can then perform the relevant tests on some subset of the generators in the bucket and communicate results back to the master node. This will work fine for the two-level testing paradigm. However, effective implementation of the concatenation scheme is more problematic. The communication of long pseudorandom number streams across processes is impractical and even the storage of such streams on individual processes is likely unfeasible. However, many of the tests are sums which provide the option of communicating partial sums back to the master node for subsequent processing.

Chapter 8

CONCLUSION

The research presented in this dissertation enhances the current methodology for testing parallel pseudorandom number generators. We have determined that stream dependencies can be missed by the two parallel testing schemes of the standard `TestU01` testing suite because neither one adequately addresses the multivariate nature of the data that is generated in parallel. We then composed a multivariate extension to this suite in C++ using statistical tests based on correlation analysis and vector time series.

We developed three examples of generators in the programming language R, referred to as the “Smoking Gun" generators, with built-in stream dependencies that pass multiple batteries of tests in `TestU01`. To provide for more stringent testing of these R based generators, an addition to the `TestU01` suite referred to as `CrushFile` was created. Each of the Smoking Gun generators passed all of the nearly 100 tests of this intensive `CrushFile` addition. The source code for `CrushFile` is included in Appendix A.

The first Smoking Gun generator was designed to create pairwise correlations between consecutive streams. Since `TestU01` requires the two-dimensional array of multiple streams to be flattened into a single long vector, we showed that these pairwise correlations are not detected by the `TestU01` methods. This is due to the correlated elements being a long distance apart in the flattened array. The second and third Smoking Gun generators illustrate that the dependencies inherent to a time series progressing across the streams are also concealed by flattening techniques. Each of these generators provide corroboration that easily seen inter-stream dependencies in parallel generated data become undetectable by the serial transformation of the multivariate parallel data in the `TestU01` package.

We have developed an extension for `TestU01` that is designed to better detect inter-stream dependencies such as those produced by our Smoking Gun generators. The source code for the extension is provided in Appendix B. Following the naming convention in `TestU01`, this extension is divided into three parts each beginning with the letter ‘`m`’ denoting “multivariate": `mcorr`, `mmult`, and `mport`.

The mcorr part addresses testing for pairwise correlations between the vectors of generated data using standard normal transformations of the Pearson's, Spearman's, and Kendall's sample correlation measures. The algorithm developed by Benjamini, Hochberg, and Yekutieli for controlling experiment-wise error rates is used in carrying out these pairwise comparisons.

The mmult component of our extension contains a method to test for pairwise dependence in streams of uniform pseudorandom numbers without the need to control for the global error rate. The array of deviates is first transformed to standard normal deviates and then a likelihood ratio test statistic for checking for significant deviations from an identity correlation matrix based on sample correlations is calculated.

Lastly, the final piece of the multivariate extension, mport, uses vector time series tests for white noise. These provide another strategy that can be used for detection of between stream dependence. Three different multivariate portmanteau tests for white noise are included in mport: the Hosking, the Li-McLeod, and the Mahdi-McLeod tests.

Our multivariate extension was shown to detect the built-in stream dependencies of each of the problematic Smoking Gun generators. The mcorr class detected significant correlation between each of the consecutive streams in all three of the Smoking Gun generators. It produced output with small P-values corresponding to correlations of successive pairs of vectors: i.e., between vector 1 and vector 2, between vector 2 and vector 3, etc. Each of the three correlation measures produced these same rejections of the null hypotheses of nonzero pairwise correlation between each of the vectors of generated data.

Similar results were produced by the mmult component. In testing for the pairwise correlation matrix equaling the identity using the likelihood ratio test, each of the Smoking Gun generators produced P-values that were nearly zero. Much the same as the conclusions found in using the mcorr class, this provides strong evidence that dependence exists among the streams of the data generated using each of the three Smoking Gun generators.

The mport module was also able to detect the dependencies in both the second and third Smoking Gun generators. In opposition to what was initially hypothesized, each of the three tests in mport were unable to detect the dependencies in the first Smoking Gun generator. We speculate that this is due to the time series only evolving over a small number of columns. The much longer time series of the second and third Smoking Gun generators allowed the mport tests to produce P-values near zero again, thereby recommending rejection of the null model of white noise of the mean-centered deviates.

These vector time series tests provide an avenue for future research. In our extension, we fit the mean-corrected data to a zeroth order vector time series model. It could also be beneficial to fit multivariate time series of order one or two (or larger) to the data. Then, the use of Bayes' Information Criterion could be used to determine which order vector autoregressive model fits the raw mean-corrected data best. A likelihood ratio test could also be performed to look at the significance of the model fit versus a constant fit of the data. One could then proceed in either case similarly to the tests for white noise described above. If one was lead to reject the null hypothesis, an estimator of the true correlation structure in the data would be provided in the selection of the order fit, thereby providing additional potentially useful information that is not available from our current methods.

The extension developed here works in serial with the two-dimensional array of values that could be produced by processors running in parallel. It should also be possible to have some parts of the testing components of this extension be computed in parallel. Some ideas on these parallel aspects in terms of the correlation matrix calculations have been presented in Chapter 7 of this dissertation. The TestU01 package could also be adapted to use the parallel programming capabilities of C++. Parallelization would improve the speed with which both the standard TestU01 suite and its extension discussed throughout are able to test large data sets of deviates.

Parallel pseudorandom number generation is an emerging field of interest for scientists. The work in this dissertation provides improved methodologies for enhancing the quality of the generators used in these parallel settings. The extension to the TestU01 suite discussed here in conjunction with the strong capabilities of the TestU01 serial tests provide researchers with a fundamental tool for checking that their methodologies will appropriately represent the random events they seek to study.

[1] S. Aluru, G. M. Prabhu, and J. Gustafson. A random number generator for parallel computers. *Parallel Computing*, 18:839–847, 1992.

[2] Y. Benjamini and D. Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, 29(4):1165–1188, 2001.

[3] J. Burkardt. Prob source code. `http://people.sc.fsu.edu/~jburkardt/cpp_src/prob/prob.html`, 2010.

[4] P. Coddington. Analysis of random number generators using Monte Carlo simulation. *Int. J. of Mod. Phys. C*, 5:54–68, 1994.

[5] P. Coddington. Random number generators for parallel computers. *The NHSE Review*, 2, 1997.

[6] W. Dunsmuir and E. Hannan. Vector linear time series models. *Advances in Applied Probability*, pages 339–364 1976.

[7] K. Entacher, A. Uhl, and S. Wegenkittl. Parallel random number generation: Long-range correlations among multiple processors. *ACPC*, pages 107–116, 1999.

[8] R. L. Eubank and A. Kupresanin. *Statistical Computing in C++ and R*. CRC Press, Boca Raton, FL, 2011.

[9] EC Fieller, HO Hartley, and ES Pearson. Tests for rank correlation coefficients. i. *Biometrika*, 44(3/4):470–481

[10] Ronald Aylmer Fisher. On the “probable error" of a coefficient of correlation deduced from a small sample. *Metron*, 1:3–32, 1921.

[11] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for $\mathbb{F}_2$-linear random number generators. *INFORMS Journal on Computing*, 20(3):385–390, Summer 2008.

[12] D. R. C. Hill. Practical distribution of random streams for stochastic high performance computing. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 1–8, 2010.

[13] J. R. M. Hosking. The multivariate portmanteau statistic. *Journal of the American Statistical Association*, 75(371):602–608, September 1980.

[14] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall Upper Saddle River, NJ, 6th edition, 2007.

[15] C. Kao and H. C. Tang. Systematic searches for good multiple recursive random number generators. *Computers and Operations Research*, 24(10):899–905, October 1997.

[16] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[17] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, Jan. - Feb. 1999.

[18] P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, April 1993.

[19] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(22), August 2007.

[20] P. L'Ecuyer and R. Simard. TestU01: A software library in ANSI C for empirical testing of random number generators, user's guide, detailed version. Technical report, Université de Montréal, 2009.

[21] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, November-December 2002.

[22] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM (JACM)*, 20(3):456–468

[23] W. K. Li and A. I. McLeod. Distribution of the residual autocorrelations in multivariate ARMA time series models. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 231–239

[24] E. Mahdi and A. I. McLeod. Improved multivariate portmanteau test. *Journal of Time Series Analysis*, 33(2):211–222, 2012.

[25] G. Marsaglia. DIEHARD: a battery of tests of randomness. `http://stat.fsu.edu/~geo/diehard.html`, 2013.

[26] G. Marsaglia and A. Zaman. The KISS generator. Technical report, University of Florida, 1993.

[27] M. Mascagni. Parallel linear congruential generators with prime moduli. *Parallel Computing*, 24:923–936, 1998.

[28] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26(3):436–461, September 2000.

[29] M. Mascagni and A. Srinivasan. Parameterizing parallel multiplicative lagged-Fibonacci generators. *Parallel Computing*, 30:899–916, 2004.

[30] M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation*, 2:179–194, 1992.

[31] M. Matsumoto and Y. Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 4(3):254–266

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

[33] M. Matsumoto and T. Nishimura. *Monte Carlo and Quasi-Monte Carlo Methods*, chapter Dynamic Creation of Pseudorandom Number Generators, pages 56–69. Springer, 2000.

[34] R. J. Muirhead. *Aspects of Multivariate Statistical Theory*. Wiley-Interscience, 2nd edition, 2005.

[35] NIST. Template numerical toolkit. `http://math.nist.gov/tnt/`, March 2004.

[36] A. V. Prokhorov. Kendall coefficient of rank correlation. *Online Encyclopedia of Mathematics*, 2001.

[37] D. Simcha. O (N log N) impl of Kendall's Tau. `https://stat.ethz.ch/pipermail/r-devel/2010-February/056745.html`, February 2010.

[38] A. Srinivasan, M. Mascagni, and D. Ceperley. Testing parallel random number generators. *Parallel Computing*, 29(1):69–94, January 2003.

