

“Randomness and Random Numbers”

Mehul Dinesh Jain

Term paper submitted to

Department of Computer Science

Dr. M. N. Nachappa (HOD)

St. Joseph’s College (Autonomous)

August 2017

ST. JOSEPH'S COLLEGE (AUTONOMOUS)
BANGALORE



TERM PAPER SUBMISSION CERTIFICATE

This is to certify that Smt. / Sri. _____ has
satisfactorily completed the course of titled
_____ under the guidance of Dr. / Prof.
_____ prescribed by St. Joseph's College
(Autonomous) of Bangalore University for the degree course in BCA/B.Sc for the year
2017-18.

Guide

Head of Department

Name: _____

Reg. No. : _____

Date Of Submission: _____

Abstract

‘Randomness’ is not a new phenomenon. Randomness has been used since ancient times to predict future events or take decisions. In the current world randomness reveals itself in many fields like statistics, biology, physics, etc. We have many random number generating algorithms, with new ones coming in all with one goal of generating ‘True Random’ Numbers. The irony of randomness in computers is that we are trying to achieve random using mathematics. We see events that seem random, but we do not question whether it is truly random. Algorithms designed to generate random numbers have taken computing further ahead, but the right one must be used by the user.

Randomness in computers is used in different fields differently depending on the type of randomness required. Is it possible to implement order using chaos? If Random is chaos then how is it that we derive information from it? How is it possible that a computer that only understands 1’s and 0’s as information creates chaos known as random? In this paper we shall see the basics of randomness, where it has been used, its current applications, how random numbers are generated in computers, and to prove why study is necessary if we are to understand the true meaning of randomness and use them in computing, games, art, cryptography and whether our understanding of random is a Pseudo understanding or True.

Keywords: *Randomness, TRNG, PRNG, CSRNG, CSPRNG, Procedural Generation, Cryptography, Keys, Quantum-mechanics, Seed, Random Numbers, Events.*

Introduction

Randomness has been used since ancient times in events like dice rolling, coin flipping, with one reason being the end result to be left to a random chance. Random Number generators in computers are similar; it’s trying to achieve an unpredictable, and random result. Mostly Random Numbers are generated use a mathematical formula. These numbers are called Pseudo-Random. They are drawn from a set of possible values, each of which has an equal chance of occurring (uniform distribution), and must be independent of the other, meaning that no one event influences the chance of the other.

Randomness is the lack of pattern or predictability in events. A random sequence of events, symbols or steps has no order and does not follow an intelligible pattern or combination.

In ancient history, the concepts of chance and randomness were intertwined with that of fate. Many ancient peoples threw dice to determine fate, and this later evolved into games of chance. At the same time, most ancient cultures used various methods of prophecy to attempt to evade randomness and fate.

The Chinese were perhaps the earliest people to formalize odds and chance 3,000 years ago. The Greek philosophers discussed randomness at length, but only in non-quantitative forms. It was only in the sixteenth century that Italian mathematicians began to formalize the odds associated with various games of chance. The invention of modern calculus had a positive impact on the formal study of randomness.

Many scientific fields are concerned with randomness like Algorithmic probability, Chaos theory, Cryptography, Game theory, Information theory, Pattern recognition, Probability theory, Quantum mechanics, Statistical mechanics, Statistics.

Applications of randomness in physical science includes, the idea of random motions of molecules in the development of statistical mechanics to explain phenomena in thermodynamics and the properties of gases.

In biology it includes, the modern evolutionary synthesis assigns the observed diversity of life to random genetic mutations followed by natural selection. The latter retains some random mutations in the gene pool due to the systematically improved chance for survival and reproduction that those mutated genes confer on individuals who possess them.

In finance it includes, the ‘random walk’ hypothesis considers that asset prices in an organized market evolve at random, in the sense that the expected value of their change is zero but the actual value may turn out to be positive or negative. More generally, asset prices are influenced by a variety of unpredictable events in the general economic environment, and so on.

For all these applications, random numbers are generated and then applied. Randomness alone does not provide information, but using randomness we can generate information. Hence we have Random Number generators.

There are generally two type of random number generators, Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs). PRNGs are algorithms that use mathematical formulae or simply pre-calculated tables to produce sequences of numbers that appear random. A good example of a PRNG is the linear congruential method which we will see later. PRNGs are efficient, meaning they can produce many numbers in a short time, and deterministic, meaning that a given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known which is called the seed. TRNGs extract randomness from physical phenomena and introduce it into a computer. The physical phenomenon can be very simple, like the little variations in somebody's mouse movements or in the amount of time between keystrokes. Examples include Noise signals like thermal noise, photoelectric effect. A really good physical phenomenon to use is a radioactive source. The points in time at which a radioactive source decays are completely unpredictable, and they can quite easily be detected and fed into a computer, avoiding any buffering

mechanisms in the operating system. It is applied in the case of cryptography to generate random keys to transmit data securely.

Defining Random Numbers and Generators

To understand the working of random number generators we need to understand what a fair, random event is. In an example from ‘Mahabharatha’, one of the two major epics in Sanskrit, India. A character called ‘Shakuni’ uses dice that rolls the number he says to gamble with ‘Pandavas’ and win over their kingdom which later leads to the epic war depicted in the text. This is an example of an event which is NOT random but decided. In a fair situation, the dice will roll the number without any bias; without any external influence.

The earliest methods for generating random numbers, such as dice, coin flipping and roulette wheels, are still used today, mainly in games and gambling as they tend to be too slow for most applications in statistics and cryptography. Random Number Generators are divided into two main categories namely Pseudo-Random Generator (PRNG) and True Random Generator (TRNG).

Pseudo-Random Generators are those algorithms that use mathematical formulae or simply pre-calculated tables to produce sequences of numbers that appear random. A good example of a PRNG is the linear congruential method (discussed later). PRNGs are efficient, meaning they can produce many numbers in a short time, and deterministic, meaning that a given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known.

True Random Generators are those that extract randomness from physical phenomena and introduce it into a computer. The physical phenomenon can be very simple, like the little variations in somebody's mouse movements or in the amount of time between keystrokes. A really good physical phenomenon to use is a radioactive source. The points in time at which a radioactive source decays are completely unpredictable, and they can quite easily be detected and fed into a computer, avoiding any buffering mechanisms in the operating system.

Application	Most Suitable Generator
Lotteries and Draws	TRNG
Games and Gambling	TRNG
Random Sampling (e.g., drug screening)	TRNG
Simulation and Modelling	PRNG
Security (e.g., generation of data encryption keys)	TRNG
The Arts	Varies

<u>Characteristic</u>	<u>Pseudo-Random Number Generators</u>	<u>True Random Number Generators</u>
Efficiency	Excellent	Poor
Determinism	Deterministic	Nondeterministic
Periodicity	Periodic	Aperiodic

Entropy

To understand randomness, we need to know what entropy is. Entropy in physics is defined as a thermodynamic quantity representing the unavailability of a system's thermal energy for conversion into mechanical work, often interpreted as the degree of disorder or randomness in the system.

In computing, entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators.

In information theory, entropy is a measure of the uncertainty associated with a random variable. The term by itself in this context usually refers to the Shannon entropy, which quantifies, in the sense of an expected value, the information contained in a message, usually in units such as bits. Equivalently, the Shannon entropy is a measure of the average information content one is missing when one does not know the value of the random variable.

In simple we can say entropy is 'disorder of a system', this means it is the unpredictability of the system and then number of states the system can take on.

Random Number Generators

We know that there are Pseudo Random and True Random Generators, and entropy used to calculate the randomness of the random number. Let us look at a few examples of Pseudo Random Generators, under which there are some old, some new generators, and some cryptographically secure ones, and finally a few examples of True Random Generators.

Pseudo-Random Generators:

Middle Square Method -

Suggested By John Von Neuman in 1946. It works by taking any number, squaring it, removing the middle digits of the resulting number as the "random number", then using that number as the seed for the next iteration. Problem is that at some point there is repetition.

Example: squaring the number '1111' yields '1234321' which can be written as '01234321', removing the middle digits gives "2343" as the "random" number. Repeating this procedure gives "4896" as the next result, and so on.

The problem with the method is that there is too much repetition as in the case of squaring a number with '0000'.

Xorshift –

Xorshift random number generators are a class of PNRGs that were discovered by George Marsaglia. Specifically, they are a subset of linear-feedback shift registers (LFSRs) which allow a particularly efficient implementation without excessively using sparse polynomials.

LSFR is a shift register whose input bit is a linear function of its previous state. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state.

The Xorshift generates the next number in their sequence by repeatedly taking the exclusive OR of a number with a bit-shifted version of itself. This makes them extremely fast on modern computer architectures.

Xorshift generators are among the fastest non-cryptographically-secure random number generators, requiring very small code and state. Although they do not pass every statistical test without further refinement, this weakness is well known and easily amended (as pointed out by Marsaglia in the original paper) by combining them with a non-linear function, example in a xorshift+ or xorshift* generator.

