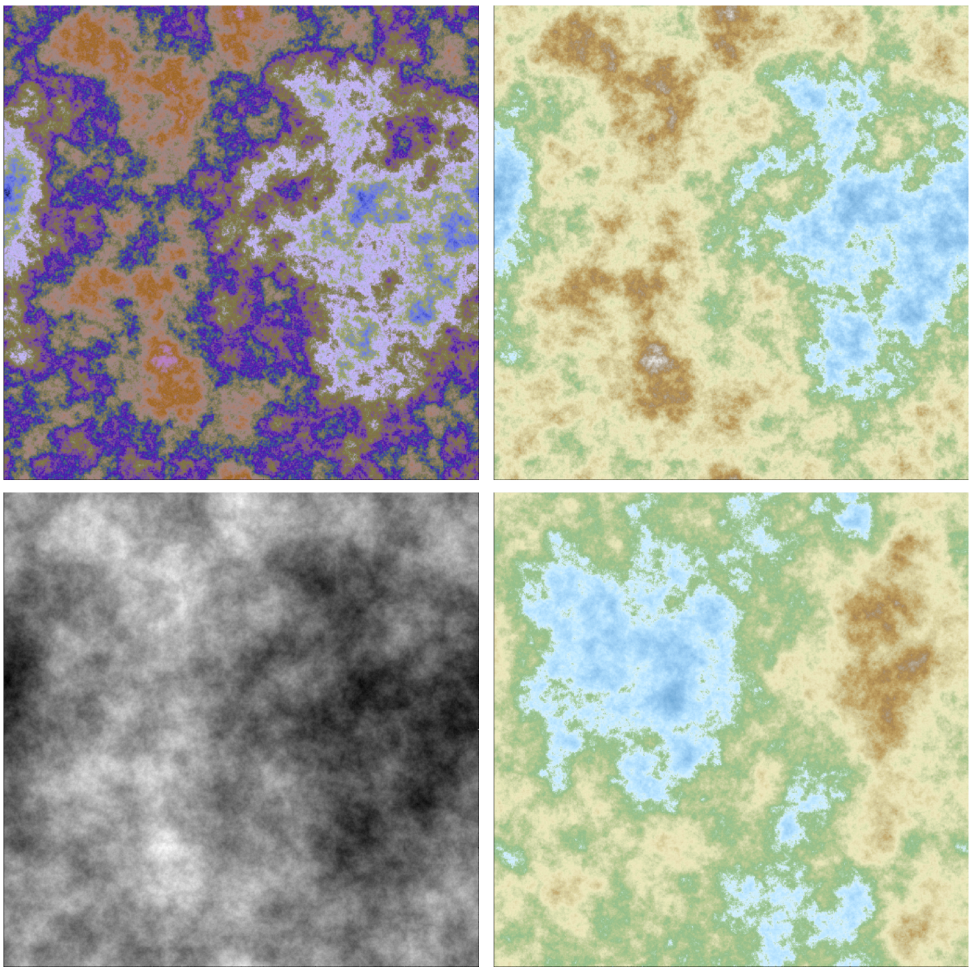
Synthetic Data



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 2.0, December 2022

```
0011111001101001011010100100101110001110110000011110011001110000000111011111001111011011\
0111000101011001110111011111010100011011011001111000111110011100010011011100100001010101\
0100101100110000100110101001101101110010101011000010010111011000101001010001010001001000\
1011111110011110100001111011011001101110101011111110000101000000101000011111110000 1001001\
1111110001000011000111101001011011000110000011100100000010010000001000000001000100101100100\
0110010111100111000110001000000000110010111100011100001110101011100010010111000111011101 0\
1001111011110010110011011011101100101110110000101001011101011101000110101111001011110100\
0001000010010001011000111010101110011111101101100010011011111010001'

size=len(sqrt2)

# Method2:
# 10,000 binary digits of SQRT(2) obtained via formula at https://mltblog.com/3REtOB9
# Implicitly uses the BigNumber Python library (https://pypi.org/project/BigNumber/)

y=2
z=5
for k in range(0,size-1):
   if z<2*y:
      y=4*y-2*z
      z=2*z+3
      digit=1
   else:
      y=4*y
      z=2*z-1
      digit=0
   print(k,digit,sqrt2[k])
```