APPENDIX A

Source Code for the CrushFile Addition

```
extern "C"{
#include "util.h"
#include "smultin.h"
#include "sknuth.h"
#include "smarsa.h"
#include "snpair.h"
#include "svaria.h"
#include "sstring.h"
#include "swalk.h"
#include "scomp.h"
#include "sspectral.h"
#include "swrite.h"
#include "sres.h"
#include "unif01.h"
#include "ufile.h"

#include "gofs.h"
#include "gofw.h"
#include "fdist.h"
#include "fbar.h"
#include "num.h"
#include "chrono.h"

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <limits.h>
}

#define LEN 120
#define NAMELEN 30
#define NDIM 200                   /* Dimension of extern arrays */
#define THOUSAND 1000
#define MILLION (THOUSAND * THOUSAND)
#define BILLION (THOUSAND * MILLION)

/* The number of tests in each battery */
#define SMALLCRUSH_NUM 10
#define CRUSH_NUM 96
#define BIGCRUSH_NUM 106
#define RABBIT_NUM 26
#define ALPHABIT_NUM 9

#define PACKAGE_STRING "TestU01 1.2.3"

double bbattery_pVal[1 + NDIM] = { 0 };
char *bbattery_TestNames[1 + NDIM] = { 0 };
int bbattery_NTests;

static char CharTemp[LEN + 1];

/* Gives the test number as enumerated in bbattery.tex. Some test applies
   more than one test, so the array of p-values does not correspond with
   the test number in the doc. */
static int TestNumber[1 + NDIM] = { 0 };


/*------------------------------ Functions ------------------------------*/
```

```
static void GetName (unif01_Gen * gen, char *genName)
{
   char *p;
   int len1, len2;

   if (NULL == gen) {
      genName[0] = '\0';
      return;
   }

   /* Print only the generator name, without the parameters or seeds. */
   /* The parameters start after the first blank; name ends with ':' */
   genName[LEN] = '\0';
   len1 = strcspn (gen->name, ":");
   len1 = util_Min (LEN, len1);
   strncpy (genName, gen->name, (size_t) len1);
   genName[len1] = '\0';
   /* For Filters or Combined generators */
   p = strstr (&gen->name[1 + len1], "unif01");
   while (p != NULL) {
      len1 += 2;
      if (len1 >= LEN)
         return;
      strcat (genName, ", ");
      len2 = strcspn (p, " \0");
      len2 = util_Min (LEN - len1, len2);
      if (len2 <= 0)
         return;
      strncat (genName, p, (size_t) len2);
      len1 = strlen (genName);
      genName[len1] = '\0';
      p += len2;
      p = strstr (p, "unif01");
   }
}


/*=========================================================================*/

static void WritepVal (double p)
/*
 * Write a p-value with a nice format.
 */
{
   if (p < gofw_Suspectp) {
      gofw_Writep0 (p);

   } else if (p > 1.0 - gofw_Suspectp) {
      if (p >= 1.0 - gofw_Epsilonp1) {
         printf (" 1 - eps1");
      } else if (p >= 1.0 - 1.0e-4) {
         printf (" 1 - ");
         num_WriteD (1.0 - p, 7, 2, 2);
         /* printf (" 1 - %.2g ", 1.0 - p); */
      } else if (p >= 1.0 - 1.0e-2)
         printf ("  %.4f ", p);
      else
```

```
         printf ("    %.2f", p);
   }
}


/*=========================================================================*/

static void WriteReport (
   char *genName,                 /* Generator or file name */
   char *batName,                 /* Battery name */
   int N,                         /* Max. number of tests */
   double pVal[],                 /* p-values of the tests */
   chrono_Chrono * Timer,         /* Timer */
   lebool Flag,                  /* = TRUE for a file, FALSE for a gen */
   lebool VersionFlag,           /* = TRUE: write the version number */
   double nb                      /* Number of bits in the random file */
   )
{
   int j, co;

   printf ("\n========= Summary results of ");
   printf ("%s", batName);
   printf (" =========\n\n");
   if (VersionFlag)
      printf (" Version:          %s\n", PACKAGE_STRING);
   if (Flag)
      printf (" File:             ");
   else
      printf (" Generator:        ");
   printf ("%s", genName);
   if (nb > 0)
      printf ("\n Number of bits:   %.0f", nb);
   co = 0;
   /* Some of the tests have not been done: their pVal[j] < 0. */
   for (j = 0; j < N; j++) {
      if (pVal[j] >= 0.0)
         co++;
   }
   printf ("\n Number of statistics:  %1d\n", co);
   printf (" Total CPU time:   ");
   chrono_Write (Timer, chrono_hms);

   co = 0;
   for (j = 0; j < N; j++) {
      if (pVal[j] < 0.0)          /* That test was not done: pVal = -1 */
         continue;
      if ((pVal[j] < gofw_Suspectp) || (pVal[j] > 1.0 - gofw_Suspectp)) {
         co++;
         break;
      }
   }
   if (co == 0) {
      printf ("\n\n All tests were passed\n\n\n\n");
      return;
   }

   if (gofw_Suspectp >= 0.01)
      printf ("\n The following tests gave p-values outside [%.4g, %.2f]",
         gofw_Suspectp, 1.0 - gofw_Suspectp);
   else if (gofw_Suspectp >= 0.0001)
      printf ("\n The following tests gave p-values outside [%.4g, %.4f]",
         gofw_Suspectp, 1.0 - gofw_Suspectp);
   else if (gofw_Suspectp >= 0.000001)
      printf ("\n The following tests gave p-values outside [%.4g, %.6f]",
         gofw_Suspectp, 1.0 - gofw_Suspectp);
   else
      printf ("\n The following tests gave p-values outside [%.4g, %.14f]",
         gofw_Suspectp, 1.0 - gofw_Suspectp);
   printf (":\n (eps  means a value < %6.1e)", gofw_Epsilonp);
   printf (":\n (eps1 means a value < %6.1e)", gofw_Epsilonp1);
   printf (":\n\n       Test                          p-value\n");
   printf (" ----------------------------------------------\n");

   co = 0;
   for (j = 0; j < N; j++) {
      if (pVal[j] < 0.0)           /* That test was not done: pVal = -1 */
         continue;
      if ((pVal[j] >= gofw_Suspectp) && (pVal[j] <= 1.0 - gofw_Suspectp))
         continue;                 /* That test was passed */
      printf (" %2d ", TestNumber[j]);
      printf (" %-30s", bbattery_TestNames[j]);
      WritepVal (pVal[j]);
      printf ("\n");
      co++;
   }

   printf (" ----------------------------------------------\n");
   if (co < N - 1) {
      printf (" All other tests were passed\n");
   }
   printf ("\n\n\n");
}


/*=========================================================================*/

static void GetPVal_Walk (long N, swalk_Res * res, int *pj,
                          const char *mess, int j2)
/*
 * Get the p-values in a swalk_RandomWalk1 test
 */
{
   int j = *pj;
   const unsigned int len = 20;

   if (N == 1) {
      bbattery_pVal[++j] = res->H[0]->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 H");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->M[0]->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 M");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);
      bbattery_pVal[++j] = res->J[0]->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 J");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->R[0]->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 R");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->C[0]->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 C");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

   } else {
      bbattery_pVal[++j] = res->H[0]->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 H");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->M[0]->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 M");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->J[0]->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 J");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->R[0]->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 R");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);

      bbattery_pVal[++j] = res->C[0]->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (CharTemp, "RandomWalk1 C");
      strncat (CharTemp, mess, (size_t) len);
      strncpy (bbattery_TestNames[j], CharTemp, (size_t) LEN);
   }

   *pj = j;
}


/*=========================================================================*/

static void GetPVal_CPairs (long N, snpair_Res * res, int *pj, char *mess,
   int j2)
/*
```

```
 * Get the p-values in a snpair_ClosePairs test
 */
{
   int j = *pj;
   const unsigned int len = 20;

   if (N == 1) {
      bbattery_pVal[++j] = res->pVal[snpair_NP];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs NP");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      bbattery_pVal[++j] = res->pVal[snpair_mNP];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs mNP");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

   } else {
      bbattery_pVal[++j] = res->pVal[snpair_NP];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs NP");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      bbattery_pVal[++j] = res->pVal[snpair_mNP];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs mNP");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      bbattery_pVal[++j] = res->pVal[snpair_mNP1];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs mNP1");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      bbattery_pVal[++j] = res->pVal[snpair_mNP2];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs mNP2");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      bbattery_pVal[++j] = res->pVal[snpair_NJumps];
      TestNumber[j] = j2;
      strcpy (CharTemp, "ClosePairs NJumps");
      strncat (CharTemp, mess, (size_t) len);
      strcpy (bbattery_TestNames[j], CharTemp);

      if (snpair_mNP2S_Flag) {
         bbattery_pVal[++j] = res->pVal[snpair_mNP2S];
         TestNumber[j] = j2;
         strcpy (CharTemp, "ClosePairs mNP2S");
         strncat (CharTemp, mess, (size_t) len);
         strcpy (bbattery_TestNames[j], CharTemp);
      }
   }
   *pj = j;
}


/*=========================================================================*/

static void InitBat (void)
/*
 * Initializes the battery of tests: sets all p-values to -1.
 */
{
   int j;
   static int flag = 0;
   for (j = 0; j < NDIM; j++)
      bbattery_pVal[j] = -1.0;
   if (0 == flag) {
      flag++;
      for (j = 0; j < NDIM; j++)
        bbattery_TestNames[j] = (char*) util_Calloc
          (LEN + 1, sizeof (char));   //Modified
   }
}



/*=========================================================================*/

static void Crush (unif01_Gen * gen, char *filename, int Rep[])
/*
 * A battery of stringent statistical tests for Random Number Generators
 * used in simulation.
 * Rep[i] gives the number of times that test i will be done. The default
 * values are Rep[i] = 1 for all i.
 */
{
   const int s = 30;
   const int r = 0;
   int i;
   chrono_Chrono *Timer;
   char genName[LEN + 1] = "";
   int j = -1;
   int j2 = 0;

   Timer = chrono_Create ();
   InitBat ();
   if (swrite_Basic) {
      printf ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n"
         "                 Starting Crush\n"
         "                 Version: %s\n"
         "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n\n\n",
         PACKAGE_STRING);
   }

   bool fileFlag;
   //Inserted
   if (NULL == gen) {
      gen = ufile_CreateReadText (filename, 2 * BILLION);
      fileFlag = TRUE;
   } else
      fileFlag = FALSE;
   //

   {
      sres_Basic *res;
      res = sres_CreateBasic ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_SerialOver (gen, res, 1, 500 * MILLION, 0, 4096, 2);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SerialOver, t = 2");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_SerialOver (gen, res, 1, 300 * MILLION, 0, 64, 4);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SerialOver, t = 4");
      }
      sres_DeleteBasic (res);
   }
   {
      smarsa_Res *res;
      res = smarsa_CreateRes ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 0, 1024 * 1024, 2);
         bbattery_pVal[++j] = res->Pois->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CollisionOver, t = 2");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 10, 1024 * 1024, 2);
         bbattery_pVal[++j] = res->Pois->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CollisionOver, t = 2");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 0, 1024, 4);
         bbattery_pVal[++j] = res->Pois->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CollisionOver, t = 4");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 20, 1024, 4);
```

```
          bbattery_pVal[++j] = res->Pois->pVal2;
          TestNumber[j] = j2;
          strcpy (bbattery_TestNames[j], "CollisionOver, t = 4");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
          smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 0, 32, 8);
          bbattery_pVal[++j] = res->Pois->pVal2;
          TestNumber[j] = j2;
          strcpy (bbattery_TestNames[j], "CollisionOver, t = 8");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
          smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 25, 32, 8);
          bbattery_pVal[++j] = res->Pois->pVal2;
          TestNumber[j] = j2;
          strcpy (bbattery_TestNames[j], "CollisionOver, t = 8");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
          smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 0, 4, 20);
          bbattery_pVal[++j] = res->Pois->pVal2;
          TestNumber[j] = j2;
          strcpy (bbattery_TestNames[j], "CollisionOver, t = 20");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
          smarsa_CollisionOver (gen, res, 10, 10 * MILLION, 28, 4, 20);
          bbattery_pVal[++j] = res->Pois->pVal2;
          TestNumber[j] = j2;
          strcpy (bbattery_TestNames[j], "CollisionOver, t = 20");
      }
      smarsa_DeleteRes (res);
   }

   {
      sres_Poisson *res;
      res = sres_CreatePoisson ();
      if (fileFlag) ufile_InitReadText (); //Inserted
#ifdef USE_LONGLONG
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
          long d;
#if LONG_MAX <= 2147483647L
          d = 1073741824L;
          smarsa_BirthdaySpacings (gen, res, 10, 10 * MILLION, 0, d, 2, 1);
#else
          d = 2*1073741824L;
          smarsa_BirthdaySpacings (gen, res, 5, 20 * MILLION, 0, d, 2, 1);
#endif
          bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 2");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 5, 20 * MILLION, 0, 2097152, 3,
      1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 3");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 5, 20 * MILLION, 0, 65536, 4, 1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 4");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 3, 20 * MILLION, 0, 512, 7, 1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 7");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 3, 20 * MILLION, 7, 512, 7, 1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 7");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 3, 20 * MILLION, 14, 256, 8, 1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 8");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_BirthdaySpacings (gen, res, 3, 20 * MILLION, 22, 256, 8, 1);
   bbattery_pVal[++j] = res->pVal2;
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 8");
}
```

```
#else
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 200, 4 * MILLION / 10, 0,
            67108864, 2, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 2");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 100, 4 * MILLION / 10, 0, 131072,
            3, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 3");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 200, 4 * MILLION / 10, 0,
            1024 * 8, 4, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 4");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 100, 4 * MILLION / 10, 0, 16, 13,
            1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 13");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 100, 4 * MILLION / 10, 10, 16,
            13, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 13");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 100, 4 * MILLION / 10, 20, 16,
            13, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 13");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_BirthdaySpacings (gen, res, 100, 4 * MILLION / 10, 26, 16,
            13, 1);
         bbattery_pVal[++j] = res->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "BirthdaySpacings, t = 13");
      }
#endif

      sres_DeletePoisson (res);
   }
   {
      lebool flag = snpair_mNP2S_Flag;
      snpair_Res *res;
      res = snpair_CreateRes ();

      snpair_mNP2S_Flag = FALSE;
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         snpair_ClosePairs (gen, res, 10, 2 * MILLION, 0, 2, 0, 30);
         GetPVal_CPairs (10, res, &j, (char*)", t = 2", j2);
      }

      snpair_mNP2S_Flag = TRUE;
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         snpair_ClosePairs (gen, res, 10, 2 * MILLION, 0, 3, 0, 30);
         GetPVal_CPairs (10, res, &j, (char*)", t = 3", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         snpair_ClosePairs (gen, res, 5, 2 * MILLION, 0, 7, 0, 30);
         GetPVal_CPairs (10, res, &j, (char*)", t = 7", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         snpair_ClosePairsBitMatch (gen, res, 4, 4 * MILLION, 0, 2);
         bbattery_pVal[++j] = res->pVal[snpair_BM];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "ClosePairsBitMatch, t = 2");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         snpair_ClosePairsBitMatch (gen, res, 2, 4 * MILLION, 0, 4);
