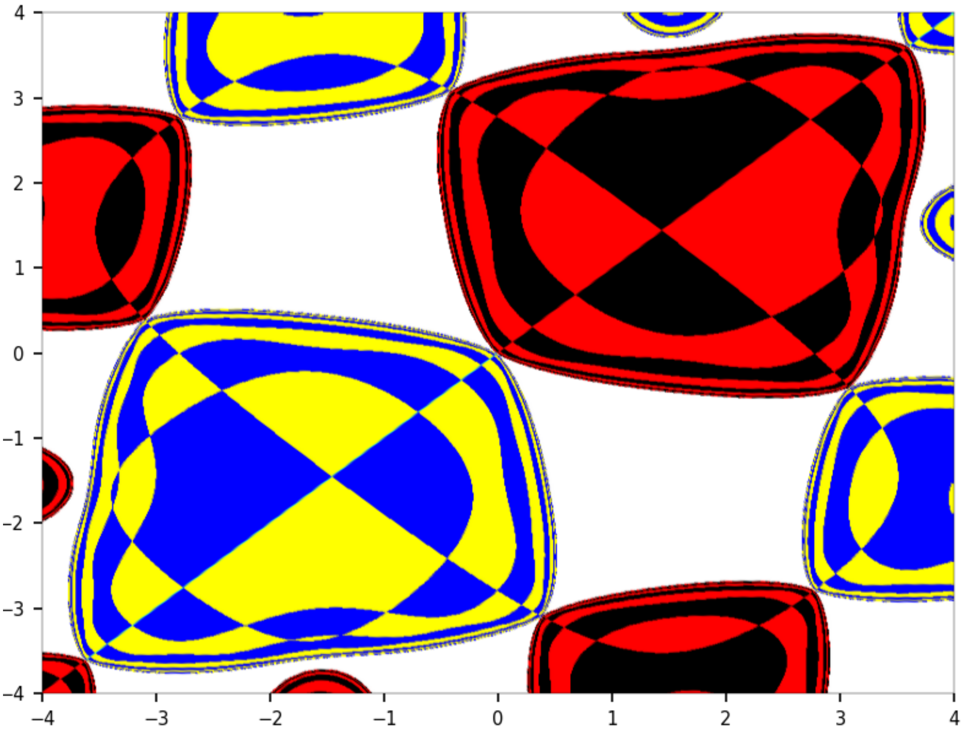
Gentle Introduction To Chaotic Dynamical Systems



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 2.0, May 2023

Chapter 5

Application: Synthetic Stock Exchange

In this chapter, I describe an original number guessing game played with real money. It can also be used as a synthetic stock exchange, leveraging the mathematics discussed in chapter 4. In particular, it is based on the dyatic map and efficient algorithms to compute the digits attached to good seeds in the dynamical systems involved.

If properly implemented, this system cannot be manipulated: formulas to win numbers are made public. Also, it simulates a neutral, efficient stock market. In short, for the participants, there is nothing random: everything is deterministic and fixed in advance, and known to all users. Yet it behaves in a way that looks perfectly random, and public algorithms offered to compute winning numbers require so much computing power, that for all purposes, they are useless – except to comply with gaming laws and to establish trustworthiness. Any attempt at predicting winning numbers based on historical data, using machine learning techniques, will fail: the sequence of winning numbers appear to be totally random, despite the fact that it is deterministic.

I use private algorithms to determine the winning numbers, and while they produce the exact same results as the public algorithms, they are incredibly more efficient, by many orders of magnitude. Also, I mathematically prove that the public and private algorithms produce the same results. To prove it on a real example may not possible: the public algorithm requires too much computing time.

The goal is to turn this application into a real platform where participants play with real money. I also discuss how it can be done given the legal barriers. The operator generates revenue via charging a fee per transaction, just like stock exchanges, and unlike lotteries that retain a portion of the payments to finance themselves. In short, all the money used by participants to purchase numbers, goes back to the participants.

5.1 Introduction

The application discussed here requires a very large number of pseudo-random digits generated in real-time, obtained as fast as possible. The digits must emulate randomness extremely well. I use the quadratic irrational random number generator discussed in section 4.4.

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, themselves purchased with real money. Another description is a lottery or number guessing game. You pay a fee per transaction, 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 input information, 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 so much computing time to obtain any winning number with certainty, to make it useless. 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 numbers. 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).

Thus, technically, this application is not a game of chance but a mathematical competition. But one that no one can win other than by luck, due to tremendous computing power required to find the winning number with certainty. The digits used in the system must be uniformly distributed. Even a tiny bias 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.

5.2 Winning, sequences and customized ROI tables

Rather than trading stocks or other financial instruments, participants (the users) purchase numbers. Sequences of winning numbers are generated all the time, and if you can predict the next winning number in a given sequence, your return is maximum. If your prediction is not too far from a winning number, you still make money, but not as much. The system has the following features:

- The algorithms to find the winning numbers are public and regularly updated. Winning is not a question of chance, since all future winning numbers are known in advance and can be computed using the public algorithm.
- The public algorithm, though very simple in appearance, is not easy to implement efficiently. In fact, it is hard enough that mathematicians or computer scientists do not have advantages over the layman, to find winning numbers.
- To each public algorithm, corresponds a private version that runs much, much faster. I use the private version to compute the winning numbers, but both versions produce the exact same numbers.
- Reverse-engineering the system to discover any of the private algorithms, is as difficult as breaking strong encryption.
- The exact return is known in advance and specified in public ROI tables. It is based on how close you are to a winning number, no matter what that winning number is. Thus, your gains or losses are not influenced by the transactions of other participants.
- The system is not rigged and cannot be manipulated, since winning numbers are known in advance.
- The system is fair: it simulates a perfectly neutral stock market.
- Participants can cancel a transaction at any time, even 5 minutes before the winning number is announced.
- Trading on margin is allowed, depending on model parameters.
- The money played by the participants is not used to fund the company or pay employees or executives. It goes back, in its entirety, to the participants. Participants pay a fee to participate.

Comprehensive tables of previous winning numbers are published, well before a new sequence – based on these past numbers – is offered to players. It entices participants to design or improve strategies to find winning numbers, even though arbitraging is technically not possible, unless there is an unknown flaw in my method. Actually, past winning numbers are part of the public data that is needed to compute the next winning numbers, both for participants and the platform operators.

Various ROI tables are offered to participants, and you can even design your own. If you are conservative, you can choose one with a return of 10% for a perfect guess, a 54% chance of winning on any submitted number, and a maximum potential loss of 4%. This table is safe enough that you could “trade” on margin. It mimics a neutral stock market. The return is what you make or lose in one day percentagewise, on one sequence. New winning numbers are issued every day for each live sequence, so your return – negative or positive – gets compounded if you play frequently. A sequence is the equivalent of a specific stock in the stock market.

Another interesting ROI table offers a return of 330% for a perfect guess, with the same 54% chance of winning on any transaction, and a maximum potential loss also of 4%. However, the payoff when your guess is close to but different from the winning number, is a lot lower than in the previous example. If you are a risk taker, you may like a table offering a maximum return of 500%, a 68% chance of winning on any transaction, and a maximum potential loss of 60%. Or another table with a maximum return of 600%, a 80% chance of winning, but a maximum potential loss of 100%. I discuss ROI tables and sequences in details later in this chapter, along with examples. By winning, I mean that your guess is close enough to the winning number so that you get financially rewarded. The reward in question might be small depending on the ROI table, compared to correctly guessing the winning number.

All the sequences currently implemented consist of 8-bit numbers: each winning number – a new one per day per sequence – is an integer between 0 and 255. I am working on a generalization to 16-bit numbers, offering payoffs of the same caliber as lottery jackpots. By design, all ROI tables including customized ones, are neutral, with an average zero gain. This beats all casinos and lotteries where the average gain is negative. It applies to all sequences. Indeed, sequences and ROI tables are independent. The participant can test various strategies: for instance:

- Try various ROI tables.
- Play every day until you experience your first win.
- Play every day until you experience your first loss.
- Play until you have achieved a prespecified goal, or exit if your losses reaches some threshold, whatever comes first.
- Increase or decrease how much you spend depending on your results.
- Look if you can find patterns in the winning numbers, then exploit them.

Any pattern in the winning numbers is likely short-lived and coincidental, in the same way that any purely random time series exhibits a number of patterns, since the number of potential patterns is infinite.

5.3 Implementation details with example

Here I show an example of a typical sequence. It illustrates how the winning numbers are computed for the sequence in question. The purpose is to explain the mechanics for one of the 8-bit systems. The 32-bit version offers more flexibility, as well as potential returns that can beat those of a state lottery jackpot. The sample 8-bit sequence is defined by the public algorithm below.

