# New Random Generators for Large-Scale Reproducible AI

Vincent Granville
vincentg@MLTechniques.com
www.GenAItechLab.com
Version 1.0, August 2024

## 1 Randomness and reproducibility: two key components

Modern GenAI apps rely on billions if not trillions of pseudo-random numbers. You find them in the construction of latent variables in nearly all deep neural networks and almost all applications: computer vision, synthetization, and LLMs. This component is overlooked. It is assumed that Python random functions do a good job, and very few seem to care about reproducibility, though customers do. Incidentally, all my GenAI apps described in my book [2] are fully reproducible, even those based on GANs (generative adversarial networks).

When producing so many random numbers or for strong encryption, you need top grade generators. The most popular one – adopted by Numpy and other libraries – is the Mersenne Twister [Wiki], and advertised as not fit for cryptography. It is known for its flaws: see chapter 4 in my book about chaos [1].

This paper has its origins in the development of a new framework to prove the conjectured randomness of the digits of infinitely many simple math constants, such as $e^{1486/3331}$ or $\sqrt{491/3127}$. By randomness, I mean that if you don't know the name of the constant in question, you can not predict the next digit no matter how many past digits you collected, even if you collected more than there are atoms in the universe. In short, for all purposes, the sequence can not be distinguished – statistically speaking – from true random numbers. Note that in practical application, you use millions of such constants, typically using a few millions digits from each one, starting at arbitrary locations. The procedure is described in chapter 4 in [1].

The remaining focuses on three main areas. First, how to efficiently compute the digits of the mathematical constants in question to use them at scale. Then, some methodology to compare two types of random numbers: those generated by Python, versus those from the irrational numbers investigated here. Finally, proposing a new type of random digit sequence based on an incredibly simple formula leading to fast computations.

One of the benefits of my proposed random bit sequences, besides stronger randomness (infinite period) and fast implementation at scale, is to not rely on external libraries that may change over time. These libraries may change and render your results non-replicable in the long term if (say) Numpy decides to modify the internal parameters of its random generator.

Also, if your generated data is more random than the numbers used for testing its randomness, it will lead to false positives, flagging your experiment as not well randomized when actually the issue if with the test. In this article, I explain how to deal with this. Some of my tests involve predicting the value of a string given the values of previous strings in a sequence, a topic at the core of many large language models (LLMs).

Methods based on neural networks – mines being an exception – are notorious for hiding the seeds used in the various random generators involved. It leads to non-replicable results. It is my hope that this article will raise awareness about this issue, while offering better generators that do not depend on which library version you use. Last but not least, the datasets used here are infinite, giving you the opportunity to work with truly big data and infinite numerical precision.

## 2 Computing the digits of special math constants

Here, the constants must lie between 0 and 1. While I use the binary numeration system, the Python code also works with integer bases $b$ other than 2. To introduce my method, I start with a textbook example: the exponential function. Let $\lambda_m(k) = (k \bmod m) + 1$. Define the following recursion with $k \geq 2$, assuming $z$ and $m$ are fixed integers with $z, m \geq 1$:

$$
\begin{aligned}
p_k(z,m) &= z \cdot \lambda_m(k-1) \cdot p_{k-1}(z,m) + 1 \\
q_k(z,m) &= z \cdot \lambda_m(k-1) \cdot q_{k-1}(z,m)
\end{aligned}
$$

The initial conditions are $p_1(m,z) = 0$ and $q_1(m,z) = z$. I also define the limit numbers

$$
\xi(z,m) = \lim_{k\to\infty} \frac{p_k(z,m)}{q_k(z,m)}, \quad \xi(z) = \lim_{m\to\infty} \xi(z,m) = \exp\left(\frac{1}{z}\right) - \frac{z+1}{z}. \tag{1}
$$

If $m$ is finite, then $\xi(z, m)$ is a rational number, albeit with a gigantic period even for rather small values of $m$. Finally, to make the discussion easier, I also introduce the notation

$$\xi_k(z, m) = \frac{p_k(z, m)}{q_k(z, m)} \tag{2}$$

The originality of my approach lies in the fact that for a fixed $k$, I am interested only in the first few digits of the rational number $\xi_k(z, m)$, that is, its prefix, skipping the periodic part. Also, under the right conditions, the number of digits in the prefix, referred to as the prefix length, increases with $k$ and eventually becomes infinite as $k \to \infty$. In addition, even at the limit as $k \to \infty$, the number $\xi(z, m)$ is rational if $m$ is finite. The implications are discussed later in this article.

## 2.1 P-adic valuations

All rational numbers have a period – a group of digits repeating itself indefinitely – preceded by a prefix. For instance, in base $b = 10$ (the ordinary decimal system), we have:

$$\frac{3011}{8325} = 0.36168168168168168\dots$$

In this example, the prefix is 36, and the period is 168. In fact, there is a simple formula to detect the length of the prefix, that is, the number of digits that it contains in a base $b$. Let us assume that $p, q$ are positive integers with $p < q$. To compute the prefix $\pi(p, q)$ of $p/q$ in base $b$, where $b$ is a prime number, we proceed as follows:

- Compute the $p$-adic valuations $\nu_b(p), \nu_b(q)$ of $p$ and $q$ in base $b$. The $p$-adic valuation of $p$ in prime base $b$ is defined as the exponent of the highest power of $b$ that divides $p$ [Wiki].
- The length of the prefix $\pi(p, q)$ is given by the formula

$$L = L(\pi(p, q)) = \max\Big[0,\ \nu_b(q) - \nu_b(p)\Big] \tag{3}$$

  It remains unchanged if you multiply $p$ and $q$ by an integer constant, even by a power of $b$. For simplicity, when there is no confusion, $L(\pi(p, q))$ is denoted as $L$.
- The prefix is obtained via the following integer division [Wiki]:

$$\pi(p, q) = (p \cdot b^L)//q = \frac{1}{q} \cdot \Big[p \cdot b^L - (p \cdot b^L \bmod q)\Big]. \tag{4}$$

  This integer is typically represented as a string of length $L$ in base $b$, with padding zeroes on the left if needed, to match the length.

