from: https://www.random.org

Introduction to Randomness and Random Numbers

by Dr Mads Haahr

Random numbers are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena and for selecting random samples from larger data sets. They have also been used aesthetically, for example in literature and music, and are of course ever popular for games and gambling. When discussing single numbers, a random number is one that is drawn from a set of possible values, each of which is equally probable, i.e., a uniform distribution. When discussing a sequence of random numbers, each number drawn must be statistically independent of the others.

Are the Numbers Really Random?

This question is surprisingly hard to answer. Before we try, let's define what exactly we mean by a random number.

When discussing single numbers, a random number is one that is drawn from a set of possible values, each of which is equally probable. In statistics, this is called a uniform distribution, because the distribution of probabilities for each number is uniform (i.e., the same) across the range of possible values. For example, a good (unloaded) die has the probability 1/6 of rolling a one, 1/6 of rolling a two and so on. Hence, the probability of each of the six numbers coming up is exactly the same, so we say any roll of our die has a uniform distribution. When discussing a sequence of random numbers, each number drawn must be statistically independent of the others. This means that drawing one value doesn't make that value less likely to occur again. This is exactly the case with our unloaded die: If you roll a six, that doesn't mean the chance of rolling another six changes.

So, why is it hard to test whether a given sequence of numbers is random? The reason is that if your random number generator (or your die) is good, each possible sequence of values (or die rolls) is equally likely to appear. This means that a good random number generator will also produce sequences that look nonrandom to the human eye (e.g., a series of ten rolls of six on our die) and which also fail any statistical tests that we might expose it to. If you flip enough coins, you will get sequences of coin flips that seen in isolation from the rest of the sequence don't look random at all. Scott Adams has drawn this as a Dilbert strip, which is funny exactly because it is true:

What Dilbert is told is correct: It is impossible to prove definitively whether a given sequence of numbers (and the generator that produced it) **is random**. It could happen that the creature in the comic strip has been generating perfectly random numbers for many years and that Dilbert simply happens to walk in at the moment when there's six nines in a row. It's not very likely, but if the creature sits there for long enough (and Dilbert visits enough times), then it will eventually happen.

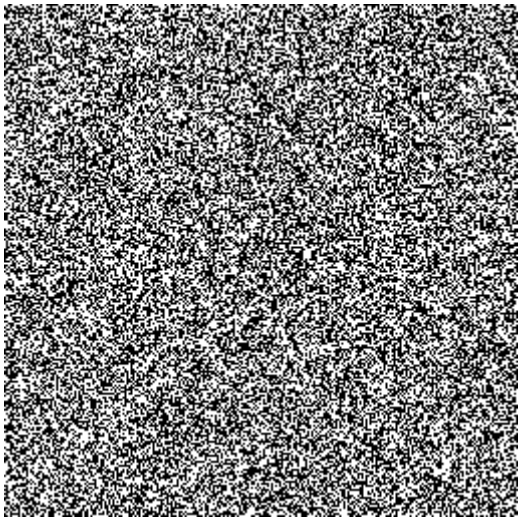
So, if it is impossible to definitively prove randomness, what can we do instead? The pragmatic approach is to take many sequences of random numbers from a given generator and subject them to a battery of **statistical tests**. As the sequences pass more of the tests, the confidence in the randomness of the numbers increases and so does the confidence in the generator. However, because we expect some sequences to appear nonrandom (like the ten rolls of six on our die), we should expect some of the sequences to fail at least some of the tests. However, if many sequences fail the tests, we should be suspicious. This is also the way you would intuitively test a die to see if it is loaded: Roll it many times, and if you see too many sequences of the same value coming up, you should be suspicious.

If you look at the Real-Time Statistics for RANDOM.ORG, you will sometimes see blocks of numbers that failed some of the tests. This does not mean that the numbers are not random. In fact, if all the blocks passed all the tests, we should be suspicious, because it would mean the generator would not be producing those sequences that don't look (but still would be) random.

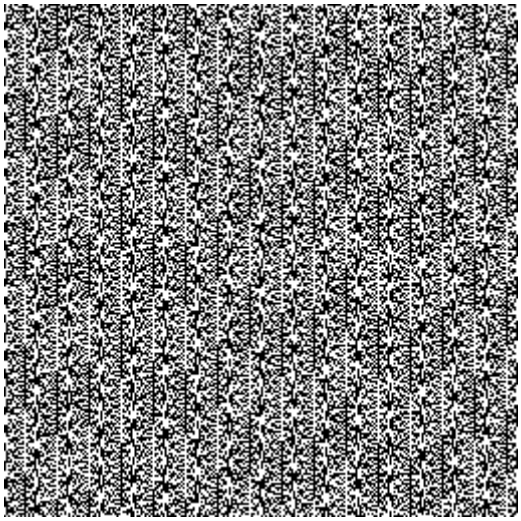
Simple Visual Analysis

One way to examine a random number generator is to create a visualisation of the numbers it produces. Humans are really good at spotting patterns, and visualisation allows you to use your eyes and brain directly for this purpose. While you shouldn't consider this type of approach an exhaustive or formal analysis, it is a nice and quick way to get a rough impression of a given generator's performance. The bitmaps shown below are sections of larger bitmaps created by Bo Allen in April 2008 to examine the

quality of two random number generators. Bo created the bitmap on the left with RANDOM.ORG's Bitmap Generator, which is of course a True Random Number Generator (TRNG), and the bitmap on the right with the rand() function from PHP on Microsoft Windows, which is a Pseudo-Random Number Generator (PRNG).