5.3.1 Seeds, public and private algorithms

I now describe the public algorithm. It works as follows. Start with initial values y_0 and z_0 that are positive integers called seeds, with $z_0 > y_0$. Then for $t = 0, 1, 2$ and so on, compute y_{t+1} and z_{t+1} iteratively as follows:

```
If $z_t < 2y_t$ then
    $y_{t+1} = 4y_t - 2z_t$
    $z_{t+1} = 2z_t + 3$
else
    $y_{t+1} = 4y_t$
    $z_{t+1} = 2z_t - 1$.
```

I discuss the seeds in section 5.3.4. In one version of the system, the seeds are public but are extremely large integers with millions of even billions of digits. In another version, I guarantee the existence of working seeds y_0, z_0 smaller than 10^3, but I keep them secret. Thus the participant would have to test up to $10^3 \times 10^3 = 10^6$ pairs of seeds to find the winning numbers. And that's just the easy part of the problem.

In this example, the public algorithm computes big numbers linked to the successive binary digits of some quadratic irrational denoted as x_0, without actually computing the digits. It is part of a family of algorithms described in section 4.3.3. The quadratic irrational x_0 depends on the seeds y_0, z_0. The player is unaware of this fact, and certainly does not know which x_0 is being used. It also guarantees that the digits – and thus the winning numbers – are uniformly distributed and uncorrelated. In short, they constitute a perfect random sequence.

The operator, aware of these facts, can pre-compute billions of binary digits of x_0 using a very fast method, or better, obtain these digits from an external source such as Sagemath, and store them in some lookup table. This constitutes the private algorithm. In Sagemath, the command to get the first 10^4 digits of (say) $\sqrt{2}$ is `N(sqrt(2),prec=10000).str(base=2)`. It takes less than one second to execute. It requires more and more time the more digits you want. The runtime is proportional to the square of the number of digits needed.

Some methods do not need to compute previous digits to get those starting at a given location: see section 5.3.3. This can significantly increase performance. It allows the operator to quickly start at iteration (say) $t = 10^8$ when creating a new set of winning numbers, skipping the previous iterations and saving a significant amount of computing time. To the contrary, the participant, having no clue as to when the winning numbers start, must go through all the iterations when using the public algorithm.

5.3.2 Winning numbers and public data

The iterations in the public algorithm represent the time. All the winning numbers for a particular sequence are successive 8-bit blocks starting at a specific machine-generated iteration T in the public algorithm. No one knows the starting time T, not even the platform operator nor its software engineers. Typically, $T > 10^8$ and can automatically be specified by the system, potentially using a random value such as the actual time (in milliseconds) when it was created by the algorithm in production mode. A different T is used for each sequence, and T can be periodically reset to increase the security of the system.

In the 8-bit system, winning numbers are always integers between 0 and 255. In the public algorithm, they occur only at iterations $t = T, T+8, T+16, T+24$ and so on. The winning number at iteration $t \geq T$ is defined as $w_t = (z_t - 256 \cdot z_{t-8} + 255)/4$. It is a positive integer. It consists of 8 successive digits in the binary expansion of the underlying quadratic irrational x_0 used in the private algorithm. The reason for skipping 7 out of 8 numbers is to make sure that winning numbers are not auto-correlated.

5.3.2.1 Using the public algorithm to win

Before any winning sequence is made available to the public and participants are allowed to play, the operator publishes the previous 2000 winning numbers attached to the sequence in question. Using the public algorithm, the participants can identify when these 2000 numbers appear in the sequence $\{w_t\}$, though they must first find the right seeds among several candidates publicly listed. Once the past winning numbers are found with the public algorithm and correct seeds, they player knows that the next number is a winning number: he can purchase that number and win with 100% certainty, thus getting the maximum payout.

In case multiple seeds (y_0, z_0) lead to the same 2000 winning numbers at some point within the prespecified number of iterations (say, one trillion), the operator must still pay the full prize to the participant, regardless of which number the participant purchased and which seed he used. As long as the participant can show how he found the 2000 numbers in question, by sharing the seeds that he used. So the operator should make sure that the probability of two different seeds yielding 2000 identical winning numbers in the right order, in less than a trillion iterations, is essentially zero, see section 5.3.5.

Because winning numbers have a random behavior, the chance of such a "collision" is essentially zero anyway. The chance of finding the 2000 winning numbers using the exact same seed as the operator (after testing a large number of them using the public algorithm on potential seeds listed by the operator), is considerably higher by many orders of magnitude. It is essentially 100%. However the limitation here is of a different nature: it requires years of computing time, and sequences (that is, seeds) and even algorithms are updated frequently to essentially make it impossible to win other than by luck. Given that a winning number has only 8 bits, the chance of winning by luck is 1/256. And even if your guess is close to the winning number, you still get a payout, albeit smaller.

5.3.2.2 Python code

The code below is also available on GitHub, here. It covers more than the public algorithm. In particular, it computes the successive digits $d(t)$ of the quadratic irrational x_0 determined by the seeds y_0, z_0, for $t = 1, 2$ and so on. Using results obtained in section 5.3.3, the implementation can be accelerated by several orders of magnitude when T is within some known range, for instance if $\tau < T < \tau + 10^4$ and $T > 10^9$. The operator typically knows τ but not T, while the player knows none of them.

```python
# w is an 8-bit winning number if t >= T and (t - T) % 8 == 0

t = 0
y = 2      # seed: 1st component, y0 (for t = 0)
z = 5      # seed: 2nd component, z0 (for t = 0)
T = 43     # must be >= 10
max = 200 # maximum for t, must be >= T
buffer = {} # to store 9 previous vales of z
x0 = 0      # irrational number represented by the digits

for t in range(1, max):

   if z < 2*y:
      y = 4*y - 2*z
      z = 2*z + 3
      d = 1     # t-th binary digit of x0
   else:
      y = 4*y
      z = 2*z - 1
      d = 0     # t-th binary digit of x0

   x0 += d * 1/2**t
   if t >= T - 8:
    buffer[t % 9] = z
    if t >= T:
        w = (z - 256*buffer[(t-8) % 9] + 255) >> 2

   if t >= T and (t - T) % 8 == 0:
      print(t, w, d, z)

print("\nNumber x0:", x0)
```

At iteration t, the array `buffer`, once updated, contains the preceding 8 values $z_{t-1}, z_{t-2}, \dots, z_{t-8}$ as well as z_t, in circular order. Actually, rather than keeping these huge values in a buffer, it is sufficient to just keep the corresponding digits instead, and compute w_t with the convolution formula

$$w_t = \sum_{k=0}^{7} 2^k d(t-k).$$

Note that $d(t) = (z_t - 2z_{t-1} + 1)/4$ is equal to either 0 or 1. It is represented by the variable `d` in the code. In the end, the whole system is justified based on the following theorem.

Theorem 5.3.1 *If x_0 is an irrational number, then the corresponding sequence $w_t, w_{t+8}, w_{t+16}, \dots$ is free of auto-correlations regardless of $t > 0$, and these numbers are uniformly distributed on $\{0, 1, \dots, 255\}$. Assuming $d(t) = (z_t - 2z_{t-1} + 1)/4$ and $z_0 > y_0$, we also have*

$$x_0 = \sum_{t=1}^{\infty} \frac{d(t)}{2^t} = \frac{-(z_0 - 1) + \sqrt{(z_0-1)^2 + 8y_0}}{4}, \tag{5.1}$$

$$w_t = \sum_{k=0}^{7} 2^k d(t-k) = \frac{z_t - 256 \cdot z_{t-8} + 255}{4} \tag{5.2}$$

Now let's consider two sequences: one with seeds y_0, z_0 and quadratic irrational x_0, and another one with seeds y_0', z_0' and quadratic irrational x_0'. If x_0, x_0' are irrational numbers linearly independent over the set of rational numbers, then the two sequences $w_t, w_{t+8}, w_{t+16}, \dots$ and $w_t', w_{t+8}', w_{t+16}', \dots$ are free of cross-correlations.

