How to Test with TestU01

M.E. O'Neill — 2017-08-14 20:32

In an earlier post (how-to-test-with-practrand.html) I showed you how to test a PRNG with PractRand. Now it's the turn of the venerable TestU01. It's a little more finicky, but quite doable. Let's see…

Downloading and Building TestU01

You can find the TestU01 website here (http://simul.iro.umontreal.ca/testu01/). It configures in the traditional way for Unix software.

I'm going to assume you don't want to do a full install of TestU01 as a general library on your system, you just want to use it briefly, so we'll do a quick and easy temporary install. The commands below are all you need to do to build a working test setup with TestU01 on a Unix system (Linux, macOS, or the Windows subsystem for Linux),

```
mkdir TestU01
cd TestU01
basedir=`pwd`
curl -OL http://simul.iro.umontreal.ca/testu01/TestU01.zip
unzip -q TestU01.zip
cd TestU01-1.2.3
./configure --prefix="$basedir"
make -j 6
make -j 6 install
cd ..
    mkdir lib-so
    mv lib/*.so lib-so/.
```

The last two lines aren't 100% necessary, but dealing with dynamic libraries is more trouble than it's worth here. This way we'll use the static libraries.

Writing a Test Program to test Xorshift-32

The TestU01 documentation explains how to test a simple 32-bit generator, and gives sample code.

```
// Adapted from TestU01 manual, Figure 2.2, Figure 2.4

#include "TestU01.h"

// Example PRNG: Xorshift 32

static unsigned int y = 2463534242U;

unsigned int xorshift (void)
{
    y ^= (y << 13);
    y ^= (y >> 17);
    return y ^= (y << 5);
}

int main()
{
    // Create TestU01 PRNG object for our generator
    unif01_Gen* gen = unif01_CreateExternGenBits("Xorshift 32", xorshift);

    // Run the tests.
    bbattery_SmallCrush(gen);

    // Clean up.
    unif01_DeleteExternGenBits(gen);

    return 0;
}
```

To compile it, we need to run

```
gcc -std=c99 -Wall -O3 -o test-xorshift-minimal test-xorshift-minimal.c -Iinclude -Lli
```

It's vital that we include the `-Iinclude -Llib -ltestu01 -lprobdist -lmylib -lm` part to let the compiler find the TestU01 libraries and link with them.

If we run the resulting executable, we get a lot of output as it goes through each test, culminating in this summary:

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

 Version:          TestU01 1.2.3
 Generator:        Xorshift 32
 Number of statistics:  15
 Total CPU time:   00:00:05.46

 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ----------------------------------------------
  1  BirthdaySpacings                 eps
  2  Collision                      1 - eps1
  6  MaxOft                         6.7e-16
  8  MatrixRank                       eps
 10  RandomWalk1 H                   5.7e-7
 ----------------------------------------------
 All other tests were passed
```

Less than six seconds of testing, and we see that unimproved XorShift generators have problems. (Truncated XorShift*, on the other hand, would have passed with flying colors.)

As short and sweet as this test code was, it's not ideal in a few ways. We can't run the code with a different seed, we can't easily change what test we run, and it doesn't also test the reversed bits (which is important with TestU01). We'll address these issues in the next examples.

Testing the 32-bit Mersenne Twister (C++)

When you add some command line options to allow the user to specify a seed, and the ability to choose test options, and so on, the code balloons a bit, but it's nothing earth shattering in its complexity:

```
#include <random>
#include <string>
#include <cstdio>
#include <cstdint>
#include <cstddef>

extern "C" {
    #include "TestU01.h"
}

// GENERATOR SECTION
//
// This is the only part of the code that needs to change to switch to
// a new generator.

const char* gen_name = "std mt19937";  // TestU01 doesn't like colons!!?!

const int MAX_SEEDS = 1;
uint64_t seed_data[MAX_SEEDS];

uint32_t gen32()
{
    static std::mt19937 rng(seed_data[0]);

    return rng();
}

// END OF GENERATOR SECTION

inline uint32_t rev32(uint32_t v)
{
    // https://graphics.stanford.edu/~seander/bithacks.html
    // swap odd and even bits
    v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
    // swap consecutive pairs
    v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
    // swap nibbles ...
    v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
    // swap bytes
    v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
    // swap 2-byte-long pairs
    v = ( v >> 16             ) | ( v               << 16);
    return v;
}

uint32_t gen32_rev()
{
    return rev32(gen32());
}

const char* progname;

void usage()
{
    printf("%s: [-v] [-r] [seeds...]\n", progname);
    exit(1);
}

int main (int argc, char** argv)
{
    progname = argv[0];

    // Config options for TestU01
    swrite_Basic = FALSE;  // Turn of TestU01 verbosity by default
                           // reenable by -v option.

