FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

FURTHER TESTU01 INTEGRATION INTO THE SPRNG LIBRARY

By

CHRISTOPHER DRAPER

A Project submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Computer Science Masters

2020

Christopher Draper defended this project on July 31, 2020.
The members of the supervisory committee were:

Michael Mascagni

Major Professor

Andy Wang

Committee Member

Sonia Haiduc

Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the project has been approved in accordance with university requirements.

TABLE OF CONTENTS

ABSTRACT

The SPRNG library is the Scalable Parallel Random Number Generator library created by M. Mascagni and A. Srinivasan in 2000. The purpose of the SPRNG library is to provide high quality randomly generated numbers, however the SPRNG library has also included a random number testing library that implements a number of the tests found proposed by Knuth and tests found in the DIEHARD library, along with implementing a number of physical model tests. However these tests have not seen much improvement over the different versions of the SPRNG library and newer and better implementations of these tests have been created. One such implementation is the TestU01 library, which was made to be an improvement over existing random number testing libraries. This masters project is focused on improving the SPRNG library's test suite by replacing it with the TestU01 test suite and to see if the improvements offered by TestU01 are worth including into the main version of SPRNG. The updated version of the library and data sets generated can be found at `https://github.com/chdraper16/SPRNGTestU01`.

CHAPTER 1

INTRODUCTION

In this paper I will go over a brief history and description of both the SPRNG library and the TestU01 library. Then, I will go through the motivations of their creation, and the motivations and goals of further integrating them together. Next I will describe the process of integrating the two libraries, along with going into detail about what parts of SPRNG could be upgraded to the TestU01 tests and what could not. After this I will go into detail about the attempts made to preserve support for parallel testing using MPI. I will then provide information about the differences between the two libraries in terms of speed. Finally I will end with a discussion on whether this full integration of TestU01 into the main SPRNG library is a worthy decision.

1.0.1 General Information

Before getting into information specific to each library, a discussion on common information between each library is important. Both the SPRNG and TestU01 libraries do related work on pseudorandom number generators. Pseudorandom number generators work by making use of some mathematical equation that produces a stream of numbers that appear to be totally random. However not all pseudorandom number generators are created equal, with the quality of the generator being affected by how random the numbers generated actually are. While producing a truly random stream of numbers from a generator is not possible, it is possible to produce a stream of random numbers that is close enough to random. With the main use of pseudorandom numbers being their application in Monte Carlo simulations, where a high enough quality random number generator is sufficient enough to produce meaningful and accurate results from the simulation. To test if a pseudorandom number generator is producing a high quality random number stream, random number tests have been made. The tests in SPRNG and TestU01 fall into one of two categories; they are ether statistical tests or physical model tests. The statistical tests work on the basis that if a pseudorandom number generator produces results that cannot be meaningfully distinguished from the results that would be gathered from a truly random stream of numbers, then the pseudorandom number generator is considered to produce high quality random numbers. For instance, one test

is the equidistribution test where the pseudorandom number stream is tested to see if the results produce a uniform distribution of numbers. If a truly random stream of numbers is large enough, then the distribution of its numbers should be uniform, with each number appearing an equal number of times. If a large enough stream of pseudorandom numbers is close enough to be statically indistinguishable from the uniform distribution of the truly random stream of numbers, then the pseudorandom number generator is free from any noticeable biases and patterns that would have a significant impact on the results of a Monte Carlo simulation. These statistical tests are mostly tests that are defined by D.E. Knuth[1] with both SPRNG and TestU01 choosing to implement many of these tests. The physical model tests work by running specific Monte Carlo simulations that have a known solution for the given starting state of the simulation. The only physical model test that is shared between the two libraries is based upon a random walk simulation. The random walk test simulates someone randomly walking around, afterwards which different properties of that walk can be tested against a known solution. Most of the tests work by compiling the results of the statistical test and then comparing it to the ideal values using a chi-squared test or something similar. The test that is used to produce a result from the initial statistical test is called the first level test. It is possible for the user to ask these tests to be run multiple times in succession over different parts of the random number stream. To condense this information into a meaningful result, a second level test is the run on the results produced by the first level test. This second level test is usually a Kolmogorov-Smirnov or Anderson-Darling test which is run by comparing the distribution of the p-values produces by the chi-squared test to their ideal distribution. The test that is used for the first level testing, the second level testing, and the distribution that the first level test's output follows varies between the libraries and what overall test is being run.

1.0.2 SPRNG

The SPRNG library is a random number generation library first released by M. Mascagni and A. Srinivasan in 2000[4]. During that time there was no standard library for high quality random number generation, with many existing libraries being ether sub-par and or developed in house. A bad pseudorandom number generator will produce results that have some sort of pattern or bias in the number stream. Then when used in Monte Carlo simulations this causes one outcome of the simulation to be over or under represented which in turn leads to incorrect results. Therefore, SPRNG was developed to provide high quality pseudorandom number generators that can be

used in almost any context. SPRNG aimed to provide a wide range of different pseudorandom number generators that each could satisfy the conditions of being scalable to any size and that could produce parallel random number streams. The ability to produce parallel random number streams from a single generator does make SPRNG unique. Most random number generators are only able to produce one random number stream from a given seed, while SPRNG is able to produce multiple streams of random numbers that differ from each other. SPRNG also has other functionality included with in its library such as sharing streams between multiple processes, being able to package the state of random number generators for storage, creation of new generators in the SPRNG format, and the inclusion of a test suite for pseudorandom number generators. It is this test suite in SPRNG that is of interest to this project. The SPRNG test suite is mostly comprised of tests that were defined in the paper by Knuth[1], along side some additional physical model tests, and tests made to specifically test parallel random number streams. SPRNG also already includes an optional set of TestU01 tests. Namely allowing the user to put the SPRNG random number generators through some of the TestU01 test suites. The SPRNG test suite is older and has not seen that many updates in between the 5 major versions of SPRNG, with the only major updates adding more physical model tests and many new implementations and versions of those same tests being released over time.