9.4 Military-grade PRNG Based on Quadratic Irrationals

If you produce simulations or create synthetic data that requires billions or trillions of random numbers (such as in section 1.3.4.1) you need a pseudo-random number generator (PRNG) that is not only fast, but in some cases, truly emulates randomness. Usually you can't have both. Congruential PRNGs are very fast and can be pretty good at emulating randomness. The Mersenne twister available in Python and in other languages has a very large period, more than enough for any practical need. Yet depending on the seed, it has flaws caused by the lack of perfect randomness in the distribution of prime numbers. These were revealed by the prime test in section 9.2.2. To the contrary, PRNGs based on irrational numbers exhibit stronger randomness if you skip the first few digits and carefully choose your numbers. But they tend to be very slow.

In this section I propose a new approach to obtain billions of trillions of digits from combinations of quadratic irrationals [Wiki] such as $\sqrt{2}$ or $\sqrt{7583}$, based on the algorithm in section 9.3.3. The goal is to produce replicable random numbers. If you want to use them for strong encryption, you need to use a seed that is hardware-generated so that the same seed is never used more than once.

9.4.1 Fast algorithm rooted in advanced analytic number theory

The idea to get a fast algorithm is simple. Instead of producing n digits from a single number, I generate r digits from m different numbers, with $n = rm$. While the Python code relies only on basic additions and very few operations, the computation of n binary digits of a single irrational number involves very large integers. The computational complexity is $O(n^2)$. If instead you generate r digits from m numbers, the computational complexity drops to $O(rm^2)$. In the most extreme and very interesting case where $r = n$ and $m = 1$, the computational complexity is $O(n)$, just as fast as the Mersenne twister. The method is based on two deep results in number theory:

- The binary digits of a quadratic irrational behave as an infinite realization of independent Bernoulli trials with equal proportions of 0 and 1. This unproven conjecture is one of the most difficult unsolved problems in mathematics. Very strong empirical results involving trillions of digits, support this hypothesis.
- Two sequences of digits from irrational numbers that are linearly independent over the set $\mathbb{Q}$ of rational numbers, have zero cross-correlation. The correlation is defined as the empirical correlation in this case.

The latter is a consequence of the following theorem: the correlation $\rho(p, q)$ between the sequences $(\{pb^k\alpha\})$ and $(\{qb^k\alpha\})$ indexed by $k = 0, 1$ and so on, where p, q are positive integers with no common divisors, $b > 1$ is an integer, and α is a positive irrational, is equal to $\rho(p, q) = (pq)^{-1}$. Here $\{\cdot\}$ denotes the fractional part function.

A proof (by William Huber) of this unpublished theorem can be found here, with additional discussion on this topic, here. Note that $\lfloor b \cdot \{b^k\alpha\}\rfloor$ is the k-th digit of α in base b. The brackets represent the integer part function. Thus, if $\alpha_1 = p\alpha$ and $\alpha_2 = q\alpha$, the correlation between the sequences $(\{b^k\alpha_1\})$ and $(\{b^k\alpha_2\})$ is $\rho(p,q) = (pq)^{-1}$. If α_1, α_2 are irrational and linearly independent over $\mathbb{Q}$, then the only way you can write $\alpha_1 = p\alpha, \alpha_2 = q\alpha$ is by letting p, q tends to infinity, thus the correlation vanishes. It implies that the correlation between the digit sequences of α_1 and α_2 is zero.

There is more number theory involved. In particular, the method uses the new algorithm described in section 9.3.3 to compute the binary digits of quadratic irrationals. It is also connected to square-free integers, and approximations of irrational by rational numbers which is linked to continued fractions. Square-free integers [Wiki] are also discussed in section 9.2.1 and 15.2.2.2. They represent 61% of all positive integers: the exact proportion is $6/\pi^2$.

9.4.2 Fast PRNG: explanations

Each positive integer c can be written as $c = ab$, where a is a square, and b is square-free. For instance, if $c = 3^5 \times 13^6 \times 19$, then $a = 3^4 \times 13^6$ and $b = 3 \times 19$. If c is a square, then $c = a$ and $b = 1$.