Proof
Formula (5.1) is proved here. To prove (5.2), recursively use the fact that $d(t) = (z_t - 2z_{t-1} + 1)/4$. That is:

$$\begin{aligned}
d(t) + 2d(t-1) &= (z_t - 4z_{t-2} + 3)/4,\\
d(t) + 2d(t-1) + 4d(t-2) &= (z_t - 8z_{t-3} + 7)/4,\\
d(t) + 2d(t-1) + 4d(t-2) + 8d(t-3) &= (z_t - 16z_{t-4} + 15)/4,\\
d(t) + 2d(t-1) + 4d(t-2) + 8d(t-3) + 16d(t-4) &= (z_t - 32z_{t-5} + 31)/4,
\end{aligned}$$

and so on. From this, it is clear that the sequence $w_t, w_{t+8}, w_{t+16}, \dots$ consists of successive blocks of 8 bits in the binary digit representation of the number x_0. Thus assuming these digits are a realization of independent Bernoulli trials with same chance of fail/win (thus assuming x_0 is a good seed of the dyadic map), then the winning numbers are independent with a uniform distribution on $\{0, \dots, 255\}$. The context here is about the empirical (observed) joint distribution of the first n digits, its limit when $n \to \infty$, and the conjecture that x_0 is a strongly normal number. The concept of strong normality is defined in section 4.2.1.1. See also section 3.1.2 where correlation is defined. Also note that $0 < x_0 < 1$ since $y_0 < z_0$. The last statement about the absence of cross-correlations is true if both x_0 and x_0' are strongly normal. See here, and Exercise 22. ■

Table 5.1 illustrates some of the quantities discussed here. It corresponds to the output of the Python code in section 5.3.2.2, assuming the winning numbers w_t start at iteration $t = T = 43$. In this case, $x_0 = -1 + \sqrt{2}$. The number z_t, needed to compute w_t, grows by a factor 2 at each iteration. In practice, T is above 10^9.

5.3.3 Optimizing computations

Regardless of $k \geq 0$, the change of seeds $y_0' = 2^{2k} y_0, z_0' = 2^k(z_0 - 1) + 1$ leads to the new quadratic irrational $x_0' = 2^k x_0$, according to formula (5.1). Thus, in theory, it allows you to get the digits of x_0 starting at location $k+1$ without computing the first k digits. And consequently, the corresponding winning numbers – no matter how far in the sequence – with little computational efforts. However, this is true as long as $y_0' < z_0'$. In practice, when k is large, this is never the case.

t	w_t	$d(t)$	z_t
43	157	1	49758216191605
51	230	0	12738103345051545
59	72	0	3260954456333195553
67	69	1	834804340821298061589
75	151	1	213709911250252303767133
83	216	0	54709737280064589764386657
91	155	1	14005692743696534979682984557
99	55	1	3585457342386312954798844046557
107	84	0	917877079650896116428504075918673
115	171	1	234976532390629405805697043435180717
123	233	1	60153992292001127886258443119406264229
131	241	1	15399422026752288738882161438568003643333
139	214	0	3942252038848585917153833328273408932693849
147	246	0	1009216521945237994791381332037992686769626073
155	11	1	258359429617980926666593621001726127813024274477
163	168	0	66140013982203117226647966976441888720134214266529
171	147	1	16931843579443998010021879545969123512354358852231757
179	186	0	4334551956337663490565601163768095619162715866171330281
187	132	0	1109645300822441853584793897924632478505655261739860552209
195	206	0	284069197010545114517707237868705914497447747005404301366073

Table 5.1: Winning number w_t, digit $d(t)$, and z_t at iteration t

A workaround consists in finding seeds y_0', z_0' leading to $x_0' = 2^k x_0 - \lfloor 2^k x_0 \rfloor$ where the brackets represent the integer part function. It involves computing integer square roots [Wiki], that is, the integer part of the square root of very large integers. There are very efficient methods to accomplish this, especially to get the binary digits. The mpmath and gmpy2 Python libraries offer specific functions for these computations. Indeed you can even use the isqrt function from the math library. Here I illustrate how to do it with the isqrt function from the gmpy2 library: it is implemented in C and possibly the fastest of all.

The starting point is to reverse the problem. Given two large positive integers α, β, you want to find the seeds y_0, z_0 for the quadratic irrational $x_0 = -\alpha + \sqrt{\beta}$. The choice of α, β must result in $0 < x_0 < 1$ for the private algorithm to work. By virtue of (5.1), this leads

$$\alpha = \lfloor \sqrt{\beta} \rfloor, \quad y_0 = 2(\beta - \alpha^2), \quad z_0 = 1 + 4\alpha. \tag{5.3}$$

To skip the first k digits, you need to find seeds y_0', z_0' such that $x_0' = -\alpha' + \sqrt{\beta'}$ with $\beta' = 2^{2k}\beta$ and $\alpha' = \lfloor \sqrt{\beta'} \rfloor$. This leads to the same solution as (5.3), but this time with y_0, z_0, α, β replaced respectively by $y_0', z_0', \alpha', \beta'$. The main challenge is the computation of α', that is, the integer square root of $2^{2k}\beta$.

I provide the code below if you want to use the gmpy2 library. Here square represents β'. For $k = 10^9$ and $\beta = 3$, it took about one minute on my laptop, excluding the time needed to print the result. The integer square root isqrt is computed in base 16, which is very handy since winning numbers are 8-bit long, thus easy to encode in base 16: each winning number consists of two consecutive digits in base 16. It would be even better to use base=256 in the code snippet. The maximum base allowed when I tested my script, was 60.

For much larger values of k, the code needs to be modified, as you will run out of memory and require a parallel architecture at some point. The time required seems to be a sublinear function of k until you use up all the memory. I suggest running it in GPU.

```
import gmpy2

beta = 3
base = 16
k = 10**9
square = beta * (2**(2*k))
isqrt = gmpy2.isqrt(square).digits(base)
print(isqrt)
```

5.3.4 Seeds with billions of digits and enhanced system

The methodology also works when the seeds are not integer numbers: Formula (5.1) and (5.2) remain valid even if y_0, z_0 are irrational numbers, as long as $0 < y_0 < z_0$. In particular, the winning numbers w_t are still 8-bit integers. The public algorithm still works, but now y_t and z_t are no longer integers. As a result, it will quickly lead to numeral inaccuracy and completely erroneous numbers unless you use special techniques to handle very high precision float, with billions of digits after the decimal point. Think about $y_0 = \log(415)$ and $z_0 = \exp(\sqrt{7\pi})$ just to give an example of the possibilities. This makes it a lot harder to win for the player. But it does not make it much more difficult for the operator since it knows x_0, and the player does not.

When y_0 and z_0 are integers, the public algorithm – if implemented in Python – automatically takes care of very large integers. This no longer works when the seeds are not integers. In the above example, the seeds are private although some minimum information about them is provided to the player, to guarantee that the set of winning numbers is uniquely defined given the public information, and that the player only has to try a finite set of seeds. Should the winning numbers be reachable via multiple sets of seeds, it may increase the player's chance of winning. Thus the operator might want to make sure that there is only one path to the winning numbers, given the public information.

5.3.4.1 Towards maximum security

What if some player figures out the connection between the public algorithm and Formula (5.1) and (5.2)? For instance, by reading this chapter. He still has to try many seeds, but he could relatively quickly compute (possibly in a number of weeks or months) billions of digits of all the potential candidates for x_0, and then get an hedge to find the winning numbers. In order to make the system robust against such hacks, there are different possibilities, in addition to frequently changing the seeds or using non-integer seeds:

- Reparametrize y and z in the public algorithm, using a mapping $(x, y) \mapsto (y', z')$ such as a linear transform. The new public algorithm will be based on y', z', using the updated recursion. The player may not see the connection between the new recursion and the version published in this chapter.
- Use an entirely different recursion attached to more general algebraic numbers rather than quadratic irrationals. This is a work in progress.
- The easiest solution is to keep the recursion as is, but instead use (say) 5 different seeds $(y_{0,k}, z_{0,k})$ with $0 \leq k < 5$. It leads to 5 different sequences, and the winning number at iteration t comes from the k-th sequence, with $k = t \bmod 5$.

The last option is simple and offers flexibility. For instance, you could use 200 rather than 5 sequences.

5.3.4.2 Real example

Here I illustrate the method with a simple example. In this case, the public data consists of two pairs of seeds $(y_{0,1}, z_{0,1})$, $(y_{0,2}, z_{0,2})$ and the past 2000 winning numbers. Can you find the next 500 winning numbers with this information? One of the two seeds leads to the winning numbers, the other one does not. Also available in the public information: you need less than one trillion iterations to identify the 2000 winning numbers in question. You may use the public algorithm in section 5.3.1 to answer this question, or better, any hack of your own. The public information is available at the following locations:

- The seeds $y_{0,1}, z_{0,1}, y_{0,2}, z_{0,2}$ are on GitHub. The links to the files are $y_{0,1}$, $z_{0,1}$, $y_{0,2}$, $z_{0,2}$. Each of them is about 3 megabytes (uncompressed).
- The file with the 2000 past winning numbers can be found here. It also contains the 500 future winning numbers. Ignore them: these are the numbers that you are supposed to find. I included them so you that can check against the winning numbers that you come up with.

