How to Test with PractRand

M.E. O'Neill — 2017-08-11 11:27

I first mentioned and gave an overview of PractRand in this blog post (pcg-passes-practrand.html). Since then, I've been having “fun” testing with PractRand and the results have shown up in several subsequent posts (../categories/practrand.html). You could be having similar fun, too! Let's see how, with a worked example: testing the Mersenne Twister (https://en.wikipedia.org/wiki/Mersenne_Twister).

Downloading and Building PractRand

You can find PractRand's website here (http://pracrand.sourceforge.net/). The instructions are a bit basic, but you basically just compile all the files, link them together to build the `RNG_test` program, and you're done.

That said, if you want it to run a bit faster, you might want to apply this patch (../downloads/practrand-0.93-bigbuffer.patch) which causes it to use a larger buffer when reading from standard input.

Here's a complete start-to-finish build process for the PractRand's test program on Linux or OS X:

```
mkdir PractRand
cd PractRand
curl -OL https://downloads.sourceforge.net/project/pracrand/PractRand_0.93.zip
unzip -q PractRand_0.93.zip
curl -sL http://www.pcg-random.org/downloads/practrand-0.93-bigbuffer.patch | patch -p
g++ -std=c++14 -c src/*.cpp src/RNGs/*.cpp src/RNGs/other/*.cpp -O3 -Iinclude -pthreac
ar rcs libPractRand.a *.o
rm *.o
g++ -std=c++14 -o RNG_test tools/RNG_test.cpp libPractRand.a -O3 -Iinclude -pthread
```

Making a Output Program

The easiest way to test with PractRand is to make a program that spews a stream of random bytes (in binary) to standard output. Writing code to do so is very straightforward when you have a generator that produces random unsigned 8-bit, 16-bit, 32-bit, 64-bit, or even 128-bit numbers. Some old generators (e.g., Unix `rand()`, `random()` and `drand()` and C++'s `std::minstd_rand`) don't have a full use-all-the-bits power-of-two output range, but general-purpose modern ones almost always do.

Testing the 32-Bit Mersenne Twister

Now we'll put together some code for to test C++'s Mersenne Twister. As you can see, the essence of the code we need is very, very simple:

```
#include <cstdio>
#include <cstdint>

#include <random>

int main()
{
    freopen(NULL, "wb", stdout);  // Only necessary on Windows, but harmless.

    std::mt19937 rng(42);

    while (1) {
        uint32_t value = rng();
        fwrite((void*) &value, sizeof(value), 1, stdout);
    }
}
```

We can compile it like this,

```
linux$ g++ -std=c++14 -O3 -Wall -o mt19937-min mt19937-min.cpp
```

(on Linux, `gcc` will warn about the `freopen` line; you can just comment it out because it's only needed if you're using Windows).

With the code compiled, we can do a quick check:

```
linux$ ./mt19937-min | hexdump -Cv | head
00000000  66 dc e1 5f b3 3d ea cb  5c 03 62 f3 0e 95 f5 2e  |f.._.=..\.b.....|
00000010  6a f4 63 bb 47 d4 99 c7  bc ae 41 99 14 2c cb 98  |j.c.G.....A..,..|
00000020  66 d6 f0 27 79 18 22 72  d2 41 ef 27 d6 f4 97 19  |f..'y."r.A.'....|
00000030  4a 91 de 0e ca 55 91 75  57 b8 bd dd 74 ed 6d 55  |J....U.uW...t.mU|
00000040  63 ac e2 99 67 eb 92 24  97 3e 44 b5 82 a0 a0 a6  |c...g..$.>D.....|
00000050  95 06 45 05 34 fd 70 0e  01 03 4c f8 57 e9 d4 b8  |..E.4.p...L.W...|
00000060  eb f5 1a d5 9d fd 44 f0  25 db 5b 36 81 09 33 00  |......D.%.[6..3.|
00000070  bf 14 8c 2e bb 93 01 fe  14 99 f3 2e a0 44 13 9e  |.............D..|
00000080  cb d1 e2 4d 39 4d 95 9c  15 70 56 86 fc 18 cf 01  |...M9M...pV.....|
00000090  eb f2 93 6e 58 6b e7 05  30 fe 8d 4a da a1 57 86  |...nXk..0..J..W.|
```