A C version of two xorshift algorithms:

```
#include <stdint.h>
/* The state word must be initialized to non-zero */
uint32_t xorshift32(uint32_t state[static 1])
{
    uint32_t x = state[0];
    x ^= x << 13;
```

```

    x ^= x >> 17;
    x ^= x << 5;
    state[0] = x;
    return x;
}

```

A variation like **xorshift* generator** takes a xorshift generator and applies an invertible multiplication (modulo the word size) to its output as a non-linear transformation, as suggested by Marsaglia.

```

#include <stdint.h>

uint64_t x; /* The state must be seeded with a nonzero value. */

uint64_t xorshift64star(uint64_t state[static 1]) {
    uint64_t x = state[0];
    x ^= x >> 12; // a
    x ^= x << 25; // b
    x ^= x >> 27; // c
    state[0] = x;
    return x * 0x2545F4914F6CDD1D; }

```

Linear Congruential Generator –

The method represents one of the oldest and best-known PRNG algorithms. The theory behind the algorithm is easy to understand, and the implementation is efficient. Especially on computer hardware which can provide $X_{i+1} = (aX_i + c) \bmod m$, $i = 0, 1, 2, \dots$ modulo arithmetic by storage-bit truncation. The recurrence relation defines the generator:

- The initial value X_0 is called the seed;
- a is called the constant multiplier;
- c is the increment;
- m is the modulus;

The selection of a , c , m and X_0 drastically affects the statistical properties such as mean and variance, and the cycle length.

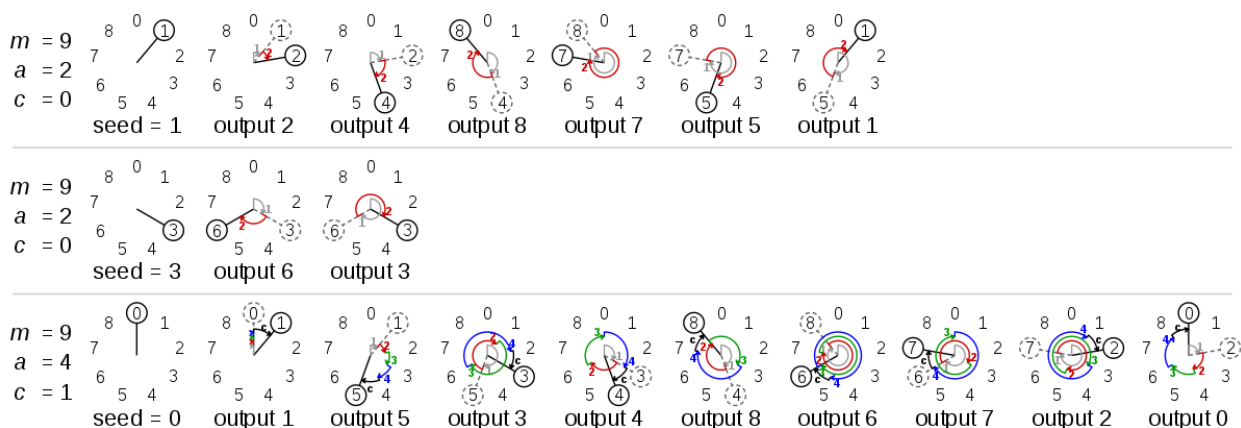
When $c \neq 0$, the form is called the *mixed congruential method*; When $c = 0$, the form is known as the *multiplicative congruential method*.

Common built-in `rand()` functions used this algorithm like Borland C/C++, Visual Basic 6 (and earlier), etc. LCGs are fast and require minimal memory (typically 32 or 64 bits) to retain state. This makes them valuable for simulating multiple independent streams. LCGs should not be used for applications where high-quality randomness is required, it is not suitable for a Monte Carlo simulation because of the serial correlation (among other things). We will learn about Monte Carlo Simulation later. LCG must not be used for cryptographic applications as well because if a linear congruential generator is seeded with a character and then iterated once, the result is a simple classical cipher called an affine cipher; this cipher is easily broken by standard frequency analysis.

Sample Python Code:

```
def lcg(modulus, a, c, seed):
    while True:
        xn_new = (a * seed + c) % modulus
        seed = xn_new
        yield xn_new
```

Visual Representation of working of Linear Congruential Generator



Mersenne Twister –

The Mersenne Twister is a PRNG, which is a general purpose, and widely used generator. Its name derives from the fact that its period length is chosen to be a Mersenne prime. In mathematics, a Mersenne prime is a prime number that is one less than a power of two. That is, it is a prime number that can be written in the form $M_n = 2^n - 1$ for some integer n . The first four Mersenne primes are 3, 7, 31, and 127. The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime $2^{19937} - 1$. The standard implementation of that, MT19937, uses a 32-bit word

length. There is another implementation that uses a 64-bit word length, MT19937-64; it generates a different sequence.

The Mersenne Twister is the default PRNG for the following software systems: Microsoft Visual C++, Microsoft Excel, GAUSS, GLib, GNU Multiple Precision Arithmetic Library, GNU Octave, GNU Scientific Library, gretl, IDL, Julia, CMU Common Lisp, Embeddable Common Lisp, Steel Bank Common Lisp, Maple, MATLAB, Free Pascal, PHP, Python, R, Ruby, SageMath, Scilab, Stata. It is also available in Apache Commons, in standard C++ (since C++11), and in Mathematica. Add-on implementations are provided in many program libraries, including the Boost C++ Libraries, the CUDA Library, and the NAG Numerical Library.

Declaration in C++

```
template<
    class UIntType,
    size_t w, size_t n, size_t m, size_t r,
    UIntType a, size_t u, UIntType d, size_t s,
    UIntType b, size_t t,
    UIntType c, size_t l, UIntType f
> class mersenne_twister_engine;

using std::mt19937 =
std::mersenne_twister_engine<std::uint_fast32_t, 32, 624, 397, 31,
                             0x9908b0df, 11,
                             0xffffffff, 7,
                             0x9d2c5680, 15,
                             0xefc60000, 18, 1812433253>

using std::mt19937_64 =
std::mersenne_twister_engine<std::uint_fast64_t, 64, 312, 156, 31,
                             0xb5026f5aa96619e9, 29,
                             0x5555555555555555, 17,
                             0x71d67fffed60000, 37,
                             0xfff7eee000000000, 43, 6364136223846793005>
```

Examples

1 seed to mt19937_64

```
#include<random>
```

```
#include<cstdint>
#include<iostream>

int main()
{
    std::random_device device;    //non-deterministic random engine.
    std::mt19937_64 eng(device()); //using non-deterministic random
engine to seed std::mt19937_64
    std::uniform_int_distribution<std::size_t> dst(0,99); //distribution
to generate [0,99] integers
    for(std::size_t i(0);i!=20;++i)    //generate 20 random integers
        std::cout<<dst(eng)<<' ';    //using distribution to
operate mt19937_64
    std::cout<<'\n';
}
```

Possible output: 38 24 1 95 16 92 27 66 52 66 85 5 55 49 3 81 8 72 79 8

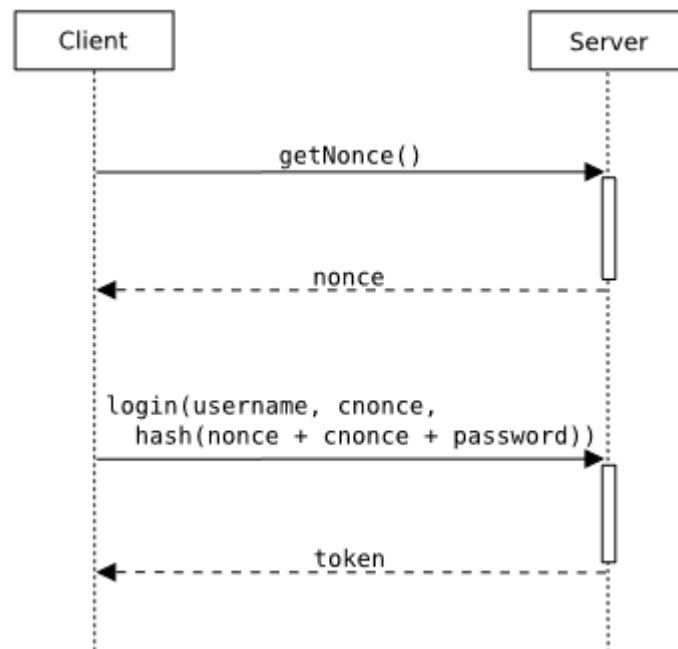
Cryptographically Secure Random Generators –

These Pseudo-Random Generators are designed to be suitable for cryptography. (CSRG) or (CSPRNG). Many aspects of cryptography require random numbers, for example:

- Key Generation
- Nonces
- One-Time Pads
- Salts in certain signature schemes.

Key generation is the process of generating keys in cryptography. A key is used to encrypt and decrypt whatever data is being encrypted/decrypted. A device or program used to generate keys is called a key generator or keygen.

Nonce is an arbitrary number that may only be used once. It is similar in spirit to a nonce word, hence the name. It is often a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. They can also be useful as initialization vectors and in cryptographic hash function.