    // Config options for generator output
    bool reverseBits = false;

    // Config options for tests
    bool testSmallCrush = false;
    bool testCrush = false;
    bool testBigCrush = false;
    bool testLinComp = false;

    // Handle command-line option switches
    while (1) {
        --argc; ++argv;
        if ((argc == 0) || (argv[0][0] != '-'))
            break;
        if ((argv[0][1]=='\0') || (argv[0][2]!='\0'))
            usage();
        switch(argv[0][1]) {
        case 'r':
            reverseBits = true;
            break;
        case 's':
            testSmallCrush = true;
            break;
        case 'm':
            testCrush = true;
            break;
        case 'b':
            testBigCrush = true;
            break;
        case 'l':
            testLinComp = true;
            break;
        case 'v':
            swrite_Basic = TRUE;
            break;
        default:
            usage();
        }
    }

    // Name of the generator
    std::string genName = gen_name;
    if (reverseBits)
        genName += " [Reversed]";

    // Determine a default test if need be


    if (!(testSmallCrush || testCrush || testBigCrush || testLinComp)) {
        testCrush = true;
    }

    // Initialize seed-data array, either using command-line arguments
    // or std::random_device.

    printf("Testing %s:\n", genName.c_str());
    printf("- seed_data[%u] = { ", MAX_SEEDS);
    std::random_device rdev;
    for (int i = 0; i < MAX_SEEDS; ++i) {
        if (argc >= i+1) {
            seed_data[i] = strtoull(argv[i],0,0);
        } else {
            seed_data[i] = (uint64_t(rdev()) << 32) | rdev();
        }
        printf("%s0x%016lx", i == 0 ? "" : ", ", seed_data[i]);
    }
    printf("}\n");
    fflush(stdout);

    // Create a generator for TestU01.

    unif01_Gen* gen =
        unif01_CreateExternGenBits((char*) genName.c_str(),
                                   reverseBits ? gen32 : gen32_rev);

    // Run tests.

    if (testSmallCrush) {
        bbattery_SmallCrush(gen);
        fflush(stdout);
    }
    if (testCrush) {
        bbattery_Crush(gen);
        fflush(stdout);
    }
    if (testBigCrush) {
        bbattery_BigCrush(gen);
        fflush(stdout);
    }
    if (testLinComp) {
        scomp_Res* res = scomp_CreateRes();
        swrite_Basic = TRUE;
        for (int size : {250, 500, 1000, 5000, 25000, 50000, 75000})
            scomp_LinearComp(gen, res, 1, size, 0, 1);
        scomp_DeleteRes(res);
        fflush(stdout);
    }

    // Clean up.

    unif01_DeleteExternGenBits(gen);


    return 0;
}
```

We can compile this code with

```
g++ -std=c++14 -Wall -O3 -o test-mt19937 test-mt19937.cpp -Iinclude -Llib -ltestu01 -l
```

And run it with, for example,

```
linux$ ./test-mt19937 -r -s
Testing std mt19937 [Reversed]:
- seed_data[1] = { 0xc37415b2ce04157e}

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

 Version:          TestU01 1.2.3
 Generator:        std mt19937 [Reversed]
 Number of statistics:  15
 Total CPU time:   00:00:06.37

 All tests were passed
```

One neat thing about this code is that by running it with the `-l` option, we can reveal the same “Linear Complexity” test failure that the BigCrush and Crush tests reveal, but do so in a few seconds rather than having to wait hours for the same result. We'll show output from this test when we test the 64-bit Mersenne Twister.

Testing the 64-bit Mersenne Twister (C++)

Testing 64-bit generators with TestU01 is a bit of a pain, because we can only test 32 bits at a time. But we can adapt the above code into a version for 64-bit generators that handles the issue by allowing us to send either the high 32 bits or the low 32 bits to be tested. It might also be good to test an interleaving of high and low bits, but that although that might detect some problems, it is also very likely to mask other problems. I'll leave implementing that test as an exercise.