You should *always* do a quick sanity check with a hex dump just to guard against face-palm-level bugs. For example, if you see output like

```
00000000  66 dc e1 5f 00 00 00 00  b3 3d ea cb 00 00 00 00  |f.._.....=......|
00000010  5c 03 62 f3 00 00 00 00  0e 95 f5 2e 00 00 00 00  |\.b.............|
00000020  6a f4 63 bb 00 00 00 00  47 d4 99 c7 00 00 00 00  |j.c.....G.......|
00000030  bc ae 41 99 00 00 00 00  14 2c cb 98 00 00 00 00  |..A......,......|
00000040  66 d6 f0 27 00 00 00 00  79 18 22 72 00 00 00 00  |f..'....y."r....|
00000050  d2 41 ef 27 00 00 00 00  d6 f4 97 19 00 00 00 00  |.A.'............|
00000060  4a 91 de 0e 00 00 00 00  ca 55 91 75 00 00 00 00  |J........U.u....|
00000070  57 b8 bd dd 00 00 00 00  74 ed 6d 55 00 00 00 00  |W.......t.mU....|
00000080  63 ac e2 99 00 00 00 00  67 eb 92 24 00 00 00 00  |c.......g..$....|
00000090  97 3e 44 b5 00 00 00 00  82 a0 a0 a6 00 00 00 00  |.>D.............|
```

it means that that the value variable is a 64-bit variable, but the output it 32 bits! (With GCC, for example, a `uint_fast32_t` holds a 32-bit value but is 64-bits wide.)

Finally, we can run PractRand to test it,

```
linux$ ./mt19937-min | ./RNG_test stdin32
RNG_test using PractRand version 0.93
RNG = RNG_stdin32, seed = 0xc98d226b
test set = normal, folding = standard (32 bit)

rng=RNG_stdin32, seed=0xc98d226b
length= 128 megabytes (2^27 bytes), time= 3.9 seconds
  no anomalies in 117 test result(s)

rng=RNG_stdin32, seed=0xc98d226b
length= 256 megabytes (2^28 bytes), time= 8.0 seconds

...
```

We can see that it runs without problem and doesn't see any statistical flaws (yet!). When starting out with a test suite, it's good to make sure everything is working by using a generator that won't fail immediately. The Mersenne Twister is one such generator, it'll fail at 256 GB of output, but it'll behave okay until then.

Testing the 64-bit Mersenne Twister

For the 64-bit Mersenne Twister, we could leave the test program almost the same. We just need to change the declared type of the generator and the type of the `value` variable,

```
#include <cstdio>
#include <cstdint>

#include <random>

int main()
{
    freopen(NULL, "wb", stdout);  // Only necessary on Windows, but harmless.

    std::mt19937_64 rng(42);

    while (1) {
        uint64_t value = rng();
        fwrite((void*) &value, sizeof(value), 1, stdout);
    }
}
```

and then run it, remembering to use the `stdin64` option,

```
linux$ ./mt19937_64-min | ./RNG_test stdin64
RNG_test using PractRand version 0.93
RNG = RNG_stdin64, seed = 0x82bfa213
test set = normal, folding = standard (64 bit)

rng=RNG_stdin64, seed=0x82bfa213
length= 128 megabytes (2^27 bytes), time= 3.6 seconds
  no anomalies in 148 test result(s)

rng=RNG_stdin64, seed=0x82bfa213
length= 256 megabytes (2^28 bytes), time= 8.1 seconds
  no anomalies in 159 test result(s)

^C
```

Notice that it's a tad faster because it's generating twice as much data at every iteration. In fact, outputting values in a one-at-a-time fashion is easy but needlessly slow. We'll get better performance if we generate a larger block of numbers to output. Adding a little bit more code to create a buffer,

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

#include <random>

int main()
{
    freopen(NULL, "wb", stdout);  // Only necessary on Windows, but harmless.

    std::mt19937_64 rng(42);
    constexpr size_t BUFFER_SIZE = 1024 * 1024 / sizeof(uint64_t);
    static uint64_t buffer[BUFFER_SIZE];

    while (1) {
        for (size_t i = 0; i < BUFFER_SIZE; ++i)
            buffer[i] = rng();
        fwrite((void*) buffer, sizeof(buffer[0]), BUFFER_SIZE, stdout);
    }
}
```

gives us results more quickly:

```
linux$ ./mt19937_64-buffer | ./RNG_test stdin64
RNG_test using PractRand version 0.93
RNG = RNG_stdin64, seed = 0xd26c00ae
test set = normal, folding = standard (64 bit)

rng=RNG_stdin64, seed=0xd26c00ae
length= 128 megabytes (2^27 bytes), time= 2.3 seconds
  no anomalies in 148 test result(s)

rng=RNG_stdin64, seed=0xd26c00ae
length= 256 megabytes (2^28 bytes), time= 5.3 seconds
  no anomalies in 159 test result(s)

rng=RNG_stdin64, seed=0xd26c00ae
length= 512 megabytes (2^29 bytes), time= 11.0 seconds
  Test Name                         Raw       Processed     Evaluation
  BCFN(2+2,13-2,T)                  R=  -7.0  p =1-1.2e-3   unusual
  [Low1/64]BCFN(2+2,13-6,T)         R=  -5.7  p =1-1.0e-3   unusual
  ...and 167 test result(s) without anomalies

rng=RNG_stdin64, seed=0xd26c00ae
length= 1 gigabyte (2^30 bytes), time= 21.7 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low1/64]DC6-9x1Bytes-1           R=  +5.5  p =  7.1e-3   unusual
  ...and 179 test result(s) without anomalies

rng=RNG_stdin64, seed=0xd26c00ae
length= 2 gigabytes (2^31 bytes), time= 43.2 seconds
  no anomalies in 191 test result(s)

rng=RNG_stdin64, seed=0xd26c00ae
length= 4 gigabytes (2^32 bytes), time= 82.8 seconds
  no anomalies in 201 test result(s)