When everything goes well, the iterations leading to Formula (1) produce integer vectors $(p_k, q_k)$ with a prefix length that on average, increases as $k$ increases. In particular, any $k$ with a prefix $\pi(p_k, q_k)$ longer than the previous ones, adds new digits to $\xi(z, m)$ while preserving digits already obtained. I explain in the next section what this means. For now, it is sufficient to know that everything works nicely in the exponential case introduced earlier. Newly added digits won't be changed later as $k$ increases.

## 2.2 Digit blocks, speed of convergence

From now on, I use $b = 2$. Also, the function $\nu_2(\cdot)$ is denoted as $\nu(\cdot)$ for convenience. When $z = 1$ and $m = \infty$ in the exponential case (1), each new iteration $k$ yields $\nu(k)$ new digits. In particular, $\nu(k) = 0$ if and only if $k$ is odd. Also $p_k$ is always odd regardless of $k$, $z$, or $m$. Thus $L(\pi(p_k, q_k)) = \nu(q_k) = \nu(k!)$ in this case. These properties are peculiar to this system. Other systems discussed later also have nice properties, albeit different.

When the digits of $\xi_k(z, m)$ computed in previous iterations are preserved when overwritten as $k$ increases, the system is said to be digit-preserving. A new set of digits added at a specific iteration is called a digit block. The goal is to find patterns (or their absence) in the non-empty blocks, and to study the asymptotic properties of the digits of $\xi_k(z, m)$ as $k \to \infty$ by focusing on the successive prefixes. The cases $m = \infty$ and $m < \infty$ are treated in the same way, even though the former results in $\xi_k(z, m)$ being rational as $k \to \infty$, while the latter leads to a known irrational number given in (1). To summarize, my new framework allows you to analyze the digit distribution of $e$ by focusing on the prefix of peculiar rational numbers, some having the same proportions of zeros and ones in the binary system, and some not, with the prefix digits sequentially matching those of $e$ when $m, k \to \infty$.

If the goal is to study the digit distribution from a theoretical point of view, speed of convergence may not be important. In applications such as pseudo-random number generators (PRNGs), speed is critical. The system presented here relies on the Taylor series of the exponential function. Binary splitting [Wiki] can accelerate convergence, along with fast multiplication [Wiki]. In general, having a numerator $p_k$ not divisible by the base $b$, works best. Likewise, the faster $\nu_b(q_k)$ increases as $k$ increases, the faster the convergence. Choosing a power of $b$ for $z$, helps achieve this goal. So far, the best improvement (several orders of magnitude) was obtained using a very efficient computation of the $p$-adic functions. To produce random bits as fast as possible, I discuss a system with very fast convergence in section 2.3, though the limit number $\xi$ is not a known constant.

Finally, if you are not interested in discovering potential patterns in the block sequence (and even if you do), you can skip the production of intermediate digits to eliminate redundancy, avoiding all the integer divisions linked to Formula (4), as well as the $p$-adic computations. That is, you can compute the $(p_k, q_k)$ vectors iteratively and get the digits in the last iteration. In the exponential case, since block sizes are known in advance, you can slice the final digit sequence accordingly to retrieve the blocks.

In practice, for random bits generation, you use multiple sequences in parallel, for instance different values of $z$ or a combination of sequences discussed in section 2.3. This further increases the speed, thanks to parallelization. Interestingly, typical congruential random number generators are similar to using $m < \infty$ in the exponential case, resulting in a finite period and less randomness. See chapter 4 in [1] for a discussion about the benefits of using irrational numbers (in this case, $m = \infty$).

### 2.3 A plethora of interesting pseudo-random sequences

The exponential case discussed at the beginning of section 2 is just one of many examples leading to interesting results and applications to random bits generation. There are other examples in the Python code, listed below. Here again, $\xi$ denotes the limiting value of $p_k/q_k$ as $k \to \infty$, while IC stands for initial conditions.

**Continued fractions**, with $\xi = (-1+\sqrt{5})/2$. IC: $p_1 = 13$, $q_1 = 21$.

$p_k = q_{k-1}$,
$q_k = p_{k-1} + q_{k-1}$.

**Second order linear recursion**, with $\xi = 1/(3-\sqrt{3})$. IC: $p_0 = 0$, $p_1 = 1$, $q_0 = 0$, $q_1 = 2$.

$p_k = 2p_{k-1} + 2p_{k-2} + 1$,
$q_k = 2q_{k-1} + 2q_{k-2}$.

**Square root**, with $\xi = \sqrt{2}/4$. IC: $p_1 = 0$, $q_1 = 2$.

$p_k = 2p_{k-1} + 1$ if $(2p_{k-1}+1)^2 < 2q_{k-1}^2$, else $p_k = 2p_{k-1}$,
$q_k = 2q_{k-1}$.

**Fast recursion**. IC: $p_1 = 1$, $q_1 = 2$.

$p_k = 2^k p_{k-1} + 3^k \bmod 2^k$,
$q_k = 2^k q_{k-1}$.

**Logarithm**, with $\xi = \log 2$. IC: $p_0 = 1$, $p_1 = 5$, $q_0 = 2$, $q_1 = 8$.

$p_k = (3k+2)p_{k-1} - 2k^2 p_{k-2}$,
$q_k = 2(k+1)q_{k-1}$.

All the above systems except the logarithm case are digit-preserving. Play with the Python code to see how block sizes evolve over the iterations, and how frequently new digits are added, in each case. The fast recursion system has this property:

$$\frac{p_k}{q_k} = \sum_{l=1}^{k} \frac{3^l \bmod 2^l}{2^{l(l+1)/2}} \to \xi = 0.673443360740852\ldots \text{ as } k \to \infty. \tag{5}$$

With the exception of the first two digit blocks $B_1, B_2$ merged together due to initial conditions, blocks $B_k$ ($k = 3, 4$ and so on) is $k$-bits long in base 2, with $B_k = 3^k \bmod 2^k$. The lower (rightmost) bit in each block is always 1. That is, $B_k \bmod 2 = 1$. For that reason, one may prefer the unbiased version, where $3^k \bmod 2^k$ is replaced by $(3^k - 1)/2 \bmod 2^k$. Finally, it is probably not difficult to prove that $\xi$ in Formula (5) is irrational.