The quadratic irrationals used here are characterized by a bivariate seed (y_0, z_0). Given a seed, the successive iterations produce the binary digits of the number

$$x_0 = \frac{-(z_0 - 1) + \sqrt{(z_0-1)^2 + 8y_0}}{4}. \tag{9.7}$$

The connection to square-free integers is as follows: $c = (z_0 - 1)^2 + 8y_0$ can not be a square. I use the notation $c = ab$ where a is the square part of c, and b is the square-free part. Thus, we must have $b > 1$. I use a large number of seeds to generate a large number of quadratic irrationals. The cross-correlation between the two digit sequences in any pair of quadratic irrationals must be zero. Based on the theory in section 9.4.1, it means that once a seed produces a specific b, any future seed with the same b must be rejected. This is accomplished using the variable `accepted` in the code, along with the hash table (dictionary in Python) `squareFreeList`. The key for this hash table is actually b.

The number of digits produced for each quadratic irrational is specified by the parameter `size`. The total number of quadratic irrationals is determined by `Niter`. Not all of them are used: a few are rejected for the reason just mentioned. All the binary digits of all the accepted quadratic irrationals are stored in the hash table `digits`. For instance, `digits[(b,k)]` is the k-th digit of the quadratic irrational with square-free part b. The seed (y_0, z_0) corresponding to this number is `squareFreeList[b]`.

Due to the particular choice of seeds in the Python code (with $y_0 = 1$), many quadratic irrationals are close to zero. More specifically, Formula (9.7) yields the following approximation: $x_0 \approx y_0/(z_0 - 1)$. So the first few digits are biased and should be skipped. This is accomplished via the parameter `offset`. There are other problematic quadratic irrationals such as those with $z_0 - 1$ being a power of 2. A future version of this algorithm can reject these quadratic irrationals, and also reject those that are too close to a number already in the hash table. However, increasing the parameter `offset` is the easiest option to eliminate these problems. The next step is to run a standard battery of tests such as the Diehard tests, and check whether this PRNG passes all of them depending on the parameters and configuration.

I haven't tested yet the algorithm with `size=1`: this is the fastest way to generate many digits (assuming `Niter` is increased accordingly), and possibly the best choice assuming `offset` is large enough. In particular, if you extract just one digit of each quadratic irrational, there is a faster way to do it, see section 9.4.4.

9.4.3 Python code

Now that I explained all the details about the algorithm, here is the Python code. It is also available on GitHub, here, under the name `stronprng.py`.

```python
# By Vincent Granville, www.MLTechniques.com

import time
import random
import numpy as np

size = 400          # number of binary digits in each number
Niter = 5000        # number of quadratic irrationals
```

```
start = 0          # first value of (y, z) is (1, start)
yseed = 1          # y = yseed
offset = 100       #   skip first offset digits (all zeroes) of each number
PRNG = 'Quadratic' # options: 'Quadratic' or 'Mersenne'
output = True      # True to print results (slow)

squareFreeList = {}
digits = {}
accepted = 0 # number of accepted seeds

for iter in range(start, Niter):

   y = yseed # you could use a non-fixed y instead, depending on iter
   z = iter
   c = (z - 1)**2 + 8*y

   # represent c as a * b where a is square and b is square-free
   d = int(np.sqrt(c))
   a = 1
   for h in range(2, d+1):
      if c % (h*h) == 0: # c divisible by squared h
         a = h*h
   b = c // a # integer division

   if b > 1 and b not in squareFreeList:
      q = (-(z - 1) + np.sqrt(c)) / 4 # number associated to seed (y, z); ~ y/(z-1)
      squareFreeList[b]=(y,z)  # accept the seed (y, z)
      accepted += 1

start = time.time()

for b in squareFreeList:

   y = squareFreeList[b][0]
   z = squareFreeList[b][1]

   for k in range(size):

      # trick to make computations faster
      y2 = y + y
      y4 = y2 + y2
      z2 = z + z

      # actual computations
      if z < y2:
         y = y4 - z2
         z = z2 + 3
         digit = 1
      else:
         y = y4
         z = z2 - 1
         digit = 0
      if k >= offset:
         digits[(b,k)] = digit

end = time.time()
print("Time elapsed:",end-start)

if output == True:
   OUT=open("strong4.txt","w")
   separator="\t" # could be "\t" or "" or "," or " "
   if PRNG == 'Mersenne':
      random.seed(205)
   for b in squareFreeList:
      OUT.write("["+str(b)+"]")
      for k in range(offset, size):
         key = (b, k)
         if PRNG == 'Quadratic':
            bit = digits[key]
         elif PRNG == 'Mersenne':
            bit = int(2*random.random())
         OUT.write(separator+str(bit))
      OUT.write("\n")
   OUT.close()

print("Accepted seeds:",accepted," out of",Niter)
```