Note that the filenames contain information about times and seeds chosen in this test. In the file with the winning numbers, the first column represents the time t. In practice, this information is not available or encrypted. Here you can retrieve the winning numbers in a matter of minutes, by using this information, with the Python code in this section. Pretend that you don't have this information, and see if you can retrieve the winning numbers in a reasonable amount of time. The size for each seed is about 3 megabytes. In an industry-grade version, this size could be several gigabytes or even terabytes, and the secrete β (see code) much larger than the one used in this test, and hard to find.

Increasing the number of candidate seeds (with only one leading to winning numbers) may not significantly increase the security of the system, as hackers may pre-compute tables with billions of digits for as many integers (β in the code) as they can [22]. A much better solution is to interlace winning numbers from multiple seeds, as discussed in the last bullet item in section 5.3.4.1. In this case, the hacker does not know which seed is used at any given iteration, as many combinations are possible. Also, working with trillions rather than billions of digits makes it considerably harder for hackers. To this day, only the first 10^{13} digits of $\sqrt{2}$ are known, see here. Typically, such computations require months of computing time.

The Python code below is private and not shared with the player. The operator uses it to generate the winning numbers. It is also on GitHub here, and named `lottery_fast.py`. The variables `y0_1`, `z0_1`, `beta_1`, and `alpha_1` in the code correspond respectively to y_0', z_0', β' and α' in section 5.3.3. In this case, the quadratic irrational is $x_0 = -\lfloor\sqrt{\beta}\rfloor + \sqrt{\beta}$. You can skip the computation of the first 10^8 digits and directly jump at iteration $1 + 10^8$ by setting `offset=10**8` in the code. The number of winning numbers produced is about one eighth of `n`.

The code allows you to determine the seed when starting at an arbitrary iteration `t+offset` rather than $t = 1$. I use this functionality to create the large public seeds leading to winning numbers. Of course, `offset` is kept private. It also allows you to compute the winning numbers starting at an even much larger iteration $t = T$, with T also kept secret. One issue is checking whether the digits produced are correct. In practice I use two different mechanisms to compute the digits, to double-check. I have seen implementations that are correct for the first few million digits, then fail later on in the sequence.

```
import gmpy2

def big_seed(offset, beta):

   beta_1 = beta * (2**(2*offset))
   alpha_1 = int(gmpy2.isqrt(beta_1))
   y0_1 = 2 * (beta_1 - alpha_1*alpha_1)
   z0_1 = 1 + 4*alpha_1
   return(y0_1, z0_1)


n = 20000      # number of digits to compute
offset = 10**7 # digits start at location 1 + offset
beta = 2
y0, z0 = big_seed(offset, beta)
y = y0
z = z0

digits = {}
winning_numbers = {}

for t in range(1, n):
   if z < 2*y:
      y = 4*y - 2*z
      z = 2*z + 3
      digits[t + offset] = 1
   else:
      y = 4*y
      z = 2*z - 1
      digits[t + offset] = 0
   if t > 8 and t % 8 == 5:
      w = 0
      for k in range(8):
         w += digits[t + offset - k] * (2**k)
      winning_numbers[t + offset - k] = w
      print(t + offset, w)

filename = "lottery_seed_y" + str(offset) + "_" + str(beta) + ".txt"
OUT=open(filename,"w")
OUT.write(str(y0))
OUT.close()

filename = "lottery_seed_z" + str(offset) + "_" + str(beta) + ".txt"
OUT=open(filename,"w")
OUT.write(str(z0))
OUT.close()

filename = "lottery_winning_numbers_" + str(offset) + "_" + str(beta) + ".txt"
```

```
OUT=open(filename,"w")
for time in winning_numbers:
   number = winning_numbers[time]
   OUT.write(str(time) + "\t" + str(number) + "\n")
OUT.close()
```

5.3.5 Collision risks

This problem is related to the probability of finding (or not) a substring in a string [27]. In this context, the player may use the wrong seeds yet by some incredible luck, find the past winning numbers in his own sequence. This is called a collision. Given the constraints of the system, I show that the chance of such a collision is incredibly close to zero.

Here the substring consists of the past 2000 winning numbers, in other words $m = 8 \times 2000 \approx 2^{14}$ bits. The string consists of numbers originating from a random-looking sequence, other than that used by the operator. In terms of string terminology, the associated alphabet – the letters – consists of two symbols: 0 and 1. Since the winning numbers are guaranteed to show up within the first trillion iterations, the size of the string (the long random-looking sequence) is about $N = 2^{40}$ bits.

Finally, if the operator publishes a set of $K+1$ seeds with one of them leading to the past winning numbers, what is the chance that at least one of the K remaining sequences also leads to the same past winning numbers, albeit (in all likelihood) starting at a different position? Now I give an approximate answer to this question. The probability of the m-bit subsequence of winning numbers being in a given larger N-bit sequence is

$$1 - \Big(1 - \frac{1}{2^m}\Big)^{N-m+1} \approx \frac{N-m+1}{2^m} \approx 2^{-15960}. \tag{5.4}$$

So, even if $K = 2^{300}$, it is still considerably smaller than 2^m, and the risk of at least one collision across the K sequences is not larger than (5.4) multiplied by K. In short, the operator can offer a selection of 2^{300} seeds, claiming that only one of them leads to the winning numbers within a trillion iterations.

5.3.6 Exercises

The following original exercises complement the theory. Several are good candidates as job interview questions for software engineers, or exam questions in computer science programs. They range in complexity from relatively simple (requiring less than one hour of work) to difficult. To help you decide which exercises to work on depending on your interests, I added a title to each of them.

Exercise 18 *Partial sums of binary digits* – Let $s_t(x_0) = d(1, x_0) + \cdots + d(t, x_0)$ be the sum of the first t binary digits of a real number $x_0 \in [0, 1[$. Thus $0 \leq s_t(x_0) \leq t$. Typically, it is easier to study these sums rather than the individual digits, as they are less volatile. Show that $s_{t+1}(x_0/2) = s_t(x)$, for $t = 0, 1, 2$ and so on. By convention, $s_0(x_0) = 0$. Also show that

$$2x_0 = \sum_{t=1}^{\infty} \frac{s_t(x_0)}{2^t}, \tag{5.5}$$

where $d(t, x_0)$ is the t-th binary digit of x_0. Finally, using a greedy algorithm, for any real number $x_0 \in [0, 1]$, show how to compute $s_t(x_0)$ without computing the individual digits.

Solution

Formula (5.5) is obtained by summing the left and right hand sides of the following equalities:

$$\begin{aligned}
x_0 &= \frac{d(1,x_0)}{2} + \frac{d(2,x_0)}{4} + \frac{d(3,x_0)}{8} + \cdots, \\
\frac{x_0}{2} &= \qquad\qquad\;\; \frac{d(1,x_0)}{4} + \frac{d(2,x_0)}{8} + \cdots, \\
\frac{x_0}{4} &= \qquad\qquad\qquad\qquad\qquad\;\; \frac{d(1,x_0)}{8} + \cdots,
\end{aligned}$$

and so on. As for the greedy algorithm, it works as follows. Let $0 \leq x_0 < 1$ and

$$h_n(x_0) = \sum_{t=1}^{n} \frac{\varphi_t(x_0)}{2^t}, \quad n = 1, 2, \ldots$$

Iteratively compute $h_n(x_0)$ for $n = 1, 2$ and so on, by defining $\varphi_n(x_0)$ as the largest integer with $0 \leq \varphi_n(x_0) \leq n$, such that $h_n(x_0) \leq 2x_0$. Then $\varphi_t(x_0) = s_t(x_0)$ for all t. Also, as $n \to \infty$, $h_n(x_0) \to 2x_0$.

Note that if you replace the constraint $0 \leq \varphi_n(x_0) \leq n$ by $0 \leq \varphi_n(x_0) \leq 1$, then you get the binary digit representation of $2x_0$ instead, with $\varphi_t(x_0) = d(t, 2x_0)$ for all t and $h_n(x_0) \to 2x_0$, assuming $x_0 < \frac{1}{2}$.

Exercise 19 *Autocorrelation in the sequence of winning numbers* – Given a seed (y_0, z_0), show that the autocorrelation in the sequence $\{w_t\}$ with $t = 1, 2$ and so on, is $\frac{1}{2}$. To the contrary, the autocorrelation in the sequence w_t, w_{t+8}, w_{t+16} and so on, is zero regardless of where you start.