RANDOM.ORG



PHP rand() on Microsoft Windows

You can click on the images for the full-size (512×512) bitmaps or visit Bo Allen's comparison page where they are available side by side and where you'll also find the source code for Bo's program. As you can see from the images, the bitmap generated by the PHP/Windows pseudo-random number generator shows clear patterns compared to the one generated by RANDOM.ORG's true random number generator. Bo also found that the PHP rand() function performed considerably better on the GNU/Linux platform than on Microsoft Windows. While Bo's comparison doesn't constitute a formal analysis of the two generators, it clearly shows how careful you need to be about random numbers, especially if your site is a game or gambling site.

In general, it should be noted that pseudo-random number generators vary a lot in quality, and while the worst are very bad, the best are actually very good. You will find more information about the differences and trade-offs between the two approaches in my essay about randomness.

Pseudo-Random Number Generators (PRNGs)

As the word ‘pseudo’ suggests, pseudo-random numbers are not random in the way you might expect, at least not if you're used to dice rolls or lottery tickets. Essentially, PRNGs are algorithms that use mathematical formulae or simply precalculated tables to produce sequences of numbers that appear random. A good example of a PRNG is

the linear congruential method. A good deal of research has gone into pseudo-random number theory, and modern algorithms for generating pseudo-random numbers are so good that the numbers look exactly like they were really random.

The basic difference between PRNGs and TRNGs is easy to understand if you compare computer-generated random numbers to rolls of a die. Because PRNGs generate random numbers by using mathematical formulae or precalculated lists, using one corresponds to someone rolling a die many times and writing down the results. Whenever you ask for a die roll, you get the next on the list. Effectively, the numbers appear random, but they are really predetermined. TRNGs work by getting a computer to actually roll the die — or, more commonly, use some other physical phenomenon that is easier to connect to a computer than a die is.

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. Efficiency is a nice characteristic if your application needs many numbers, and determinism is handy if you need to replay the same sequence of numbers again at a later stage. PRNGs are typically also *periodic*, which means that the sequence will eventually repeat itself. While periodicity is hardly ever a desirable characteristic, modern PRNGs have a period that is so long that it can be ignored for most practical purposes.

These characteristics make PRNGs suitable for applications where many numbers are required and where it is useful that the same sequence can be replayed easily. Popular examples of such applications are simulation and modeling applications. PRNGs are not suitable for applications where it is important that the numbers are really unpredictable, such as data encryption and gambling.

It should be noted that even though good PRNG algorithms exist, they aren't always used, and it's easy to get nasty surprises. Take the example of the popular web programming language PHP. If you use PHP for GNU/Linux, chances are you will be perfectly happy with your random numbers. However, if you use PHP for Microsoft Windows, you will probably find that your random numbers aren't quite up to scratch as shown in this visual analysis from 2008. Another example dates back to 2002 when one researcher reported that the PRNG on MacOS was not good enough for scientific simulation of virus infections. The bottom line is that even if a PRNG will serve your application's needs, you still need to be careful about which one you use.

True Random Number Generators (TRNGs)

In comparison with PRNGs, TRNGs extract randomness from physical phenomena and introduce it into a computer. You can imagine this as a die connected to a computer, but typically people use a physical phenomenon that is easier to connect to a computer than a die is. The physical phenomenon can be very simple, like the little variations in somebody's mouse movements or in the amount of time between keystrokes. In practice,

however, you have to be careful about which source you choose. For example, it can be tricky to use keystrokes in this fashion, because keystrokes are often buffered by the computer's operating system, meaning that several keystrokes are collected before they are sent to the program waiting for them. To a program waiting for the keystrokes, it will seem as though the keys were pressed almost simultaneously, and there may not be a lot of randomness there after all.

However, there are many other ways to get true randomness into your computer. 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. The HotBits service at Fourmilab in Switzerland is an excellent example of a random number generator that uses this technique. Another suitable physical phenomenon is atmospheric noise, which is quite easy to pick up with a normal radio. This is the approach used by RANDOM.ORG. You could also use background noise from an office or laboratory, but you'll have to watch out for patterns. The fan from your computer might contribute to the background noise, and since the fan is a rotating device, chances are the noise it produces won't be as random as atmospheric noise.

Thunderstorms generate atmospheric noise

As long as you are careful, the possibilities are endless. Undoubtedly the visually coolest approach was the lavarand generator, which was built by Silicon Graphics and used snapshots of lava lamps to generate true random numbers. Unfortunately, lavarand is no longer operational, but one of its inventors is carrying on the work (without the lava lamps) at the LavaRnd web site. Yet another approach is the Java EntropyPool, which gathers random bits from a variety of sources including HotBits and RANDOM.ORG, but also from web page hits received by the EntropyPool's own web server.

Regardless of which physical phenomenon is used, the process of generating true random numbers involves identifying little, unpredictable changes in the data. For

example, HotBits uses little variations in the delay between occurrences of radioactive decay, and RANDOM.ORG uses little variations in the amplitude of atmospheric noise.

The characteristics of TRNGs are quite different from PRNGs. First, TRNGs are generally rather *inefficient* compared to PRNGs, taking considerably longer time to produce numbers. They are also *nondeterministic*, meaning that a given sequence of numbers cannot be reproduced, although the same sequence may of course occur several times by chance. TRNGs have no period.

https://www.random.org/integer-sets/?mode=advanced