1.0.3 TestU01

Created by Pierre L'Ecuyer and Richard Simard in 2007[2], the TestU01 library was created for similar reasons to the SPRNG library. At the time there were a number of other random number generators testing libraries available, however each of them were found to be insufficient in some way. With many of the other testing libraries found to be of poor quality or had limitations placed on them which made them unsuitable for ease of use. TestU01 was made to be an effective replacement for many of those libraries by providing many of the same tests, but placed in a more usable and higher quality library. TestU01 also implements many of the tests found in the Knuth[1] paper. This means that TestU01 implements many of the same tests that SPRNG implements. Since TestU01 was designed to be a random number testing library first, the implementation of the Knuth tests found in TestU01 should be higher quality than the versions of the Knuth tests found in SPRNG. Replacing the implementation of the tests within SPRNG to instead use TestU01 should provide better results, whether that be in terms of quality or speed of the results.

1.0.4 Goals of this Project

The purpose of this project is to improve the quality of the SPRNG test suite by making use of the testing functions provided by TestU01. Having a better set of tests in SPRNG will better prove that the random number generators in SPRNG are high quality generators. TestU01 has already been partially integrated into SPRNG and if the user has SPRNG installed there is a strong chance that they have TestU01 installed as well. So more fully integrating the two libraries should not be a big burden for the users. To accomplish the main goal of this project I have created three smaller goals which we will explore in this paper. The first goal is to research which of the SPRNG library tests can be replaced with TestU01 tests and then implement those changes. Not every test in SPRNG has been implemented in TestU01 and TestU01 offers a number of different implementations of each test, so identifying which TestU01 test matches the SPRNG test the closest is important as minimizing the disruption to the user's experience is important. The second goal is to see if it is possible to preserve the MPI support that allows the SPRNG tests to be parallelized so that the TestU01 tests can be parallelized as well. Many of the tests can take a significant time to run so finding a way to allow the TestU01 tests to be run in a parallelized way would be very helpful. The third goal is to determine if the TestU01 tests offer any improvements over the original SPRNG tests and if the improvements offered by TestU01 are worth any potential draw backs. These improvements will be looked at in terms of the information that TestU01 provides to the user that SPRNG does not, the speed at which the tests run, and if any features of SPRNG have to be sacrificed to run the TestU01 tests.

CHAPTER 2

TESTU01 INTEGRATION

The SPRNG's test suite is made up of a total of 21 different files. Five of these files implement supporting pieces of code that implement shared utility between the different files. The remaining 16 files are implementations of all the tests made for SPRNG. The test suite also includes a small sub directory that allows the user's to easily run the SPRNG random number generators through a number of TestU01 test suites, notably the small crush and big crush test suites. In total 9 out of the available 16 SPRNG tests were able to be replaced with TestU01 tests. A full list of the files and which ones were changed are listed below in Table 2.1. The updated files have been placed with in the testu01 sub directory of the testing directory, with the original testing files left in the main testing directory. This has been partially done for ease of comparison between the old and the new tests, but also to preserve the old tests for reasons discussed in the conclusion of the paper.

2.0.1 Unchanged Files

Unfortunately not every SPRNG test has been implemented in TestU01. With TestU01 choosing not to implement each of these tests for one reason or another. This means that these tests will be left as is since they cannot be upgraded. Alongside that, many of the supporting file will be left alone as well in order to allow the unchanged tests to still function properly.

Equidistribution Test: The equidistribution test is a statistical test however it was not implemented in TestU01. Even though there are tests that work on similar principles to the equidistribution test, TestU01 has no test that implements the equidistribution test as found in SPRNG. The only reason I can think of why this is the case is that the equidistribution test is a very simple test, where integer numbers are generated in a fixed ranged to see if there is a uniform distribution of each of the numbers generated. The equidistribution test is implemented in *equidist.cpp*.

File Name	File Type	Changed?
chisquare.cpp	Supporting	No
collisions.cpp	Test	Yes
communicate.cpp	Supporting	No
coupon.cpp	Test	Yes
equidist.cpp	Test	No
fft.cpp	Test	No
gap.cpp	Test	Yes
init_tests.cpp	Supporting	No
maxt.cpp	Test	Yes
metropolis.cpp	Test	No
perm.cpp	Test	Yes
poker.cpp	Test	Yes
random_walk.cpp	Test	Yes
runs.cpp	Test	Yes
serial.cpp	Test	Yes
stirling.cpp	Supporting	No
sum.cpp	Test	No
util.cpp	Supporting	No
wolff.cpp	Test	No
wolffind.cpp	Test	No
wolfftest.cpp	Test	No

Table 2.1: Table of SPRNG testing related files. Listing which ones have been modified and which ones were not.