Typical client-server communication during a nonce-based authentication process including both a server nonce and a client nonce.

One-Time Pad (OTP) is an encryption technique that cannot be cracked, but requires the use of a one-time pre-shared key the same size as, or longer than, the message being sent. In this technique, a plaintext is paired with a random secret key (referred to as a one-time pad). Then, each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition. If the key is truly random, is at least as long as the plaintext, is never reused in whole or in part, and is kept completely secret, then the resulting ciphertext will be impossible to decrypt or break. We see the use of this in our daily lives as One-Time-Password (OTP) to log in to various applications over the internet.

Salt is random data that is used as an additional input to a one-way function that "hashes" a password or passphrase. Salts are closely related to the concept of nonce. The primary function of salts is to defend against dictionary attacks or against its hashed equivalent.

Salts are used to safeguard passwords in storage. Historically a password was stored in plaintext on a system, but over time additional safeguards developed to protect a user's password against being read

from the system. A salt is one of those methods. Cryptographic salts are broadly used in many modern computer systems, from UNIX system credentials to Internet security.

Here is a simplified example usage of a salt value for storing passwords. This first table has two username and password combinations. The password is not stored.

Username	Password
user1	password123
user2	password123

The salt value is generated at random and can be any length, in this case the salt value is 8 bytes (64-bit) long. The hashed value is the hash of the salt value appended to the plaintext password. Both the salt value and hashed value are stored

Username	Salt Value	String to be Hashed	Hashed Value = SHA256(Password + Salt Value)
user1	E1F53135E5	password123+E1F53	72ae25495a7981c40622d49f9a52e4f1565c90f0
	59C253	135E559C253	48f59027bd9c8c8900d5c3d8
user2	84B03D034B	password123+84B03D	b4b6603abc670967e99c7e7f1389e40cd16e78ad38eb14
	409D4E	034B409D4E	68ec2aa1e62b8bed3a

As you can see from the hashed value, even though the plaintext passwords are the same, the hashed values are different. This is why the salt value prevents dictionary attacks, with the added benefit that users with the same password will not both be compromised if one password is cracked. While this does protect from an attacker using a leaked database of password hashes to attack other accounts with the same password, it does not protect a user from having their simple (commonly used or easily guessable) password guessed. The purpose of a hash and salt process in password security is not to prevent a password from being guessed, but to prevent a leaked password database from being used in further attacks.

The "quality" of the randomness required for these applications varies. On the other hand, generation of a master key requires a higher quality, such as more entropy. And in the case of one-time pads, the information-theoretic guarantee of perfect secrecy only holds if the key material comes from a true random source with high entropy. From an information-theoretic point of view, the amount of randomness, the entropy that can be generated, is equal to the entropy provided by the system. But sometimes, in practical situations, more random numbers are needed than there is entropy available. Also the processes to extract randomness from a running system are slow in actual practice. In such instances, a CSPRNG can sometimes be used. A CSPRNG can "stretch" the available entropy over more bits.

CSPRNG designs are divided into three classes:

1. Based on cryptographic primitives such as ciphers and cryptographic hashes,
2. Based upon mathematical problems thought to be hard, and
3. Special-purpose designs.

The last often introduce additional entropy when available and, strictly speaking, are not "pure" pseudorandom number generators, as their output is not completely determined by their initial state. This addition can prevent attacks even if the initial state is compromised.

Designs based on cryptographic primitives –

A secure **block cipher** can be converted into a CSPRNG by running it in counter mode. This is done by choosing a random key and encrypting a 0, then encrypting a 1, then encrypting a 2, etc. The counter can also be started at an arbitrary number other than zero.

Block cipher is a deterministic algorithm operating on fixed-length groups of bits, called a *block*, with an unvarying transformation that is specified by a symmetric key. Block ciphers operate as important elementary components in the design of many cryptographic protocols, and are widely used to implement encryption of bulk data. A **block cipher mode of operation** is an algorithm that uses a block cipher to provide an information service such as confidentiality or authenticity. A block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits called a block. A mode of operation describes how repeatedly to apply a cipher's single-block operation securely to transform amounts of data larger than a block. A cryptographically secure hash of a counter might also act as a good CSPRNG in some cases. Keeping the initial state a secret, most stream ciphers work by generating a pseudorandom stream of bits that are combined (almost always XORed) with the plaintext; running the cipher on a counter will return a

new pseudorandom stream, possibly with a longer period. The cipher can only be secure if the original stream is a good CSPRNG, although this is not necessarily the case.

Design based upon mathematical problems –

The **Blum Blum Shub** algorithm is a complicated design which uses the quadratic residuosity problem which can only be solved by integer factorization, it is used only when extreme security is needed.

Blum Blum Shub (B.B.S.) is PRNG proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub that is derived from Michael O. Rabin's oblivious transfer mapping.

Blum Blum Shub takes the form

$$x_{n+1} = x_n^2 \bmod M$$

Where $M = pq$ is the product of two large primes p and q . At each step of the algorithm, some output is derived from x_{n+1} ; the output is commonly either the bit parity of x_{n+1} or one or more of the least significant bits of x_{n+1} .

The seed x_0 should be an integer that is co-prime to M (i.e. p and q are not factors of x_0) and not 1 or 0.

Special Purpose designs –

A few PRNG that have been designed to be cryptographically secure:

The **Yarrow algorithm** is a family of cryptographic PRNG devised by John Kelsey, Bruce Schneier and Niels Ferguson.

The design of Yarrow consists of four major components including an entropy, reseed mechanism, generation mechanism and reseed control.

The **Fortuna algorithm**, the successor to Yarrow, which does not attempt to evaluate the entropic quality of its inputs.

Fortuna is a family of secure PRNGs; its design leaves some choices open to implementers. It is composed of the following pieces:

- The generator itself, which once seeded will produce an indefinite quantity of pseudo-random data.

- The entropy accumulator, which collects genuinely random data from various sources and uses it to reseed the generator when enough new randomness has arrived.
- The seed file, which stores enough state to enable the computer to start generating random numbers as soon as it has booted.

The function CryptGenRandom provided in Microsoft's Cryptographic Application Programming Interface -

In Win32 programs, Microsoft recommends its use anywhere random number generation is needed. The security of a cryptosystem's CSPRNG is significant because it is the origin for dynamic key material.

True Random Generators:

Hardware random number generators generally produce a limited number of random bits per second. In order to increase the data rate, they are often used to generate the 'seed' for a faster CSPRNG, which then generates the pseudorandom output sequence.

One early way of producing random numbers was by a variation of the same machines used to play keno or select lottery numbers. These mixed numbered ping-pong balls with blown air, perhaps combined with mechanical agitation, and used some method to withdraw balls from the mixing chamber.

On 29 April 1947, RAND Corporation began generating random digits with an "electronic roulette wheel", consisting of a random frequency pulse source of about 100,000 pulses per second gated once per second with a constant frequency pulse and fed into a five-bit binary counter.

Physical phenomena with random properties –

The outcome of quantum-mechanical events cannot in principle be predicted, hence these are the 'gold standard' for random number generation. Some quantum phenomena used for random number generation include:

Shot noise, a quantum mechanical noise source in electronic circuits. A simple example is a lamp shining on a photodiode. Arriving photons create noise in the circuit. Collecting the noise for use poses some problems, but this is an especially simple random noise source.

A nuclear decay radiation source (as, for instance, from some kinds of commercial smoke detectors), detected by a Geiger counter attached to a PC.

Photons travelling through a semi-transparent mirror. The mutually exclusive events (reflection/transmission) are detected and associated to '0' or '1' bit values respectively. Example - Lasers: The use of lasers allows for true random number generators that overcome the obstacle of slow production. In laser-based generators, entropy can be obtained by several different means. Having two photons race to a destination is one RANDOM NUMBER GENERATION 10 method that is currently implemented. Another high-speed technique is measuring the varying intensity of a chaotic laser. Prototype systems in this second category have been created that can produce random bits at rates of over ten Gigabits per second. The prototypes exhibited a natural bias toward one value over the other, so a post processor needed to be applied to create truly random sequences. A commonly used tactic is to take several bits at a time and XOR them together to remove the unwanted bias. The stream of bits that emerged from this process was able to pass the most stringent randomness tests that are used for generators dealing with cryptography. Laser generators are capable of increased speeds, but they are complex to install and prohibitively expensive.

Thermal phenomena are easier to detect. They are somewhat vulnerable to attack by lowering the temperature of the system. Some of the thermal phenomena used include:

- Thermal noise from a resistor, amplified to provide a random voltage source.^[9]
- Avalanche noise generated from an avalanche diode.
- Atmospheric noise, detected by a radio receiver attached to a PC (though much of it, such as lightning noise, is not properly thermal noise, but most likely a chaotic phenomenon).