The system leading to $\sqrt{2}/4$ behaves differently. In base 2, all non-empty blocks consist of a variable number of zeros on the left, with 1 for the rightmost bit. In other words, the block sequence represents the successive runs of zeros in the binary expansion of $\sqrt{2}/4$. The non-empty blocks are 01, 01, 1, 01, 01, 000001, 001, 1, 1, 1, 001, 1, and so on. If you put them together, you get the binary expansion of $\sqrt{2}/4$.

Nice systems are the exception. Most are not digit preserving. In particular, the logarithm system defined in the above list, is not digit-preserving with the proposed initial conditions. Likewise, if you change the initial conditions to $p_1 = 1, q_1 = 2$ in the continued fraction system, $p_k/q_k$ still converges to the same value but the digit-preserving property is lost.

The continued fraction system described here leads to the golden ratio $\xi = (-1+\sqrt{5})/2$. It is an extreme case. Here $p_k, q_k$ are two consecutive Fibonacci numbers. However, $\nu(q_k)$ grows extremely slowly. Even though $\nu(p_k) = 0$ when $\nu(q_k) \neq 0$ (thus boosting convergence), non-empty blocks become increasingly rare, though there are still infinitely many of them. The result: after $10^6$ iterations, only 20 binary digits of $\xi$ are computed. By contrast, the fast recursion system yields $5 \times 10^{11}$ digits after $10^6$ iterations.

# 3 Testing random number generators

In chapter 4 in [1], I compare traditional congruential generators with those based on irrational numbers. The goal here is different. Rather than testing the raw sequences, I focus on the digit blocks discussed earlier. These blocks may exhibit strong patterns, not compatible with randomness. This is the case for the square root system discussed in section 2.3. However non-randomness in the block distribution does imply non-randomness in the raw digit sequence. In the end, the binary digits of constants such as $\sqrt{2}$ better mimic randomness, compared to random bit sequences produced by Numpy with its Mersenne Twister.

Indeed, the goal here is to find patterns, to better understand the digit distributions of the irrational numbers in question. The hope is that it leads to a better understanding of the underlying theoretical distributions. In turn, it could help prove that some of these digit sequences behave exactly like true random sequences. See section 3.1 for an example.

The methodology is as follows. I compare statistics based on the actual blocks, with those based on blocks consisting of random digits generated by Numpy. When some discrepancy is found, I use theoretical results to assess whether Python, my blocks, or both fail at mimicking randomness. Even when both pass the test, it is possible to check which one gets a better grade. Rather than sharing all the results, I present the methodology in a teaching style to help the reader learn how to use it in various contexts.

A word of caution before digging deeper into the details. The probability that a 20-bit sequence consists of zeroes only, is 0.0001%. If you look at $10^6$ such sequences, the chance that at least one of them consists of 20 zeroes, is 63%. In other words, if you test a large number of independent sequences, for instance those produced by the digits of square roots of many prime numbers, you are bound to find one failing the randomness test, just by chance. Conversely, if you apply a large number of independent tests to any single sequence, chances are that it will fail some of the tests just by chance.

## 3.1 Theoretical properties of the digits of $\sqrt{2}$

In the system leading to $\sqrt{2}/4$, the blocks are anything but random: the rightmost digit is 1, preceded by a variable number of zeros. In single-digit blocks, the digit is also 1. The first block is $B_2 = 01$. It is followed by an empty block $B_3$, then $B_4 = 01$. Since $q_k = 2^k$, $\nu(q_k) = k$, and $p_k < q_k$, we have $0 \leq \nu(p_k) < k$. Then, the length of the prefix satisfies $L(\pi(p_k, q_k)) = k - \nu_k(p_k) \leq k$ and block $B_k$ is non-empty if and only if $\nu(p_k) = 0$.

Furthermore, the total number of digits of $\sqrt{2}/4$ collected over the first $k$ blocks is equal to the sum of the lengths of these blocks. All of this allows you to make statements about the digit distribution. In particular, this distribution is fully characterized by the lengths of the successive runs of zeros (including runs of length zero). That is, by the lengths $|B_k|$ of the non-empty blocks $B_k$. Each of them ends with the digit 1, terminating the current run of zeros. For non-empty blocks $B_k$ with $k > 2$, the length (the number of digits) satisfies:

$$|B_k| = k - \underset{l<k}{\text{argmax}}\Big\{\nu(p_l) = 0\Big\} = 1 + \nu(p_{k-1}) < k.$$

The argmax part represents the largest $l$ such that $\nu(p_l) = 0$ and $l < k$. It is conjectured that $|B_k| < 1 + \log_2 k$. Also, if $\nu(p_k) \neq 0$, then $|B_k| = 0$ and $\nu(p_k) = 1 + \nu(p_{k-1})$. The number of ones in the cumulated digits collected at any given time, is equal to the number of non-empty blocks obtained.

I now check whether block lengths have a geometric distribution. If they don't, it means that the randomness assumption is violated. If they do, then the expected frequency is $f_n = 2^{-n}$ for non-empty blocks of length $n$, with $n = 1, 2$ and so on, corresponding respectively to blocks 1, 01, 001, 0001, and so on. It would also imply that exactly 50% of the binary digits are zero, when $k$ becomes infinite. In Figure 1, the `Python` chart bars correspond to simulated run lengths using the `random` function in Numpy with 3 different seeds and 24000 digits, while `Actual` correspond to the block lengths in the square root system, with the same sample size.

For lengths $n > 6$, counts are very small due to the small sample size, explaining the volatility. Thus, the randomness assumption is not violated in the run test. It is worth trying with larger sample sizes to check if the good results extend beyond $n = 6$. Another test worth trying is checking the autocorrelations in subsequences of run lengths. They should all be close to zero if the data is random enough.