         bbattery_pVal[++j] = res->pVal[snpair_BM];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "ClosePairsBitMatch, t = 4");
      }
      snpair_DeleteRes (res);
      snpair_mNP2S_Flag = flag;
   }

   //SECOND RUN ENDS HERE (CrushVMA_out3.txt)
   {
      sres_Chi2 *res;
      res = sres_CreateChi2 ();

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_SimpPoker (gen, res, 1, 40 * MILLION, 0, 16, 16);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SimpPoker, d = 16");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_SimpPoker (gen, res, 1, 40 * MILLION, 26, 16, 16);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SimpPoker, d = 16");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_SimpPoker (gen, res, 1, 10 * MILLION, 0, 64, 64);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SimpPoker, d = 64");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_SimpPoker (gen, res, 1, 10 * MILLION, 24, 64, 64);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SimpPoker, d = 64");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_CouponCollector (gen, res, 1, 40 * MILLION, 0, 4);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CouponCollector, d = 4");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_CouponCollector (gen, res, 1, 40 * MILLION, 28, 4);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CouponCollector, d = 4");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_CouponCollector (gen, res, 1, 10 * MILLION, 0, 16);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CouponCollector, d = 16");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_CouponCollector (gen, res, 1, 10 * MILLION, 26, 16);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "CouponCollector, d = 16");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_Gap (gen, res, 1, 100 * MILLION, 0, 0.0, 0.125);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Gap, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_Gap (gen, res, 1, 100 * MILLION, 27, 0.0, 0.125);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Gap, r = 27");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_Gap (gen, res, 1, 5 * MILLION, 0, 0.0, 1.0/256.0);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Gap, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sknuth_Gap (gen, res, 1, 5 * MILLION, 22, 0.0, 1.0/256.0);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Gap, r = 22");
      }
```

```
    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_Run (gen, res, 1, 500 * MILLION, 0, TRUE);
       bbattery_pVal[++j] = res->pVal2[gofw_Mean];
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "Run of U01, r = 0");
    }

    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_Run (gen, res, 1, 500 * MILLION, 15, FALSE);
       bbattery_pVal[++j] = res->pVal2[gofw_Mean];
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "Run of U01, r = 15");
    }

    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_Permutation (gen, res, 1, 50 * MILLION, 0, 10);
       bbattery_pVal[++j] = res->pVal2[gofw_Mean];
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "Permutation, r = 0");
    }

    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_Permutation (gen, res, 1, 50 * MILLION, 15, 10);
       bbattery_pVal[++j] = res->pVal2[gofw_Mean];
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "Permutation, r = 15");
    }
    sres_DeleteChi2 (res);
}
{
    sknuth_Res2 *res;
    res = sknuth_CreateRes2 ();
    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_CollisionPermut (gen, res, 5, 10 * MILLION, 0, 13);
       bbattery_pVal[++j] = res->Pois->pVal2;
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "CollisionPermut, r = 0");
    }

    if (fileFlag) ufile_InitReadText (); //Inserted
    ++j2;
    for (i = 0; i < Rep[j2]; ++i) {
       sknuth_CollisionPermut (gen, res, 5, 10 * MILLION, 15, 13);
       bbattery_pVal[++j] = res->Pois->pVal2;
       TestNumber[j] = j2;
       strcpy (bbattery_TestNames[j], "CollisionPermut, r = 15");
    }
    sknuth_DeleteRes2 (res);
}
```

```
{
   sknuth_Res1 *res;
   res = sknuth_CreateRes1 ();

   if (fileFlag) ufile_InitReadText (); //Inserted
   ++j2;
   for (i = 0; i < Rep[j2]; ++i) {
      sknuth_MaxOft (gen, res, 10, 10 * MILLION, 0, MILLION / 10, 5);
      bbattery_pVal[++j] = res->Chi->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft, t = 5");
      bbattery_pVal[++j] = res->Bas->pVal2[gofw_AD];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft AD, t = 5");
   }

   if (fileFlag) ufile_InitReadText (); //Inserted
   ++j2;
   for (i = 0; i < Rep[j2]; ++i) {
      sknuth_MaxOft (gen, res, 5, 10 * MILLION, 0, MILLION / 10, 10);
      bbattery_pVal[++j] = res->Chi->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft, t = 10");
      bbattery_pVal[++j] = res->Bas->pVal2[gofw_AD];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft AD, t = 10");
   }

   if (fileFlag) ufile_InitReadText (); //Inserted
   ++j2;
   for (i = 0; i < Rep[j2]; ++i) {
      sknuth_MaxOft (gen, res, 1, 10 * MILLION, 0, MILLION / 10, 20);
      bbattery_pVal[++j] = res->Chi->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft, t = 20");
      bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft AD, t = 20");
   }

   if (fileFlag) ufile_InitReadText (); //Inserted
   ++j2;
   for (i = 0; i < Rep[j2]; ++i) {
      sknuth_MaxOft (gen, res, 1, 10 * MILLION, 0, MILLION / 10, 30);
      bbattery_pVal[++j] = res->Chi->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft, t = 30");
      bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "MaxOft AD, t = 30");
   }
   sknuth_DeleteRes1 (res);
}
{
   sres_Basic *res;
   res = sres_CreateBasic ();

   if (fileFlag) ufile_InitReadText (); //Inserted
   ++j2;
```

```
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_SampleProd (gen, res, 1, 10 * MILLION, 0, 10);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SampleProd, t = 10");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_SampleProd (gen, res, 1, 10 * MILLION, 0, 30);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SampleProd, t = 30");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_SampleMean (gen, res, 10*MILLION, 20, 0);
         bbattery_pVal[++j] = res->pVal2[gofw_AD];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SampleMean");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_SampleCorr (gen, res, 1, 500 * MILLION, 0, 1);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "SampleCorr");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_AppearanceSpacings (gen, res, 1, 10 * MILLION, 400 * MILLION,
            r, 30, 15);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AppearanceSpacings, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         svaria_AppearanceSpacings (gen, res, 1, 10 * MILLION, 100 * MILLION,
            20, 10, 15);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AppearanceSpacings, r = 20");
      }
      sres_DeleteBasic (res);
   }
   {
      smarsa_Res2 *res2;
      sres_Chi2 *res;
      res = sres_CreateChi2 ();
```

```
if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   svaria_WeightDistrib (gen, res, 1, 2 * MILLION, 0, 256, 0.0, 0.125);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "WeightDistrib, r = 0");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   svaria_WeightDistrib (gen, res, 1, 2 * MILLION, 8, 256, 0.0, 0.125);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "WeightDistrib, r = 8");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   svaria_WeightDistrib (gen, res, 1, 2 * MILLION, 16, 256, 0.0, 0.125);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "WeightDistrib, r = 16");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   svaria_WeightDistrib (gen, res, 1, 2 * MILLION, 24, 256, 0.0, 0.125);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "WeightDistrib, r = 24");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   svaria_SumCollector (gen, res, 1, 20 * MILLION, 0, 10.0);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "SumCollector");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, MILLION, r, s, 2 * s, 2 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 60 x 60");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, MILLION, 20, 10, 2 * s, 2 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 60 x 60");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, 50 * THOUSAND, r, s, 10 * s, 10 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 300 x 300");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, 50 * THOUSAND, 20, 10, 10 * s,
      10 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 300 x 300");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, 2 * THOUSAND, r, s, 40 * s, 40 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 1200 x 1200");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_MatrixRank (gen, res, 1, 2 * THOUSAND, 20, 10, 40 * s, 40 * s);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "MatrixRank, 1200 x 1200");
}

if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_Savir2 (gen, res, 1, 20 * MILLION, 0, 1024*1024, 30);
   bbattery_pVal[++j] = res->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "Savir2");
}
sres_DeleteChi2 (res);

res2 = smarsa_CreateRes2 ();
if (fileFlag) ufile_InitReadText (); //Inserted
++j2;
for (i = 0; i < Rep[j2]; ++i) {
   smarsa_GCD (gen, res2, 1, 100 * MILLION, 0, 30);
   bbattery_pVal[++j] = res2->GCD->pVal2[gofw_Mean];
   TestNumber[j] = j2;
   strcpy (bbattery_TestNames[j], "GCD, r = 0");
```

```
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         smarsa_GCD (gen, res2, 1, 40 * MILLION, 10, 20);
         bbattery_pVal[++j] = res2->GCD->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "GCD, r = 10");
      }
      smarsa_DeleteRes2 (res2);
   }
   {
      swalk_Res *res;
      res = swalk_CreateRes ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, 50 * MILLION, r, s, 90, 90);
         GetPVal_Walk (1, res, &j, " (L = 90)", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, 10 * MILLION, 20, 10, 90, 90);
         GetPVal_Walk (1, res, &j, " (L = 90)", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, 5 * MILLION, r, s, 1000, 1000);
         GetPVal_Walk (1, res, &j, " (L = 1000)", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, MILLION, 20, 10, 1000, 1000);
         GetPVal_Walk (1, res, &j, " (L = 1000)", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, MILLION / 2, r, s, 10000, 10000);
         GetPVal_Walk (1, res, &j, " (L = 10000)", j2);
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         swalk_RandomWalk1 (gen, res, 1, MILLION / 10, 20, 10, 10000, 10000);
         GetPVal_Walk (1, res, &j, " (L = 10000)", j2);
      }
      swalk_DeleteRes (res);
   }
   {
      scomp_Res *res;
      res = scomp_CreateRes ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         scomp_LinearComp (gen, res, 1, 120 * THOUSAND, r, 1);
         bbattery_pVal[++j] = res->JumpNum->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LinearComp, r = 0");
         bbattery_pVal[++j] = res->JumpSize->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LinearComp, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         scomp_LinearComp (gen, res, 1, 120 * THOUSAND, 29, 1);
         bbattery_pVal[++j] = res->JumpNum->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LinearComp, r = 29");
         bbattery_pVal[++j] = res->JumpSize->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LinearComp, r = 29");
      }
      scomp_DeleteRes (res);
   }
   {
      sres_Basic *res;
      res = sres_CreateBasic ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         scomp_LempelZiv (gen, res, 10, 25, r, s);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LempelZiv");
      }
      sres_DeleteBasic (res);
   }
   {
      sspectral_Res *res;
      res = sspectral_CreateRes ();

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sspectral_Fourier3 (gen, res, 50 * THOUSAND, 14, r, s);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_AD];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Fourier3, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sspectral_Fourier3 (gen, res, 50 * THOUSAND, 14, 20, 10);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_AD];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Fourier3, r = 20");
      }
      sspectral_DeleteRes (res);
   }
   {
      sstring_Res2 *res;
      res = sstring_CreateRes2 ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_LongestHeadRun (gen, res, 1, 1000, r, s, 20 + 10 * MILLION);
         bbattery_pVal[++j] = res->Chi->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LongestHeadRun, r = 0");
         bbattery_pVal[++j] = res->Disc->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LongestHeadRun, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_LongestHeadRun (gen, res, 1, 300, 20, 10, 20 + 10 * MILLION);
         bbattery_pVal[++j] = res->Chi->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LongestHeadRun, r = 20");
         bbattery_pVal[++j] = res->Disc->pVal2;
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "LongestHeadRun, r = 20");
      }
      sstring_DeleteRes2 (res);
   }
   {
      sres_Chi2 *res;
      res = sres_CreateChi2 ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_PeriodsInStrings (gen, res, 1, 300 * MILLION, r, s);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "PeriodsInStrings, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_PeriodsInStrings (gen, res, 1, 300 * MILLION, 15, 15);
         bbattery_pVal[++j] = res->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "PeriodsInStrings, r = 15");
      }
      sres_DeleteChi2 (res);
   }
   {
      sres_Basic *res;
      res = sres_CreateBasic ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
```

```
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingWeight2 (gen, res, 100, 100 * MILLION, r, s, MILLION);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingWeight2, r = 0");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingWeight2 (gen, res, 30, 100 * MILLION, 20, 10, MILLION);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingWeight2, r = 20");
      }
      sres_DeleteBasic (res);
   }
   {
      sstring_Res *res;
      res = sstring_CreateRes ();
      /* sstring_HammingCorr will probably be removed: less sensitive than
         svaria_HammingIndep */
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingCorr (gen, res, 1, 500 * MILLION, r, s, s);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingCorr, L = 30");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingCorr (gen, res, 1, 50 * MILLION, r, s, 10 * s);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingCorr, L = 300");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingCorr (gen, res, 1, 10 * MILLION, r, s, 40 * s);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingCorr, L = 1200");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, 300 * MILLION, r, s, s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 30");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, 100 * MILLION, 20, 10, s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 30");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, 30 * MILLION, r, s, 10 * s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 300");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, 10 * MILLION, 20, 10, 10 * s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 300");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, 10 * MILLION, r, s, 40 * s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 1200");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_HammingIndep (gen, res, 1, MILLION, 20, 10, 40 * s, 0);
         bbattery_pVal[++j] = res->Bas->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "HammingIndep, L = 1200");
      }
      sstring_DeleteRes (res);
   }
   {
      sstring_Res3 *res;
      res = sstring_CreateRes3 ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_Run (gen, res, 1, 1 * BILLION, r, s);
         bbattery_pVal[++j] = res->NRuns->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Run of bits, r = 0");
         bbattery_pVal[++j] = res->NBits->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Run of bits, r = 0");
      }
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_Run (gen, res, 1, 1 * BILLION, 20, 10);
         bbattery_pVal[++j] = res->NRuns->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Run of bits, r = 20");
         bbattery_pVal[++j] = res->NBits->pVal2[gofw_Mean];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "Run of bits, r = 20");
      }
      sstring_DeleteRes3 (res);
   }
   {
      sres_Basic *res;
      res = sres_CreateBasic ();
      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_AutoCor (gen, res, 10, 30 + BILLION, r, s, 1);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AutoCor, d = 1");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_AutoCor (gen, res, 5, 1 + BILLION, 20, 10, 1);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AutoCor, d = 1");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_AutoCor (gen, res, 10, 31 + BILLION, r, s, s);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AutoCor, d = 30");