In the absence of quantum effects or thermal noise, other phenomena that tend to be random, although in ways not easily characterized by laws of physics, can be used. When several such sources are combined carefully (as in, for example, the Yarrow algorithm or Fortuna CSPRNGs, enough entropy can be collected for the creation of cryptographic keys and nonces, though generally at restricted rates. The advantage is that this approach needs, in principle, no special hardware. The disadvantage is that a sufficiently knowledgeable attacker can secretly modify the software or its inputs, thus reducing the randomness of the output, perhaps substantially. The primary source of randomness typically used in such approaches is the precise timing of the interrupts caused by mechanical input/output devices, such as keyboards and disk drives, various system information counters, etc.

Software engineers without true random number generators often try to develop them by measuring physical events available to the software. An example is measuring the time between user keystrokes, and then taking the least significant bit (or two or three) of the count as a random digit. A similar approach measures task-scheduling, network hits, disk-head seek times and other internal events.

Random number servers –

There are a few websites that claim to provide numbers generated from truly a random source:

Random.org:

Random.org is a website that produces "true" random numbers based on atmospheric noise. In addition to generating random numbers in a specified range and subject to a specified probability distribution, it has free tools to simulate events such as flipping coins, shuffling cards, and rolling dice. It also offers paid services to generate longer sequences of random numbers and act as a third-party arbiter for raffles, sweepstakes, and promotions. The website was created in 1998 by Mads Haahr, a doctor and computer science professor at Trinity College in Dublin, Ireland. Random numbers are generated based on atmospheric noise captured by several radios tuned between stations.

HotBits:

The other popular free Internet-based random number generator is referred to as HotBits. This site generates its random number sequences based off radioactive decay.

Other examples –

<http://www.randomnumbers.info/>

<http://appincredible.com/online/random-number-generator/>

Tests for Randomness –

Many sequences that appear random may actually be easy to predict. For this reason, it is important to thoroughly test any generator that claims to produce random results. Whenever professional forecasters make data-driven predictions, they apply formulas to determine the probability that the results were not random. These formulas can likewise be used to verify that a result was random, and the only thing that must change is the passing criteria. No test can conclusively prove randomness; the best that can be accomplished is that with enough testing, users of the generators can be confident that the sequence is random enough. There are scholars who believe that the source of a random sequence should dictate what tests to run. For example, true random generators tend to exhibit bias toward values, and this trait worsens as the hardware wears down. As a result, these scholars claim that if a random sequence comes from a true random generator, extra tests should be performed that check for bias. Other scholars believe that random is random regardless of where it came from, and that it is appropriate to test all random sequences the same.

Statistical Tests:

With this approach, each test examines a different quality that a random number generator should have. For example, a random generator would go through a chi-squared test to ensure a uniform distribution, and then a reverse arrangements test to see if the sequences contained any trends. Confidence in the generator's randomness is only gained after it passes an entire suite of tests which comes at it from different directions. These tests should be run on more than just a single sequence to ensure that the test results are accurate. Making the act of testing more difficult is the fact that failing a test does not indicate that a generator is not random. When outputs are truly random, then there will be some isolated sequences produced that appear non-random.

Chi-squared test –

The chi-squared test is used to ensure that available numbers are uniformly utilized in a sequence. This test is easy to understand and set up, so it is commonly used. The formula for the test is:

$$\chi^2 = \sum (O_i - E_i)^2 / E_i,$$

Where the summation is of all the available categories. O represents the actual number of entries in the category, and E is the expected number of entries. For example, if the random numbers were scattered one through six, then there would be six categories, and the number of times each value appeared in the sequence would become the values of O. When the resulting value is above the chosen significance level, then it can be said that the values in the sequence are uniformly distributed. Because this test is simple, it can be run many times on different sequences with relative ease to increase the chance of its accuracy.

The runs test –

An important trait for a random sequence is that it does not contain patterns. The test of runs above and below the median can be used to verify this property. In this test, the number of runs, or streaks of numbers, above or below the median value are counted. If the random sequence has an upward or downward trend, or some kind of cyclical pattern, the test of runs will pick up on it. The total number of runs and the number of values above and below the median are recorded from the sequence. Then, these values are used to compute a z-score to determine if numbers are appearing in a random order. The formula for the score is:

$$z = ((u \pm 0.5) - \text{MEAN}u) / \sigma u,$$

Where σ denotes the standard deviation of the sequence. Just like the chi-squared test, a test for runs is easy to implement, and can be run frequently.

Next bit test –

When testing pseudo random generators for cryptography applications, the next bit test is a staple. In its theoretical form, the next bit test declares that a generator is not random if given every number in the generated sequence up to that point there is an algorithm that can predict the next bit produced with significantly greater than 50% accuracy. This definition makes the next bit test virtually impossible to implement, because it would require trying every conceivable algorithm to predict the next bit. Instead, it can be used after a pattern is discovered to cement the fact that a generator is insecure.

The universal next bit test developed in 1996 was the first to allow the next bit test to be administered, but it was shown that this test would pass non-random generators. However, this test required a large amount of resources to run, limiting its usefulness. The next bit test remains relevant in cryptography because it has been proven that if a generator can pass the theoretical next bit test, then it will pass every other statistical test for randomness.

Matrix-based tests:

These tests will take the values of the generator, and sequentially add them into a hypercube, which is a cube with k dimensions. From here the tests vary, although many of them conclude with a chi-squared calculation on their output. In a nearest pair test, Euclidean distances are computed and the nearest pairs are used to look for time-lagged patterns. It has been proposed that a large number of pseudo random generators based on circuits will fail the nearest pairs test, and it is one of the more rigorous tests available. Multidimensional tests are often specifically tailored to look at random number generators, unlike the linear tests which are multipurpose.

Exploratory Analysis:

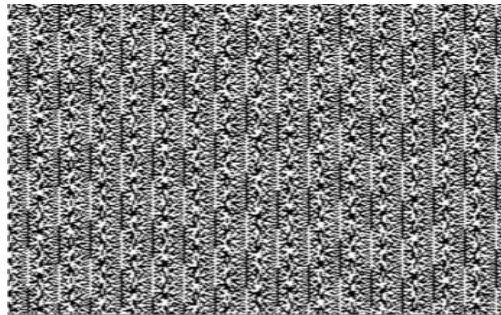
Visualizing random sequences is a good preliminary method to understand the data. There are several ways that sequences can be plotted in an attempt to bring out any oddities inside the random generator. Figure below is a bitmap obtained from the `rand()` function in PHP on Windows. By turning the sequence into a plot, it becomes more apparent that the given generator has some patterns in its sequences. In an ideal generator, the bitmap would seem like complete static, but in this example sections of black and white are clearly grouped. From this point, more specific tests can be decided on to determine just how severe the patterns actually are.

There is overlap between the problems that graphs can bring out and the problems that statistical techniques look for. A run sequence plot is designed to look for trends in a sequence, much like the test for runs. To create this plot, actual sequence values on the y axis are compared against the

values' indexes in the sequence. If there are trends in the sequence, this plot will make them easier to see.

A histogram plot has a comparable purpose to the chi-squared test. It is composed of a bar graph, with the different categories on the x-axis plotted against the number of times that value appears on the y-axis. Any kind of non-uniformity will be brought out quickly this way, signaling that more tests concerning uniform distribution should be run.

Another unique exploratory tool is the autocorrelation plot. An autocorrelation plot examines the correlation of a value to the values that came before it at various intervals, called lags. If the plot displays no correlations between values at any lag, then the numbers are most likely independent of each other, which is a good indication of randomness. Using these exploratory plots allows analysts to get a feeling for the faults of a generator and better decide on which tests to run on the number sequences.



Obtained from Random.org

NIST:

The NIST suite reigns as the industry standard. The NIST suite was designed to test bit sequences, with the idea that passing all NIST tests means that a generator is fit for cryptographic purposes. Even new true random number generators have their preliminary results run through the NIST battery to demonstrate their potential. The NIST suite contains fifteen well-documented statistical tests. Cryptography has the most stringent requirements for randomness out of all the categories, hence, a generator that passes the NIST suite is also random enough for all other applications. However when a generator fails the NIST suite, it could still be random enough to serve in areas such as gaming and simulation, since the consequences of using less than perfectly random information is small. NIST does not look at factors such as rate of production, so passing the NIST suite should not be the only factor when determining a generator's quality.

Diehard:

This suite was invented by George Marsaglia in 1995. It was made to be an update for the original random number test suite, Knuth. Knuth is named after Donald Knuth and was published in the 1969 book *The Art of Computer Programming, Volume 2*. All of the tests are available free online.

BSI evaluation criteria:

The German Federal Office for Information Security has established four criteria for quality of deterministic random number generators or Pseudo Random Generators:

K1 – There should be a high probability that generated sequences of random numbers are different from each other.

K2 – A sequence of numbers which is indistinguishable from 'true random' numbers according to specified statistical tests. These requirements are a test of how well a bit sequence has zeros and ones equally often after a sequence of n zeros (or ones), the next bit a one (or zero) with probability one-half; and any selected subsequence contains no information about the next element(s) in the sequence.

K3 – It should be impossible for any attacker (for all practical purposes) to calculate, or otherwise guess, from any given subsequence, any previous or future values in the sequence, nor any inner state of the generator.

