Vu Nguyen
CS 255
Dr.Teng Moh

Improving Pseudo Random Number generator by using Chaotic iterations

In computer science, random numbers serve as the foundation of many algorithms like statistical sampling, simulation and cryptography. However the method in which a computer would generate a random number is split between two: hardware and software. While hardware random number generators are able to produce true randomness due to physical process rather than algorithmist approach , its production is limited and slow due to entropy. In contrast, software methods produce faster random numbers by relying on algorithms to produce a random number, but often cannot truly be random due to algorithm limitation and computer deterministic properties . A software method of producing a random number is called a pseudo-random number generator or deterministic random bit generator. Due to this limitation, many algorithms that rely on consuming and using massive amounts of random numbers cannot use hardware produced numbers, but instead rely on software methods. By improving the current method of producing pseudo random numbers, we can improve the accuracy of our statistical analysis by having a more random randomized algorithm procuring these results in a non bias and non deterministic way. Using chaotic iterations on top of an existing random number algorithm can greatly improve the quality of the number generated, enhancing its desired chaotic properties.

In order to produce true randomness, we need to use physical processes (hardware) . This is because physical processes are considered stochastic, completely unpredictable. However they are slow due to the nature of these random processes. In recent years there is a phenomenon of using quantum random numbers generators chips, typically involving a quantum source; usually a dedicated quantum processor to generate sequences of random numbers at the speed of a software generator This method is expensive and often only available for enterprise use.

Currently, the best known software algorithm to generate a random number is using a linear congruential generator(LCGs) as stated above. This method is able to produce a sequence of pseudo-randomized numbers very fast and requires little memory to do so. This makes it very desirable when doing real time simulations. However there are specific weaknesses that LCG has. First, this algorithm is not intended to be used for cryptographic applications because it does not satisfy the randomize properties cryptographic required, second LCGs do not satisfy the Birthday theorem, an important statistical property where a true random number generator with 32 bits of output is expected to begin duplicating earlier outputs after $\sqrt{m} \approx 2^{16}$ results. LCG algorithm will only produce duplicates after its full period elapses, this is a major statistical flaw that can lead to inaccurate results when sampling. Third, it becomes a disadvantage to use LCG if the modulus is too low; the quality of the random number produced by LCG will be low quality and will not pass the spectral test, a test given to any pseudorandom number generators to test its quality. In order to pass the spectral test, LCG needs to use a modulus much greater than the cube of the number of random samples which are required. So a 32 bit LCG can only obtain about a thousand random, a 64 bit LCG can only produce a good 2^{31} random samples. For this reason, LCG is not suitable for large-scale Monte Carlo simulation. A Sample code of LCG in python:

```
def lcg (modulus, a, c, seed):
    """Linear congruential generator."""
    while True:
```

```
seed = (a * seed + c) % modulus
yield seed
```

So what about the programming language libraries we use to generate random numbers, libraries like java.util.random found in java, or random.py in python? They all use versions of Linear Congruential Generator (LCG) to generate their number. Java.util.random for example is taken for unix rand48 implementation, which uses 3 different value a,c, and m parameters for LCG. “... we begin with some start or "seed" number which ideally is "genuinely unpredictable", and which in practice is "unpredictable enough". For example, the number of milliseconds— or even nanoseconds— since the computer was switched on is available on most systems. Then, each time we want a random number, we multiply the current seed by some fixed number, *a*, add another fixed number, *c*, then take the result modulo another fixed number, *m*. The number *a* is generally large”. However there are drawbacks with this method, given that the current seed is unpredictable, however the next seed is based on the previous seed. Since for a given "current seed" value, the "next seed" will always be completely predictable based on that value, the series of numbers *must* repeat after at most *m* generations. This is called the **period** of the random number generator. In the case of java.util.Random, *m* is 2^{48} and the other values have indeed been chosen so that the generator has its maximum period. Therefore, randomness only happens during a small period.

We understand the shortcoming of pseudo random number algorithms, however given what we learned in Lemma 5.2 ; given the online hiring problem, it is in our advantage that we picked our candidate at random; that ois if there is a true random order in our array iteration of application, then the average case is o(log(n)) of hired & fired. We must ensure that our order is truly random to obtain O(log(n)) hiring and firing only a small subset of our candidate or else we risk going to our worst case is O(n) ie. hiring and firing every single candidate. So how can we ensure randomness with our random sampling? We can add another layer on top of our already produced random number; a naive approach is to shuffle and slice the array using Array.shuffle(); and take the n elements of the shuffled array based on our random generator. This technique will produce a true random result, however the time taken increases linearly so it is not very efficient to use for large arrays.

Work Cited

Bahi, Jacques M., et al. “An Optimization Technique on Pseudorandom Generators Based on Chaotic Iterations.” *ArXiv:1706.08773 [Nlin]*, June 2017. *arXiv.org*, http://arxiv.org/abs/1706.08773.