Performance Analysis and Simulation of Communication Systems: Project A

Adam Temmel (adte1700) & Fredrik Sellgren (frse1700)

2021/08/31

1 PRNG and Random Variables

1.1 Create a function that implements a linear congruential generator (LCG), accepting as input the parameters: seed, m, a,and c.

We implemented our LCG as a class in order to keep it somewhat similiar to the STL generators. The code used is as follows:

```
class Lcg {
public:
    using seed_t = uint32_t;

    Lcg(seed_t seed, seed_t m = 100, seed_t a = 13,
        seed_t c = 1)
        : seed(seed), m(m), a(a), c(c) {}

    seed_t operator()() {
        seed = ((a * seed) + c) % m;
        return seed;
    }

    // These are kept public to make it easier to change
    // them later in the lab, a more authentic generator
    // would probably keep them private.
    seed_t seed, m, a, c;
};
```

Figure 1: LCG Class

1.2 Generate 1000 values uniformly distributed in the range [0,1] using your PRNG. For this case use m=100, a=13 c=1and seed =1;

The code used to generate 100 uniformly distributed values is presented in *2*.

```cpp
template<typename T>
double normalize(T x, T min, T max) {
    return static_cast<double>(x - min)
        / static_cast<double>(max - min);
}

void testPrng() {
    Lcg lcg(1);

    auto urv = CreateObject<UniformRandomVariable> ();
    constexpr size_t n = 1000;

    std::vector<double> ourResults(n);

    std::generate(ourResults.begin(), ourResults.end(),
            [&]() {
        return normalize(lcg(), 0u, lcg.m);
    });

    std::vector<double> theirResults(n);

    std::generate(theirResults.begin(), theirResults.end(),
            [&]() {
        return urv->GetValue(0., 1.);
    });

    {
        std::ofstream ourFile("our_results.txt");
        for(auto f : ourResults) {
            ourFile << f << '␣';
        }
    }

    {
        std::ofstream theirFile("their_results.txt");
        for(auto f : theirResults) {
            theirFile << f << '␣';
        }
    }
}
```

Figure 2: Code used to write 1000 uniformly generated values from each generator to disk

1.3 Compare the distribution of your values with the distribution of values generated using the `UniformRandomVariable()` of ns-3.

Two histograms comparing our generator to the Ns3 generator are visualized in *figure 3*.