Solution
The sequence $w_t, w_{t+8}, w_{t+16}, \ldots$ consists of successive blocks of non-overlapping bits in the binary representation of x_0, or in other words, digits in base 256. These blocks are independent and thus uncorrelated assuming x_0 is a strongly normal number. To the contrary, the sequence $w_t, w_{t+1}, w_{t+2}, \ldots$ consists of overlapping blocks. For instance, w_t and w_{t+1} both have 8 bits, but share 7 bits. Thus the autocorrelation.

Exercise 20 *Some digit sequences are more random than others* – Identify some quadratic irrationals that exhibit a less random behavior than others, in their binary digit sequence. Can you explain why?

Solution
If you pick up a million quadratic irrationals and look at the first million digits for each of them, you are bound to find some extremes. Actually, failure to find such extremes would be an indication of lack of randomness. You would expect these numbers, at least some of them, given enough digits, to *locally* exhibit some patterns.

In the end, the number of zeros and ones in a random sequence of n digits is governed by the law of the iterated logarithm, see section 4.2.1.2. It is typically different from $n/2$, with a discrepancy in the order $\sqrt{n}$. The maximum discrepancy is of the order $\sqrt{n \log \log n}$. A lower discrepancy, say of order $n^{1/4}$, is actually a sign of non-randomness, contrarily to intuition: see section 1.5.2 for examples and discussion.

Other metrics such as the length of the maximum run, are governed by similar laws. I looked at those statistics to identify outliers, and shared my results in Table 4.1. For instance the seed $y_0 = 90, z_0 = 91$ leading to $x_0 = (-45 + \sqrt{2203})/2$, fails the prime test of randomness discussed in section 4.2.1.1, at least for the first 20,000 digits.

In general, it is a good idea to skip the first million digits, as increased randomness typically kicks in later in sequences that are otherwise initially less random.

Exercise 21 *Square-free integers and seeds to avoid* – Numbers of the form $(z_0 - 1)^2 + 8$ where $z_0 > 1$ is an integer, rarely contain a square factor. These numbers appear in Formula (5.1) with $y_0 = 1$, and thus satisfy $y_0 < z_0$ as required in Theorem 5.3.1. To the contrary, among all positive integers, only a proportion $6/\pi^2$ consists of square-free integers. Identify integers of the form $(z_0 - 1)^2 + 8$ that are not square-free, and show how rare they are.

Solution
You can use the Python code in section 4.4.3 to answer this question. Among the first `Niter` integers of the form $(z_0 - 1)^2 + 8$, the variable `accepted` counts those that are square-free. Seeds (y_0, z_0) leading to non square-free integers $(z_0 - 1)^2 + 8y_0$ must be removed because they introduce cross-correlations among the various sequences of winning numbers across multiple seeds. The fact that these seeds are rare means that very few must be rejected.

Exercise 22 *Cross-correlation between binary digit sequences* – This is a difficult exercise. Let $X = x_0$ be a positive real number with random digits, each digit with probability $\frac{1}{2}$ of being one. If p, q are two strictly positive co-prime integers, free of factors of the form 2^k for $k = 1, 2$ and so on, then the correlation between the binary digit sequences of pX and qX, is equal to $(pq)^{-1}$. By digit sequences, I mean the digits starting after the decimal point. How would you use this fact to prove that the binary digits of $x_0 = -1 + \sqrt{2}$ and $x'_0 = -2 + \sqrt{7}$ are uncorrelated? Also show that the correlation between the binary digits of $-4 + \sqrt{18}$ and $-7 + \sqrt{50}$ is $1/15$.

Solution
Use the Python code in Exercise 23 to do some preliminary investigations, starting with $p = 1$ and $q = 3$, to empirically confirm that the correlation is $(pq)^{-1}$.

If $x_0 = -1 + \sqrt{2}$ and $x'_0 = -2 + \sqrt{7}$, then there is no real number X such that $x_0 = pX, x'_0 = qX$ and p, q are positive integers. Otherwise $x_0/x'_0 = p/q$ would be a rational number, and we know that x_0/x'_0 is irrational. Indeed, the only way to make p/q irrational is to have $p, q \to \infty$. Then the correlation $(pq)^{-1}$ in question (for the digit sequences of x_0 and x'_0) is zero, the desired result.

To prove that the correlation is $(pq)^{-1}$, for the digit sequences of pX and qX assuming p, q are co-prime positive integers free of powers of 2, see here. The proof is unfinished. It uses the same notations as in the Python code in Exercise 23. It relies on analyzing the carry-over mechanism when doing the multiplications pX and qX using the grade school algorithm (implemented in the Python code). I invite you to finish my proof.

Finally, $\sqrt{18} = 3\sqrt{2}$ and $\sqrt{50} = 5\sqrt{2}$. Using $X = \sqrt{2}, p = 3$, and $q = 5$, the correlation between the two corresponding digit sequences is $(pq)^{-1} = 1/15$. It can be verified empirically. This assumes that $X = \sqrt{2}$ is strongly normal, a conjecture widely believed to be true.

Exercise 23 *Grade school multiplication* – Write a program that computes the binary digits of pX. Here p is a positive integer, and X is a positive real number in $[0, 1]$. Use this program to compute the correlation between the sequences of binary digits of pX and qX, where p, q are positive integers, and X a number in $[0, 1]$ with random binary digits. Focus on the digits after the decimal point (ignore the other ones).

Solution

Here is the Python code. It is also on GitHub, here. It creates the digits of X, then those of pX and qX, starting backward with the last digits. Finally it computes the correlation in question, assuming the digits of X are random. It works if X has a finite number of digits, denoted as `kmax` in the code. By increasing `kmax`, you can approximate any X with infinitely many digits, arbitrarily closely.

```
# Compute binary digits of X, p*X, q*X backwards (assuming X is random)
# Only digits after the decimal point (on the right) are computed
# Compute correlations between digits of p*X and q*X
# Include carry-over when performing grade school multiplication

import numpy as np

# main parameters
seed = 105
np.random.seed(seed)
kmax = 1000000
p = 5
q = 3

# local variables
X, pX, qX = 0, 0, 0
d1, d2, e1, e2 = 0, 0, 0, 0
prod, count = 0, 0

# loop over digits in reverse order
for k in range(kmax):

   b = np.random.randint(0, 2) # digit of X
   X = b + X/2

   c1 = p*b
   old_d1 = d1
   old_e1 = e1
   d1 = (c1 + old_e1//2) %2 # digit of pX
   e1 = (old_e1//2) + c1 - d1
   pX = d1 + pX/2

   c2 = q*b
   old_d2 = d2
   old_e2 = e2
   d2 = (c2 + old_e2//2) %2 #digit of qX
   e2 = (old_e2//2) + c2 - d2
   qX = d2 + qX/2

   prod += d1*d2
   count += 1
   correl = 4*prod/count - 1

   if k% 10000 == 0:
      print("k = %7d, correl = %7.4f" % (k, correl))

print("\np = %3d, q = %3d" %(p, q))
print("X = %12.9f, pX = %12.9f, qX = %12.9f" % (X, pX, qX))
print("X = %12.9f, p*X = %12.9f, q*X = %12.9f" % (X, p*X, q*X))
print("Correl = %7.4f, 1/(p*q) = %7.4f" % (correl, 1/(p*q)))
```

5.4 Customized ROI tables and business model

The platform may work like a stock exchange. Here I use the word participant or player interchangeably. The operator, sometimes called the house in casino terminology, is the company managing the platform. Buying a number (also called a guess) is sometimes referred to as making a bet. The purchase price is called a wager.

The participant pays a small transaction fee for each purchased number. Buying a number is similar to buying a stock on the stock market. If the number is a winning number or close to a winning number, your invested money (the wager) increases in value, generating a positive ROI. Otherwise, it decreases, generating a negative ROI. Once the winning number is announced, the player can redeem his money (wager plus gain or minus loss), or leave it on his account and use it for future bets. The operator may offer mechanisms to automatically buy numbers regularly so that the money invested by the player is actively "traded".

If the participant purchases all the 256 potential numbers for a given sequence at a given time and places the same dollar amount on each of them, the return is zero. In other words, the operator does not make money by "taxing" or taking a share on positive returns: the game is perfectly fair. The operator makes money via the fixed transaction fees only.

Of course, the participant can pick up a number randomly, and has a 1/256 chance to win the largest return in the 8-bit system. Indeed, there is no way to do better than pure luck, in order to win. The reason is because the public algorithm requires too much computing time to find the future winning numbers, given a sequence of past winning numbers and a choice of public seeds. By the time you find them – which could take years – the sequence in question has been archived, and new sequences are offered. That said, the player can use any algorithm to find patterns in the data, and test them in the hope to increase her odds to get a positive return, as well as for learning purposes. In the end, the system is similar to the real stock market – a mathematical competition – except that in principle, you know in advance with absolute certainty which stocks (in this case numbers) will win at any given time.