K4 – It should be impossible, for all practical purposes, for an attacker to calculate, or guess from an inner state of the generator, any previous numbers in the sequence or any previous inner generator states.

For cryptographic applications, only generators meeting the K3 or K4 standard are acceptable.

Applications of Randomness

We have seen Random Generation, and different tests of Random Generation. Let us look at the applications of Pseudo-Random Numbers mainly in –

1. Simulation
2. Electronic Games
3. Generative Art
 - a. Visual Art
 - b. Music
 - c. Literature

Simulation:

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. They are used in applications of physics and mathematics when it is difficult or impossible to use other approaches. In physics-related problems, Monte Carlo methods are useful for simulating systems with many coupled degrees of freedom, such as fluids, disordered materials, strongly coupled solids, and cellular structures. Other examples include modeling phenomena with significant uncertainty in inputs such as the calculation of risk in business and, in math, evaluation of multidimensional definite integrals with complicated boundary conditions.

In its pure mathematical form, the Monte Carlo method consists of finding the definite integral of a function by choosing a large number of independent-variable samples at random from within an interval or region, averaging the resulting dependent-variable values, and then dividing by the span of the interval or the size of the region over which the random samples were chosen. This differs from the classical method of approximating a definite integral, in which independent-variable samples are selected at equally spaced points within an interval or region.

The Monte Carlo method is most famous for its use during the Second World War in the design of the atomic bomb. It has also been used in diverse applications, such as the analysis of traffic flow on superhighways, the development of models for the evolution of stars, and attempts to predict fluctuations in the stock market. The scheme also finds applications in integrated circuit (IC) design, quantum mechanics, and communications engineering.

The main idea behind this method is that the results are computed based on repeated random sampling and statistical analysis. The Monte Carlo simulation is in fact random experimentations, in

the case that, the results of these experiments are not well known. Monte Carlo simulation methods do not always require truly random numbers to be useful. Many of the most useful techniques use deterministic, pseudorandom sequences, making it easy to test and re-run simulations. The only quality usually necessary to make good simulations is for the pseudo-random sequence to appear "random enough" in a certain sense.

What this means depends on the application, but typically they should pass a series of statistical tests. Testing that the numbers are uniformly distributed or follow another desired distribution when a large enough number of elements of the sequence are considered is one of the simplest, and most common ones. Weak correlations between successive samples is also often desirable/necessary.

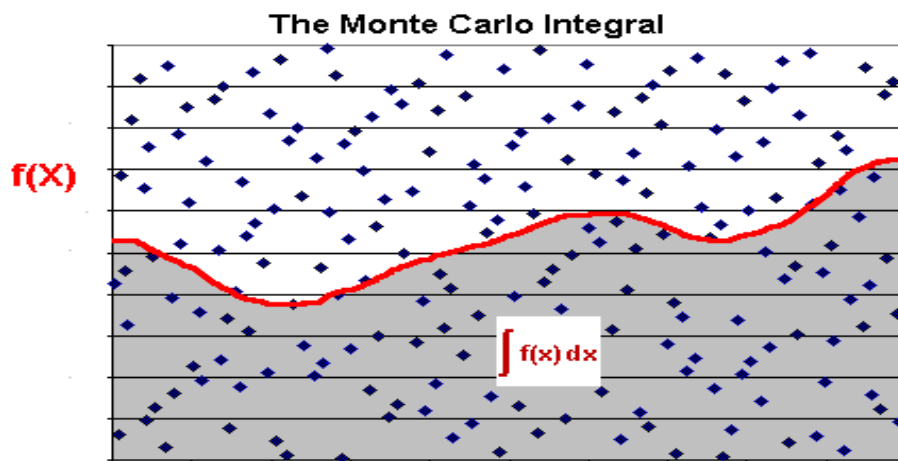
Sawilowsky lists the characteristics of a high quality Monte Carlo simulation:

- PRNG has certain characteristics (*e.g.*, a long "period" before the sequence repeats)
- PRNG produces values that pass tests for randomness
- There are enough samples to ensure accurate results
- The proper sampling technique is used
- The algorithm used is valid for what is being modeled
- It simulates the phenomenon in question.

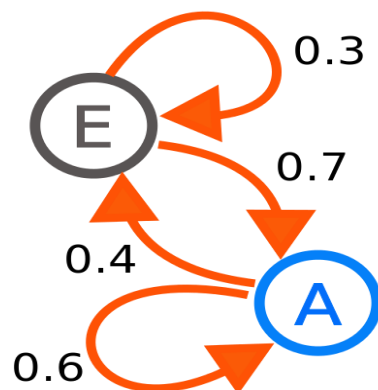
Pseudo-random number sampling algorithms are used to transform uniformly distributed pseudo-random numbers into numbers that are distributed according to a given probability distribution.

Low-discrepancy sequences are often used instead of random sampling from a space as they ensure even coverage and normally have a faster order of convergence than Monte Carlo simulations using random or pseudorandom sequences. Methods based on their use are called quasi-Monte Carlo methods.

In probability theory and related fields, a Markov process, named after the Russian mathematician Andrey Markov, is a stochastic process that satisfies the Markov property sometimes characterized as "memorylessness". Roughly speaking, a process satisfies the Markov property if one can make predictions for the future of the process based solely on its present state just as well as one could knowing the process's full history, hence independently from such history; i.e., conditional on the present state of the system, its future and past states are independent.



Monte Carlo Integral Example



A diagram representing a two-state Markov process, with the states labelled E and A. Each number represents the probability of the Markov process changing from one state to another state, with the direction indicated by the arrow.

For our example, let's start with a circle of radius 1 inscribed within a square with sides of length 2.

We can then use Monte Carlo simulation to randomly sample points from within the square. More specifically, we can randomly sample points using a uniform distribution where the minimum is -1 and the maximum is +1:

$$-1 \leq x \leq 1$$

$$-1 \leq y \leq 1$$

If $x^2 + y^2 \leq 1$, then the point falls inside the circle.

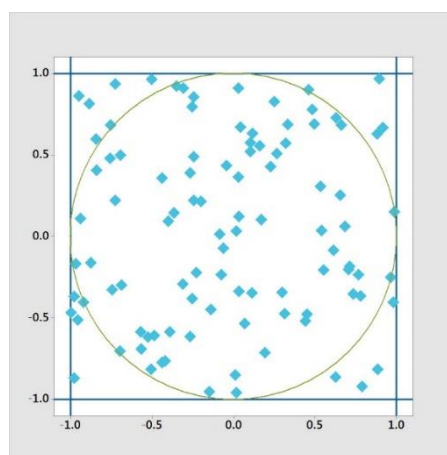
Since:

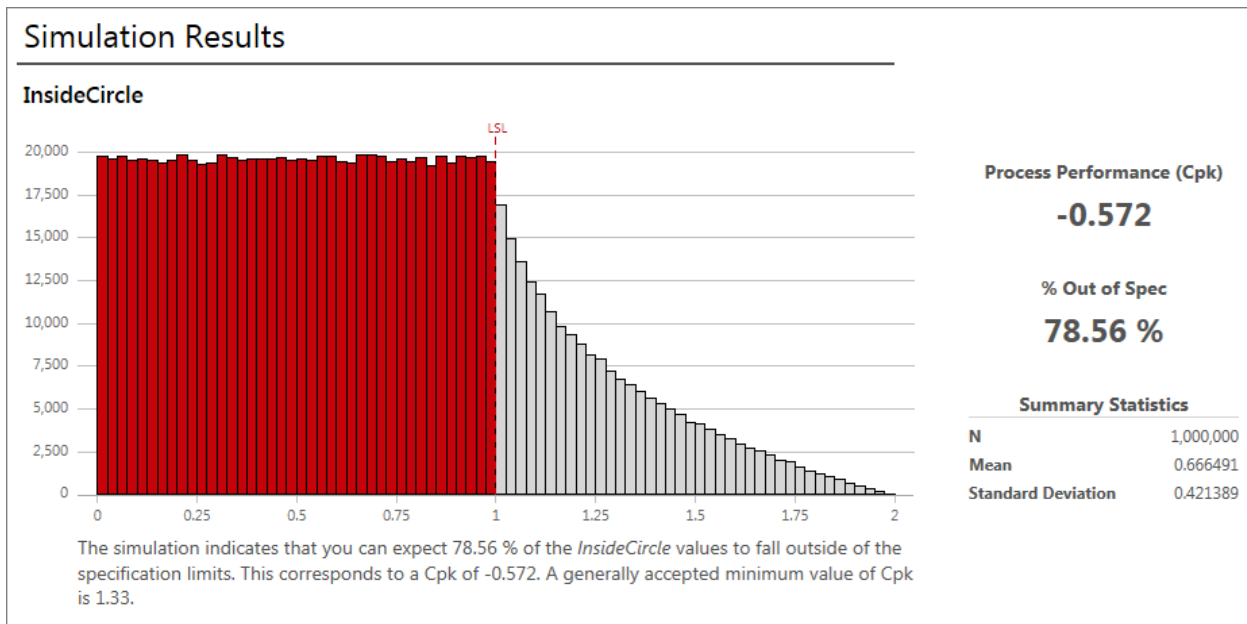
$$\text{Area of the circle} = \pi * r^2 = \pi * 1^2 = \pi$$

$$\text{Area of the square} = \text{side length}^2 = 2^2 = 4$$

Then the ratio of the two areas - we'll call it r - can be represented as:

$$r = \frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}$$





As shown in the output, the results of the simulation are that 78.56% of the values will fall outside of the circle. (Since Monte Carlo simulation uses random sampling, this number will not be exactly the same every time you run a simulation).