FFT Test: The FFT test, or Fast Fourier Transform test, is a statistical test however it is not a test that was implemented in TestU01. The SPRNG version of the FFT test also requires parallel streams of random numbers which TestU01 would not have used since TestU01 tests single streams of random numbers. The test works by filling each column of a 2D array with a different number stream of a parallel random number generator and then applying the Fourier transformation too it so that the result can be compared to the expected values. The FFT test is implemented in *fft.cpp*.

Metropolis Test: The Metropolis test is a special kind of test that differs from the standard statistical test. It works by running the Metropolis algorithm which is a Monte Carlo simulation of the Ising model for which the exact solution is known, the results of the simulation and the exact solution can then be compared to test the quality of the random number generator used. Since the Metropolis test is not a more standard kind of statistical test it was not included in the TestU01 library. The Metropolis Test is implemented in *metropolis.cpp*.

Sum Test: The Sum test is a statistical test however unlike many of the other tests it is a test that requires parallel streams of random numbers. It works by repeatedly summing n numbers for multiple parallel streams of random numbers from one generator and then testing if these sums have a uniform distribution. TestU01 only has tests meant for non-parallel streams of random numbers meaning that this test is not a part of the TestU01 library and cannot be upgraded. The Sum test is implemented in *sum.cpp*.

Wolff Test: The Wolff test is a different implementation of the same kind of test that the Metropolis test runs. The Wolff test is just a different Monte Carlo algorithm that runs a simulation of the Ising model. This means that the Wolff test is also not a part of the TestU01 library as it is also not a more standard kind of statistical test. The Wolff tests are implemented in the files *wolff.cpp*,*wolffind.cpp*, and *wolfftest.cpp*.

2.0.2 Changed Files

The process of upgrading the files from the original SPRNG code to the new TestU01 code was not too hard once I figured out how to upgrade the first test. The process followed a simple loop of work that allowed me to upgrade most of the tests without any difficultly. First I identified what TestU01 function call implemented the same test that I was working on in SPRNG. Next I had to translate the input that SPRNG asked for into the input that TestU01 is asking for. For a few tests the input format is slightly different and I wanted to make it so that I could run ask many of the new tests using the same input that the old SPRNG tests asked for. This was not possible in all of the tests and with any changes being detailed in the descriptions of the new tests. Third I modified the SPRNG files to make use of TestU01 code. This process required me to extern the needed C libraries that implement the TestU01 code, translate the SPRNG user input and random number generator into the TestU01 format, and then call the needed functions to run the TestU01 tests. After this I just needed to go through the process of cleaning up the unneeded SPRNG code and leave behind only what I needed to run the TestU01 tests.

Each test is run via command line input with the user providing a number of arguments for the test to make use of. The first seven arguments are shared between each of the SPRNG tests, after that the arguments vary by which test is being called. The arguments are called as follows:

test.sprng rngtype nstreams ncombine seed param nblocks skip test_arguments

The argument *test.sprng* is the name of the executable of the test which is in the format of the test name with the .sprng extension at the end of it. The argument *rngtype* expects an integer number from 0 to 5 referring to which random number generator the user wishes to use. The options are as follows: 0 Modified Additive Lagged Fibonacci Generator, 1 48-bit Linear Congruential Generator, 2 64-bit Linear Congruential Generator, 3 Combined Multiple Recursive Generator, 4 Multiplicative Lagged Fibonacci Generator, and 5 Prime Modulus Linear Congruential Generator. The argument *ncombine* is the number of parallel random number streams that are interleaved to form a new stream of random numbers. While *nstreams* refers to the number of combined streams that are created for testing. With each stream of random numbers being tested in *nblocks* blocks of random numbers. The argument *seed* is the number that is used as the encoding of the starting state of the random number generator, parallel generators are not seeded directly with a seed value but instead use the given value to create multiple starting states for each stream. Lastly the argument *param* is used to set unique parameters to the generators, as some of the generators have slightly variations that can be selected using this value. *skip* is used to note how many random numbers should be skipped over going from testing one block of random numbers to the next, however TestU01 does not have this functionality so for SPRNG tests that make use of TestU01 will ignore this input. The *skip* argument is being left in the TestU01 tests in order to still allow the new code to run with the same test cases generated for previous versions of SPRNG. *test_arguments* is for the list of arguments specific to each test. I will detail the arguments needed by each test bellow, along with detailing what each test does and any changes in the arguments and tests between the old SPRNG tests and new TestU01 tests. For example the coupon test can be run with the following command:

./coupon.sprng 1 4 2 0 0 3 0 100000 8

Collision Test: d n

The collision test divides [0,1) into d equal segments to create d^2 subdivisions. The test then generates n pairs of points in [0,1) and counts the number of points that fall into a subdivision that already contains at least one point. The observed number of collisions is then compared to the expected number of collisions using a chi-squared test. The SPRNG version of this test made use of a variable range of integer numbers to test in that could be broken up into a different number of subsequences, however the TestU01 version of the test makes use of a fixed range of floating point

numbers so one argument of the test had to be dropped as it was unneeded. The SPRNG version of the test makes use of only 1 dimension however the TestU01 version of the test will often complain when making use of 1 dimension saying that it is experiencing negative variance when running the test, so it has been increased to two dimensions. The SPRNG version of the test also asked the user for $\log_2$d, this has been replaced by asking for just d in order to simplify the test arguments as TestU01 also asks for d.