9.4.4 Computing a digit without generating the previous ones

If `offset` is large, you spend time computing digits (at the beginning of the binary expansion of each quadratic irrational) that you will use only to get the subsequent digits. There is a more efficient approach: use a different seed (y_0', z_0'). The new seed has the benefit of keeping the square-free part b unchanged. To get the digit in position k in one shot (assuming the first position corresponds to $k = 0$), use the seed $y_0' = 2^{2k}y_0, z_0' = 2^k(z_0 - 1) + 1$. Then $x_0' = 2^k x_0$, so your quadratic irrational is multiplied by 2^k. This is a direct consequence of Formula (9.7). By digits, I mean the binary digits on the right starting after the decimal point.

Finally, the position k that you select for the digit may be randomized: it can depend on the quadratic irrational. This makes it very difficult to reverse-engineer your sequence of random digits.

9.4.5 Security and comparison with other PRNGs

The spreadsheet `strongprng.xlsx` (on GitHub) compares the quadratic irrational PRNG with the Mersenne twister. The test involves 5000 quadratic irrationals, one per row. Based on the acceptation rule, only 4971 were used. For each of them, I computed 400 binary digits and skipped the first 100 hundreds as suggested in the methodology. So in total, there are 4971 blocks, each with 300 digits. The tab corresponding to the Mersenne twister has the same block structure with the same numbers of digits: it was produced using the option `PRNG='Mersenne'` in the Python code. The numbers in brackets in column D represent the block ID: the number b in the case of the quadratic irrational PRNG, and nothing in particular in the case of the Mersenne twister.

I computed some summary statistics: digit expectation and variance for each column and for each row, as well as cross-correlations between rows, and also between columns. The results are as expected and markedly similar when comparing the two PRNG's. You would expect any decent PRNG to pass these basic tests, so this is not a surprise. You need more sophisticated tests to detect hard-to-find departure from randomness; that was the purpose of section 9.2.2 with the prime test. Figure 9.4 shows 5000 correlations values. They are not statistically different from zero, and independent despite the fact that they correspond to seeds produced in chronological order. Indeed, their joint distribution is identical to that produced with the Mersenne twister (not shown in the picture, but included in the spreadsheet).

Whether using the Mersenne twister or quadratic irrationals, the compression factor – based on the zip tool applied to the 1.5 million raw digits – is about the same and equal to almost 8. This is the worst achievable compression factor: each character (8 bits) representing a digit is turned into 1 bit of information. It means that the zip tool is unable to detect any pattern in the digits that would allow for any amount of real compression.

9.4.5.1 Important comments

There are very few serious articles in the literature dealing with digits of irrational numbers to build PRNG's. It seems that this idea was abandoned long ago due to the computational complexity and the erroneous belief that it defeats the non-deterministic nature of randomness. It is my hope that my quadratic irrational PRNG debunks all these myths. Note that there has been attempts to use chaotic dynamical systems to build PRNG's. See for instance [105]. My upcoming book on dynamical systems explores many new related methods in great details.

Also, my PRNG's, by combining thousands to billions of quadratic irrationals, present some similarity to combined linear congruential generators [Wiki]. It avoids the drawbacks that these generators are facing. In the context of congruential generators, combining is used to increase the period and randomness. In the case of quadratic irrational PRNG's, the period is always infinite, and the goal is to increase security, but not randomness which is already maximum with just one number. Also using many quadratic irrationals each with a few digits runs a lot faster than one number with many digits. In addition, using many quadratic irrationals leads to a very simple implementation using a distributed architecture.