How much ROI you can get depends on the strategy that you select. The operator offers different strategies, and the player can also create and upload customized strategies, as long as they are perfectly fair (neutral). Templates are provided to create customized fair strategies. A strategy is a ROI table that tells you how much you can win or lose depending on how close you are to the winning number. Some provide gains or losses comparable to trading major indexes, and are rather safe: your ROI on a single transaction may never be less than -10% or more than 20%. Some are very aggressive and comparable to playing the lottery (except that lotteries are not fair games). In this case you could lose 50% of your wager in a worst case scenario – if your number is very far from the winner – or multiply it by a factor 10 if your number is a winner.

A player can submit multiple numbers for a given sequence at any given time. Or she may decide to submit only one number, or continue to play on the same sequence (new winning numbers are issued regularly) until she wins some positive ROI, or until her loss reaches some threshold, whichever comes first.

Thus, the core of the methodology, from a business standpoint, are the strategies and the public ROI tables attached to them. These tables allow you to choose a strategy that matches your risk tolerance. I now describe them in details. I explain the 32-bit system later in section 5.4.1.2.

5.4.1 Bracketed and parametric ROI tables: examples

In the 8-bit system, both winning numbers and guesses submitted by the player are integers between 0 and 255 inclusive. Let's define the distance between a winning number w, and the guess w' made by a player, as

$$d(w, w') = \min\Big(|w - w'|, 256 - |w - w'|\Big). \tag{5.6}$$

Thus the distance is an integer between 0 and 128 inclusive. Assuming guesses and winning numbers are perfectly random, the distance is a random variable D with the following distribution:

$$P(D = d) = \begin{cases} \frac{1}{256}, & d = 0 \text{ or } 128 \\ \frac{1}{128}, & 0 < d < 128 \end{cases} \tag{5.7}$$

Formula (5.7) is at the core of the business model. It allows you to compute expected profits and losses, as well as the variance in financial metrics, for the operator. Before diving into this in section 5.4.2, I now define the closely related concept of ROI table. An ROI table is a file with three columns, generated by the operator and offered to the players. These columns are:

1. The distance d between a guess w' and a winning number w. It is an integer number between 0 and 128 inclusive, and the key of the ROI table, from a database standpoint.

2. The multiplier $m(d)$. If you bet one dollar on a number w', and the distance to the winning number w is $d = d(w, w')$, then your one dollar is multiplied by $m(d)$. The multiplier is a positive real number. When smaller than one, you lose some money. Otherwise you make money. The function $m(d)$ is decreasing, and maximum when $d = 0$, that is, when you correctly guess the winning number.
3. The return $r(d)$ is the ROI on the guessed number. Thus, $r(d) = m(d) - 1$. Positive returns of 500% are possible depending on the ROI table. However, you can not lose more than the amount you put on an number. This is in contrast with the real stock market where shorting may lead to losses bigger than the principal.

The second and third columns are redundant, so you may use only one of them. All the ROI tables offered by the operator are neutral (also called fair or unbiased) in the sense that $m(d)$ must satisfy

$$\sum_{d=0}^{128} m(d) \cdot P(D = d) = 1. \tag{5.8}$$

The operator generates revenue via a small transaction fee on each bet. Thus the return, for any player, is not influenced by the gains and losses of other players. This is contrast to the the real stock market or other systems where the collected money is simply redistributed to participants, minus a fixed portion kept by the operator.

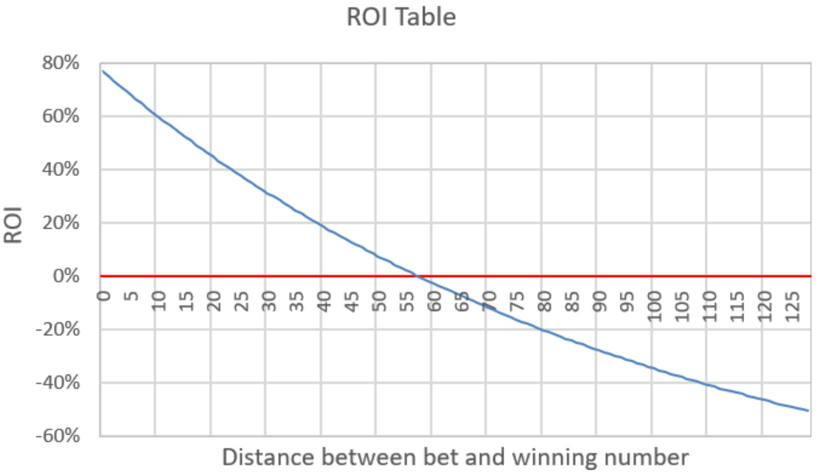


Figure 5.1: Smooth neutral ROI table with moderate risk / reward

5.4.1.1 Types of ROI tables, with examples

Several examples of ROI tables are featured in my spreadsheet `lottery_ROI_tables.xlsx` available on GitHub, here. They range from high risk / high reward with a small chance of winning big, similar to a lottery, to low risk / low reward for risk-adverse players, in this case similar to trading major indexes on the stock market.

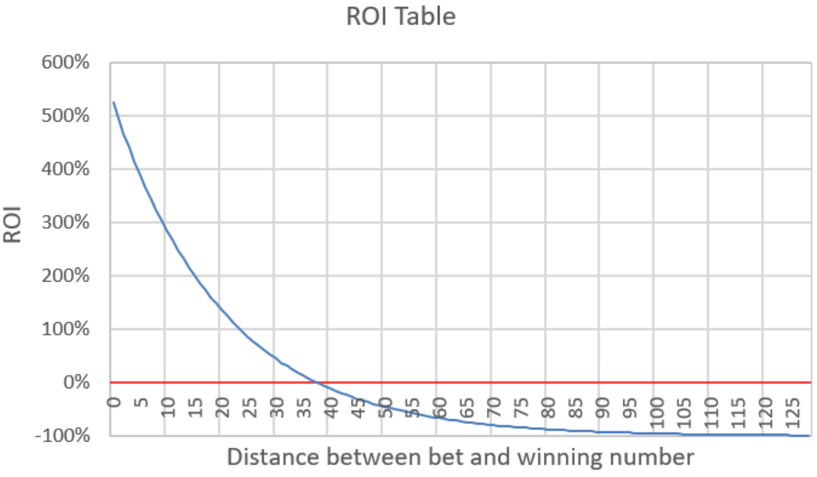


Figure 5.2: Smooth neutral ROI table with high risk / high reward

In addition, the ROI curve may appear smooth as in Figures 5.1– 5.3, or piecewise-linear with brackets as in Figure 5.4. The former is a called the parametric type because the ROI curve is a smooth decaying math function governed by one or two parameters, such as $m(d) = \mu\lambda^d$ with $\lambda < 1$. Depending on the parameters, it is designed for risk takers (Figure 5.2) or risk-adverse players (Figure 5.3).

Again, all these tables are fair and designed for the 8-bit system. Tables for the 32-bit system are described in section 5.4.1.2. The Y-axis in Figures 5.1– 5.4 represents $r(d)$, while the X-axis represents d.