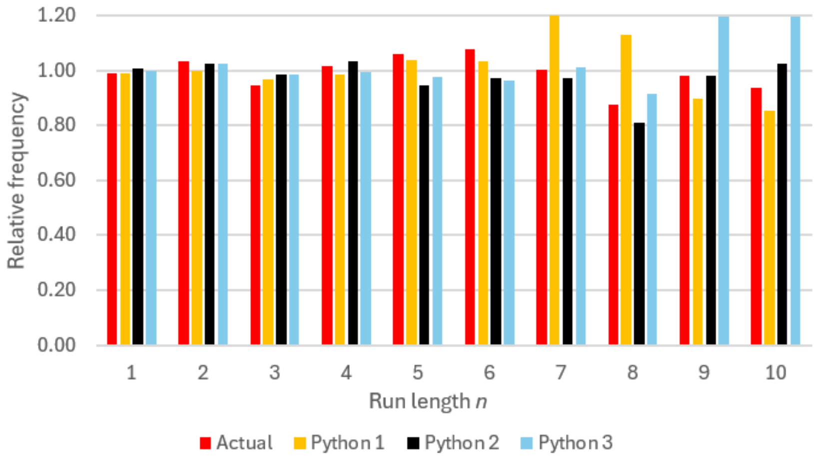


Figure 1: Relative frequencies of zero runs in 4 binary digits sequences

### 3.2 Fast recursion and congruential equidistribution

The square root and fast recursion systems share some properties: $q_k$ is a power if 2, and each block $B_k$ represents an odd integer. In the square root system, $q_k = 2^k$ and $p_k$ is just one bit. However, in the fast recursion system, $q_k = 2^{k(k+1)/2}$ and $p_k$ is $k$ bits long. Thus, convergence is much faster, and you can concatenate the successive blocks to create random bit sequences growing quadratically in size as $k$ increases. The fact that $B_k$ is always odd, is a weakness in this case but not in the square root system. To avoid this problem, let's start by generalizing the fast recursion system, as follows:

$$p_k = 2^k p_{k-1} + A_k \bmod 2^k,$$

$$q_k = 2^k q_{k-1},$$

with the initial conditions $p_1 = A_1 \bmod 2$, and $q_1 = 2$. Here $(A_k)$ is a sequence of integers, with $A_k$ much larger than $2^k$. In the first version, $A_k = 3^k$. Again, $B_k = A_k \bmod 2^k$, possibly with extra zeroes on the left as usual, so that $B_k$ is $k$-bit long when converted to a string. If the digits are random enough, one would expect that over many blocks, $B_k \bmod m$ covers all integer values (called residues) between 0 and $m-1$, each with about the same frequency, regardless of $m > 1$. This property is called asymptotic congruential equidistribution. It is not satisfied when $m$ is even, if $A_k = 3^k$.

| | | Fast Recursion | | | Python Library | | |
|---|---|---|---|---|---|---|---|
| $m$ | Exp. | Model1 | Model2 | Model3 | Seed1 | Seed2 | Seed3 |
| 2 | 5000 | 4998.50 | 0.5 | 92.50 | 1.50 | 102.50 | 63.50 |
| 3 | 3333 | 48.63 | 41.82 | 53.96 | 10.03 | 25.66 | 29.13 |
| 4 | 2500 | 2498.75 | 0.43 | 65.48 | 37.71 | 51.91 | 40.90 |
| 5 | 2000 | 37.22 | 54.95 | 40.25 | 59.92 | 38.23 | 30.94 |
| 6 | 1667 | 1665.84 | 40.27 | 42.13 | 37.75 | 36.85 | 30.98 |
| 7 | 1429 | 22.41 | 20.53 | 29.68 | 26.51 | 25.61 | 22.47 |
| 8 | 1250 | 2163.11 | 0.33 | 43.66 | 33.28 | 29.86 | 28.98 |
| 9 | 1111 | 36.10 | 32.00 | 23.59 | 19.64 | 28.39 | 24.03 |
| 10 | 1000 | 999.26 | 29.62 | 34.70 | 32.89 | 32.69 | 36.29 |
| 11 | 909 | 26.23 | 20.84 | 35.34 | 19.86 | 17.90 | 45.75 |
| 12 | 833 | 832.69 | 30.02 | 30.22 | 27.71 | 19.75 | 28.48 |
| 13 | 769 | 21.48 | 28.82 | 23.03 | 22.25 | 30.50 | 40.53 |
| 14 | 714 | 713.39 | 22.39 | 25.56 | 27.83 | 24.42 | 24.32 |
| 15 | 667 | 24.59 | 27.13 | 23.31 | 22.81 | 19.49 | 27.04 |
| 16 | 625 | 1080.69 | 0.24 | 27.31 | 26.90 | 18.44 | 19.17 |

Table 1: Stdev for congruential equidistribution test (10,000 blocks)

One would think that choosing $A_k = (3^k - 1)/2$ would fix the issue. Indeed, it improves the situation, yet the property is not satisfied if $m$ is a multiple of 4. What about $A_k = 3^k + k$? Now the property seems to be satisfied, but with some problem when $m$ is a power of 2. For these $m$'s, the residues are evenly distributed – perfectly – which is not compatible with randomness. You need the right amount of variance: not too much, not too little. This is accomplished with $A_k = 5^k//2^k$. Here "//" stands for the integer division, also denoted as $5^k >> k$ (bit shift) since the denominator is a power of 2. It is equal to $\lfloor 5^k/2^k \rfloor$. You cannot use $A_k = 3^k//2^k$ because $A_k$ must be much larger than $2^k$, otherwise the property is violated for obvious reasons.

In the literature, congruential equidistribution is referred to as uniform distribution for integer sequences, a term that is misleading. Its equivalent for real numbers is called equidistribution modulo 1. If $\theta$ is an irrational number, then the integer sequence $A_k = \lfloor k\theta \rfloor$ satisfies the property. It also works with $A_k = \lfloor b^k\theta \rfloor$, where $b > 1$ is an integer (the base), assuming $\theta$ is a normal number in base $b$. Well known counter-examples: when $A_k$ is a polynomial in $k$ of degree 2 or higher with integer coefficients, and when $A_k = \lfloor k!\, e \rfloor$. Finally, the numbers $\sqrt{2}, e, \log 2$ and $\pi$ are conjectured to be normal in any base. Section 3.1 in this article sets new foundations to study the normality of $\sqrt{2}$ in base $b = 2$. For more details on congruential equidistribution, see chapter 5 in [3].