      }

      if (fileFlag) ufile_InitReadText (); //Inserted
      ++j2;
/*    util_Assert (j2 <= CRUSH_NUM, "Crush:  j2 > CRUSH_NUM");  */
      for (i = 0; i < Rep[j2]; ++i) {
         sstring_AutoCor (gen, res, 5, 11 + BILLION, 20, 10, 10);
         bbattery_pVal[++j] = res->pVal2[gofw_Sum];
         TestNumber[j] = j2;
         strcpy (bbattery_TestNames[j], "AutoCor, d = 10");
      }
      sres_DeleteBasic (res);
   }

   bbattery_NTests = ++j;
   GetName (gen, genName);
   WriteReport (genName, (char*)"Crush", bbattery_NTests,
      bbattery_pVal, Timer, FALSE, TRUE, 0.0);
   chrono_Delete (Timer);
}


/*=========================================================================*/

void bbattery_Crush (unif01_Gen * gen)
{
   int i;
   int Rep[NDIM + 1] = {0};
   for (i = 1; i <= CRUSH_NUM; ++i)
      Rep[i] = 1;
   Crush (gen, NULL, Rep); //Modified
}


void bbattery_CrushFile (char *filename)
{
    int i;
    int Rep[1 + NDIM] = {0};
    for (i = 1; i <= CRUSH_NUM; ++i)
        Rep[i] = 1;
    Crush (NULL, filename, Rep);
}

/*=========================================================================*/

void bbattery_RepeatCrush (unif01_Gen * gen, int Rep[])
{
  Crush (gen, NULL, Rep);
}



/*=========================================================================*/
#if 0
static void WriteTime (time_t t0, time_t t1)
{
   int y1;
   double y = 0;

   y = difftime (t1, t0);
   /* printf (" Total time: %.2f sec\n\n", y); */
   printf (" Total time: ");
   y1 = y / 3600;
   printf ("%02d:", y1);
   y -= y1 * 3600.0;
   y1 = y / 60;
   printf ("%02d:", y1);
   y -= y1 * 60.0;
   printf ("%.2f\n\n", y);
}
#endif


/*=========================================================================*/

static void DoMultinom (lebool fileFlag, /* */
   unif01_Gen * gen,              /* */
   double nb,                     /* Number of bits */
   int *pj,                       /* j */
   int j2,                        /* Test number in the battery */
   int Rep[]                      /* Number of replications */
   )
/*
 * Do the smultin_MultinomialBits in Rabbit
 */
{
   const long NLIM = 10000000;
   long n, N;
   int L, t;
   double x;
   int i;
   int j = *pj;
   smultin_Res *res;
   smultin_Param *par = NULL;
   double ValDelta[] = { -1 };

   util_Assert (nb > 0.0, "MultinomialBits:   nb <= 0");
   par = smultin_CreateParam (1, ValDelta, smultin_GenerCellSerial, -3);
   res = smultin_CreateRes (par);
   if (fileFlag)
      ufile_InitReadBin ();

#ifdef USE_LONGLONG
   /* Limit sample size n to NLIM because of memory limitations. */
   /* Determine number of replications N from this. */
   N = 1 + nb / NLIM;
   n = nb / N;
   /* Time limit on test: N = 30 */
   N = util_Min (30, N);
   /* Set n as a multiple of s = 32 */
   n -= n % 32;
   L = num_Log2 (n / 200.0 * n);
   L = util_Max (4, L);
   for (i = 0; i < Rep[j2]; ++i) {
      smultin_MultinomialBitsOver (gen, par, res, N, n, 0, 32, L, TRUE);
      strcpy (bbattery_TestNames[++j], "MultinomialBitsOver");
      bbattery_pVal[j] = res->pColl;
      TestNumber[j] = j2;
   }

#else
   x = nb / 32.0;
   N = 1 + x / NLIM;
   n = x / N;
   N = util_Min (30, N);
   L = 16;
   t = 32 / L;
   /* We want a number of collisions >= 2 */
   while ((L > 1) && (n / num_TwoExp[L] * n * t * t < 2.0)) {
      L /= 2;
      t = 32 / L;
   }
   n = n * (32 / L);
   /* We want a density n / k < 2 to use case Sparse = TRUE */
   if (n > 2 * num_TwoExp[L]) {
      N = n / num_TwoExp[L] * N;
      n /= N;
      while ((double) N * n * L > nb)
         n--;
   }
   while (n * L % 32 > 0)
      n--;
   if (n > 3) {
      for (i = 0; i < Rep[j2]; ++i) {
         smultin_MultinomialBits (gen, par, res, N, n, 0, 32, L, TRUE);
         strcpy (bbattery_TestNames[++j], "MultinomialBits");
         bbattery_pVal[j] = res->pColl;
         TestNumber[j] = j2;
      }
   }
#endif
   *pj = j;
   smultin_DeleteRes (res);
   smultin_DeleteParam (par);
}


/*-------------------------------------------------------------------------*/

static void DoAppear (lebool fileFlag, /* */
   unif01_Gen * gen, double nb,    /* Number of bits to test */
   int *pj,                        /* j */
   int j2,                         /* Test number in the battery */
   int Rep[]
   )
/*
 * Do the svaria_AppearanceSpacings test in Rabbit
 */
{
   sres_Basic *res;
   const long NLIM = 2000000000;
   int L;
   long N, Q;
   int i;
   int j = *pj;
   double temp = nb * (30.0 / 32.0) / 20.0;

   res = sres_CreateBasic ();
   if (num_TwoExp[30] < temp / 30.0)
      L = 30;
   else if (num_TwoExp[15] < temp / 15.0)
      L = 15;
   else if (num_TwoExp[10] < temp / 10.0)
      L = 10;
   else if (num_TwoExp[6] < temp / 6.0)
      L = 6;
   else if (num_TwoExp[5] < temp / 5.0)
      L = 5;
   else if (num_TwoExp[3] < temp / 3.0)
      L = 3;
   else
      L = 2;
   temp = nb / 2;
   temp *= 30.0 / 32.0;
   temp /= L;
   N = 1 + temp / NLIM;
   Q = temp / N;
   N = 1;

   if (Q < 50)
      return;
   if (fileFlag)
      ufile_InitReadBin ();
   for (i = 0; i < Rep[j2]; ++i) {
      svaria_AppearanceSpacings (gen, res, N, Q, Q, 0, 30, L);
      j++;
      if (N == 1)
         bbattery_pVal[j] = res->pVal2[gofw_Mean];
      else
         bbattery_pVal[j] = res->pVal2[gofw_Sum];
      TestNumber[j] = j2;
      strcpy (bbattery_TestNames[j], "AppearanceSpacings");
   }
   sres_DeleteBasic (res);
   *pj = j;
}


/*---------------------------------------------------------------------------*/

static void DoWalk (lebool fileFlag,  /* */
   unif01_Gen * gen,               /* */
   double nb,                      /* Number of bits to test */
   int *pj,                        /* j */
   int j2,                         /* Test number in the battery */
   int Rep[]
   )
/*
 * Do 3 swalk_RandomWalk1 tests in Rabbit
 */
{
   swalk_Res *res;
   long n, N, L;
   double z;
   int i;

   L = 128;
   z = nb / L;
   N = 1 + z / BILLION;
   n = z / N;
   N = 1;
   while (n < 100) {
      L /= 2;
      n *= 2;
   }
   if (L < 4)
      return;
   n = nb / (L * N);
   n = util_Min (n, 500 * MILLION);
   if (L < 32) {
      while (32 * n > nb)
         n--;
   }
   if (n < 30)
      return;

   res = swalk_CreateRes ();
   ++j2;
   if (fileFlag)
      ufile_InitReadBin ();
   for (i = 0; i < Rep[j2]; ++i) {
      swalk_RandomWalk1 (gen, res, N, n, 0, 32, L, L);
      GetPVal_Walk (N, res, pj, "", j2);
   }
   if (L < 96)
      return;

   L = 1024;
   z = nb / L;
   N = 1 + z / BILLION;
   n = z / N;
   n = util_Min (n, 50 * MILLION);
   N = 1;
   while ((double) n * L > nb)
      n--;
   if (n < 30)
      return;

   ++j2;
   if (fileFlag)
      ufile_InitReadBin ();
   for (i = 0; i < Rep[j2]; ++i) {
      swalk_RandomWalk1 (gen, res, N, n, 0, 32, L, L);
      GetPVal_Walk (N, res, pj, " (L = 1024)", j2);
   }

   L = 10016;
   z = nb / L;
   N = 1 + z / BILLION;
   n = z / N;
   n = util_Min (n, 5 * MILLION);
   N = 1;
   while ((double) n * L > nb)
      n--;
   if (n < 30)
      return;
   ++j2;
   if (fileFlag)
      ufile_InitReadBin ();
   for (i = 0; i < Rep[j2]; ++i) {
      swalk_RandomWalk1 (gen, res, N, n, 0, 32, L, L);
      GetPVal_Walk (N, res, pj, " (L = 10016)", j2);
   }

   swalk_DeleteRes (res);
}


/*=========================================================================*/

static double ProbabiliteLHR (long j, double Lnl)
/*
```

```
 * Returns the probability that the longest series of successive 1 has
 * length = j.
 */
{
   double x, temp;
   temp = (j + 1) * num_Ln2 - LnI;
   x = exp (-exp (-temp));
   temp += num_Ln2;
   x = exp (-exp (-temp)) - x;
   return x;
}

/*-------------------------------------------------------------------------*/

static double GetPLongest (int longest)
/*
 * Get the probabilities for the longest run of 1 or 0 over 20000 bits.
 */
{
   double pLeft, pRight;
   double LnLen;
   int j;

   LnLen = log (20000.0);
   pLeft = 0.0;
   for (j = 0; j < longest; j++)
      pLeft += ProbabiliteLHR (j, LnLen);
   pRight = 1.0 - pLeft;
   pLeft += ProbabiliteLHR (longest, LnLen);
   return gofw_pDisc (pLeft, pRight);
}


/*-------------------------------------------------------------------------*/

static void WriteReportFIPS_140_2 (
   const char *genName,                /* Generator or file name */
   lebool Flag,                 /* = TRUE for a file, FALSE for a gen */
   int nbit,                     /* Number of bits */
   int longest0,                 /* Longest string of 0 */
   int longest1,                 /* Longest string of 1 */
   int nrun0[],                  /* Number of 0 runs */
   int nrun1[],                  /* Number of 1 runs */
   int ncount[]                  /* Number of 4 bits values */
   )
{
   int i, j;
   double X;
   fmass_INFO Q;
   double p, pLeft, pRight;
   lebool failFlag = FALSE;

   printf
      ("\n============= Summary results of FIPS-140-2 =============\n\n");
   if (Flag) {
      printf (" File:              ");
   } else {
      printf (" Generator:         ");
   }
```

```
printf ("%s", genName);
printf ("\n Number of bits:   20000\n");

printf ("\n       Test          s-value         p-value    FIPS Decision\n");
printf (" -----------------------------------------------------------------\n");

/* Monobit results */
j = 0;
printf (" %-20s", bbattery_TestNames[j]);
printf (" %5d        ", nbit);
Q = fmass_CreateBinomial (20000, 0.5, 0.5);
pLeft = fdist_Binomial2 (Q, nbit);
pRight = fbar_Binomial2 (Q, nbit);
fmass_DeleteBinomial (Q);
p = gofw_pDisc (pLeft, pRight);
gofw_Writep0 (p);
if ((nbit <= 9725) || nbit >= 10275) {
   printf (" %10s", "Fail");
   failFlag = TRUE;
} else
   printf (" %10s", "Pass");

printf ("\n");

/* Poker results */
X = 0;
for (i = 0; i < 16; i++)
   X += (double) ncount[i] * ncount[i];
X = 16 * X / 5000 - 5000;
j = 1;
printf (" %-16s", bbattery_TestNames[j]);
printf ("%10.2f        ", X);
p = fbar_ChiSquare2 (15, 12, X);
gofw_Writep0 (p);
if ((X <= 2.16) || X >= 46.17) {
   printf (" %10s", "Fail");
   failFlag = TRUE;
} else
   printf (" %10s", "Pass");
printf ("\n\n");

/* Run results */
printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[1]);
if ((nrun0[1] <= 2315) || nrun0[1] >= 2685) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[2]);
if ((nrun0[2] <= 1114) || nrun0[2] >= 1386) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");
printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[3]);
if ((nrun0[3] <= 527) || nrun0[3] >= 723) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[4]);
if ((nrun0[4] <= 240) || nrun0[4] >= 384) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[5]);
if ((nrun0[5] <= 103) || nrun0[5] >= 209) {
   failFlag = TRUE;
   printf (" %25s", "Fail");
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun0[6]);
if ((nrun0[6] <= 103) || nrun0[6] >= 209) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[1]);
if ((nrun1[1] <= 2315) || nrun1[1] >= 2685) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[2]);
if ((nrun1[2] <= 1114) || nrun1[2] >= 1386) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[3]);
if ((nrun1[3] <= 527) || nrun1[3] >= 723) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[4]);
if ((nrun1[4] <= 240) || nrun1[4] >= 384) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[5]);
if ((nrun1[5] <= 103) || nrun1[5] >= 209) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d", nrun1[6]);
if ((nrun1[6] <= 103) || nrun1[6] >= 209) {
   printf (" %25s", "Fail");
   failFlag = TRUE;
} else
   printf (" %25s", "Pass");
printf ("\n\n");

/* Longest run results */
printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d       ", longest0);
p = GetPLongest (longest0);
gofw_Writep0 (p);
if (longest0 >= 26) {
   printf (" %10s", "Fail");
   failFlag = TRUE;
} else
   printf (" %10s", "Pass");
printf ("\n");

printf (" %-20s", bbattery_TestNames[++j]);
printf (" %5d       ", longest1);
p = GetPLongest (longest1);
gofw_Writep0 (p);
if (longest1 >= 26) {
   printf (" %10s", "Fail");
   failFlag = TRUE;
} else
   printf (" %10s", "Pass");
printf ("\n");

if (!failFlag) {
   printf (" ----------------------------------------------------------------\n");
   printf (" All values are within the required intervals of FIPS-140-2\n");
}
```

```
   printf ("\n\n\n");
}


/*-----------------------------------------------------------------------*/

#define SAMPLE 625                  /* 625 * 32 = 20000 */
#define MASK4  15                   /* Mask of 4 bits */

static void FIPS_140_2 (unif01_Gen * gen, char *filename)
{
   int i, j;
   int nbit = 0;                    /* Number of bits */
   int longest0 = 0;                /* Longest string of 0 */
   int longest1 = 0;                /* Longest string of 1 */
   int nrun0[7] = { 0 };            /* Number of 0 runs */
   int nrun1[7] = { 0 };            /* Number of 1 runs */
   int ncount[16] = { 0 };          /* Number of 4 bits values */
   int prevBit;                     /* Previous bit */
   int len = 0;                     /* Length of run */
   unsigned long jBit;              /* Current bit */
   unsigned long Z;                 /* Block of 32 bits */
   unsigned long Bits[SAMPLE + 1];
   lebool fileFlag = FALSE;
   char genName[LEN + 1] = "";

   InitBat ();
   if (swrite_Basic) {
      printf ("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n"
         "                  Starting FIPS_140_2\n"
         "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\n\n\n");
   }
   util_Assert (NULL == gen || NULL == filename,
      "bbattery_FIPS_140_2:   one of gen or filename must be NULL");
   util_Assert (!(NULL == gen && NULL == filename),
      "bbattery_FIPS_140_2:   no generator and no file");
   util_Assert (!(NULL == gen && !(strcmp (filename, ""))),
      "bbattery_FIPS_140_2:   no generator and no file");

   if ((NULL == gen) && filename && strcmp (filename, "")) {
      gen = ufile_CreateReadBin (filename, SAMPLE);
      fileFlag = TRUE;
   }

   for (j = 0; j < SAMPLE; j++)
      Bits[j] = unif01_StripB (gen, 0, 32);

   if (fileFlag) {
      ufile_DeleteReadBin (gen);
      strncpy (genName, filename, (size_t) LEN);
   } else {
      GetName (gen, genName);
   }

   /* Make sure to count the first run; set prevBit != {0, 1} */
   prevBit = 2;

   for (j = 0; j < SAMPLE; j++) {
      /* Count the number of 1 */
      Z = Bits[j];
      while (Z > 0) {
         Z &= Z - 1;                /* Clear lowest 1 bit */
         ++nbit;
      }

      /* Count the number of 4 bits values */
      Z = Bits[j];
      for (i = 0; i < 8; i++) {
         (ncount[Z & MASK4])++;
         Z >>= 4;
      }

      /* Count the number of runs and get the longest runs */
      Z = Bits[j];
      jBit = bitset_maskUL[31];

      while (jBit > 0) {
         if (Z & jBit) {            /* bit 1 */
            if (prevBit != 1) {
               if (len < 6)
                  (nrun0[len])++;
               else
                  (nrun0[6])++;
               if (len > longest0)
                  longest0 = len;
               len = 1;
            } else {
               len++;
            }
            prevBit = 1;

         } else {                   /* bit 0 */
            if (prevBit != 0) {
               if (len < 6)
                  (nrun1[len])++;
               else
                  (nrun1[6])++;
               if (len > longest1)
                  longest1 = len;
               len = 1;
            } else {
               len++;
            }
            prevBit = 0;
         }
         jBit >>= 1;
      }
   }

   strcpy (bbattery_TestNames[0], "Monobit");
   strcpy (bbattery_TestNames[1], "Poker");
   j = 1;
   strcpy (bbattery_TestNames[++j], "0 Runs, length 1: ");
   strcpy (bbattery_TestNames[++j], "0 Runs, length 2: ");
   strcpy (bbattery_TestNames[++j], "0 Runs, length 3: ");
   strcpy (bbattery_TestNames[++j], "0 Runs, length 4: ");
   strcpy (bbattery_TestNames[++j], "0 Runs, length 5: ");
   strcpy (bbattery_TestNames[++j], "0 Runs, length 6+: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 1: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 2: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 3: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 4: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 5: ");
   strcpy (bbattery_TestNames[++j], "1 Runs, length 6+: ");

   strcpy (bbattery_TestNames[++j], "Longest run of 0: ");
   strcpy (bbattery_TestNames[++j], "Longest run of 1: ");

   WriteReportFIPS_140_2 (genName, fileFlag, nbit, longest0, longest1,
      nrun0, nrun1, ncount);
}


/*-------------------------------------------------------------------------*/

void bbattery_FIPS_140_2 (unif01_Gen * gen)
{
   FIPS_140_2 (gen, NULL);
}


/*-------------------------------------------------------------------------*/

void bbattery_FIPS_140_2File (char *filename)
{
   FIPS_140_2 (NULL, filename);
}


/*=========================================================================*/
```