Coupon Test: n d

The coupon collector test generates a random series of integers in [0,d-1], and counts how many numbers need to be generated before each of the possible numbers in [0,d-1] appears at least once. This test will then be repeated n times, counting the number of times that a sequence is exactly length s. The chi-square test is used to compare the expected and observed number of sequences for each value of s. In the SPRNG version of the test sequences with a length greater than the given t would be grouped together, however TestU01 does not offer this option and so it was removed from the arguments for this test.

Gap Test: a b n

The gap test generate n floating point numbers in [0,1) and then counts how many numbers are generated before a number is generated in the interval [a,b]. The test counts the number of times a sequence length is exactly s. The chi-square test is used to compare the expected and observed number of sequences for each value of s. In the SPRNG version of the test sequences with a length greater than the given t would be grouped together, however TestU01 does not offer this option and so it was removed from the arguments for this test. In the SPRNG version of the test n was the number of gaps that the test would generate before completing, however in TestU01 n was updated to be the number of values to be generated. So n has been updated to be the number of values to be generated when running the test.

Maximum of t Test: n t [c]

The Maximum of t test generates n groups of t numbers in [0,1) and then marks down the largest number generated for each grouping. The distribution of these numbers is then compared to the theoretical distribution of x^t. c is an optional flag for TestU01 that says how many categories should be created so that the maximum values of t can be made into a uniform distribution so that

the chi-square test can be applied to it. c is given the default value of 3. In the SPRNG version of the test there was no c argument, but since it is an optional flag the old tests will still run with this version of the test.

Permutations Test: m n

The permutation tests generates m numbers and then ranks each number in increasing order. With the smallest number being rank 1 and the largest number being rank m. This forms one possible permutation of the numbers [1,m]. This process is repeated n times to gain n permutations of [,m]. The number of times each permutation appears is counted and is compared to the expected values using a chi-square test. TestU01 adds the restriction that m should be in [2,18].

Poker Test: n k d

The poker test generates k integer numbers in [0,d-1] and repeats this n times. The test counts the number of times a group has s unique integer numbers in it. The chi-square test is used to compare the expected and observed number of sequences for each value of s. TestU01 adds the restrictions that both k and d should less than 128.

Random Walk Test: n l0 l1

The random walk test applies a number of different tests all based on a random walk across the set of integers. The walk starts at 0 and then uses a randomly generated number to determine whether it should move left or right. An ideal random number generator should have an equal chance of moving both left and right at any given point in the random number stream. Each test uses this idea to compare the observed properties of the random walk to the expected. Some of the tests name use of the argument n for the length of the run. Other tests are composed of multiple random walks with lengths [l0,l1], however they only use every other length meaning they start at walk length l0, then l0+2, l0+4, until l1 is reached. There are six total tests done on the random walk. They are, counting the number of times the random walk takes a step to the right, finding the maximum value that the random walk goes too, counting the amount of time the random walk spends to the right of the origin, marking the time the random walk first enters the integer value s for each value of s, the number of times the random walk passes through the origin, and the number of times the random walk changes sign by crossing between positive and negative numbers. The test has the restriction that l0 is less than or equal to l1 and that they are both even numbers.

The TestU01 version of the test is more significantly different from the SPRNG version of the test than compared to any other of the changed tests. First the SPRNG version of the test works in two dimensions, performing the random walk over a grid instead of across the number line. The SPRNG version of the test by counting how many times the random walk crosses over x = 0 or y = 0 and which quadrants of the grid it is in when it crosses over. This test is similar enough to the second to last and last random walk test in TestU01 that the same thing is being tested between the SPRNG and TestU01 tests. If this test is determined to be two different to be included in the official release of SPRNG it will still be included here as it is easier to remove a test than add a new one in.

Runs up Test: n

The runs up test generates n random numbers in [0,1) and counts the number of successively increasing sequences of numbers that are length s. Runs of length 6 or higher are merged into one grouping of numbers. This test does not make use of the chi-square test, it instead makes use of a test defined in Knuth's *The Art of computer Programming, volume 2*[1] on page 67 as Equation 10. In the SPRNG version of the test sequences with a length greater than the given t would be grouped together, however TestU01 does not offer this option and uses a fixed value of 5 for t so it was removed from the arguments for this test.

Serial Test: d n

The serial test divides [0,1) into d equal segments to create d^2 subdivisions. The test then generates n pairs of points in [0,1) and counts the number of points that falls into each of the d^2 subdivisions. The chi-square test is then used to compare the observed values to the expected values. The SPRNG version of this test generates pairs of integers in [0,d-1] instead of dividing [0,1) into d segments.

CHAPTER 3

PARALLEL SUPPORT FOR NON-PARALLEL TESTS

The SPRNG library makes use of the Open MPI to provide parallel implementations of the tests. The original SPRNG library tests worked by breaking up the core loop of the tests across multiple processes. Each run of the loop running the test once. Then MPI functions are used to gather the data generated from each of the processes so that they could be consolidated into a final result. The issue comes that the TestU01 library does not have this functionality. All of the tests in TestU01 run in only one process which means that some of the larger tests the user could run will run significantly slower when compared to the parallel versions of the SPRNG test suite. My goal here is to find a way to run the non-parallel TestU01 tests in a parallel way. This parallel version of running TestU01 should be able to work without causing issues in the results of the test, still allowing for error messages from TestU01 to appear, and to not be too disruptive to the user experience when running the tests in parallel. The parallel tests have been implemented in the *testu01mpi* sub directory similar to how the SPRNG parallel versions of the tests are implemented in the *mpitests*.