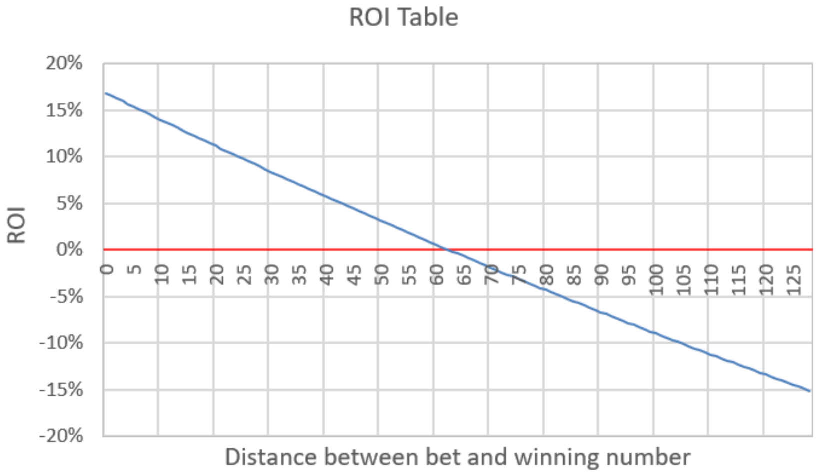


Figure 5.3: Smooth neutral ROI table with low risk / low reward

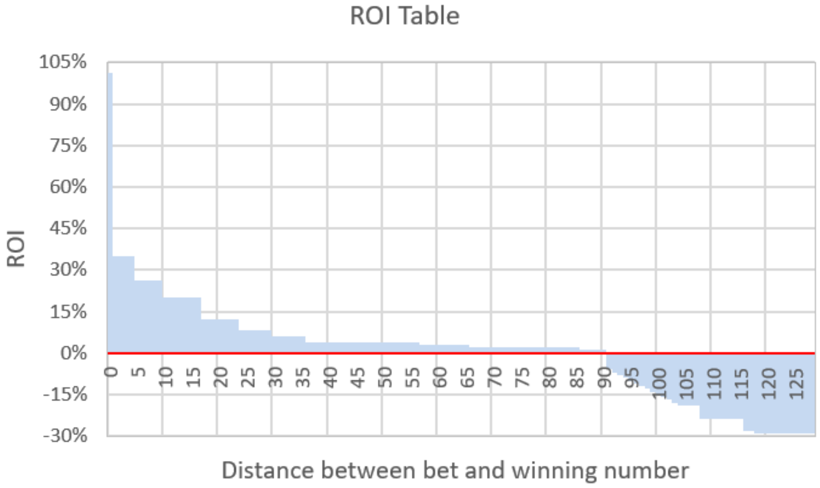


Figure 5.4: Bracketed neutral ROI table with about 20 brackets

5.4.1.2 The 32-bit system

In the 32-bit system, winning numbers consist of 4 successive winning numbers from the 8-bit system. Now the player makes a 32-bit guess, and uses 32-bit ROI tables provided by the operator. The distance function in (5.6) and everything else easily generalize to 32-bit.

distance d to winning number	multiplier $m(d)$	proportion of guesses
0 – 19	1,000,000	0.00000091%
20 – 26,065	1000	0.0012%
26,066 – 11,530,000	10	0.54%
11,530,001 – 1,100,765,000	1.15	50.7%
1,100,765,001 – 2,147,483,648	0.70	48.7%

Table 5.2: Example of a fair, bracketed 32-bit ROI table

This system allows the operator to create tables with a very small chance to win a very large payout, with a potential multiplier $m(d) = 10^6$ for the winning number ($d = 0$), compared to 10^3 for aggressive tables in the 8-bit system. It is more risky both for the operator and the player, despite the average ROI still being zero. However, it may attract more users, but also more attention from regulatory agencies. Table 5.2 is an example of a fair ROI table in the 32-bit system, with extraordinary return when correctly guessing the 32-bit winning number.

With table 5.2, if your guess is within 19 points of the winning number (it will happen to about 39 people out of 4.3 billion) then your money (the amount that you bet) is multiplied by a factor one million. About 48.7% of the guesses result in a 30% loss. The next bracket is a 15% gain, and 50.7% of all guesses fall in that category. About one in two hundred (0.54%) results in a 900% ROI. One in 100,000 would boost your wager by a factor 1000.

5.4.1.3 Free simulator, time to positive return

The most basic ROI table works as follows. You place a bet on a number. The multiplier function $m(d)$ is such that you either win \$1 with probability $\frac{1}{2}$, or lose \$1 with probability $\frac{1}{2}$. The cumulative gains or losses by repeating this experiment over time, follow a symmetric random walk. It is well known that the probability of eventually returning to zero – that is, erasing all your losses at some point if you are in a losing streak – is 100%. The random variable measuring the time to reach such an event (hitting a target) is known as an hitting time. The topic is discussed in section 1.5.1. See also chapter 12 on random walks in [15], and lecture 5 entitled "Random walks – advanced methods" in [35].

Indeed the random walk will almost surely hit a positive value or gain (and thus any arbitrary large gain) in finite time if you play long enough, see [33] page 102. The proof is based on the Wald martingale [Wiki] derived from the random walk. However, the expected number of bets to reach any prespecified value (positive or negative) is infinite. You would think that the operator will eventually go bankrupt (if there were no transaction fee), as each participant can play long enough until realizing a gain. But this is not the case: see section 5.4.2. Anyway, this makes for a great selling point to attract players.

Offering free simulations to participants, or a free trial, is another way to attract users. They can test various ROI tables and see what they could earn by playing with the platform for a while, without committing real money. Half of them are expected to be in positive territory by the end of the trial period. The free version (where you don't play with real money) could be monetized via advertising, attracting sponsors selling financial products, educational material, or gambling platforms. It could also offer the possibility to enter new numbers (or the same one) in real time across multiple sequences and/or multiple ROI tables.

5.4.2 Daily payouts for the operator: Brownian motion

It is impossible to predict the total amount to pay each day to the participants. On average, the amount is zero when combining gains and losses, but the daily variations over thousands of bets can potentially be large depending on the ROI tables in use. The purpose of this section is to study these variations, and show how the aggregates follow a Brownian motion over time. Properties of Brownian motions are discussed in chapter 1.

Without loss of generality, let us assume that the wager v is always a fixed \$20. For an individual bet, the player's payout is $v \cdot m(d)$, where $m(d)$ is the multiplier based on the ROI table, and d the distance to the winning number. Thus the operator must pay back $v \cdot m(d)$ to the player. This dollar amount includes the initial wager. The profit (or loss) on this transaction is $v - v \cdot m(d)$ for the operator if we ignore transaction fees. On average it is zero when using fair ROI tables: see Formula (5.8).

Assuming the number chosen by the participant is random, the variance for the operator's profit/loss on a single bet in the 8-bit system, given a ROI table, is

$$\begin{aligned}
\sigma^2 &= \text{Var}[v - v \cdot m(D)] \\
&= v^2 \text{Var}[m(D)] && (5.9) \\
&= v^2 \Big(\text{E}[m^2(D)] - \text{E}^2[m(D)] \Big) \\
&= v^2 \Big(\text{E}[m^2(D)] - 1 \Big) \\
&= v^2 \left[\Big(\sum_{d=0}^{128} m^2(d) P(D = d) \Big) - 1 \right], && (5.10)
\end{aligned}$$

where D – a random variable – is the distance to the winning number, with $P(D = d)$ given by (5.7). Aggregated over n bets in a single day (assuming to be independent), the standard deviation is thus $\sigma_n = \sqrt{n} \cdot \sigma$. Since these profits/losses follow a Gaussian distribution of zero mean when n is large, it is easy to compute the 99.99 percentile: a worst case occurring once every 10,000 days on average – when multiple players win big on a single day. Or the 99.995 percentile: the extreme occurring once every 20,000 days on average.

The most aggressive ROI table with the highest σ^2 corresponds to $m(0) = 256$ and $m(d) = 0$ if $d > 0$. You win only when you correctly guess the winning number. The odds are 1/256 per guess. Then your wager v is multiplied by 256. Otherwise you lose the money you placed on that number, in its entirety. In this case, it is easy to verify that $\sigma_n^2 = 255 \cdot v^2 n$. If the daily volume of transactions attached to this ROI table is 10,000 bets at \$20 each, then $\sigma_n \approx$ \$31,937. The 99.99 percentile of a standard normal distribution is about 3.72. Thus the worst daily loss expected in 30 years is about 3.72 × \$31,937 ≈ \$118,805. Aggressive 32-bit ROI tables produce far worse extremes.

Finally, you need to decide on the transaction fee. At 1 cent on a \$20 wager, your average daily revenue in the above example would be 0.01 × 10,000 × \$20 = \$2000, minus the finance charges to process credit cards or other financial instruments used to accept payments. This is more than enough to offset potentially severe aggregated losses due to the growing volatility of the underlying Brownian motion attached to the payouts. It also guarantees a strong linear revenue growth. Without the transaction fees, losses due to payouts may take millions of years to naturally recover on their own. They always do eventually, in the same way that players always reach a positive return at some point, absent of transaction fees.