| | Fast Recursion | | | Python Library | | |
|---|---|---|---|---|---|---|
| String | Model1 | Model2 | Model3 | Seed1 | Seed2 | Seed3 |
| 0 | 24,986,553 | 24,986,515 | 24,993,579 | 24,995,827 | 24,998,289 | 24,995,167 |
| 1 | 25,008,446 | 25,008,484 | 25,001,420 | 24,999,172 | 24,996,710 | 24,999,832 |
| 00 | 8,327,584 | 8,326,983 | 8,330,446 | 8,333,231 | 8,335,234 | 8,332,216 |
| 01 | 12,497,442 | 12,497,468 | 12,496,467 | 12,495,140 | 12,494,518 | 12,498,799 |
| 10 | 12,497,441 | 12,497,469 | 12,496,468 | 12,495,141 | 12,494,517 | 12,498,799 |
| 11 | 8,338,874 | 8,338,302 | 8,335,781 | 8,334,393 | 8,334,637 | 8,333,329 |
| 000 | 3,566,418 | 3,566,701 | 3,570,993 | 3,571,832 | 3,572,551 | 3,569,057 |
| 001 | 6,247,215 | 6,245,949 | 6,248,383 | 6,249,023 | 6,250,880 | 6,248,691 |
| 010 | 4,996,856 | 4,997,481 | 4,996,396 | 4,996,259 | 4,995,501 | 4,998,112 |
| 011 | 6,251,062 | 6,249,773 | 6,250,134 | 6,250,077 | 6,250,783 | 6,249,944 |
| 100 | 6,247,215 | 6,245,949 | 6,248,383 | 6,249,023 | 6,250,879 | 6,248,691 |
| 101 | 5,000,084 | 5,000,724 | 4,998,010 | 4,996,975 | 4,996,353 | 4,999,168 |
| 110 | 6,251,061 | 6,249,774 | 6,250,134 | 6,250,078 | 6,250,783 | 6,249,945 |
| 111 | 3,575,452 | 3,575,762 | 3,572,484 | 3,573,978 | 3,572,508 | 3,572,486 |
| $\chi^2$ | 7.19 | 7.24 | 0.92 | 0.17 | 0.04 | 0.33 |

Table 2: String occurences in sequences with 49,994,999 digits

Table 1 shows a statistical summary for each residue class modulo $m$, with $2 \leq m \leq 16$, for six digit sequences based on $N = 10{,}000$ blocks: using the `random` Python library with seeds 0, 1, 2 on the right, and three different versions of the fast recursion system on the left. The number of digits per sequence, called length, is $N(N-1)/2-1 = 49{,}994{,}999$. For each block $B_k$ and for each modulus $m$, I computed the residue $B_k \bmod m$. Then I counted the occurrences of each residue over the $N$ blocks, for each $m$. The expected value given $m$, is $N/m$ and featured in the second column. The six rightmost columns show the observed standard deviations for these values. In case of randomness (and thus equidistribution), the standard deviation should be small, but not too small. Also, it should decrease roughly at the same speed as $1/\sqrt{m}$ as you go down the rows in the table. The only sequence compatible with these requirements is Model3. The other ones exhibit patterns not consistent with randomness. The 3 models under "Fast Recursion" are, from left to right, $A_k = 3^k$, $A_k = 3^k + k$, and $A_k = 5^k//2^k$. See code in section 4.1.

Table 2 shows the observed frequencies for various strings, in the 6 digit sequences. The means seem correct at first glance: for instance, about 50% of zeroes and ones when you look at the first 2 rows. Model1 and Model2 exhibit more variance, while the Python library (the three rightmost columns) produce lower variance. Which ones are correct? It turns out that all but Model1 and Model2 are acceptable, with Model3 being the best and outperforming the Python library. To come to this conclusion, I computed the following statistics:

$$\chi^2(S) = \sum_{s\in S} \frac{(X_s - \mathrm{E}[X_s])^2}{\mathrm{E}[X_s]}, \tag{6}$$

where $S$ is a set of strings for instance $S = \{00, 01, 10, 11\}$, $X_s$ is the number of occurrences of $s$ in the digit sequence, and $\mathrm{E}[X_s]$ is the expected number of occurrences if the digits were random. It has a $\chi^2$ distribution with $|S| - 1$ degrees of freedom, where $|S|$ is the number of strings in $S$. I computed $\chi^2$ for $S = \{0, 1\}$. The results are displayed in the bottom row. In this case, the $\chi^2$ expectation is 1.00. The chance that it is above 7.19 is about 0.73%, while the chance that it is below 0.04 is 15.82%. Clearly, Model3 is the best.

The computation of $\chi^2$ for 2-bit and 3-bit strings – say, $S = \{00, 01, 10, 11\}$ – is more challenging because the strings overlap in the digit sequence. Thus, even in case of perfect randomness, these strings have uneven frequencies, and the independence requirement to apply formula (6) is violated. Yet, it can be addressed with more advanced probability theory.

More tests are needed to assess the quality of Model3: the run test in section 3.1, conditional independence (see section 3.3), auto-correlations and so on. The tests performed so far suggest that it outperforms Python libraries to generate random bits. Not only that, but with efficient implementation, it should also run faster. Note that if $A_k = 5^k//2^k$, then $A_k = 10^k >> 2k$. Finally, we need to test with different values of $N$.

## 3.3 Exponential system: predicting the next block

Constants such as $e$ have been thoroughly tested and passed all the randomness tests. The goal here is to focus on the blocks rather than the digits. They may exhibit patterns, at least at the beginning of the sequence. Studying them may help understand the mechanisms at play, possibly leading to theoretical results, for instance the fact that any binary string is found in the digit sequence, infinitely many times. I discuss three topics:

- Non-random behavior in one-digit blocks, with statistical analysis to back it.
- The first occurrence of any string, in the block sequence.
- Can we predict the next block given the current block?

### 3.3.1 Pattern in one-digit blocks: more ones than zeroes?

The pattern in question is not incompatible with randomness: it is balanced by more zeroes than ones in blocks consisting of multiple digits. I expect the pattern to weaken when considering a very large number of blocks. Here, I looked at the first 22,500 non-empty blocks, totaling 44,991 digits. The proportion of zeroes is 49.78%, which seems fine. However 11,250 of these blocks are one-digit, thus totaling 11,250 digits, but with a proportion of zeroes equal to only 48.07%

Now, let's see if this could happen by chance or not, and whether there is an explanation. Using Formula (6) with $S = \{0, 1\}$, we have $\chi^2 = 0.88$ for the first ratio, 49.78%. This is within range for a $\chi^2$ with 1 degree of freedom: the expected value is 1.00. But for 48.07%, we have $\chi^2 = 16.74$, and $P[\chi^2 \geq 16.74] = 0.00004$. That's the probability that it could happen by chance, also called $p$-value.