3.0.1 Challenges to Implementation

There are a few problems that needed to be solved when implementing parallel support for TestU01. First it is possible that TestU01 cannot handle being run in parallel. TestU01 may have been coded in a way that running the same function across multiple processes causes the code to ether crash or produce incorrect results. Second TestU01 primarily conveys information to the user via standard output. So there will need to be some way to gather the results generated by TestU01 and consolidate them into one final result. Alongside that, without proper changes made while running the tests in a parallel environment the console will be flooded with a vast amount of garbled text that is being produced by TestU01 as it processes the information. Any information that is printed to the user should still be readable. Lastly, once the information has been collected from TestU01 it should be consolidated into a final result.

3.0.2 Implementation

Fortunately TestU01 can be run in parallel using MPI without any issues. Since MPI runs by making entirely separate processes instead of making threats all of the TestU01 code runs as if it was being run normally. There is one possible issue that may come up relating to TestU01 in this parallel code. In the functional calls to TestU01 the number of replications of the test has been set to one. This was done in order to only run the test the exact number of times the user wanted while also preserving functionality related to testing parallel streams of numbers. At lower values of n, which is generally the number of random numbers or sub tests that are used in the test, TestU01 is more likely to throw errors about detecting poor test quality, errors that SPRNG would not have thrown since TestU01 has a higher standard of quality for their tests. This issue can easily be avoided by only making use of the parallel tests when you wish to test a large amount of random numbers.

When running any of the tests they each produce a p-value that tells the user how significantly the observed and expected results different from each other. The p-value produced by this test will be different, provided the inputted random number stream is different, each time the test runs. This p-value should follow a normal distribution. Since if the random number stream is truly random then each subsection of that random number stream has an equal chance to appear more or less random when compared to the ideal. This means that if a test on a random number generator is run multiple times then the p-value produced by each of these tests can be then compared to the uniform distribution in a Kolmogorov-Smirnov test. This multilevel testing has already been implemented in both SPRNG and TestU01 separately, and now needs to be brought together. The first step to doing this is going to be gaining access to the p-value that is produced by each TestU01 test. The first attempt I made at doing this was to capture the output generated by TestU01 and to then parse down the text so that I could read in the p-value. This did not work for multiple reasons, however after finding the expanded documentation for TestU01 I was able to find a solution. In the functional calls to TestU01 there is the provided option to pass a structure that will be filled with all of the information generated through the testing of the random number generator. This includes the final p-value that is produced by TestU01 to determine the quality of the stream. Making use of the original of the original SPRNG code structure I was able to call the TestU01 tests multiple times, gathering up the needed data, and eventually passing the data to the primary process to be

consolidated into a final result. One thing to note about the parallel implementation of the TestU01 test calls is that it is able to preserve all of the functionality that the SPRNG library offers. This includes the functionality of the *skip* and *nstreams* command like arguments that the previous non parallel tests were not able to support. This is because the parallel version of the test makes TestU01 run no replications of the test. So the code needed to implement the *skip* and *nstreams* arguments can be run in between each run of the test. This method was done for the parallel tests so that the TestU01 code can be integrated into SPRNG as seamlessly as possible since a number of other functions used later on heavily rely on having the correct number of p-values to test upon. Alongside that, if TestU01 was allowed to run its own replications of the test then the final test would be a three level testing process. With each level of testing having less data to work with, as the third level of testing would only have as many p-values to work with as processes spawned. So the final result will be weakened due to the lower amount of data being provided. With the test requiring an excessive amount of random numbers, tests, and processes to produce reliable results. Since the parallel versions of the TestU01 do not run multiple replications of a test that means TestU01 does not run any second level tests. Instead only SPRNG's single second level test is run. This means that the parallel TestU01 tests will not produce as much information as the nonparallel tests, but their versions of the single level tests will still be run. To solve the issue of large amounts of clutter being printed to standard out, the standard output stream was closed for every spawned process except for the process that will produce the final results. This still allows the user to see some of the output generated by the TestU01 tests, while keeping it readable and not overwhelming. Alongside allowing TestU01 to still tell the users about any errors that occur. Since the errors in TestU01 are mostly related to bad user input, if one test encounters an error then every test will encounter the same error so the user will be informed about the errors.

Once I have collected the needed p-values from each run of the TestU01 test I need to make use of MPI to send all the data to one of the processes spawned by MPI in order to run the second level test. To do this I made use of already existing SPRNG code. SPRNG has the function *getKSdata(double *V, long ntests)* that gathers the p-values from the different processes and delivers them to the main process. SPRNG also has the functions *KS(double *V, long NTESTS, double (*F)(double)* and *KSpercent(double value, long NTESTS)*. The KS() function runs the Kolmogorov Smirnov test on the data in V against the cumulative distribution function that is defined by

the function *F, it returns the Kolmogorov Smirnov statistic. The KSpercent() function takes in the Kolmogorov Smirnov statistic produced by the previous test and the total number of tests performed and returns the p-value. The cumulative distribution function wanted by the KS() function needs to be the cumulative uniform distribution function. However SPRNG only has functions for the cumulative normal function and the incomplete gamma function, so the cumulative uniform distribution function had to be added to the available list of functions. This has caused the files *tests.h* and *chisquare.cpp* to be changed with in the *testu01mpi* sub directory.

