Utah Valley University

Modular Pseudo-Random Number Generator using Math Equations, Existing Generators, and Noise Generators

Created by

Daniel Espinel

UVID: 10743799

August 13, 2021

Problem Statement

Due to the nature of technology, random number generators aren't truly random. We can get close through various mathematical theorems such as the Minimal Standard Generator and the Box-Muller Method. The least predicable a random number generator is, the more secure it is. In cryptography, this is crucial. If you can retrace the output back to the seed, you effectively defeat the randomness of the generator. But, how predictable would a generator be if it was not only constantly changing it's parameters mid-generation, but also modular? For this random number generator, I will be testing the viability of a generator like this. I will be using a pool of 10 mathematical equations, 10 existing random number generators, and the noise generator Fast Noise Lite created by Jordan Peck for the purposes of this research paper.

Methodology

There are 4 main steps when generating a number, create a seed, generate a chunk based on that seed, insert that chunk into the final number, repeat until finished. To get the most out the modularity of the system. There are many of what I like to call “Shuffling Steps”. These steps are the core part of the system and why I went with a multi threaded approach.

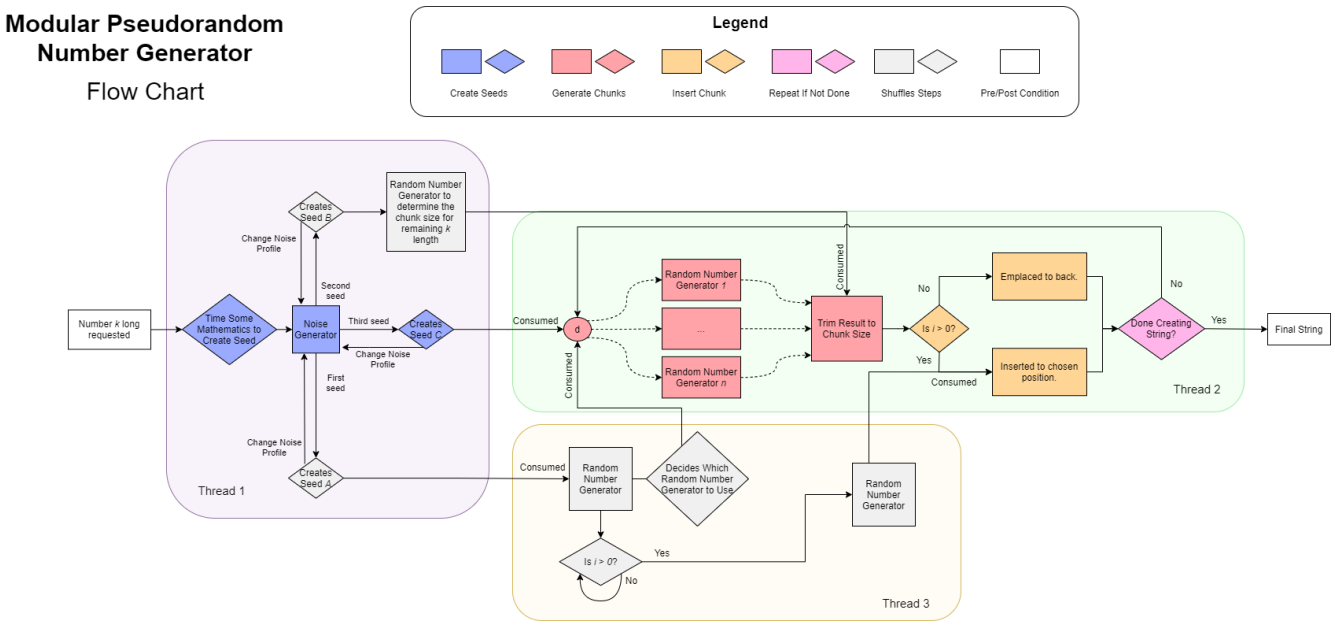


Figure 1: System Flow Chart

Looking at Figure 1, we can see that the first generates three different seeds. Before it starts this, it needs an initial seed to start working. This will be created by timing some mathematical computation to the nanosecond. The work equation in the default position will use the system time as it’s variable. The time that elapsed is then used as a seed for the noise generator. From here, we can start generating seeds. While one seed is meant for generating chunks, the other two are also very important. The Seed A goes to the third thread. This thread takes that seed and uses it to generate two indexes. One index picks which generator will be used to generate a chunk, while the other chooses which where to insert

the generated chunks into. Both of these will use the generator in the default position (for the purposes of this paper, this is the Default_Random_Engine). Going back to thread 1, Seed B will be used to generated chunk sizes. Much like Thread 3, this will also be using the Default_Random_Engine. Every time a seed is generated in Thread 1, it is then used as a variable for more random work to be timed. This result is then used to change the settings of the noise generator and it's seed.