APPENDIX B

Source code for TestU01 extension including MAKE files

```
//mcorr.cpp
#include "./tntjama/tnt.h"
using namespace TNT;
#include "kendall2.c"
#include "mcorr.h"
#include <cmath>
using std::cout;
using std::endl;
using std::ios;
using std::swap;

//Constructor
mcorr::mcorr(int N, int P, TNT::Array2D <long double> Mat, double Alpha ){
    cout << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl
         << "                 Starting Multivariate Extension" << endl
         << "                 Version: TestU01 1.2.3" << endl
         << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl << endl << endl;

  if(N >= 1 && P >= 1){
    n = N;
    p = P;
    mat = Mat;
    numCorrs = mcorr_binomCoef(p, 2);
    alpha = Alpha;
    corrData = TNT::Array2D <long double>(numCorrs, 3);
    pVals = TNT::Array2D <long double>(numCorrs, 3);
    Z = TNT::Array2D <long double>(numCorrs, 3);
    Ranks = TNT::Array2D <long double>(n, p);
  }
  else
    std::cout << "Number of rows and columns must be positive integers."
              << std::endl;
};

//Private method definitions
unsigned int mcorr::mcorr_binomCoef(unsigned int N, unsigned int K){
  if(K == 0 || K == N)
    return 1;
  else
    return (N*mcorr_binomCoef(N - 1, K - 1))/K;
}

void mcorr::mcorr_quickSort(TNT::Array2D<long double> matrix,
                            int cols, int left, int right) {
      int i = left, j = right;
      long double pivot = matrix[(left + right) / 2][0];

      // Partitioning
      while (i <= j) {
        while (matrix[i][0] < pivot)
          i++;
        while (matrix[j][0] > pivot)
          j--;
        if (i <= j) {
          std::swap(matrix[i][0], matrix[j][0]);
          for(int k = 1; k < cols; k++){
            std::swap(matrix[i][k], matrix[j][k]);
          }
          i++;
          j--;
        }
      }

      // Recursive calls
      if (left < j)
        mcorr_quickSort(matrix, cols, left, j);
      if (i < right)
        mcorr_quickSort(matrix, cols, i, right);
}

//Public method definitions
void mcorr::mcorr_corr(TNT::Array2D <long double> matrix){
    int count = 0;
    TNT::Array1D <long double> mean(p, 0.0);
    TNT::Array2D <long double> Cov(p, p, 0.0);

     // Two pass algorithm
    //Calculate the means of each column
    for(int j = 0; j < p; j++){
      for (int i = 0; i < n; i++)
        mean[j] += mat[i][j];
      mean[j] /= n;
    }
    //Compute covariance matrix
    for(int k = 0; k < p; k++){
      for(int j = 0; j < p; j++){
        for (int i = 0; i < n; i++)
          Cov[k][j] += (mcorr::mat[i][j] - mean[j])
            *(mcorr::mat[i][k] - mean[k]);
        Cov[k][j] /= (n - 1);
      }
    }

    //Compute the Pearson correlation coefficient matrix
      //(only bottom triangular to save space and time)
    for(int k = 0; k < mcorr::p; k++)
      for(int j = 0; j < k; j++){
        if(Cov[k][k]*Cov[j][j] == 0.0 && Cov[k][j] >= 0)
          mcorr::corrData[count][0] = 1.0000;
        else if(Cov[k][k]*Cov[j][j] == 0.0 && Cov[k][j] < 0)
          mcorr::corrData[count][0] = -1.0000;
        else
          mcorr::corrData[count][0] = Cov[k][j]/sqrt(Cov[k][k]*Cov[j][j]);

        mcorr::corrData[count][1] = k;
        mcorr::corrData[count][2] = j;
        count++;
      }
}

void mcorr::mcorr_fisherTrans(int type){
  for(int i = 0; i < mcorr::numCorrs; i++){
    mcorr::Z[i][0] = sqrt( (mcorr::n - 3)/1.06 ) * 0.5
      * log( (1.0 + mcorr::corrData[i][0])
             / (1.0 - mcorr::corrData[i][0]) );
    if (type == 1)
```

```
      mcorr::Z{i][0] *= sqrt(1.06)
    mcorr::Z[i][1] = mcorr::corrData[i][1];
    mcorr::Z[i][2] = mcorr::corrData[i][2];
  }
}

void mcorr::mcorr_getPVals(){
  for(int i = 0; i < mcorr::numCorrs; i++){
    //Two tailed P-value from Z test
    mcorr::pVals[i][0] = erfc(fabs(mcorr::Z[i][0]) / sqrt(2) );
    mcorr::pVals[i][1] = mcorr::Z[i][1];
    mcorr::pVals[i][2] = mcorr::Z[i][2];
  }
}

int mcorr::mcorr_BHY(){
  double q;
  int k;
  sortedPVals = TNT::Array2D <long double> (mcorr::numCorrs, 4);
  double tempSum;

  //Arrange P-values in ascending order
  mcorr_quickSort(mcorr::pVals, 3, 0, mcorr::numCorrs - 1);

  //Prepare rankings
  for(int i = 0; i < mcorr::numCorrs; i++){
    sortedPVals[i][0] = i;
    for(int j = 0; j < 3; j++)
      sortedPVals[i][j+1] = mcorr::pVals[i][j];
  }

  //Define q
  for(double j = 1.0; j <= mcorr::numCorrs; j++)
    tempSum += 1/j;
  q = alpha/tempSum;

  //Compute k
  k = 0;
  for(double i = 1.0; i <= mcorr::numCorrs; i++){
    if(sortedPVals[i-1][1] <= q*(long double)(i/mcorr::numCorrs))
      k++;
  }

  return k;
}


void mcorr::mcorr_spearman(){
  TNT::Array1D<long double> rankings(n, 0.0);
  TNT::Array2D<long double> temp(n, 3, 0.0);
  int j, ji, jt;
  double rank;

  for(int k = 0; k < p; k++){
    for(int i = 0; i < n; i++){
      temp[i][0] = mcorr::mat[i][k];
      temp[i][1] = i; //Original position (to sort by later)
    }
    j = 1;
    //Sort by first column and copy sorted list to rankings
    mcorr_quickSort(temp, 2, 0, n - 1);
    for(int i = 0; i < n; i++)
      rankings[i] = temp[i][0];


    //Rank the sorted vector (including midranks for ties)
    while (j < n) {
      if (rankings[j] != rankings[j - 1]) { //Not a tie.
        rankings[j-1] = j;
        ++j;
      }
      else { //A tie:
        for (jt = j + 1; jt <= n
              && rankings[jt - 1] == rankings[j - 1]; jt++);
        rank = 0.5 * (j + jt - 1); //Mean rank of the tie
        for (ji = j; ji <= (jt - 1); ji++)
          //Enter mean rank into all tied entries
          rankings[ji - 1] = rank;
        j = jt;
      }
    }
    //If the last element was not tied, this is its rank
    if (j == n)
      rankings[n - 1] = n;

    //Swap first column and second column and insert
    //rankings as third column of temp
    for(int i = 0; i < n; i++){
      temp[i][2] = rankings[i];
      swap(temp[i][0], temp[i][1]);
    }

    //Sort by original position
    mcorr_quickSort(temp, 3, 0, n - 1);

    //Place each rank column into the rank matrix that will be passed
    //into the Pearson correlation function
    for(int i = 0; i < n; i++)
      mcorr::Ranks[i][k] = temp[i][2];
  }

  //Compute Spearman correlation matrix (only bottom triangular)
  mcorr::mcorr_corr(Ranks);

}


void mcorr::mcorr_kendall(){
  //Since kendallNlogN requires floating pointers
  double* arr1 = new double[n];
  double* arr2 = new double[n];

  int m = 0;
  TNT::Array2D <long double> newtemp(n, 2, 0.0);
   for(int k = 0; k < p; k++)
     for(int j = 0; j < k; j++){
        for(int i = 0; i < n; i++){
          newtemp[i][0] = mcorr::mat[i][k];
          newtemp[i][1] = mcorr::mat[i][j];
        }

        //Sort in lockstep by column 1
        mcorr_quickSort(newtemp, 2, 0, n - 1);

        for(int i = 0; i < n; i++){
          arr1[i] = newtemp[i][0];
          arr2[i] = newtemp[i][1];
        }

        mcorr::corrData[m][0] = kendallNlogN(arr1, arr2, n, 1);
        mcorr::corrData[m][1] = k;
        mcorr::corrData[m][2] = j;
        m++;
      }
  delete[] arr1;
  delete[] arr2;
}

void mcorr::mcorr_kendallNormal(){
  for(int i = 0; i < mcorr::numCorrs; i++){
    mcorr::Z[i][0] = mcorr::corrData[i][0] /
      sqrt( (2.0* (2.0*n + 5.0)) / (9.0 * n * (n - 1) ) );
    mcorr::Z[i][1] = mcorr::corrData[i][1];
    mcorr::Z[i][2] = mcorr::corrData[i][2];
  }
}

void mcorr::mcorr_pairCorr(int corrType){
  int k = 0;

  cout << "————————————————————————————————" << endl
       << "PairCorr test ";
  if (corrType == 0)
    cout << "for Pearson Correlations:" << endl;
  else if (corrType == 1)
    cout << "for Spearman Correlations:" << endl;
  else
    cout << "for Kendall Correlations:" << endl;
  cout << "————————————————————————————————" << endl
       << "   n =  " << n << ",  p = " << p << endl << endl << endl;
  if (corrType == 0)
    mcorr::mcorr_corr(mat);
  else if (corrType == 1)
    mcorr::mcorr_spearman();
  else
    mcorr::mcorr_kendall();
  if (corrType == 0){
    //Transform Pearson correlations into normals using the
    //Fisher r to z Transform
    mcorr::mcorr_fisherTrans(1);
  }
  else if (corrType == 1){
    //Transform Spearman correlations into normals using the
    //Fisher r to z Transform
    mcorr::mcorr_fisherTrans(0);
  }
  else{
    //Transform Kendall correlations into normals
    mcorr::mcorr_kendallNormal();
  }

  mcorr::mcorr_getPVals();
  k = mcorr::mcorr_BHY();

  cout << "————————————————————————————————————————" << endl;
  cout << "Test results using Benjamini/Hochberg/Yekutieli" << endl;
  if (corrType == 0)
    cout << "for Pearson Correlations:" << endl;
  else if (corrType == 1)
    cout << "for Spearman Correlations:" << endl;
  else
    cout << "for Kendall Correlations:" << endl;
  cout << "Alpha = " << alpha << endl << endl;

  if(k == 0){
    cout << "Reject none of the null hypotheses"
         << endl << endl << endl;
    return;
  }

  cout << "Reject the null hypotheses of nonzero correlation "
       << "corresponding to" << endl;
  for(int i = 0; i < k; i++){
    cout << "P–value_(" << i + 1 << "):  Vector "
         << (int)(sortedPVals[i][3] + 1)
         << " and Vector " << (int)(sortedPVals[i][2] + 1) << endl;
  }
  cout << endl << endl;

}
```

```
//mcorr.h
#ifndef mcorr_H
#define mcorr_H

#include "./tntjama/tnt.h"
using namespace TNT;

class mcorr{
  //private class methods
  unsigned int mcorr_binomCoef(unsigned int N,
                               unsigned int K);
  void mcorr_quickSort(TNT::Array2D<long double> matrix,
                       int cols, int left, int right);

 public:
  //public class members
  int n; //number of rows of pseudorandom data
  int p; //number of columns of pseudorandom data
  TNT::Array2D <long double> mat;
  int numCorrs;
  double alpha;
  TNT::Array2D <long double> corrData;
  TNT::Array2D <long double> pVals;
  TNT::Array2D <long double> sortedPVals;
  TNT::Array2D <long double> Z;
  TNT::Array2D <long double> Ranks;

  //constructor & destructor
  mcorr(int N, int P, TNT::Array2D <long double> Mat,
        double Alpha);
  ~mcorr(){};

  //public class methods
  void mcorr_corr(TNT::Array2D <long double> matrix);
  int mcorr_BHY();
  void mcorr_fisherTrans(int type);
  void mcorr_getPVals();
  void mcorr_spearman();
  void mcorr_kendall();
  void mcorr_kendallNormal();
   //corrType -> 0 = Pearson, 1 = Spearman, 2 = Kendall
  void mcorr_pairCorr(int corrType);

};
#endif
```

```
//mmult.cpp
#include "./tntjama/tnt.h"
#include "./tntjama/jama_lu.h"
using namespace TNT;
#include "mmult.h"
#include <cmath>
using std::cout;
using std::endl;
using std::ios;
using std::swap;

//Constructor
mmult::mmult(int N, int P, TNT::Array2D <long double> Mat,
             double Alpha ){
    cout << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl
         << "                Starting Multivariate Extension" << endl
         << "                Version: TestU01 1.2.3" << endl
         << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl << endl << endl;

  if(N >= 1 && P >= 1){
    n = N;
    p = P;
    mat = Mat;
    alpha = Alpha;
    normMat = TNT::Array2D <long double> (N, P, 0.0);
    A = TNT::Array2D <long double> (P, P, 0.0);
    C = TNT::Array2D <long double> (A.dim1(), A.dim2(), 0.0);
    obsTS = 0.0;
    pValue = 0.0;
  }
  else
    std::cout << "Number of rows and columns must be positive integers."
              << std::endl;
};

//Private method definitions
//Determine trace of inputted square matrix
long double mmult::trace(TNT::Array2D <long double> A){
  long double matTrace = 0.0;
  for(int i = 0; i < A.dim1(); i++)
    matTrace += A[i][i];
  return matTrace;
}

//Multiply matrix by a constant
TNT::Array2D <long double> mmult::multConst (TNT::Array2D <long double> A,
                                              long double b){
  TNT::Array2D <long double> newMat (A.dim1(), A.dim2(), 0.0);
  for(int i = 0; i < newMat.dim1(); i++)
    for(int j = 0; j < newMat.dim2(); j++)
      newMat[i][j] = b * A[i][j];

  return newMat;
}

//Copy matrix row into Array1D
template<class T>
```

```
TNT::Array1D<T> mmult::copyRowToVec(const TNT::Array2D<T> &M, int rowNum){
  TNT::Array1D<T> vec(M.dim2());
  for(int c = 0; c < M.dim2(); ++c)
    vec[c] = M[rowNum][c];
  return vec;
}

//Computes an outer product of two inputted vectors
template <class T>
TNT::Array2D<T> mmult::outerProd(const TNT::Array1D<T> &v,
                                 const TNT::Array1D<T> &v2){
  //declare variable to store matrix
  TNT::Array2D<T> outerMat(v.dim(), v2.dim(), 0.0);

  //multiply components in vector
  for(int i = 0; i < v.dim(); i++)
    for(int j = 0; j < v2.dim(); j++)
      outerMat[i][j] = v[i] * v2[j];

  //return answer
  return outerMat;
}

//Public method definitions
void mmult::mmult_LRT(){
  //Convert uniform (0, 1) deviate matrix to normal(0, 1) deviates
  for(int i = 0; i < mat.dim1(); i++)
    for(int j = 0; j < mat.dim2(); j++)
      normMat[i][j] = normal_01_cdf_inv (mat[i][j]); //from prob.cpp

  //Compute A matrix
  for(int i = 0; i < n; i++){
    TNT::Array1D <long double> temp1(normMat.dim2(), 0.0);
    temp1 = copyRowToVec(normMat, i);
    A += outerProd(temp1, temp1);
  }

  //Muirhead calculations using C = n^{-1} A
  C = multConst(A, (long double) (1.0 / (long double) n));

  JAMA::LU<long double> luC(C);

  obsTS = n * (trace(C) - log(luC.det()) - p);

  cout << "--------------------------------------------------" << endl
       << "Likelihood Ratio Test for " << endl
       << "     Pairwise Correlation Matrix = Identity:" << endl
       << "Alpha = " << alpha << endl << endl
       << "--------------------------------------------------" << endl
       << "   n =  " << n << ",  p = " << p << endl << endl << endl;

  cout << "--------------------------------------------------" << endl
       << "LR Test Statistic                :  " << obsTS << endl
       << "p-value of test                  :  ";

  if(!std::isnan(obsTS))
    pValue = 1 - chi_square_cdf(obsTS, p * (p + 1.0) / 2.0) ;
  cout << pValue << endl << endl;
}
```

```
//mmult.h
#ifndef mmult_H
#define mmult_H

#include "./prob/prob.hpp"

class mmult{
  //private class methods
  long double trace (TNT::Array2D <long double> A);
  TNT::Array2D <long double> multConst (TNT::Array2D <long double> A,
                                        long double b);
  template<class T> TNT::Array1D<T>
    copyRowToVec(const TNT::Array2D<T> &M, int rowNum);
  template<class T> TNT::Array2D<T>
    outerProd(const TNT::Array1D<T> &v, const TNT::Array1D<T> &v2);

 public:
  //public class members
  int n; //number of rows of pseudorandom data
  int p; //number of columns of pseudorandom data
  TNT::Array2D <long double> mat; //inputted matrix
  double alpha; //significance level
  TNT::Array2D <long double> normMat;  //normalized input
  TNT::Array2D <long double> A; //related to covariance matrix
  TNT::Array2D <long double> C; //n^{-1} A
  long double obsTS; //observed likelihood ratio test statistic
  double pValue; //corresponding p-value

  //constructor & destructor
  mmult(int N, int P, TNT::Array2D <long double> Mat, double Alpha);
  ~mmult(){};

  //public class methods
  void mmult_LRT();

};
#endif
```

```
//mport.cpp
#include "./tntjama/jama_cholesky.h"
#include "./tntjama/jama_lu.h"
#include "./tntjama/jama_qr.h"
#include "./tntjama/tnt.h"
#include "mport.h"
#include <iomanip>
#include <cmath>
using std::cout;
using std::endl;
using std::ios;
using std::setw;

//Constructor
mport::mport(int N, int P, TNT::Array2D <long double> Mat,
             long double Alpha){
    cout << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl
         << "                 Starting Multivariate Extension" << endl
         << "                 Version: TestU01 1.2.3" << endl
         << "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