^C
```

We see a couple of blips here where PractRand observes slightly unusual output. With lots of tests applied, that'll happen once in a while and doesn't necessarily indicate a real problem. But if we could set the seed, we could do another run to be sure. I'll leave writing the code as an exercise for you, but here's the full test results for the 64-bit Mersenne Twister with that functionality added and code to print the generator name and seed to standard error:

```
linux$ ./mt19937_64 | ./RNG_test stdin64
std::mt19937_64(0xcfa51619e9100201) initialized.
RNG_test using PractRand version 0.93
RNG = RNG_stdin64, seed = 0x42cee57d
test set = normal, folding = standard (64 bit)


rng=RNG_stdin64, seed=0x42cee57d
length= 128 megabytes (2^27 bytes), time= 2.6 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low4/64]DC6-9x1Bytes-1           R=  -4.8  p =1-2.2e-3   unusual
  ...and 147 test result(s) without anomalies

rng=RNG_stdin64, seed=0x42cee57d
length= 256 megabytes (2^28 bytes), time= 5.5 seconds
  no anomalies in 159 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 512 megabytes (2^29 bytes), time= 11.7 seconds
  no anomalies in 169 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 1 gigabyte (2^30 bytes), time= 21.7 seconds
  no anomalies in 180 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 2 gigabytes (2^31 bytes), time= 41.6 seconds
  no anomalies in 191 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 4 gigabytes (2^32 bytes), time= 78.7 seconds
  no anomalies in 201 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 8 gigabytes (2^33 bytes), time= 160 seconds
  no anomalies in 212 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 16 gigabytes (2^34 bytes), time= 298 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low16/64]BCFN(2+9,13-4,T)        R= +11.5  p =  5.6e-5   unusual
  ...and 222 test result(s) without anomalies

rng=RNG_stdin64, seed=0x42cee57d
length= 32 gigabytes (2^35 bytes), time= 623 seconds
  no anomalies in 233 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 64 gigabytes (2^36 bytes), time= 1186 seconds
  no anomalies in 244 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 128 gigabytes (2^37 bytes), time= 2351 seconds
  no anomalies in 255 test result(s)
```

```
rng=RNG_stdin64, seed=0x42cee57d
length= 256 gigabytes (2^38 bytes), time= 4505 seconds
  no anomalies in 265 test result(s)

rng=RNG_stdin64, seed=0x42cee57d
length= 512 gigabytes (2^39 bytes), time= 9219 seconds
  Test Name                         Raw       Processed     Evaluation
  BRank(12):24K(1)                  R=+99759  p~= 0           FAIL !!!!!!!!
  [Low16/64]BRank(12):16K(1)        R= +1165  p~=  1.3e-351   FAIL !!!!!!!
  [Low1/64]BCFN(2+0,13-0,T)         R=  +8.2  p =  6.0e-4   unusual
  ...and 273 test result(s) without anomalies
```

So there we go! In PractRand's tests, the 64-bit Mersenne Twister is good to 256 GB, but fails (badly!) at 512 GB. (TestU01 has a more specialized and sensitive test that can detect the same linear complexity issues in only 50,000 outputs, but arguably PractRand's more generalized test provides a more realistic measure of the seriousness of the issue.)

Integrating into PractRand Instead?

Instead of using the standard-input interface, you can integrate your code directly into PractRand. It actually already has a built-in Mersenne Twister, so we can see if using it offers any advantage:

```
linux$ ./RNG_test mt19937
RNG_test using PractRand version 0.93
RNG = mt19937, seed = 0xc5937f7
test set = normal, folding = standard (32 bit)

rng=mt19937, seed=0xc5937f7
length= 128 megabytes (2^27 bytes), time= 3.0 seconds
  no anomalies in 117 test result(s)

rng=mt19937, seed=0xc5937f7
length= 256 megabytes (2^28 bytes), time= 6.4 seconds
  Test Name                         Raw       Processed     Evaluation
  DC6-9x1Bytes-1                    R=  -3.9  p =1-7.2e-3   unusual
  ...and 123 test result(s) without anomalies

rng=mt19937, seed=0xc5937f7
length= 512 megabytes (2^29 bytes), time= 11.7 seconds
  no anomalies in 132 test result(s)

^C
```

Meh. It's faster but not a *lot* faster. I don't think it's worth the hassle of learning the internals of PractRand or the risks (e.g., no more `hexdump` checks to make sure we've not made a silly blunder). I like having a separate program I can test. But if you want to go that way, go for it!

Conclusion

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

(I recommend you don't try Xoroshiro128+ or Xorshift128+ as your first generator to test; you might think your setup isn't working. Try something else first.)

mt19937 **practrand** **testing**