```
#include <random>
#include <string>
#include <cstdio>
#include <cstdint>
#include <cstddef>

extern "C" {
    #include "TestU01.h"
}

// GENERATOR SECTION
//
// This is the only part of the code that needs to change to switch to
// a new gnerator.

const char* gen_name = "std mt19937_64"; // TestU01 doesn't like colons!!?!

const int MAX_SEEDS = 1;
uint64_t seed_data[MAX_SEEDS];

uint64_t gen64()
{
    static std::mt19937_64 rng(seed_data[0]);

    return rng();
}

// END OF GENERATOR SECTION

inline uint32_t rev32(uint32_t v)
{
    // https://graphics.stanford.edu/~seander/bithacks.html
    // swap odd and even bits
    v = ((v >> 1) & 0x55555555) | ((v & 0x55555555) << 1);
    // swap consecutive pairs
    v = ((v >> 2) & 0x33333333) | ((v & 0x33333333) << 2);
    // swap nibbles ...
    v = ((v >> 4) & 0x0F0F0F0F) | ((v & 0x0F0F0F0F) << 4);
    // swap bytes
    v = ((v >> 8) & 0x00FF00FF) | ((v & 0x00FF00FF) << 8);
    // swap 2-byte-long pairs
    v = ( v >> 16             ) | ( v               << 16);
    return v;
}

inline uint32_t high32(uint64_t v)
{
    return uint32_t(v >> 32);
}

inline uint32_t low32(uint64_t v)
{
    return uint32_t(v);
}
```

```
uint32_t gen32_high()
{
    return high32(gen64());
}


uint32_t gen32_high_rev()
{
    return rev32(high32(gen64()));
}

uint32_t gen32_low()
{
    return low32(gen64());
}

uint32_t gen32_low_rev()
{
    return rev32(low32(gen64()));
}

const char* progname;

void usage()
{
    printf("%s: [-v] [-r] [seeds...]\n", progname);
    exit(1);
}

int main (int argc, char** argv)
{
    progname = argv[0];

    // Config options for TestU01
    swrite_Basic = FALSE;  // Turn of TestU01 verbosity by default
                           // reenable by -v option.

    // Config options for generator output
    bool reverseBits = false;
    bool highBits = false;

    // Config options for tests
    bool testSmallCrush = false;
    bool testCrush = false;
    bool testBigCrush = false;
    bool testLinComp = false;

    // Handle command-line option switches
    while (1) {
        --argc; ++argv;
        if ((argc == 0) || (argv[0][0] != '-'))
            break;
        if ((argv[0][1]=='\0') || (argv[0][2]!='\0'))
            usage();
        switch(argv[0][1]) {
        case 'r':
            reverseBits = true;
            break;
        case 'h':
            highBits = true;
            break;
        case 's':
            testSmallCrush = true;
            break;
        case 'm':
            testCrush = true;
            break;
        case 'b':
            testBigCrush = true;
            break;
        case 'l':
            testLinComp = true;
            break;
        case 'v':
            swrite_Basic = TRUE;
            break;
        default:
            usage();
        }
    }

    // Name of the generator

    std::string genName = gen_name;
    genName += highBits ? " [High bits]" : " [Low bits]";
    if (reverseBits)
        genName += " [Reversed]";

    // Determine a default test if need be

    if (!(testSmallCrush || testCrush || testBigCrush || testLinComp)) {
        testCrush = true;
    }

    // Initialize seed-data array, either using command-line arguments
    // or std::random_device.