         << endl << endl << endl;

  if(N >= 1 && P >= 1){
    n = N;
    p = P;
    mat = Mat;
    alpha = Alpha;
    lagOrder = 20;  //Specify default value if none given
  }
  else
    std::cout << "Number of rows and columns must be positive integers."
              << std::endl;
};


//Private method definitions

//Creates identity matrix
TNT::Array2D<long double> mport::identity(int size){
  TNT::Array2D <long double> iMat(size, size, 0.0);
  for(int j = 0; j < iMat.dim1(); j++)
    iMat[j][j] = 1.0;
  return iMat;
}

//From http://wiki.cs.princeton.edu/index.php/TNT
//Compute transpose of a matrix
template<class T>
TNT::Array2D<T> mport::transpose(const TNT::Array2D<T> &M)
{
  TNT::Array2D<T> tran(M.dim2(), M.dim1() );
  for(int r = 0; r < M.dim1(); ++r)
    for(int c = 0; c < M.dim2(); ++c)
      tran[c][r] = M[r][c];
  return tran;
}

//Get diagonal values of matrix in 1D vector
```

```
template<class T>
TNT::Array1D<T> mport::diag(const TNT::Array2D<T> &M)
{
  TNT::Array1D<T> diagVec(M.dim1(), 0.0);
  for(int r = 0; r < M.dim1(); r++)
    diagVec[r] = M[r][r];
  return diagVec;
}


//Computes an outer product of two inputted vectors
template <class T>
TNT::Array2D<T> mport::mport_outerProd(const TNT::Array1D<T> &v,
                                       const TNT::Array1D<T> &v2)
{
  //declare variable to store matrix
  TNT::Array2D<T> outerMat(v.dim(), v2.dim(), 0.0);

  //multiply components in vector
  for(int i = 0; i < v.dim(); i++)
    for(int j = 0; j < v2.dim(); j++)
      outerMat[i][j] = v[i] * v2[j];

  //return answer
  return outerMat;
}


//Added from http://wiki.cs.princeton.edu/index.php/TNT
//Invert matrix
TNT::Array2D <long double>
mport::invert(const TNT::Array2D<long double> &M){
  assert(M.dim1() == M.dim2()); // square matrices only please

  // solve for inverse with LU decomposition
  JAMA::LU<long double> lu(M);

  // create identity matrix
  TNT::Array2D<long double> id(M.dim1(), M.dim2(), 0.0);
  for (int i = 0; i < M.dim1(); i++)
    id[i][i] = 1;

  // solves A * A_inv = Identity
  return lu.solve(id);
}

//Mean center inputted matrix
void mport::mport_centerMat (){
  TNT::Array2D <long double> meanMat;
  centMat = TNT::Array2D <long double> (mat.dim1(),
                                        mat.dim2(), 0.0);

  meanMat = TNT::Array2D <long double> (1, mat.dim2(), 0.0);
  //Means of columns of matrix
  for(int j = 0; j < mat.dim2(); j++){
    for(int i = 0; i < mat.dim1(); i++)
      meanMat[0][j] += mat[i][j];
    meanMat[0][j] /= mat.dim1();
  }
  TNT::Array2D <long double> repmat (mat.dim1(),
                                     mat.dim2(), 0.0);
  for(int i = 0; i < repmat.dim1(); i++)
    for(int j = 0; j < repmat.dim2(); j++)
      repmat[i][j] = meanMat[0][j];

  //Mean center
  centMat = mat - repmat;

}

//Used to construct Toeplitz Block matrix
//(matrix of matrices for Mahdi-McLeod test)
void mport::mport_blockFill(long double** pIn,
                            TNT::Array2D <long double> pToFill,
                            int rowLoc, int colLoc, int blkSize){
  //put in the blkSize-by-blkSize array in the
  //(rowLoc, colLoc) block
  if(colLoc > rowLoc) //top triangle
    for(int i = 0; i < blkSize; i++)
      for(int j = 0; j < blkSize; j++)
        pToFill[i + blkSize*rowLoc][j + blkSize*colLoc]
          = pIn[i][j];
  else //otherwise put in the transpose (bottom triangle)
    for(int i = 0; i < blkSize; i++)
      for(int j = 0; j < blkSize; j++)
        pToFill[i + blkSize*rowLoc][j + blkSize*colLoc]
          = pIn[j][i];
}

//Hosking and Li-McLeod tests
void mport::mport_portmanteauTests(int lagOrder){
  TNT::Array2D <long double> c0(p, p, 0.0);
  TNT::Array2D <long double> c0inv(c0.dim1(), c0.dim2(), 0.0);
  TNT::Array1D <long double> d;
  TNT::Array2D <long double> dd;
  TNT::Array2D <long double> L;
  TNT::Array2D <long double> matcTemp1;
  TNT::Array2D <long double> matcTemp2;
  TNT::Array2D <long double> tempMult;
  TNT::Array3D <long double> cl(lagOrder, p, p, 0.0);
  //Li-McLeod Portmanteau observed test statistic
  long double lmp = 0.0;
  //Vectorized/Kronecker product used in Hosking and LM
  long double impCalc = 0.0;
  long double tempDiv = 0.0;
  long double hoskTemp = 0.0;
  TNT::Array2D <long double> vec_cl(p * p, 1, 0.0);
  TNT::Array2D <long double> innerMult;
  double pValue_lmp = 0.0;
  //Hosking Portmanteau observed test statistic
  long double hosk = 0.0;
  double pValue_hosk = 0.0;

  //Compute lag zero correlation matrix
  c0 = matmult(transpose(centMat), centMat);
  d = diag(c0);
  dd = mport_outerProd(d, d);
```

```
for(int i = 0; i < dd.dim1(); i++)
  for(int j = 0; j < dd.dim2(); j++)
    dd[i][j] = sqrt(dd[i][j]);

for(int i = 0; i < dd.dim1(); i++)
  for(int j = 0; j < dd.dim2(); j++)
    c0[i][j] /= dd[i][j];

c0inv = invert(c0);

//Compute lag el (el = 0, ... lagOrder - 1)
//correlation matrices and store in 3D array
for(int el = 0; el < cl.dim1(); el++){
  matcTemp1 = TNT::Array2D <long double> (n - el - 1,
                                          centMat.dim2(), 0.0);
  matcTemp2 = TNT::Array2D <long double> (n - el - 1,
                                          centMat.dim2(), 0.0);
  for(int r = 0; r < matcTemp1.dim1(); r++)
    for(int c = 0; c < matcTemp1.dim2(); c++)
      matcTemp1[r][c] = centMat[r][c];

  for(int r = 0; r < matcTemp2.dim1(); r++)
    for(int c = 0; c < matcTemp2.dim2(); c++)
      matcTemp2[r][c] = centMat[r + el + 1][c];

  tempMult = matmult(transpose(matcTemp1), matcTemp2);
  for(int r = 0; r < cl.dim2(); r++)
    for(int c = 0; c < cl.dim3(); c++)
      cl[el][r][c] = tempMult[r][c];

  for(int i = 0; i < dd.dim1(); i++)
    for(int j = 0; j < dd.dim2(); j++)
      cl[el][i][j] /= dd[i][j];
}

//Compute covariance matrix in Hosking & Li-McLeod statistics

//Compute Kronecker product of c0inv with itself
TNT::Array2D <long double> rr(c0inv.dim1() * c0inv.dim1(),
                              c0inv.dim2() * c0inv.dim2(),
                              0.0);
for(int r = 0; r < c0inv.dim1(); r++)
  for(int c = 0; c < c0inv.dim2(); c++)
    for(int i = 0; i < c0inv.dim1(); i++)
      for(int j = 0; j < c0inv.dim2(); j++)
        rr[r * c0inv.dim1() + i][c * c0inv.dim2() + j]
          = c0inv[r][c] * c0inv[i][j];

//Vectorize cl matrices
for(int el = 0; el < cl.dim1(); el++){
  for(int i = 0; i < cl.dim2(); i++)
    for(int j = 0; j < cl.dim3(); j++)
      vec_cl[i + j * cl.dim2()][0] = cl[el][i][j];

  innerMult = matmult(matmult(transpose(vec_cl), rr),
                      vec_cl);
  impCalc += innerMult[0][0];
  tempDiv = innerMult[0][0] / (long double) (n - el - 1);
```

```
    hoskTemp += tempDiv;
  }

  //Li—McLeod test results
  lmp = n * impCalc + p * p * lagOrder * (lagOrder + 1)
    / (long double) (2.0 * n);
  if(std::isnan(lmp)){
    cout << "Li—McLeod Test Statistic is infinite.  "
        << "Program exiting." << endl;
    return;
  }
  df = p * p * lagOrder; //modelOrder = 0
  pValue_lmp = 1 − gamma_inc( (double) df / 2.0,
                              (double) lmp / 2.0);

  cout << "———————————————————————————————————————————————"
       << endl
       << "Portmanteau Test for White Noise (Li—McLeod):"
       << endl
       << "Alpha = " << alpha << endl << endl
       << "———————————————————————————————————————————————"
       << endl
       << "   n =  " << n << ",  p = " << p
       << ",  Lag order = " << lagOrder
       << endl << endl << endl;

 cout << "———————————————————————————————————————————————"
      << endl
      << "Li—McLeod Test Statistic            :  "
      << lmp << endl
      << "Degrees of Freedom                  :  "
      << df << endl
      << "p—value of test                     :  "
      << pValue_lmp << endl << endl << endl;

  //Hosking test results
  hosk = n * n * hoskTemp;
  if(std::isnan(hosk)){
    cout << "Hosking Test Statistic is infinite. "
         << "Program exiting." << endl;
    return;
  }
  pValue_hosk = 1 − gamma_inc ( (double) df / 2.0,
                                (double) hosk / 2.0);

  cout << "———————————————————————————————————————————————"
       << endl
       << "Portmanteau Test for White Noise (Hosking):"
       << endl
       << "Alpha = " << alpha << endl << endl
       << "———————————————————————————————————————————————"
       << endl
       << "   n =  " << n << ",  p = " << p
       << ",  Lag order = " << lagOrder
       << endl << endl << endl;

 cout << "———————————————————————————————————————————————"
      << endl
      << "Hosking Test Statistic              :  "
      << hosk << endl
      << "Degrees of Freedom                    :  "
      << df << endl
      << "p-value of test                       :  "
      << pValue_hosk << endl << endl << endl;
}

//Mahdi-McLeod test
void mport::mport_mahdiMcLeod(int lagOrder){
  TNT::Array2D <long double> cov0(p, p, 0.0);
  TNT::Array2D <long double> cov0inv(cov0.dim1(),
                                     cov0.dim2(), 0.0);
  TNT::Array2D <long double> matcTemp1;
  TNT::Array2D <long double> matcTemp2;
  TNT::Array2D <long double> tempMult;
  TNT::Array3D <long double> covl(lagOrder, p, p, 0.0);
  TNT::Array3D <long double> Rl(lagOrder, p, p, 0.0);
  TNT::Array2D <long double> mahdiMat(Rl.dim2()
                                      * (lagOrder + 1), Rl.dim3()
                                      * (lagOrder + 1), 0.0);
  long double gv = 0.0;
  long double df_gv = 0.0;
  double pValue_gv = 0.0;

  //Compute lag zero covariance matrix
  cov0 = matmult(transpose(centMat), centMat);

  for(int i = 0; i < cov0.dim1(); i++)
    for(int j = 0; j < cov0.dim2(); j++)
      cov0[i][j] /= (long double) n;

  // LL' = cov0^{-1}
  cov0inv = invert(cov0);
  JAMA::Cholesky<long double> chol(cov0inv);
  TNT::Array2D <long double> L = chol.getL();

  //Compute lag el covariance matrix
  for(int el = 0; el < covl.dim1(); el++){
    matcTemp1 = TNT::Array2D <long double> (n - el - 1,
                                            centMat.dim2(), 0.0);
    matcTemp2 =  TNT::Array2D <long double> (n - el - 1,
                                             centMat.dim2(), 0.0);
    for(int r = 0; r < matcTemp1.dim1(); r++)
      for(int c = 0; c < matcTemp1.dim2(); c++)
        matcTemp1[r][c] = centMat[r][c];

    for(int r = 0; r < matcTemp2.dim1(); r++)
      for(int c = 0; c < matcTemp2.dim2(); c++)
        matcTemp2[r][c] = centMat[r + el + 1][c];

    tempMult = matmult(transpose(matcTemp1), matcTemp2);
    for(int r = 0; r < covl.dim2(); r++)
      for(int c = 0; c < covl.dim3(); c++)
        covl[el][r][c] = tempMult[r][c];


    for(int i = 0; i < covl.dim2(); i++)
      for(int j = 0; j < covl.dim3(); j++)
        covl[el][i][j] /= (long double) n;
  }

  //Rl = L' * covl * L (matrix multiplication)
  for(int el = 0; el < lagOrder; el++){
    TNT::Array2D <long double> tempL(p, p, 0.0);
    TNT::Array2D <long double> tempL2(p, p, 0.0);
    for(int i = 0; i < Rl.dim2(); i++)
      for(int j = 0; j < Rl.dim3(); j++)
        tempL[i][j] = covl[el][i][j];
    tempL2 = transpose(matmult(matmult(transpose(L), tempL), L));
    for(int i = 0; i < Rl.dim2(); i++)
      for(int j = 0; j < Rl.dim3(); j++)
        Rl[el][i][j] = tempL2[i][j];
  }

  //Initialize diagonal to 1
  for(int i = 0; i < mahdiMat.dim1(); i++)
    for(int j = 0; j < mahdiMat.dim2(); j++){
      if(i == j)
        mahdiMat[i][j] = 1.0;
      else
        mahdiMat[i][j] = 0.0;
    }

  //Fill the upper triangle of the matrix of matrices
  for(int i = 0; i < lagOrder + 1; i++)
    for(int j = i + 1; j < lagOrder + 1; j++)
      mport_blockFill(Rl[j - i - 1], mahdiMat, i, j, Rl.dim2());

  //now fill the lower part
  for(int i = 1; i < lagOrder + 1; i++)
    for(int j = 0; j < i; j++)
      mport_blockFill(Rl[i - j - 1], mahdiMat, i, j, Rl.dim2());

  JAMA::LU<long double> lu(mahdiMat);

  //Generalized Variance test results
  gv = (-3.0 * n) / (2.0 * lagOrder + 1.0) * log(lu.det());
  if(std::isnan(gv)){
    cout << "GV Test Statistic is infinite.  "
         << "Program exiting." << endl;
    return;
  }
  df_gv = p * p * (1.5 * lagOrder * (lagOrder + 1)
                   / (2.0 * lagOrder + 1.0) ); //modelOrder = 0
  pValue_gv = 1 - gamma_inc ( (double) df_gv / 2.0,
                              (double) gv / 2.0);

  cout << "----------------------------------------------------"
       << endl
       << "Portmanteau Test for White Noise (Mahdi-McLeod):"
       << endl
       << "Alpha = " << alpha << endl << endl
       << "----------------------------------------------------"
       << endl
       << "   n =  " << n << ",  p = " << p
       << ",  Lag order = " << lagOrder
       << endl << endl << endl;
  cout << "———————————————————————————————————"
       << endl
       << "Determinant of Toeplitz Block matrix:  "
       << lu.det() << endl
       << "Mahdi–McLeod Test Statistic         :  "
       << gv << endl
       << "Degrees of Freedom                  :  "
       << df_gv << endl
       << "p–value of test                     :  "
       << pValue_gv << endl << endl << endl;
}
```

```
//mport.h
#ifndef mport_H
#define mport_H

#include "./prob/prob.hpp"

class mport{
  //private class methods
  void mport_blockFill(long double** pIn,
                       TNT::Array2D <long double> pToFill,
                       int rowLoc, int colLoc, int blkSize);
  TNT::Array2D <long double> identity(int size);
  template<class T> TNT::Array2D<T>
    transpose(const TNT::Array2D<T> &M);
  template<class T> TNT::Array1D<T>
    diag(const TNT::Array2D<T> &M);
  template<class T> TNT::Array2D<T>
    mport_outerProd(const TNT::Array1D<T> &v,
                    const TNT::Array1D<T> &v2);
  TNT::Array2D <long double>
    invert(const TNT::Array2D<long double> &M);

 public:
  //public class members
  int n; //number of rows of pseudorandom data
  int p; //number of columns of pseudorandom data
  //inputted matrix of pseudorandom data
  TNT::Array2D <long double> mat;
  //mean centered matrix to test for white noise
  TNT::Array2D <long double> centMat;
  long double alpha;  //to compare p-values against
  //upper limit of summation in test statistics
  int lagOrder;
  int df; //degrees of freedom for Hosking & Li-McLeod tests
  int mahdiDf; //degrees of freedom for Mahdi-McLeod test

  //constructor & destructor
  mport(int N, int P, TNT::Array2D <long double> Mat,
        long double Alpha);
  ~mport(){};

  //public class methods
  //mean center inputted matrix values
  void mport_centerMat();
  //Li-McLeod and Hosking Tests
  void mport_portmanteauTests(int lagOrder);
  //Mahdi-McLeod test
  void mport_mahdiMcLeod(int lagOrder);

};
#endif
```

```
//extDriver_smokGun.cpp
//Testing the first Smoking Gun generator
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "./tntjama/tnt.h"
#include "mcorr.h"
#include "mmult.h"
#include "mport.h"
#include "./prob/prob.cpp"
using std::cout;
using std::endl;
using std::ifstream;
using std::setw;

int main(){
  int n = 10000, p = 10;
  TNT::Array2D <long double> mat(n, p);
  double Alpha;

  std::ifstream fin;
  fin.open("vma1_2.txt");

  if(!fin.is_open()){
    std::cout<< "Error opening input file!" << std::endl;
    exit(1);
  }

  //Input file one row after another
  for(int i = 0; i < n; i++)
    for(int j = 0; j < p; j++)
      fin >> mat[i][j];

  fin.close();

  Alpha = 0.01;

  mcorr mcorr1 = mcorr(n, p, mat, Alpha);

  //Pearson
  mcorr1.mcorr_pairCorr(0);
  //Spearman
  mcorr1.mcorr_pairCorr(1);
  //Kendall
  mcorr1.mcorr_pairCorr(2);

  //Testing if correlation matrix = identity
  mmult mmult1 = mmult(n, p, mat, Alpha);
  mmult1.mmult_LRT();

  //Portmanteau tests for white noise with lag = p
  mport mport1 = mport(n, p, mat, Alpha);
  mport1.mport_centerMat();
  mport1.mport_portmanteauTests(p);
  mport1.mport_mahdiMcLeod(p);

  return 0;
}
```