Electronic Games and Graphics:

Procedural generation is a method of creating data algorithmically as opposed to manually. In computer graphics, it is also called random generation and is commonly used to create textures and 3D models. In video games, it is used to automatically create large amounts of content in a game. Advantages of procedural generation include smaller file sizes, larger amounts of content, and randomness for less predictable gameplay.

The term procedural refers to the process that computes a particular function. Fractals are geometric patterns which can often be generated procedurally. Commonplace procedural content includes textures and meshes. Games like Minecraft use a seed to generate levels with different terrains.

Random numbers appear everywhere in games, from the online slots games to MMORPG games like World of Warcraft. In video games, random "drops" of items often significantly affect the course of the game. Racing games may use random number generators to determine how fast or slow the other cars move, card games may use random number generators to shuffle the deck. Some popular games, such as Pokémon, are almost entirely driven by encounters which are attached to

random number generators. In fact, you would be hard-pressed to find a game today that does not include random number generators in some form.

Games that feature co-operative play may use procedural generation to construct unique experiences for the players as they complete the game. The Left 4 Dead series, based on a zombie apocalypse setting, uses an AI Director, an artificial intelligence that monitors the behaviour of the players and creates a dynamic experience to keep players alert. It can sense when players have become comfortable and spawn new zombies and hordes to attack the players in mid-level and alter which routes are available for players to take. Similar concepts are used for games like 'Warhammer: End Times – Vermintide'.

Among other reasons, random number generators in games can be used for:

- Unit stats: Giving random stats to enemies instead of cloning the same level of difficulty on every single enemy encountered. Dynamic values for Attack, Defence, Speed, Damage, etc. can create an interesting gameplay pattern. Random ranges of values will generate stronger and weaker versions of enemies, or even friends.
- Map generation: Drawing a map by hand is optional, but if an unpredictable and complex game is required, a random map will be a better choice.
- Sprite generation: Randomly placing pieces together to create sprites rather than hand-drawing every sprite in a game is a good option. If you have ever played around with avatar options, like those of Skyrim, you can see the potential to create thousands of different sprites or character designs with random generators. The same thing can be done for buildings, cars, trees, etc.
- Artificial intelligence: Allows the game's AI to make a random rather than a fixed decision, making the gameplay less predictable to the player.



A great example of randomness at work. Spelunky: randomly generated levels, randomly selected tiles and randomly populated enemies



Minecraft generates new worlds each time, implementing a random generating algorithm. The algorithm creates a world vast enough to never let the user feel bored.

Example: In slots games, card games and other games of chance, the random number generator is essential. An improperly coded random number generator could potentially pay out more often than it should (or not at all). A great example are Gaming Club slot machine games which use pseudo-random numbers rather than true random numbers to control the game not allowing players to



Slots and card games like the ever-popular Hearthstone depend on pseudo-random number generators predict when a certain number will be hit.

Unit stats –

Randomness can be used to give enemies some personality by giving them randomized stats. Using a dynamic range of values, creating stronger and weaker enemies makes the gameplay more interesting. Unless you are generating a large number of units, modern processors can handle generating these values dynamically.

Ex- Pokémon uses Linear Congruential random number generator in the GBA and NDS games as follows: The V and VI Gen use the following 64 bit LCRNG.(Linear Congruential Random Number Generator). Seed is a number between 0 and 0xFFFFFFFF. If the generator has been called previously, seed is the result value from the previous call to the generator.



Encountering a Pokémon is randomly picked from a pool for the particular area

Alakazam	3	Lv.40	
DEX NO.	065	Alakazam	
HP	96 / 96		
ATTACK	52		
DEFENSE	51		
SP. ATK	121		
SP. DEF	81		MOVES LEARNED
SPEED	107		
NATURE	Serious		
ABILITY	Inner Focus		
ITEM	Alakazite		
			Thunder Wave
			Hidden Power
			Psycho Cut
			Recover

Each Pokémon having its unique stats and Nature within a range for each, randomly generated

Sprite generation –

Instead of hand drawing sprites, creating a generator system that randomly puts pieces together, and create random sprites dynamically. If you have played around with an avatar generator such as eightbit.me or the Mii generator on Wii or the XBox Live Arcade avatar generator, but with a random selector in charge of picking the hair, eyes, etc. You can do this to randomly generate other things, such as buildings, procedurally, as well.

Terrain Generators are projects that combine art and programming to generate a nature-like environment known as terrain. Terrain can contain slopes, dirt, trees, cliffs, water, and other things present in the environment. They are often created using stamping and pen techniques and tagged under games, simulations, and art. They can range from simple 2D textured blocks to complex 3D terrains.

Colour –

Randomizing colours is a good option to create art. It can be used for applications that need a dynamic appeal every time, best example would be in games. If you are clever in how you pick your random colours, you can come up with colour schemes that work nicely. You can either pre-define a palette of colours or randomly select one, or you can randomly select R, G, B or H, S, V numbers and create a colour at runtime. You can experiment with different mathematical tweaks to shape and constrain the randomness.

Example java code –

```
public Colour generateRandomColour(Colour mix)
{
    Random random = new Random();
    int red = random.nextInt(256);
    int green = random.nextInt(256);
    int blue = random.nextInt(256);

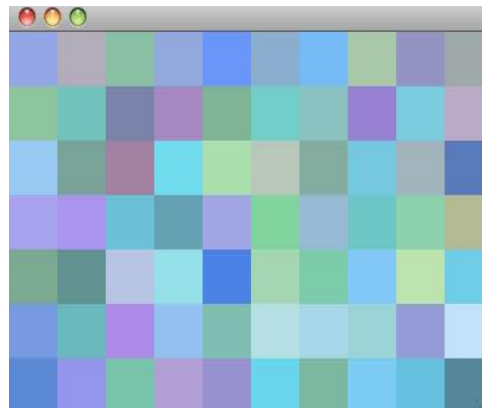
    // mix the colour
    if (mix != null) {
        red = (red + mix.getRed()) / 2;
        green = (green + mix.getGreen()) / 2;
        blue = (blue + mix.getBlue()) / 2;
    }

    Colour colour = new Colour(red, green, blue);
}
```



```
    return colour;  
}
```

Output:



Map Generation –

Good random map generator is preferred because it saves time from hard designing the map from scratch and even if the algorithm isn't near perfect it is still creative, economic and easy to refine to suit your needs. Procedurally generated content in general is a good use for random functions. You can use a random number function to create a deterministic sequence of generated values that is always the same. This is used to good effect in Atari 2600 games, Pitfall. A pseudo-random function, using a fixed seed, is used to generate each screen in the game. This achieves a very high information density, since the data that was needed to represent each screen could not be stored on a 4kb ROM, but a generator function that creates that data easily could. This technique is not used very much in modern game development since storage isn't much of an issue any more, but it is still a very interesting technique and one which merits study.


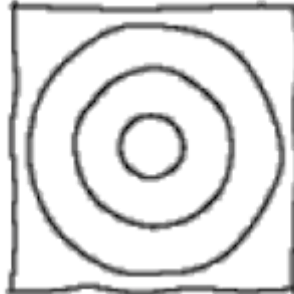


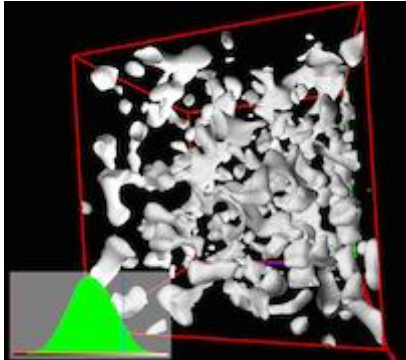

Minecraft is specifically using Perlin noise calculations, like the kind you'd use to create a rough-looking texture for a 3D model. It starts out on a very broad level, painting a basic topographical map, and adds "noise" through finer terrain details like lakes, shrubbery and animals. Importantly, it has just enough freedom to create unexpected figures, like the elaborate rock structure you see above as in the real world.

Perlin noise works by generating a noise map (a flat surface covered in dots grouped together in semi-random patterns). It makes it look like a topographical map. The computer creates a noise map, then smoothening it from chunky parts to finer details to make a map in Minecraft. It

creates a noise map creating sky and ground (ground below sky always). Then adds another noise map to use it to distinct between hills, valleys, craggy areas. Finishing with trees, lakes and clouds. These generation are performed with specific rules to give the players an infinite number of worlds to play in that are not completely random defying logic like sheep underwater.