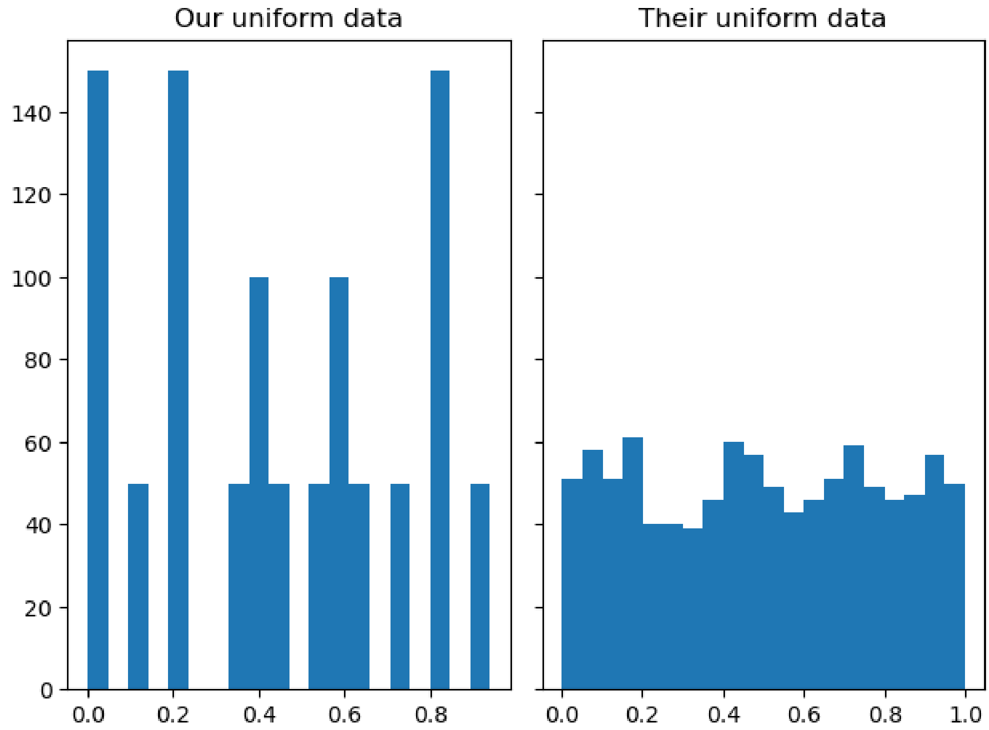

Figure 3: The corresponding plots gathered from the data above

1.4 Comment on the difference in the results and propose values of m, a, and c which gives you better results.

The main issue we found with the parameters given comes from the divisor used when performing the modulus operation, m. As m was initially set to 100, we are guaranteed to only generate numbers in $[0, 99]$. It then follows that our generator can only generate 100 different values. This might be enough for some cases, but seeing as we wish to normalize it into a continuous value between $[0, 1]$, the lack of representable numbers provided by the generator will have a negative impact on the final result. As such, it is in our interest to increase the divisor. One suggested method of improving the generator is to use a divisor that is a power of 2, such as 2^{32} or 2^{64} combined with keeping c at 0[1]. Another possible competitor for good divisors are the *Mersenne Primes*, such as $2^{32} - 1$ or $2^{64} - 1$[2], which also happens to be exactly the fix we implemented. This improvement proved to make a noticeable difference, which is illustrated in *figure 4*.

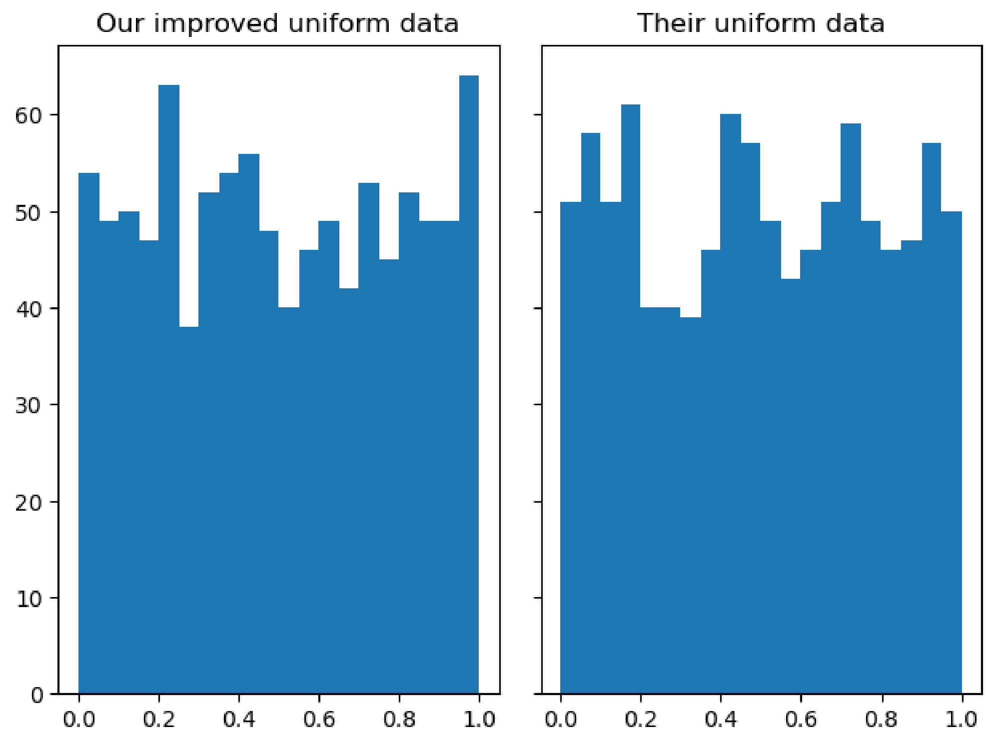

Figure 4: Our improved generator compared to the Ns3 generator

1.5 What PRNG does ns-3 use? What method does ns-3 use to generate a normal random variable?

The PRNG that is used in Ns3 is the MRG32k3a generator[3]. The underlying implementation comes from Pierre L'Ecuyer's (among others) random number generator package[4]. MRG32k3a implements a way to generate random numbers capable of being separated into disjoint substreams that are independent from one another. Ns3 states that the total period of the generator is $3.1 \cdot 10^{57}$[3].

1.6 Using the `time` system command of Linux compare the execution time for the generation of the uniform distribution using your function and ns-3 function

```
# our class:
time ./waf --run scratch/project  2.54s user 0.20s
system 107% cpu 2.547 total
# their class:
time ./waf --run scratch/project  2.59s user 0.17s
system 107% cpu 2.556 total
```

Figure 5: The results from timing a program that generated 1000 random numbers using `time`

As we can see, our implementation is slightly faster (around 0.01 seconds) than the Ns3 implementation.