MAKE file for testing first Smoking Gun

```
ext_smokGun : extDriver_smokGun.o mcorr.o mmult.o mport.o
        g++ -Wall extDriver_smokGun.o mcorr.o mmult.o mport.o -o ext_smokGun

mcorr.o : mcorr.cpp mcorr.h
        g++ -c -Wall mcorr.cpp

mmult.o : mmult.cpp mmult.h
        g++ -c -Wall mmult.cpp

mport.o : mport.cpp mport.h
        g++ -c -Wall mport.cpp

extDriver_smokGun.o : extDriver_smokGun.cpp mcorr.h mmult.h mport.h
        g++ -c -Wall extDriver_smokGun.cpp
```

```
//extDriver_warmBarA.cpp
//Testing the second Smoking Gun generator
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "./tntjama/tnt.h"
#include "mcorr.h"
#include "mmult.h"
#include "mport.h"
#include "./prob/prob.cpp"
using std::cout;
using std::endl;
using std::ifstream;
using std::setw;

int main(){
  int n = 10000, p = 10;
  TNT::Array2D <long double> mat(n, p);
  double Alpha;

  std::ifstream fin;
  fin.open("mvaTS.txt");

  if(!fin.is_open()){
    std::cout<< "Error opening input file!" << std::endl;
    exit(1);
  }

  //Input file one row after another
  for(int i = 0; i < n; i++)
    for(int j = 0; j < p; j++)
      fin >> mat[i][j];

  fin.close();

  Alpha = 0.01;

  mcorr mcorr1 = mcorr(n, p, mat, Alpha);

  //Pearson
  mcorr1.mcorr_pairCorr(0);
  //Spearman
  mcorr1.mcorr_pairCorr(1);
  //Kendall
  mcorr1.mcorr_pairCorr(2);

  //Testing if correlation matrix = identity
  mmult mmult1 = mmult(n, p, mat, Alpha);
  mmult1.mmult_LRT();

  //Portmanteau tests for white noise with lag = p
  mport mport1 = mport(n, p, mat, Alpha);
  mport1.mport_centerMat();
  mport1.mport_portmanteauTests(p);
  mport1.mport_mahdiMcLeod(p);

  return 0;
}
```