3.0.3 Running the Parallel Tests

To run the parallel versions of the TestU01 tests you will need to make use of the sub directory *testu01mpi*. If both the MPI and TestU01 flags were enabled during the configuring of the SPRNG library then this sub directory will be compiled and from this sub directory the MPI versions of the test can be run. This was done to match the same style of integration that the SPRNG library did to the parallel versions of its tests. In SPRNG the tests with MPI enables are kept in a sub directory that is only compiled if the MPI flag has been enabled with configuring the library. To run the parallel version of the tests the user needs to make use of the same command line arguments mentioned in the previous section except they need to add 'mpirun -np X' before their command line argument. With X being the number of separate processes that should be spawned to run this code. As example for the coupon test:

mpirun -np 4 ./coupon.sprng 1 4 2 0 0 3 0 100000 8

This will split the four random number streams created by one generator across four different processes. With each stream generating three blocks of random numbers to be tested independently from each other. The exact distribution of the streams generated by SPRNG across the different processes is determined by SPRNG.

CHAPTER 4

TESTING

4.0.1 Testing Introduction

My goal for testing the speed of the two libraries was to gain an idea of the performance differences between SPRNG and TestU01. While the speed of the tests is not the most important factor when considered the differences between the two libraries, it is still a major factor to be considered. To create the speed tests I needed to find comparable input parameters for each library as there are some differences in the type and amount of input each testing library wants. I did this by taking the inputs that would use for a SPRNG test and then making the changes needed for it to run in a TestU01 test while still doing about the same amount of work. For most of the tests the parameters between the two tests are not different in a way that would impact how much work a test would need to do. For the Collision test the parameters did have to be changed since the two tests wanted their input in a different format for each test, however they should still do an equivalent amount of work. For the random walk test this was not possible as the TestU01 version of the test requires extra parameters which impact the amount of work done. A comparison between the two versions of the tests shall still be done but the fact that the TestU01 test can take a variable amount of extra time depending on these extra parameters should be kept in mind.

The tests were run on a Virtual box running Ubuntu 16.04.6 LTS with 2048MB of ram and one CPU processor. Two sets of tests were performed. For both sets of tests the SPRNG version of a random number generator test was compared to the TestU01 version of that random number generator test. So the SPRNG collision test was compared to the TestU01 collision test, but I did not test the SPRNG collision test against the TestU01 coupon test. The first set of tests were a series of unpaired two tailed t test. Each test was preformed five times for both the TestU01 and SPRNG version of the random number generator test, results of the speed test in Table 4.1. Each random number generator test was then compared between the versions using an unpaired two tailed t-test to see if the speed of the two tests was significantly different. A fixed version of n was used for all of these tests, for most tests it was n = 100,000 but for a few of the faster tests it was

raised to 500,000. The goal of this test was to allow the tests to run for a decently long time and then to compare them to see if there is a difference in the run times. The second set of tests was a linear regression between the two generators. Each random number generator test was preformed 5 times for each generator with n starting at n = 100,000 or 500,000 and increasing by 100,000 for each test. The collision test had to start at 10,000 and increase by only 10,000 each test due to the test having an upper limit on n. The goals of this test was to confirm the results of the previous test by including test results that included a growing value of n and to see the rate of growth for the time needed to complete each random number generator test as n grows for each library. The full set of data and the tests performed on that data can be found at the Github link in the file *Data.xlsx*.

4.0.2 Looking at the Data

Test	Library	1	2	3	4	5	Average
Collision	TestU01	1.22	1.29	1.4	1.62	1.24	1.354
Collision	SPRNG	0.19	0.19	0.18	0.27	0.17	0.2
Coupon	TestU01	5.28	5.43	5.2	5.62	5.62	5.43
Coupon	SPRNG	3.2	3.12	3.12	4.29	3.08	3.362
Gap	TestU01	1.58	1.57	1.56	1.63	1.59	1.586
Gap	SPRNG	0.74	0.72	0.74	0.75	0.75	0.74
Maxt	TestU01	15.8	18.2	17.75	19.12	18.23	17.82
Maxt	SPRNG	9.44	8.91	9.03	8.36	9.27	9.002
Perm	TestU01	4.14	3.49	3.53	3.48	3.34	3.596
Perm	SPRNG	3.52	3.5	3.47	3.47	3.58	3.508
Poker	TestU01	22.879	22.261	21.855	22.214	22.711	22.384
Poker	SPRNG	11.88	11.97	11.84	11.98	11.88	11.91
Random Walk	TestU01	12.34	12.94	12.85	12.6	12.99	12.744
Random Walk	SPRNG	0.16	0.17	0.17	0.17	0.19	0.172
Runs	TestU01	11.29	11.23	11.32	11.31	11.62	11.354
Runs	SPRNG	12.8	12.79	18.15	12.95	13.23	13.984
Serial	TestU01	2.85	2.95	2.91	2.82	2.99	2.904
Serial	SPRNG	1.29	1.27	1.92	1.28	1.31	1.414