1.7 Write a second function that generates an exponential distribution with mean $\beta > 0$ from a uniform distribution generated using the LCG; Choose one of the methods for generating RV covered in the course and motivate your choice with respect to the specific task.

The course brought up the concept of *The Inverse Transform Algorithm*. It operates as follows:

1. Generate $u \in U(0, 1)$.
2. Define a function $F(X)$, which represents the distribution of the random data you wish to generate. In our case, this was an exponential distribution, so something akin to $F(X) = 1 - e^x$.
3. Solve the equation $F(X) = U$ for X, finding the inverse $X = F^{-1}(U)$. In our case, $F^{-1}(U) = -ln(1 - U)$.
4. By inserting u into F^{-1}, we can now convert a uniform $[0, 1]$ variable into the desired distribution.

We argued that the Inverse Transform Algorithm was the most appropriate method for the task, as we have a specific target function we wish to reach, which rules out a few of the alternative methods. Furthermore, all the steps involved in this algorithm are arguably less complex compared to, say, The Rejection Method.

```
double expDist(double value, double lambda) {
    return -log(1 - value) / lambda;
}
```

Figure 6: Resulting code after implementing the Inverse Function Transform

1.8 Compare your exponential distribution with ns-3 `ExponentialRandomVariable()` and the theoretical expression of the probability density function.

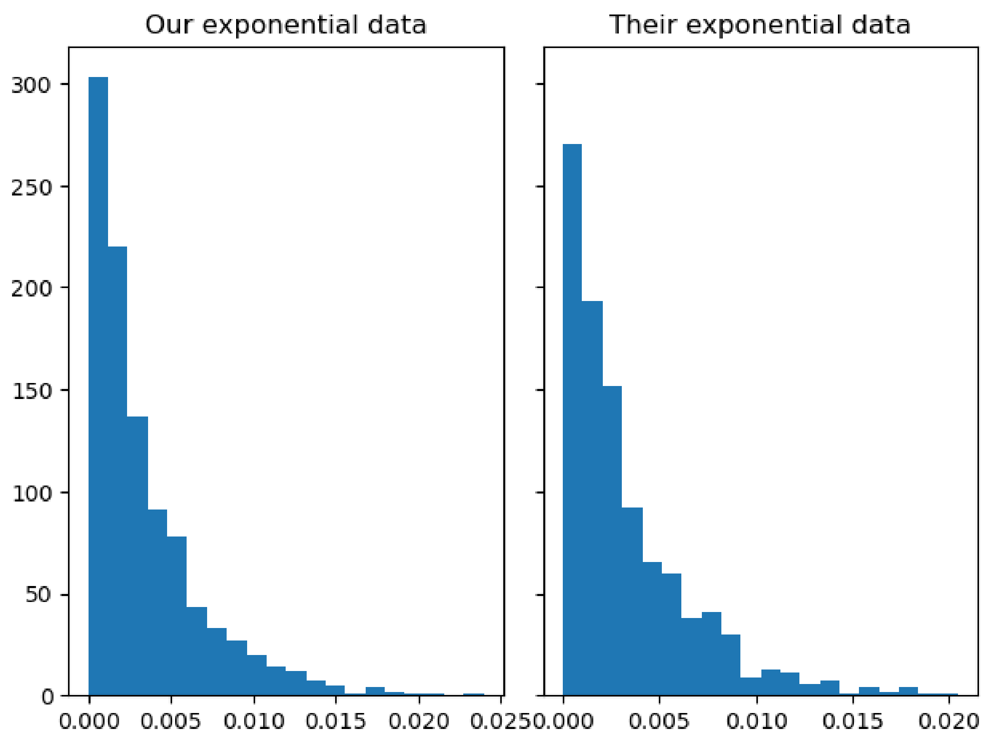

Figure 7: Our exponentially distributed generator compared to the Ns3 counterpart

As is illustrated in *figure 7*, the two generators ended up performing on a somewhat equal level.

Citations

References

[1] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Professional, 1997. Chap. Seminumerical Algorithms, pp. 10–26.

[2] Wikipedia. *Mersenne Primes*. URL: `https://en.wikipedia.org/wiki/Mersenne_prime` (visited on 08/29/2021).

[3] Ns3. *Random Variales*. URL: `https://www.nsnam.org/docs/manual/html/random-variables.html` (visited on 08/29/2021).

[4] PIERRE L'ECUYER, PIERRE L'ECUYE, E. JACK CHEN, et al. "AN OBJECT-ORIENTED RANDOM-NUMBER PACKAGEWITH MANY LONG STREAMS AND SUBSTREAMS". In: (2000).