MAKE file for testing second Smoking Gun

```
ext_warmBarA : extDriver_warmBarA.o mcorr.o mmult.o mport.o
        g++ -Wall extDriver_warmBarA.o mcorr.o mmult.o mport.o -o ext_warmBarA

mcorr.o : mcorr.cpp mcorr.h
        g++ -c -Wall mcorr.cpp

mmult.o : mmult.cpp mmult.h
        g++ -c -Wall mmult.cpp

mport.o : mport.cpp mport.h
        g++ -c -Wall mport.cpp

extDriver_warmBarA.o : extDriver_warmBarA.cpp mcorr.h mmult.h mport.h
        g++ -c -Wall extDriver_warmBarA.cpp
```

```
//extDriver_warmBarB.cpp
//Testing the third Smoking Gun generator
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "./tntjama/tnt.h"
#include "mcorr.h"
#include "mmult.h"
#include "mport.h"
#include "./prob/prob.cpp"
using std::cout;
using std::endl;
using std::ifstream;
using std::setw;

int main(){
  int n = 10000, p = 10;
  TNT::Array2D <long double> mat(n, p);
  double Alpha;

  std::ifstream fin;
  fin.open("mvaTSa.txt");

  if(!fin.is_open()){
    std::cout<< "Error opening input file!" << std::endl;
    exit(1);
  }

  //Input file one row after another
  for(int i = 0; i < n; i++)
    for(int j = 0; j < p; j++)
      fin >> mat[i][j];

  fin.close();

  Alpha = 0.01;

  mcorr mcorr1 = mcorr(n, p, mat, Alpha);

  //Pearson
  mcorr1.mcorr_pairCorr(0);
  //Spearman
  mcorr1.mcorr_pairCorr(1);
  //Kendall
  mcorr1.mcorr_pairCorr(2);

  //Testing if correlation matrix = identity
  mmult mmult1 = mmult(n, p, mat, Alpha);
  mmult1.mmult_LRT();

  //Portmanteau tests for white noise with lag = p
  mport mport1 = mport(n, p, mat, Alpha);
  mport1.mport_centerMat();
  mport1.mport_portmanteauTests(p);
  mport1.mport_mahdiMcLeod(p);

  return 0;
}
```

MAKE file for testing third Smoking Gun

```
ext_warmBarB : extDriver_warmBarB.o mcorr.o mmult.o mport.o
        g++ -Wall extDriver_warmBarB.o mcorr.o mmult.o mport.o -o ext_warmBarA

mcorr.o : mcorr.cpp mcorr.h
        g++ -c -Wall mcorr.cpp

mmult.o : mmult.cpp mmult.h
        g++ -c -Wall mmult.cpp

mport.o : mport.cpp mport.h
        g++ -c -Wall mport.cpp

extDriver_warmBarB.o : extDriver_warmBarB.cpp mcorr.h mmult.h mport.h
        g++ -c -Wall extDriver_warmBarB.cpp
```

```
//extDriver_mers.cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "./tntjama/tnt.h"
#include "mcorr.h"
#include "mmult.h"
#include "mport.h"
#include "./prob/prob.cpp"
using std::cout;
using std::endl;
using std::ifstream;
using std::setw;

int main(){
  int n = 10000, p = 10;
  TNT::Array2D <long double> mat(n, p);
  double Alpha;

  std::ifstream fin;
  fin.open("mers.txt");

  if(!fin.is_open()){
    std::cout<< "Error opening input file!" << std::endl;
    exit(1);
  }

  //Input file one row after another
  for(int i = 0; i < n; i++)
    for(int j = 0; j < p; j++)
      fin >> mat[i][j];

  fin.close();

  Alpha = 0.01;

  mcorr mcorr1 = mcorr(n, p, mat, Alpha);

  //Pearson
  mcorr1.mcorr_pairCorr(0);
  //Spearman
  mcorr1.mcorr_pairCorr(1);
  //Kendall
  mcorr1.mcorr_pairCorr(2);

  //Testing if correlation matrix = identity
  mmult mmult1 = mmult(n, p, mat, Alpha);
  mmult1.mmult_LRT();

  //Portmanteau tests for white noise with lag = p
  mport mport1 = mport(n, p, mat, Alpha);
  mport1.mport_centerMat();
  mport1.mport_portmanteauTests(p);
  mport1.mport_mahdiMcLeod(p);

  return 0;
}
```

MAKE file for testing the Mersenne Twister

```
ext_mers : extDriver_mers.o mcorr.o mmult.o mport.o
        g++ -Wall extDriver_mers.o mcorr.o mmult.o mport.o -o ext_mers

mcorr.o : mcorr.cpp mcorr.h
        g++ -c -Wall mcorr.cpp

mmult.o : mmult.cpp mmult.h
        g++ -c -Wall mmult.cpp

mport.o : mport.cpp mport.h
        g++ -c -Wall mport.cpp

extDriver_mers.o : extDriver_mers.cpp mcorr.h mmult.h mport.h
        g++ -c -Wall extDriver_mers.cpp
```

```
//extDriver_MRG32k3a.cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include "./tntjama/tnt.h"
#include "mcorr.h"
#include "mmult.h"
#include "mport.h"
#include "./prob/prob.cpp"
extern "C"{
#include "gdef.h"
#include "unif01.h"
#include "ulec.h"
}
using std::cout;
using std::endl;
using std::ifstream;
using std::setw;

int main(){
  int n = 10000, p = 10;
  TNT::Array2D <long double> mat(n, p);
  double Alpha;

  unif01_Gen *gen;

  //MRG32k3a
  gen = ulec_CreateMRG32k3a (123., 123., 123., 123., 123., 123.);

  //Generate values one row after another
  for(int i = 0; i < n; i++)
    for(int j = 0; j < p; j++)
      mat[i][j] = unif01_StripD(gen, 0);

  ulec_DeleteGen (gen);
  Alpha = 0.01;

  mcorr mcorr1 = mcorr(n, p, mat, Alpha);

  //Pearson
  mcorr1.mcorr_pairCorr(0);
  //Spearman
  mcorr1.mcorr_pairCorr(1);
  //Kendall
  mcorr1.mcorr_pairCorr(2);

  //Testing if correlation matrix = identity
  mmult mmult1 = mmult(n, p, mat, Alpha);
  mmult1.mmult_LRT();

  //Portmanteau tests for white noise with lag = p
  mport mport1 = mport(n, p, mat, Alpha);
  mport1.mport_centerMat();
  mport1.mport_portmanteauTests(p);
  mport1.mport_mahdiMcLeod(p);

  return 0;
}
```

MAKE file for testing MRG32k3a

```
ext_MRG32k3a : extDriver_MRG32k3a.o mcorr.o mmult.o mport.o
        g++ -Wall extDriver_MRG32k3a.o mcorr.o mmult.o mport.o -o ext_MRG32k3a

mcorr.o : mcorr.cpp mcorr.h
        g++ -c -Wall mcorr.cpp

mmult.o : mmult.cpp mmult.h
        g++ -c -Wall mmult.cpp

mport.o : mport.cpp mport.h
        g++ -c -Wall mport.cpp

extDriver_MRG32k3a.o : extDriver_MRG32k3a.cpp mcorr.h mmult.h mport.h
        g++ -c -Wall extDriver_MRG32k3a.cpp
```