Example of Perlin Noise in Minecraft

Noise Dimension	Raw Noise (Grayscale)	Use Case
1		 <p>Using noise as an offset to create handwritten lines.</p>
2		 <p>By applying a simple gradient, a procedural fire texture can be created.</p>
3		 <p>Perhaps the quintessential use of Perlin noise today, terrain can be created with caves and caverns using a modified Perlin Noise implementation.</p>

Some pictures of Noise in different dimensions and some of their uses at runtime

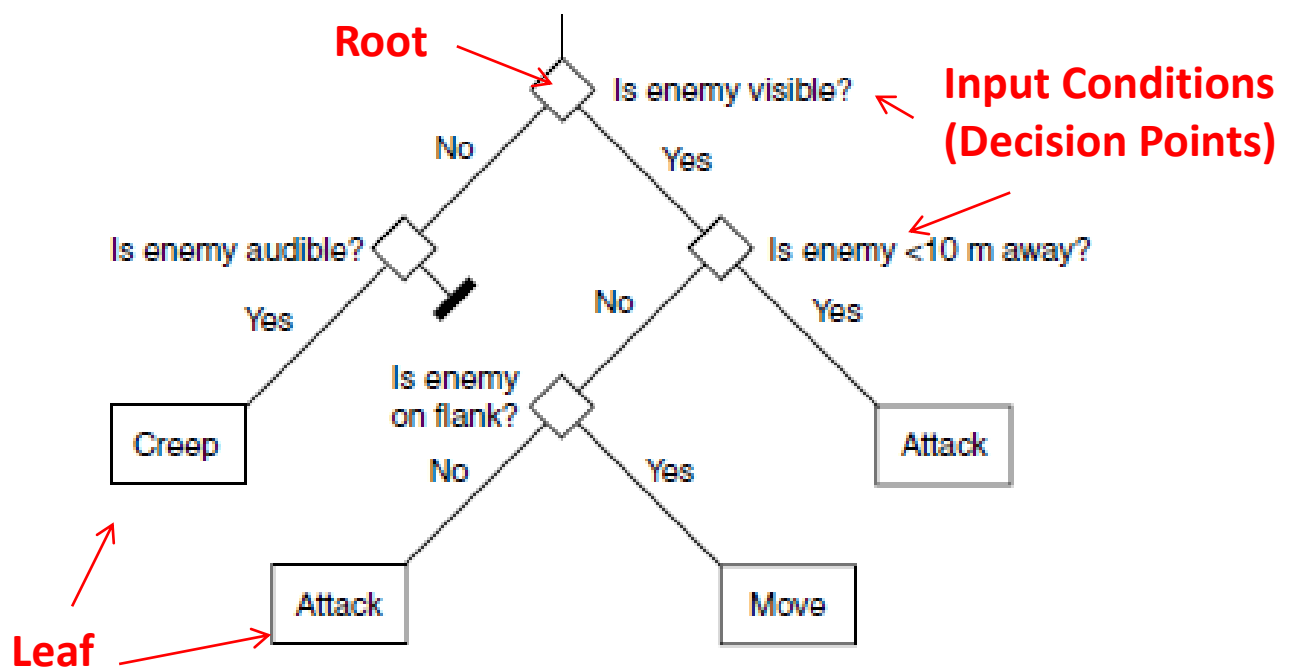
Artificial Intelligence –

In the case of Artificial Intelligence, it requires a lot of planning while using randomness in its decision making. Whenever your AI needs to make a decision, you potentially can use randomness to make that decision less predictable. Weighted probability is important here, as completely random AI behaviour is erratic and seems crazy, while an AI that occasionally does something unexpected will seem tricky or deceptive or clever.

Example of applying randomness to an AI Soldier in a game: In Artificial Intelligence there is a concept of ‘Decision Trees’. A decision tree is a tree made of a ‘Root’, from the root ‘Options’ and finally at the end or leaf nodes are the ‘Decisions’.

Tree has starting decision (root)

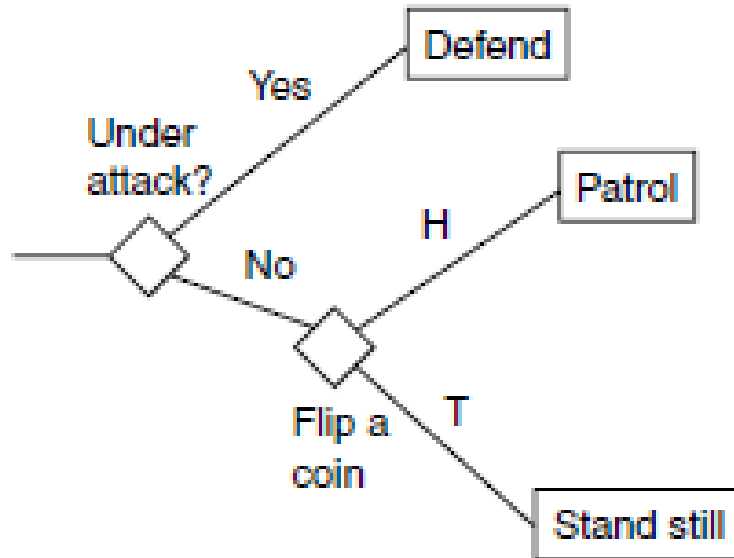
For each decision (starting from root), one of a set of ongoing options is chosen. Choice is made based on conditions derived from the character’s knowledge/values, and continue along the tree until the decision process has no more decisions to make. Each leaf is an action, to be executed



Sample decision tree of a soldier character

To introduce random choices in a Decision Tree, decision making process needs to be stable. The rules can be that, if there is no relevant changes in world state, there should be no change

in decision. Consecutive frames should stay with the chosen random decision until some world state changes. Implementation should be allowing the random decision to keep track of what it did last time, so that it knows the previous choice to take when the same decision is encountered.



In the first decision (if not under attack), it chooses randomly to patrol or stand still. Subsequently, continues on with the previously chosen action. If the parent decision takes the ‘under attack’ choice (a different branch), it gets rid of the stored choice. Finally repeat. Using a time-out scheme (a stop timer) to reset the previous action, and initiate a new random choice stops the AI from doing the same thing forever.

Generative Art:

Generative art refers to art that in whole or in part has been created with the use of an autonomous system. An autonomous system in this context is generally one that is non-human and can independently determine features of an artwork that would otherwise require decisions made directly by the artist. In some cases the human creator may claim that the generative system represents their own artistic idea, and in others that the system takes on the role of the creator.

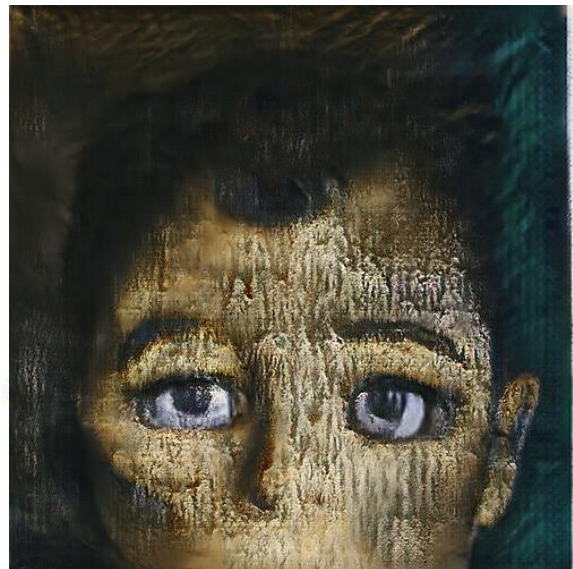
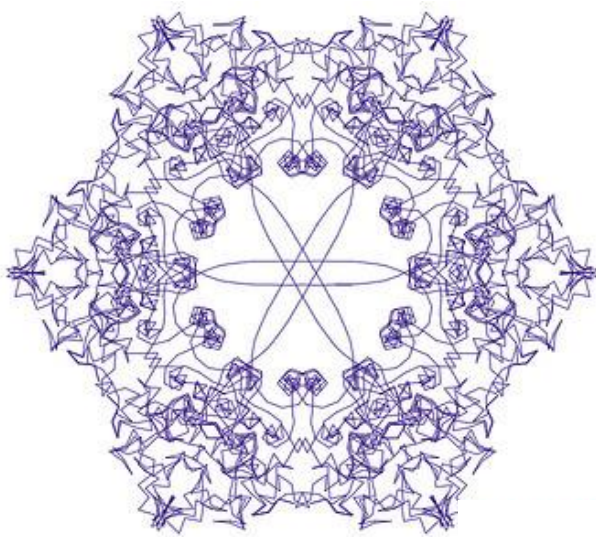
Art can include images, music, videos, and even literature and poetry.