What caused the imbalance is a sharp increase in ones at the end of the sequence in question. Big enough to cause a noticeable and long-lasting drop in the proportion of zeroes. While still within possible values in the digit sequence, it is well outside of the 99.9% range when looking at single-digit blocks. There is a lot of literature on simple random walks (those with equal probability of zero and one) as well as their time-continuous equivalent, Wiener processes. The cumulative digits of $e$ are conjectured to follow a simple random walk. The maximum discrepancy between the cumulative sums of zeroes and ones at any iteration $k$ is well studied, and its expectation is $\sqrt{\pi k/2}$. The law of the iterated logarithm offers bounds for infinite sequences. It applies to the digits, not the blocks. More on this in chapter 1, in my book [1].

### 3.3.2 Block coverage problem

In the square root system in section 3.1, non-empty blocks have deterministic digits: each block consists of a single 1 in the rightmost position, usually preceded by 0's on the left. But the successive block lengths appear to be unpredictable. In the exponential system, the opposite is true. Block digits are randomly distributed, but the length of block $B_k$ is deterministic, and equal to $\nu(k)$. By definition, A block of length $n$ represents an $n$-bit integer. How long does it take to cover all potential $2^n$ values? In other words, how many successive blocks of length $n$ are needed until all potential combinations of $n$ bits are covered?

Here I address this problem. Let's first look at what to expect if the digits were random. In that case, at some point we see the first block of length $n$. In the exponential system, it is block $B_k$ with $k = 2^n$. What can we say about the second block of length $n$? It would be block $B_k$ with $k = 3 \cdot 2^n$ in the exponential system. What are the chances that it is different from the first one, assuming randomness?

More generally, let $T_{n,i}$ be the random variable denoting the number of blocks of length $n$ visited until we cover $i$ distinct ones. Also, let $\Delta_{n,i} = T_{n,i} - T_{n,i-1}$ with $i > 0$ and $T_{n,0} = 0$. We have:

$$\mathrm{E}[\Delta_{n,i}] = \sum_{j=1}^{\infty} j \cdot P(\Delta_{n,i} = j) = \sum_{j=1}^{\infty} j \cdot \Big(\frac{i-1}{2^n}\Big)^{j-1} \cdot \frac{2^n - (i-1)}{2^n} = \frac{2^n}{2^n - i + 1}. \tag{7}$$

Now, let $T_n$ be the total number of blocks of length $n$ visited until we cover all the $2^n$ distinct combinations of zeroes and ones. We have:

$$\mathrm{E}[T_n] = \sum_{i=2}^{2^n} \mathrm{E}[\Delta_{n,i}] = \sum_{i=2}^{2^n} \frac{2^n}{2^n - i + 1} = 2^n \cdot \sum_{i=1}^{2^n - 1} \frac{1}{i}. \tag{8}$$

It follows that $\mathrm{E}[T_n]$ is asymptotically equal to $(\log 2) \cdot 2^n \cdot n$ as $n$ tends to infinity. Now, I compare the predicted time arrivals $T_{n,1}, T_{n,2}, \dots, T_{n,2^n}$ with the observed ones.

| $i$ | $\mathrm{E}[T_{n,i}]$ | $T^*_{n,i}$ | $T_{n,i}$ | $B_{k_i}$ | $k_i$ |
|---|---|---|---|---|---|
| 1 | 1.00 | 1 | 1 | 0000 | 16 |
| 2 | 2.07 | 2 | 2 | 1011 | 48 |
| 3 | 3.21 | 3 | 3 | 1100 | 80 |
| 4 | 4.44 | 4 | 4 | 1110 | 112 |
| 5 | 5.77 | 6 | 5 | 1001 | 144 |
| 6 | 7.23 | 7 | 6 | 0110 | 176 |
| 7 | 8.83 | 8 | 9 | 0011 | 272 |
| 8 | 10.61 | 10 | 10 | 1101 | 304 |
| 9 | 12.61 | 12 | 11 | 0111 | 336 |
| 10 | 14.89 | 15 | 14 | 0001 | 432 |
| 11 | 17.56 | 16 | 16 | 1111 | 496 |
| 12 | 20.76 | 17 | 22 | 0101 | 688 |
| 13 | 24.76 | 20 | 32 | 0010 | 1008 |
| 14 | 30.09 | 30 | 33 | 1000 | 1040 |
| 15 | 38.09 | 39 | 34 | 0100 | 1072 |
| 16 | 54.09 | 45 | 76 | 1010 | 2416 |

Table 3: New block arrival times $T_{n,i}$ ($n = 4$)

Table 3 shows the $2^n$ arrival times $T_{n,i}$ ($i = 1, 2, \dots, 2^n$) for first occurrence of a new block of length $n$, here with $n = 4$. The expected value $\mathrm{E}[T_{n,i}]$ is based on Formula (7). For comparison purposes, I also included the values obtained by replacing the block strings by random strings using the `random` function in Numpy. These are denoted as $T^*_{n,i}$. They are based on the same block structure, and obtained by setting `random=True` in the Python code in section 4.2. I noticed that the last unseen block in the exponential system takes longer than expected to show up, not just for $n = 4$. This may be true only for small $n$. It is not incompatible with strong randomness in the full digit sequence. Note that $k_i = 2^n \cdot (2T_{n,i} - 1)$ and $T_{n,1} = 1$.

### 3.3.3 Predicting the next block

Table 4 shows counts for single-digit blocks 0 and 1, when the previous non-empty block is 00, 01, 10, or 11. Again, I looked at the first 22,500 non empty blocks.

| Block | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 0 | 684 | 674 | 701 | 640 |
| 1 | 757 | 766 | 690 | 713 |
| $\chi^2$ | 3.698 | 5.878 | 0.087 | 3.938 |

Table 4: Counts for blocks 0 and 1 given previous block

In case of independence and uniform distribution, the eight counts should be roughly the same. But they exhibit too much variance: Formula (6) yields $\chi^2 = 17.534$, here with $8 - 1 = 7$ degrees of freedom. The $p$-value is $P[\chi^2 > 17.534] = 0.014$. Thus, the probability for this to happen by chance is 1.4%.