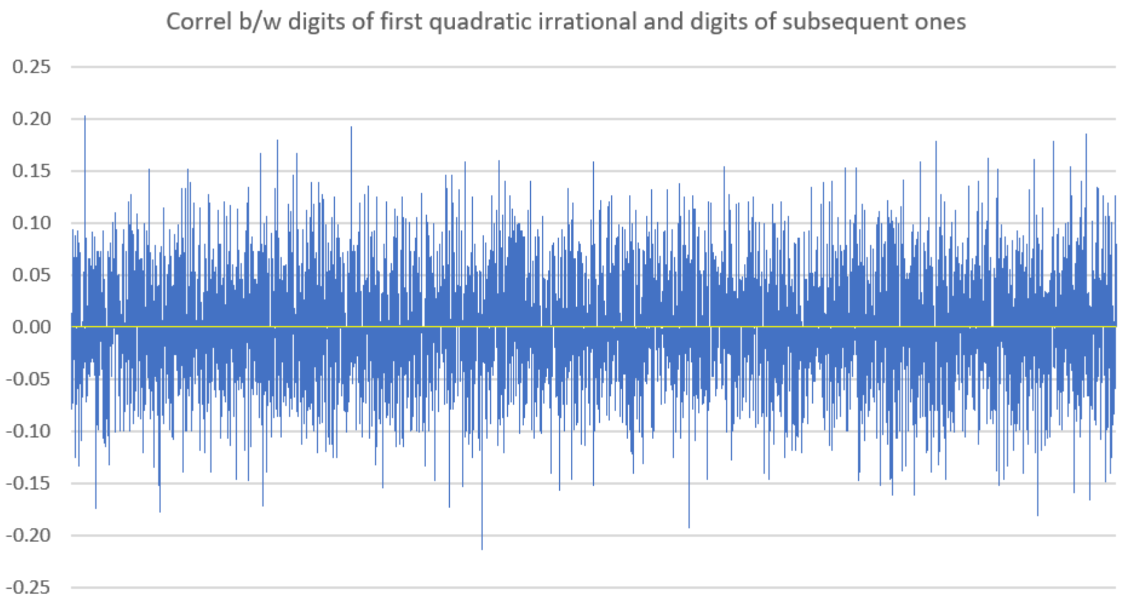


Figure 9.4: Correlations are computed on sequences consisting of 300 binary digits

Finally, while it sounds like a cave-man idea to publish a table of random digits, it servers a very important purpose that has been forgotten in modern scientific research and benchmarking: replicability. You are free to reuse my Excel spreadsheet if you want your research to be replicable. Of course you would get the same digits if you use the Python code with the same seeds. This is not true with the Mersenne twister: the digits may depend on which version of Python you use. Also, one advantage of the quadratic irrational PRNG is its portability, and the fact that you will get the same digits regardless of your programming language. Time permitting, I will publish a much larger table with at least a trillion digits. In the meanwhile, if you need such a table, feel free to email me at vincentg@MLTechniques.com.

9.4.6 Curious application: a new type of lottery

The following application requires a very large number of pseudo-random numbers generated in real-time, as fast as possible. The digits must emulate randomness extremely well. It would benefit from using the quadratic irrational PRNG. The idea consists of creating a virtual, market-neutral stock market where people buy and sell stocks with tokens. In short, a synthetic stock market where you play with synthetic money (tokens). Another description is a lottery or number guessing game. You pay a fee per transaction (with real money), and each time you make a correct guess, you are paid a specific amount, also in real money. The participant can select different strategies ranging from conservative and limited to small gains and low volatility, to aggressive with a very small chance to win a lot of money.

The algorithm that computes the winning numbers is public; it requires some data input, also publicly published (the equivalent of a public key in cryptography). So you can use it to compute the next winning number and be certain to win each time, which would very quickly result in bankruptcy for the operator. However the public algorithm necessitates billions of years of computing time to obtain any winning number with certainty. But you can guess the winning number: your odds of winning by pure chance (in a particular example) is 1/256.

The operator uses a private algorithm that very efficiently computes the next winning number. From the public algorithm, it is impossible to tell – even if you are the greatest computer scientist or mathematician in the world – that there is an alternative that could make the computations a lot faster: the private algorithm is the equivalent of a private key in cryptography. The public algorithm takes as much time as breaking an encryption key (comparable to factoring a product of two very large primes), while the private version is equivalent to decoding a message if you have the private key (comparable to finding the second factor in the number in question if you know one of the two factors – the private key).

Needless to say, any very slight deviation resulting from flaws in the PRNG will quickly lead to either the bankruptcy of the operator, or the operator enriching itself and being accused of lying about the neutrality of the simulated market. I made a presentation on this topic at the Operations Research Society conference (INFORMS) in 2019, in a session exploring biases in algorithms. See the abstract here. The full paper will be included in my upcoming book “Experimental Math and Probabilistic Number Theory”.