    printf("Testing %s:\n", genName.c_str());
    printf("- seed_data[%u] = { ", MAX_SEEDS);
    std::random_device rdev;
    for (int i = 0; i < MAX_SEEDS; ++i) {
        if (argc >= i+1) {
            seed_data[i] = strtoull(argv[i],0,0);
        } else {
            seed_data[i] = (uint64_t(rdev()) << 32) | rdev();
        }
        printf("%s0x%016lx", i == 0 ? "" : ", ", seed_data[i]);
    }
```

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

    // Create a generator for TestU01.

    unif01_Gen* gen =
        unif01_CreateExternGenBits((char*) genName.c_str(),
          reverseBits ? (highBits ? gen32_high_rev : gen32_low_rev)
                      : (highBits ? gen32_high     : gen32_low));

    // Run tests.

    if (testSmallCrush) {
        bbattery_SmallCrush(gen);
        fflush(stdout);
    }
    if (testCrush) {
        bbattery_Crush(gen);
        fflush(stdout);
    }
    if (testBigCrush) {
        bbattery_BigCrush(gen);
        fflush(stdout);
    }
    if (testLinComp) {
        scomp_Res* res = scomp_CreateRes();
        swrite_Basic = TRUE;
        for (int size : {250, 500, 1000, 5000, 25000, 50000, 75000})
            scomp_LinearComp(gen, res, 1, size, 0, 1);
        scomp_DeleteRes(res);
        fflush(stdout);
    }

    // Clean up.

    unif01_DeleteExternGenBits(gen);

    return 0;
}
```

Again, we compile with

```
g++ -std=c++14 -Wall -O3 -o test-mt19937_64 test-mt19937_64.cpp -Iinclude -Llib -ltest
```

and here's a run (piped through a couple of greps to strip blank and unimportant lines):

```
$ ./test-mt19937_64 -h -l | grep . | egrep -v '^(HOST|Generator)'
Testing std mt19937_64 [High bits]:
- seed_data[1] = { 0x1981b0ad2c8efea9}
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:

-----------------------------------------------
   N =  1,  n = 250,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    1
Chi2 statistic for size of jumps      :    1.17
p-value of test                       :    0.28
-----------------------------------------------
Normal statistic for number of jumps  :    1.11
p-value of test                       :    0.13
-----------------------------------------------
CPU time used                    :  00:00:00.00
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 500,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    2
Chi2 statistic for size of jumps      :    2.20
p-value of test                       :    0.33
-----------------------------------------------
Normal statistic for number of jumps  :    1.22
p-value of test                       :    0.11
-----------------------------------------------
CPU time used                    :  00:00:00.00
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 1000,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    3
Chi2 statistic for size of jumps      :    3.57
p-value of test                       :    0.31
-----------------------------------------------
Normal statistic for number of jumps  :    0.95
p-value of test                       :    0.17
-----------------------------------------------
CPU time used                    :  00:00:00.00
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 5000,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    5
Chi2 statistic for size of jumps      :    1.78
p-value of test                       :    0.88
```

```
-----------------------------------------------
Normal statistic for number of jumps  :  -0.053
p-value of test                       :    0.52
-----------------------------------------------
CPU time used                    :  00:00:00.01
***********************************************************

std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 25000,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    8
Chi2 statistic for size of jumps      :    9.45
p-value of test                       :    0.31
-----------------------------------------------
Normal statistic for number of jumps  :    1.44
p-value of test                       :    0.07
-----------------------------------------------
CPU time used                    :  00:00:00.45
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 50000,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    9
Chi2 statistic for size of jumps      :    5.63
p-value of test                       :    0.78
-----------------------------------------------
Normal statistic for number of jumps  :  -32.26
p-value of test                       : 1 - eps1    *****
-----------------------------------------------
CPU time used                    :  00:00:01.35
***********************************************************
std mt19937_64 [High bits]
scomp_LinearComp test:
-----------------------------------------------
   N =  1,  n = 75000,  r =  0,    s = 1
-----------------------------------------------
Number of degrees of freedom          :    9
Chi2 statistic for size of jumps      :   10.25
p-value of test                       :    0.33
-----------------------------------------------
Normal statistic for number of jumps  :  -91.51
p-value of test                       : 1 - eps1    *****
-----------------------------------------------
CPU time used                    :  00:00:01.68
```

Here, in 1.68 seconds of testing, we learn exactly how/where the Mersenne Twister fails. It survives up to 50,000 outputs, but then fails. We could run BigCrush instead, but we'd wait more than three hours to learn less about this failure. If there were other failures, we'd learn about them, too, but there aren't any; this is the only one. Linear complexity is the test that catches out all (G)LFSR generators that have insufficiently strong output functions.

Conclusion

Using a random-number test suite isn't hard. You can do it, too!

testing **testu01**