However, when treated separately in four conditional groups as in Table 4, the difference between the two counts in a same column is almost within normal range, except for column 01 ($p$-value = 0.015). But since we computed four $\chi^2$, it is not surprising that one of them is off. In any case, the probability that the current block is 1 given that the previous one is 01, is too high, at least in the first 22,500 non empty blocks. Note that for the exponential system, a block of length 2 can only be followed by a block of length 1, excluding the empty block between both. Finally, the findings in this section overlap with and corroborate those in section 3.3.1.

# 4 Python code

The main code is in section 4.2. The short code in section 4.1 covers the congruential equidistribution test for the fast recursion system discussed in section 3.2.

## 4.1 Fast recursion

This program is also on GitHub, here. I used it to produce Tables 1 and 2.

```
import numpy as np
import random
seed = 1
random.seed(seed) # try 0, 1, 2

def int_to_binstring(x, n):
   # convert integer x into n-bit string
   str = bin(x).replace("0b",'')
   while len(str) < n:
      str = '0' + str
   return(str)

def hash_update(key, hash):
   if key in hash:
      hash[key] += 1
   else:
      hash[key] = 1
   return(hash)

def summary(hash, m, N):
   # summary stats for congruences modulo m
   mean = 0
   std = 0
   for j in range(m):
      # loop on residue classes modulo m
      if (m, j) in hash:
         count = hash[(m, j)]
      else:
         count = 0
      mean += count
      std += count*count
   total = mean
   mean /= m
   std = np.sqrt(std/m - mean*mean)
   # normalize
   # mean /= N-M-1
   # std /= np.sqrt(N-M-1)
   return(mean, std)

M = 20   # test moduli up tp M
N = 10000 # last block visited

hres1 = {} # residues modulo m: counts based on model
hres2 = {} # residues modulo m: counts based on random numbers
fstring = "" # digit sequence based on model
```

```
rstring = "" # digit sequence based on random numbers

for k in range(2, N):

   # fastint = (3**k) % (2**k)
   # fastint = ((3**k + k)) % (2**k)
   fastint = ((5**k) >> k) % (2**k)
   fstring = fstring + int_to_binstring(fastint, k)
   randint = random.getrandbits(k)
   rstring = rstring + int_to_binstring(randint, k)

   for m in range(2,M):

      res1 = fastint % m
      res2 = randint % m
      key1 = (m, res1)
      key2 = (m, res2)
      if k > m:
         hres1 = hash_update(key1, hres1)
         hres2 = hash_update(key2, hres2)

for m in range(2, M):
   for res in range(0,m):
      key = (m, res)
      if key not in hres1:
         hres1[key] = 0
         print(">>> Missing:", key)
      if key not in hres2:
         hres2[key] = 0
      print(key, hres1[key], hres2[key])

print("\nresidue classes: summary")
for m in range(2, M):
   (fmean, fstd) = summary(hres1, m, N)
   (rmean, rstd) = summary(hres2, m, N)
   # fmean = rmean, thus nor showing rmean
   print("%4d %8.2f %8.2f %9.2f" % (m, fmean, fstd, rstd))

print("\nSubstring counts in digit sequence")
substrings = ('0','1','00','01','10','11',
          '000','001','010','011',
          '100','101','110','111',)
for str in substrings:
   print("%4s %10d %10d" % (str, fstring.count(str), rstring.count(str)))

print(fstring[0:1000])
```

## 4.2 Main code

The code is also on GitHub, here. It computes $p_k, q_k$, the prefix $\pi(p_k, q_k)$ and the blocks $B_k$ for the various systems described in section 2. In the code, $B_k$ is named `new_digits` and it is stored as a binary string, see lines 177–170. Since full digit sequences of increasing lengths are computed at each iteration $k$ in order to find more and more digits, you should not use this algorithm if your goal is to simply get a large number of digits, for the mathematical constants in question. In that case, you don't need to identify the various blocks, and you can remove all intermediary block and prefix computations except in the last iteration. This will dramatically increase the speed.

```
1 import gmpy2
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib as mpl
5 from matplotlib import pyplot
6
7 def update_hash(hash, key, count):
```

```
    if key in hash:
        hash[key] += count
    else:
        hash[key] = count
    return(hash)

def p_adic(k, base):
    # return the largest integer v such that base**v divides k
    # https://www.geeksforgeeks.org/python-program-to-find-whether-a-no-is-power-of-two/
    if base == 2:
        div = k & (~(k - 1))
        v = len(bin(div)) - 3
    else:
        div = 1
        v = 0
        while k % div == 0:
            div *= base
            v += 1
        div = div // base
        v -= 1
    return(v, div)

def initialize(mode, z=1, m=0):

    p_buffer = []
    q_buffer = []

    if 'Exponential' in mode:
        # p/q tends to exp(x)-1-1/x as k -> infty
        p_buffer.append(0)
        p_buffer.append(0)
        q_buffer.append(0)
        q_buffer.append(z)

    elif mode == 'Linear':
        # p/q tends to 1/(3-SQRT(3)) as k --> infty
        p_buffer.append(0)
        p_buffer.append(1)
        q_buffer.append(0)
        q_buffer.append(2)

    elif mode == 'ContinuedFractions':
        # p/q tends to (-1+sqrt(5))/2 as k -> infty
        p_buffer.append(0)
        p_buffer.append(13)
        q_buffer.append(0)
        q_buffer.append(21)

    elif mode == 'Special':
        p_buffer.append(0)
        p_buffer.append(0)
        q_buffer.append(1)
        q_buffer.append(2)

    elif mode == 'Special2':
        p_buffer.append(0)
        p_buffer.append(0)
        q_buffer.append(1)
        q_buffer.append(2)

    elif mode == 'Special3':
        p_buffer.append(0)
        p_buffer.append(1)
        q_buffer.append(0)
        q_buffer.append(2)