Table 4.1: Table of speed test with results timed in the number of seconds it took for each test to complete in real time

Even looking by eye at the data found in Table 4.1 we can see there are some significant differences in terms of the speed of the SPRNG tests and the TestU01 tests. Only two tests fail to

Test	T value 2 Tail	Crit Val	Reject Null
Collision	15.26369568	2.776445105	Yes
Coupon	8.333291764	2.570581836	Yes
Gap	63.76964938	2.446911851	Yes
Maxt	15.15299228	2.570581836	Yes
Perm	0.623189454	2.776445105	No
Poker	56.34832383	2.776445105	Yes
Random Walk	103.5864617	2.776445105	Yes
Runs	-2.512519964	2.776445105	No
Serial	11.42039106	2.776445105	Yes

Table 4.2: Testing if the tests reject the Null Hypothesis that the tests run at the same speed using a two tailed t test.

reject the null hypothesis that the tests run at the same speed. While having TestU01 being just as fast or even faster than SPRNG would have been ideal even looking at the t-test data this is clearly not the case. Taking a look at the Regression Tests we can see that for the tests that rejected the null hypothesis the TestU01 test was the slower of the two tests. While the amount varied by the speed of the test using the TestU01 version of the test slowed down the test by a few seconds for the faster tests and for almost half a minute for the slower tests in our range of n. To figure out the exact differences in terms of speed between the two libraries I ran the regression tests again, but I ran two regression tests one for the data created by the SPRNG library and one for the data created by the TestU01 library. For example, the Maximum of t test was found to be one of the slower tests when comparing the SPRNG and TestU01 run times. When Running Regression on the TestU01 data and the SPRNG data separately I got the following two regression equations. TestU01: Y = 2.993+0.00132*n. SPRNG: Y = 2.012 + 0.00006604*n. With the time it takes for TestU01 to complete a test growing at twice the rate it takes SPRNG to complete that same test. Taking a look at table 4.3 this trend continues for most of the tests. TestU01 consistently runs slower than the SPRNG library tests. Ignoring the two outlying values of Collision and Random Walk, TestU01 is on average 1.9905 times slower than SPRNG.

The two exceptions to SPRNG being the faster of the two libraries are the Runs and Permutation tests that failed to reject the null hypothesis in the T-test. Taking a closer look at the permutation tests we can see that the two tests are actually running at the same speed. With the coefficient for the generator type having a p-value of p=0.149. This means that the two versions of the test

were not statistically different from each other in terms of speed. For the Runs test the opposite situation was found, where all the p-values in the regression test were found to be significant however it showed that the TestU01 version of the test was actually faster. With the TestU01 test being about 1.25 times faster than the SPRNG version of the test.

Library	Test	Regression Line	Comparison
TestU01	Collision	Y = 0.027 + 0.0000135*n	33.75
SPRNG	Collision	Y = 0.08 + 0.0000004*n	
TestU01	Coupon	Y = 1.129 + 0.00003677*n	2.2684
SPRNG	Coupon	Y = 1.537 + 0.00001621*n	
TestU01	Gap	Y = 0.411 + 0.00001045*n	1.968
SPRNG	Gap	Y = 0.135 + 0.00000531*n	
TestU01	Max of t	Y = 2.993 + 0.00013207*n	1.9998
SPRNG	Max of t	Y = 2.012 + 0.00006604*n	
TestU01	Permutation	Y = 0.337 + 0.00002575*n	1.3064
SPRNG	Permutation	Y = 1.297 + 0.00001971	
TestU01	Poker	Y = 2.996 + 0.00017858*n	2.1955
SPRNG	Poker	Y = 3.934 + 0.00008134*n	
TestU01	Random Walk	Y = 4.121 + 0.00007747*n	56.547
SPRNG	Random Walk	Y = 0.017 + 0.00000137	
TestU01	Runs	Y = 3.392 + 0.000014*n	0.8883
SPRNG	Runs	Y = 4.064 + 0.00001576*n	
TestU01	Serial	Y = 0.683 + 0.00000377*n	3.307
SPRNG	Serial	Y = 0.66 + 0.00000114*n	

Table 4.3: This table contains all the individual regression lines for each individual test, the regression takes in n which is the number of random numbers to be generated or sub tests to be run and produced y which is the amount of time in seconds the test takes to complete. With a comparison value which is how much slower TestU01 is compared to SPRNG.

The exact reason why one TestU01 test is faster or slower than the SPRNG test may vary however there is one common factor between all the TestU01 tests that causes a slowdown in the tests. The SPRNG test calculates the Kolmogorov-Smirnov test and chi-square tests during the process to see if the generator passes the test. The TestU01 calculates the Kolmogorov-Smironv test twice for both the maximal and minimal difference between the expected and observed distribution, TestU01 calculates the Anderson-Darling test to compare the expected and observed distribution, a number of chi-squared tests, and many of the tests provide extra analysis specific to the test being run. This means that the TestU01 tests will run slower since they are also producing more results

than the SPRNG version of the test. The Collision and Random Walk tests are outliers in terms of how much slower they were compared to the SPRNG tests. This is because when upgrading from the SPRNG test to the TestU01 test they are the only two tests that had major differences in how the tests were implemented. With collision working in two dimensional space instead of one dimensional space for where the points are generated and Random Walk consisting of multiple tests on a random walk instead of just one test.