The reason why timed work a fundamental part of generating seeds, is due to the fact that computation in the time-scale of nanoseconds varies greatly. Some computer hardware might be slower in calculating then others and your CPU might by more busy with system calls or other tasks at the time of calculation. The asynchronous nature of modern systems make this in interesting avenue worth researching.

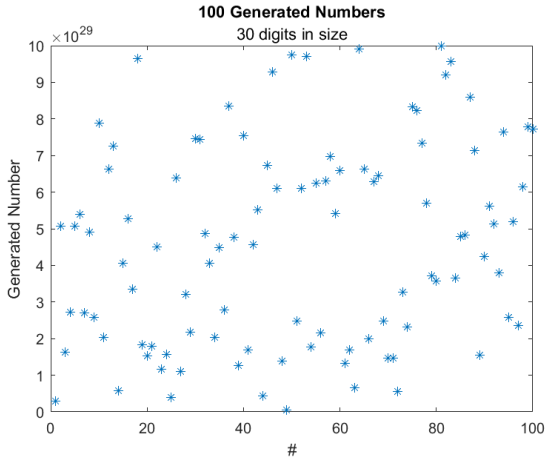
A list of the generators (including technical links) and mathematical equations used for this paper are included as a comment block in Libraries.h.

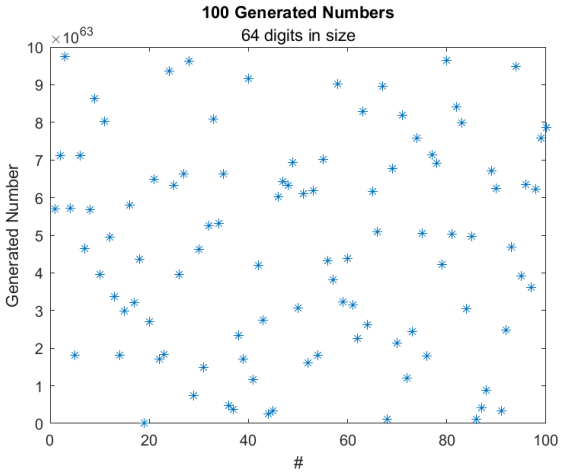
Software Design

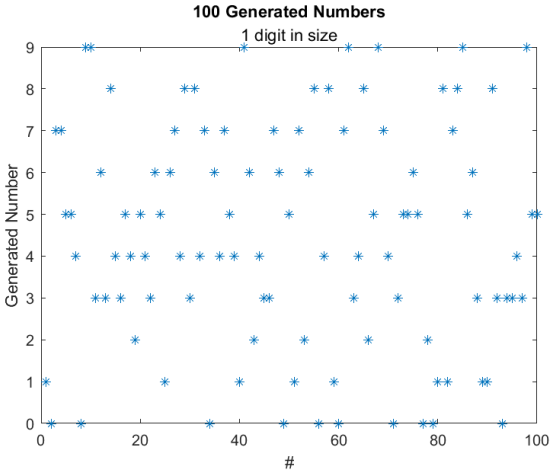
The system is multi-threaded and written in C++ due to it's efficiency in computation; and that it recently standardized threading. The generator is a producer-consumer style architecture that uses static resources declared in the library header file. Since there is allot of seeds being generated, there are multiple mutexes and points of race conditions. For this reason, I have also invested allot of time in error handling. Each thread is wrapped in a try catch block since threads have to handle their own exceptions and I have multiple assertions; particularly in Thread 3 because this is where most of the failures can occur.

When it comes to the modular part of the system, I have made importing code swift. The pools of generators and mathematical equations are set up as an object, this was mainly done for organizational purposes. The only things needed to add or remove from these pools, are is the reference to it/them in the object, and updating the total count in the enumeration listed above. The rest is handled by the generator.

Results







When importing the data into Matlab, I found out that the standard deviation of the data slightly from round $2.5 \text{ x } 10^{digits}$. Even in an extreme example of 1 digit sized numbers, this seemed to be the standard deviation of the current setup the generator.

Running the Generator

There are two ways to use this generator. Using the included executable, or compiling an executable yourself. If you are running Windows, then you can use the included executable to start generate numbers. Run NumGen.exe and follow the command line prompts. You will first be asked how large you want the generated number to be. This can range from 1 to INT_MAX (2147483648). Afterwards, you will be asked if you want to print 100 numbers to a file. I included this option to make analyzing in MatLab easier. The delimiter is '#'. If you only want to generate one number, then select no. Either way, the number(s) will be saved to a file after being generated. Due to the standardization of threading still being early on C++ (particularly the implementation of waking threads), the program may halt or crash during generation. If this happens, then run the program again.

If you are running a computer not running on Windows. You can't run the executable if you are not on Windows due the threading library being used in this compiled executable uses the win32 library. However, you can still run this generator if you recompile using posix style threads. To do this, please follow the instructions in README.txt.