Visual Art –

Every Human has ‘Pareidolia’. The ability or the response of finding faces, objects or patterns, at places where none exist. The best example is when we see clouds shaped as different

objects and ask our friend's if that looks like a dog or a cat. Using this psychological phenomenon artists create random patterns on a computer screen which makes us believe there is an object in that image, where in reality, there isn't. Procedural art or generative art isn't just for exploiting this phenomenon of us humans but also to create some fantastic pieces of art that could have never been achieved by hand. This is because the system provides an infinite canvas, with an infinite number of options, theoretically, for an artist to create his art using randomness or chaos either completely or partially.

Examples of Images created using procedural generation:



Music –

Music is nothing but a sequence of intertwined sounds made by vibrating particles that tricks the brain into releasing dopamine. Using a computer system to be creative using sound waves has been achieved over the last two to three decades.

‘Aleatoric music’ is music in which some element of the composition is left to chance, and/or some primary element of a composed work's realization is left to the determination of its performer(s). The term is most often associated with procedures in which the chance element involves a relatively limited number of possibilities.

‘Generative music’ is a term popularized by Brian Eno to describe music that is ever-different and changing, and that is created by a system.

Example:

June 1, 2016, Google’s art machine wrote a 90 second piano melody created through a trained neural network, provided with just four notes up front. The drums and orchestration weren't generated by the algorithm, but added for emphasis after the fact.

In January of 2017, Brian Eno released his latest ambient album, “Reflection,” as both a standalone production and as an evolving soundscape via an app developed for the iPhone. With the app, fans can immerse themselves in an infinite transformation of the album, finding something new and exciting every time as music shifts from entertainment to companionship. The app will make changes each time the song is played making it feel like you listened to a new song though there was only a few changes made to the original using a randomizing system.

There are already websites that hint at a world where music is no longer concrete. ‘Infinite Jukebox’ is a great example of that. It is an online app that lets a user input one of their favorite songs and let it play infinitely, by looping and connecting different sections of the song then decomposing the track into individual beats, then choosing randomly between identical beats in the song, it can make a new song infinite number of times.

Literature –

Literature is a complex fields for a computer to learn. Many AI designers or Machine learning algorithm developers have a hard time teaching a computer literature. The reason being that literature, many a times contains a meaning that is hidden. It is the hidden meaning that makes literature interesting. The second meaning behind the using of particular phrase or word. Poetry magic

is created not just by words, but language, and placement of words. Designing algorithms to create literature requires similar yet different sets of skills for a computer.

Examples include writers such as Tristan Tzara, Brion Gysin, and William Burroughs used the cut-up technique to introduce randomization to literature as a generative system. Jackson Mac Low produced computer-assisted poetry and used algorithms to generate texts; Philip M. Parker has written software to automatically generate entire books. Jason Nelson used generative methods with Speech-to-Text software to create a series of digital poems from movies, television and other audio sources.

What is actually ‘Random’?

We believe that dice and coins are random because of ignorance. If we could know every initial condition, the exact forces and properties that play for a particular flip or roll, we could theoretically calculate the result before it even happened. Researchers at Stanford have built coin flipping robots that can precisely control a flip to get the result they want 100 percent of the time.

Is there anything you could not predict even if you knew everything? A process determined by nothing? We are not sure if there are any patterns at the randomness we are looking at or looking for the wrong pattern. A random process that we know can, will occasionally produce patterns. YouTube URLs are random. A unique one is made for every uploaded video, but sometimes, YouTube generates a URL that contains, by chance, a word. Example this statistics lecture consists of the word “JAVA -<https://www.youtube.com/watch?v=JaVA-nXKVRI>”

You can search for a video with a word inside its URL using this syntax in Google:

‘allinurl:[your_word_here] site:youtube.com/watch’

It is easier to be certain that something is not random than that it is. We generally call predictable things ‘random’. Example if you met new friends at a party they were not random in a mathematical sense. This non-statistical use of the word random irritates some people but the word’s understood and used meaning is not too far from the original meaning of the word. In the 1300s, random meant running or at great speed. Later, it would be used to describe things that have no definite purpose. It was until the 1800s that random took on a particular mathematical definition. Then in the 1970s, MIT’s student paper popularized the use of the word random to mean strange. Just because something is strange does not mean it has no discoverable cause. Many theories revolve around the amount up information and new people we are confronted with at an increasing rate; now more than ever before being the reason we call predictable things random.

Even though a die and a coin are extremely sensitive to their initial conditions, and, over the course of normal use, are quite unpredictable, they do over time exhibit certain biases. Biases that make them a bit more predictable and a little less random. Precision dice are only quality controlled within a few micrometers.

Research shows what happens when coins are spun and flipped. For instance, the US nickel is just the right diameter and thickness to wind up landing not heads up or tails up when flipped, but on its side, about once every six thousand times it's flipped. Researchers at Stanford have found if you flip a coin, the side facing up, for statistical and physical reasons, before the flip begins, actually has a fifty one percent chance to be the side facing up again.

To achieve more fairness do not let the coin touch the ground, because it is found that when a coin spins larger biases come into play. The shape of its edge, its center of gravity. The heavier side tends to go down quite often. In the case of some coins as often as eighty percent of the time. It has been found that a one Euro coin will spin, and land heads up, more often than not. Theoretically if we knew everything about the initial conditions of a coin flip or a die roll, we could calculate their outcome beforehand. This is extremely difficult to calculate. High precision would be required because the smallest difference between two initial conditions can be magnified over time leading to chaotic, unpredictable results.

'Random.org', a website stated earlier, which gives a true random number, uses atmospheric noise, but technically still is a deterministic system. If we could just find out those initial conditions we could, theoretically, predict their outcomes.

Conclusion

We have seen 'what is randomness', 'beginning of randomness' 'entropy', 'use of randomness in our computer systems for various applications', 'tests of the random generators' and also 'if our understanding of randomness is correct'. To answer the question of what truly is 'Random', we need to first answer what is 'Not Random'.

There are those events that we know will certainly happen and certainly not. There are events that we can assume might happen, and might not, but for the real random events we should not even be able to make a slight assumption. Example we can ascertain that water will freeze at 0 degree Celsius; we can also be certain that the entire universe will not disappear the next day.

We know this because everything in the universe is made of 12 fundamental particles, and they interact in four predictable ways. A fundamental particle is a particle whose substructure is unknown; thus, it is unknown whether it is composed of other particles. An elementary particle can be a fermion or a boson. Fermions are the building blocks of matter and have mass, while bosons behave

The second law of thermodynamics states that entropy in the universe increases with time. It is because as the universe expands, the atoms have more space and hence can be in more states at a given time. Given that the entropy is increasing, this means the information in the universe is increasing as well. Quantum mechanics describes how the 12 fundamental particles behave. Though this is an interesting and successful field, this is only a probabilistic theory. This means that at no given instance, one can predict the position of an electron at a given time in the future, but probabilities can be calculated as to where the electron could be found. Let us say an electron is located at a particular point, you have gained information, you could not have predicted this with certainty beforehand.

We just described the properties of quantum sized objects as probabilities, or chances. We know the various physical phenomenon associated with True random generators previously that can only be determined by looking at the atom at that given instance of time. It is determined by a randomness determined by the universe itself.

Thus a new information is being generated every time a quantum event occurs. Hence it could mean that these quantum events are driving up the information of the universe, creating new information all of the time, increasing the disorder in the universe. This means that all of us do not function randomly. Randomness is essential for information. This further strengthens a philosophical ideology which says “Light and Darkness are not Opposites, but they Co-Exist”. This means one cannot exist without the other. Thus, all our algorithms, applications that generate this randomness, creating information from nothing and applying it in all sorts of fields from which we derive meaning from, in reality is chaos.

References

<http://iopscience.iop.org/article/10.1088/1367-2630/18/6/065004>

Andrea Rock, Paris-London University- "Pseudorandom Number Generators for Cryptographic Applications"

David DiCarlo, Liberty University- “Random Number Generation: Types and Techniques”

http://www.gamasutra.com/blogs/CharlotteWalker/20141021/228292/Develophttps://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/ping_Games_with_Random_Number_Generators.php

<https://csanyk.com/2012/08/random-number-generation-in-game-programming/>

<https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/>

<https://imgur.com/a/gsxww>

Discrete-Event System Simulation FOURTH EDITION

Jerry Banks Independent Consultant :-- . John S. Carson II · Brooks Automation Barry L Nelson

Northwestern University David M. Nicol University of Illinois, Urbana-Champaign

<http://whatis.techtarget.com/definition/Monte-Carlo-method-or-Monte-Carlo-analysis>

<http://pcg.wikidot.com/pcg-algorithm:generative-art>

<https://flafla2.github.io/2014/08/09/perlinnoise.html>

<http://faculty.rhodes.edu/wetzel/random/mainbody.html>

<http://faculty.rhodes.edu/wetzel/random/level23intro.html>

http://www.nytimes.com/2010/10/31/magazine/31FOB-onlanguage-t.html?_r=3&ref=on_language

http://commons.wikimedia.org/wiki/Dice_by_number_of_sides#D100

<http://www.awesomedice.com/blog/353/d20-dice-randomness-test-chessex-vs-gamescience/>

<http://arxiv.org/pdf/1008.4559.pdf>