CHAPTER 5

CONCLUSION

The decision whether to make use of this version of the SPRNG library with the TestU01 integration is dependent on a couple of factors. The speed of the tests, the quality of the tests, and the features available. No one factor will be the deciding factor on whether the TestU01 integration is fully included in the next release of SPRNG. The changes and what they offer, and what they cost, should be looked at as a whole. In this section I will be summarizing both the disadvantages and advantages of the TestU01 integration, before coming to my suggestion on if the TestU01 integration is a good idea or not.

5.0.1 Discussion of Differences

The disadvantages to TestU01 mostly comes down to the slower speed of TestU01 and the loss of functionality around the *skip* and *nstreams* command line arguments. Ignoring the outlying Random Walk and Collision tests the TestU01 library is on average twice as slow as the SPRNG library. This is unfortunate for the TestU01 library as it is not unusual to perform tests on extremely large sets of random numbers. Looking at some of the tests that are used in the TestU01 BigCrush test suite, there is an example of calling the Coupon Collector test with an n value of $2*10^8$. This would take the TestU01 test over two hours to run while it would only take SPRNG 54 minutes to run. The slower nature of TestU01 would be an unfortunate cost of upgrading the SPRNG tests. The loss of the functionality of the *skip* and *nstreams* arguments is a bigger detriment to making use of the TestU01 functions. SPRNG is a library for random number generators that can produce parallel streams of random numbers and the ability to test these parallel streams is important. The parallel version of the TestU01 code does still support both the *skip* and *nstreams* arguments so the functionality will not be completely removed. However this requires the installation of the optional MPI library. Furthermore, the current implementation of the parallel tests removes some of the benefits of TestU01. TestU01 provides multiple second level tests to see if the random number generator passes the test, however the current parallel test implementation only provides the one SPRNG already offers. This could be changed to provide those multiple second level tests,

however then the functionality of the *skip* and *nstreams* arguments would have to be removed from the parallel tests as well. I did investigate if it was possible to keep support for the *skip* and *nstreams* arguments. It may be theoretically possible to modify the code so that the *skip* and *nstreams* arguments are fully supported. However it would not be possible for every single test, and would require working through the TestU01 code to see exactly how many random numbers each test requires to run and if there are any signs that tell when a TestU01 test is going from one replication of the random number test to the next one. I was unable to confirm if ether of these things were possible to do and I am reasonable certain that the second one is impossible.

The benefits of making use of TestU01 over the original SPRNG code come down to one main factor which is that TestU01 provides more information and higher quality tests to the user. While speed is an important factor for the tests, it is not a main factor. For the larger tests of random number generators, like the Coupon collector test mentioned above, the test will only really be run once or twice. After a generator has been confirmed to run correctly with a large test, there is very little need to run the test again. Adding an extra hour of run time to an already long test time is worth it provided that the test provides a better quality result, which TestU01 does by providing more second level tests. SPRNG only provides the Kolmogorov-Smirnov test, where the maximum absolute value difference between the expected and observed values is tested to determine if the random number generator is sufficiently random or not. TestU01, on the other hand, provides at a minimum three second level test statistics. First, TestU01 provides the Kolmogorov-Smirnov test where the maximum difference is between the expected values and observed values that are greater than the expected values. Second, TestU01 provides the similar Kolmogorov-Smirnov test where the maximum difference is between the expected values and the observed values that are less than the expected values. Finally, TestU01 also provides the Anderson-Darling test which gives more weight to the values at the tails of the observed values and is more sensitive to errors in the distribution. With just these three tests TestU01 provides more information to the user about how a particular generator fails to pass a test and will catch more errors than what SPRNG would catch. In addition to those three tests, TestU01 also provides extra second level tests that are specific to the statistical test being run. For instance, the Coupon Collector test sums up the random numbers generated in each replication for the test and then compares the distribution of the sums to the expected distribution.

5.0.2 Recommendation on Upgrading to TestU01

I support the integration of TestU01 into the SPRNG library if the integration means adding the TestU01 tests into a separate directory where they user can choose to run them or not. This allows the random number generators to be tested using the higher quality tests provided by TestU01 while still allowing the user to test the parallel stream functions provided by the SPRNG random number generators by using the original SPRNG tests. This type of integration is what is currently available in the Github where I have placed the updated SPRNG library. I can not recommend the type of integration of TestU01 into SPRNG where the SPRNG tests are fully replaced by the TestU01 tests. The resulting loss of the functionality relating to the *skip* and *nstreams* arguments is too great for the non MPI versions of the tests and the MPI versions of the TestU01 tests do not offer anything that SPRNG does not already offer.

BIBLIOGRAPHY

[1] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 1998.

[2] Pierre Lecuyer and Richard Simard. Testu01. *ACM Transactions on Mathematical Software*, 33(4), Jan 2007.

[3] Pierre L'Ecuyer and Richard Simard. Testu01. `http://simul.iro.umontreal.ca/testu01/tu01.html`.

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

[5] M. Mascagni. The scalable parallel random number generators library (sprng). `http://sprng.org/`.

[6] Michael Mascagni. Sprng: A scalable library for pseudorandom number generation. *Recent Advances in Numerical Methods and Applications II*, 1999.