```

```
    return(p_buffer, q_buffer)


#--- [1] main loop

# parameters
mode = 'Exponential' # options: 'Exponential', 'Exponential1', 'Exponential2'
                     # 'Linear', 'ContinuedFractions', 'Special', 'Special2', 'Special3'
random = False
base = 2  # base must be a prime number
n = 45000 # number of iterations
m = 0     # used in Exponential mode (if m=0, p/q tends to exp(z)-1-1/z)
z = 1     # used in Exponential mode (integer > 0)
seed = 659 # used if random = True
np.random.seed(seed)

# local variables
digits = ""
new_digits = ""
hash1 = {}
hash2 = {}
hash_nu = {}
bprefix = ""
lmax = 0
k_old = 0
zeros = 0
ones = 0

(p_buffer, q_buffer) = initialize(mode, z, m)

for k in range(2, n+1, 1):

    # p = p[k], p_buffer[1] = p[k-1], p_buffer[2] = p[k-2]
    # q = q[k], q_buffer[1] = q[k-1], q_buffer[2] = q[k-2]

    if mode == 'Exponential':
        p = z*k*p_buffer[1] + 1
        q = z*k*q_buffer[1]

    elif mode == 'Exponential1' and m > 0:
        p = z*min(k, m)*p_buffer[1] + 1
        q = z*min(k, m)*q_buffer[1]

    elif mode == 'Exponential2' and m > 0:
        p = z*((k-1)%m + 1) * p_buffer[1] + 1
        q = z*((k-1)%m + 1) * q_buffer[1]

    elif mode == 'Linear':
        p = 2*p_buffer[1] + 2*p_buffer[0] + 1
        q = 2*q_buffer[1] + 2*q_buffer[0]

    elif mode == 'ContinuedFractions':
        p = q_buffer[1]
        q = p_buffer[1] + q_buffer[1]

    elif mode == 'Special':
        if (2*p_buffer[1]+ 1)**2 < 2 * q_buffer[0]**2:
            p = 2*p_buffer[1] + 1
        else:
            p = 2*p_buffer[1]
        q = 2*q_buffer[1]

    elif mode == 'Special2':
        p = 2**k * p_buffer[1] + ((3**k - 1)//2) % (2**k)
        q = 2**k * q_buffer[1]

```

```
    elif mode == 'Special3':
        p = 2**k * p_buffer[1] + (3**k) % (2**k)
        q = 2**k * q_buffer[1]

    p_buffer[0] = p_buffer[1]
    p_buffer[1] = p
    q_buffer[0] = q_buffer[1]
    q_buffer[1] = q
    nu_k, div_k = p_adic(k, base)
    nu_p, div_p = p_adic(p, base)
    nu_q, div_q = p_adic(q, base)
    if p > q:
        print("Warning: p >= q (unauthorized)")
        exit()
    l = max(0, nu_q - nu_p) # length of prefix

    if l > lmax:

        # process additional digits found
        lmax = l
        prefix = (base**l) * p // q
        bprefix_old = bprefix
        l_old = len(bprefix_old)
        bprefix = gmpy2.mpz(prefix).digits(base)
        while len(bprefix) < l:
            bprefix = '0' + bprefix
        if bprefix[0:l_old] != bprefix_old:
            match = 'Fail'
        else:
            match = 'Success'
        new_digits_old = new_digits

        if random:
            new_int = np.random.randint(base**(l-l_old))
        else:
            new_int = prefix % (base**(l - l_old))

        new_digits = gmpy2.mpz(new_int).digits(base)
        while len(new_digits) < l - l_old:
            new_digits = '0' + new_digits

        delta = k - k_old
        zeros += new_digits.count('0')
        ones += new_digits.count('1')
        size = len(new_digits)
        print("===>", k, l, size, delta, nu_k, nu_p, nu_q, match, zeros+ones,
               zeros, ones, ">>", new_digits)

        digits += new_digits
        k_old = k
        key = (size, k)
        if new_digits not in hash1:
            hash_nu[key] = new_digits
        update_hash(hash1, new_digits, 1)
        key = (new_digits, new_digits_old)
        update_hash(hash2, key, 1)


#--- [2] Output results

#- [2.1] block counts

patterns = () # list of all potential combos of t binary digits
t = 5
for k in range(0,2**t,1):
    bint = bin(k)
```

```
    bint = bint[2:len(bint)]
    while len(bint)<t:
       bint = "0"+bint
    patterns = (*patterns, bint)

print("\nblock counts\n")
hash1_print = {}

for key in hash1:
    klen = len(key)
    hash1_print[(klen, key)] = hash1[key]

for keyx in sorted(hash1_print):
    key = keyx[1]
    klen = len(key)
    count = hash1[key]
    if count > 1:
       print(klen, key, count)

#- [2.2] first occurrence of block

print()
print("first occurrence of block")
old_size = 0
count = 0
for key in sorted(hash_nu):
    size = key[0]
    if size == old_size:
       count = count+1
    else:
       print()
       count = 1
       old_size = size
    print(count, hash_nu[key], key)

#- [2.3] conditional block counts

print("\nconditional block counts\n")
hash2_print = {}

for key in hash2:
    klenA = len(key[0])
    klenB = len(key[1])
    hash2_print[(klenA, klenB, key)] = hash2[key]

for keyx in sorted(hash2_print):
    key = keyx[2]
    old = key[0]
    new = key[1]
    count = hash2[key]
    if count > 5:
       print("(old, new):",key, hash2[key])

#- [2.4] high level summary

print()

print("Zeros vs ones:", zeros, ones)
print("Proportion of zeros:", zeros/(zeros + ones))

sum1 = 0
ndigits1 = min(80,len(digits))

for k in range(0,ndigits1,1):
    sum1 += int(digits[k])/base**(k+1)

```

```
sum2 = 0
ndigits2 = min(80,len(bprefix))

for k in range(0,ndigits2,1):
   sum2 += int(bprefix[k])/base**(k+1)

print("dCheck:", sum1)
print("bCheck:", sum2)
print("Number:", p/q)

if 'Exponential' in mode:
   print("Target:", np.exp(1/z)-(1+1/z))
elif mode == 'Linear':
   print("Target:", 1/(3-np.sqrt(3)))
elif mode == 'ContinuedFractions':
   print("Target:", (-1+np.sqrt(5))/2)
elif mode == 'Special':
   print("Target:", np.sqrt(2)/4)

print("Digits per n:", (zeros+ones)/n)
```

## References


[1] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [Link]. 1, 3, 4, 7

[2] Vincent Granville. *State of the Art in GenAI & LLMs – Creative Projects, with Solutions*. MLTechniques.com, 2024. [Link]. 1

[3] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences*. Dover, 2012. [Link]. 6