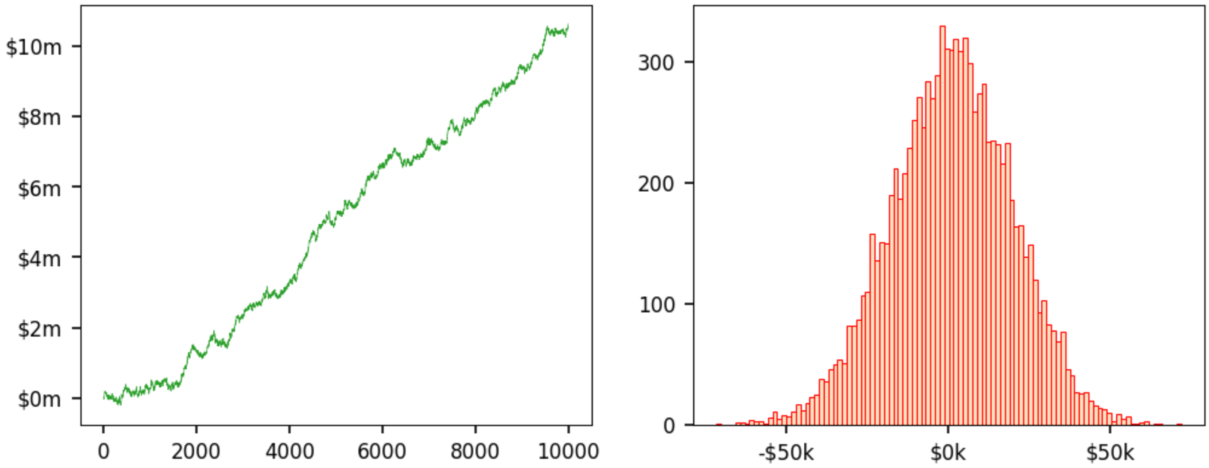


Figure 5.5: Cumulative profits (left) and distribution of daily profits (right), with a 0.025% fee

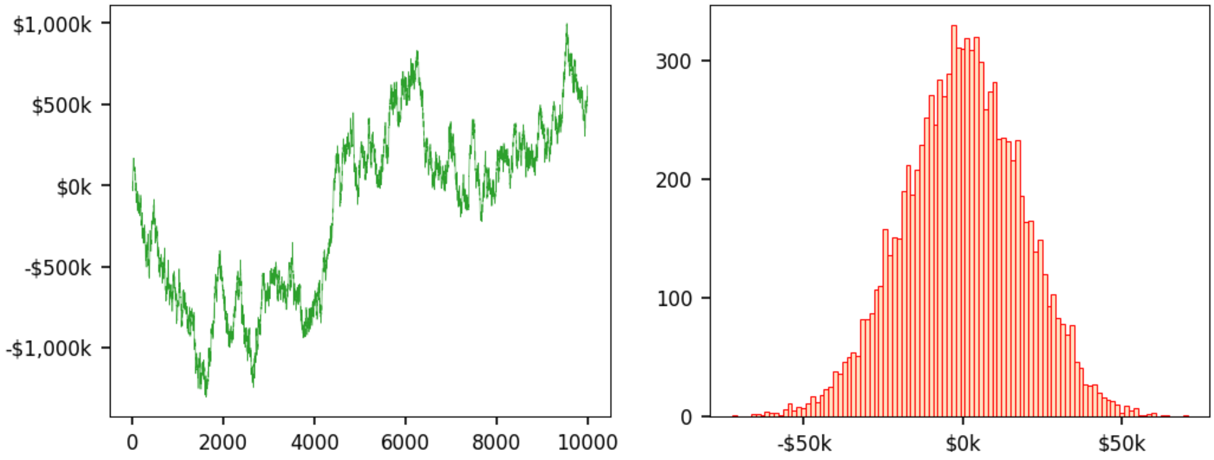


Figure 5.6: Cumulative profits (left) and distribution of daily profits (right), with no fee

5.4.2.1 Example with truncated geometric distribution

I now illustrate the profit for the operator over time, and the statistical distribution of daily profits, based on the transaction fee and the type of ROI table. Examples in Figures 5.1– 5.3 involved a truncated geometric function for $m(d)$. In other words, $m(d) = \mu\lambda^d$ for $0 \leq d \leq 128$, with μ a normalization factor so that (5.8) is satisfied, and $0 < \lambda < 1$. I use the same function here. With $\lambda = 1$ and no transaction fee, there is zero profit or loss either for the operator or the player. The most volatile case, with maximum potential payout, corresponds to $\lambda \to 0$. In that case, $m(d) = 256$ if $d = 0$, otherwise $m(d) = 0$: the player loses the entirety of his wager.

The cumulative profits over time are extremely sensitive to the seed when there is no transaction fee. Each seed leads to a different Brownian motion path. But the introduction of even a tiny transaction fee, completely changes the picture to the benefit of the operator. It turns the Brownian path into an almost straight line with positive slope, thus consistently accumulating profits. In figure 5.5, the transaction fee is 0.025% (half of a cent on a \$20 wager). The time period is 10^4 days, that is, about 30 years. By contrast, there is no transaction fee in figure 5.6. The two right plots look identical, but they aren't: the one in Figure 5.5 is not centered at zero, but shifted by a miniscule amount, to the right. The impact of this shift on the left plot (cumulative profit) is dramatic.

The Brownian motion in Figure 5.6 (left plot) is going nowhere. It is unpredictable, and will oscillate infinitely many times between positive and negative values, with the daily standard deviation increasing linearly over time. I encourage you to play with the code, and try different transaction fees, different seeds, and different values for λ.

5.4.2.2 Python code and algorithm to find optimum transaction fee

For simplicity, rather than digits of quadratic irrationals, the Python code uses random numbers generated by the Mersenne twister via the `randint` function. Thus, the number of potential seeds is limited to 2^{32}. With quadratic irrationals, the number of potential seeds is infinite, making it harder to crack.

To determine the minimum transaction fee, I suggest the following approach: compute the R-squared ρ^2 between the random curve on the left plot in Figures 5.5– 5.6, and the "best fit" straight line. In short, perform a simple linear regression. You want to choose a transaction fee satisfying $\rho^2 > \rho^2_{\text{min}}$, where ρ^2_{min} is a prespecified threshold. This guarantees that the operator makes steady profits, while offering a very low transaction fee.

Increasing the transaction fee will increase ρ^2. Even with a modest fee – say 0.1% instead of 0.025% in my example – ρ^2 is extremely close to 1. In short, the curve (almost) becomes a straight line. The minimum transaction fee depends on the ROI table, with aggressive tables (high σ^2) requiring higher minimum transaction fees to operate a predictably profitable business.

In practice, you use much higher fees than the minimum required, to accumulate profits faster. However fees that are too high will deter potential players. The solution is to do some price elasticity analysis, to find out the ideal fees that maximizes profits, for each ROI table.

In the Python code, variables starting with `arr_` represent arrays. The code is also on GitHub, here.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.ticker as mtick

def set_ROI_table(llambda):

   mu = 0
   ROI_table = np.empty(129)

   for d in range(129):
      multiplier = llambda**d

      ROI_table[d] = multiplier
      if d == 0 or d == 128:
         mu += multiplier/256
      else:
         mu += multiplier/128

   ROI_table /= mu
   return(ROI_table)

def compute_profits(llambda, fee, seed):

   arr_profits = []
   arr_cumul_profits = []

   np.random.seed(seed)
   ROI_table = set_ROI_table(llambda)
   for day in range(ndays):
      collected_fees = fee * v * nbets
      if day % 1000 == 0:
         print("seed: %4d day: %5d" % (seed, day))
      winner = np.random.randint(0, 256, nbets)
      guess = np.random.randint(0, 256, nbets)
      delta = abs(winner - guess)
      d = np.minimum(delta, 256 - delta)
      d_unique, d_counts = np.unique(d, return_counts=True)
      d_nvals = len(d_unique)
      if d_nvals == 129:
         profits = np.dot(d_counts, ROI_table) # fast computation
      else:
         profits = 0
         for index in range(d_nvals):
            d_value = d_unique[index]
            profits += d_counts[index]*ROI_table[d_value]
      today_profit = collected_fees + v*nbets - v*profits
      arr_profits.append(today_profit)
      if day == 0:
         arr_cumul_profits.append(today_profit)
      else:
         yesterday_profit = arr_cumul_profits[day - 1]
         arr_cumul_profits.append(today_profit + yesterday_profit)

   return(arr_profits, arr_cumul_profits, ROI_table)

#--- main part

ndays = 10000 # time period = ndays
nbets = 10000 # bets per day
v    = 20     # wager
llambda = 0.30 # strictly between 0 and 1
transaction_fee = 0.005
seed = 26

arr_profits, arr_cumul_profits, ROI_table = compute_profits(llambda,
   transaction_fee, seed)

#--- plot daily profit/loss distribution, and aggregated numbers over time

x = np.arange(ndays)
y = arr_cumul_profits

# custom tick functions

def currency_ticks_k(x, pos):
   x = int(x / 1000) # plotted values will be in thousand $
   if x >= 0:
      return ’${:,.0f}k’.format(x)
   else:
      return ’-${:,.0f}k’.format(abs(x))

def currency_ticks_m(x, pos):
   x = int(x / 1000000) # plotted values will be in million $
   if x >= 0:
      return ’${:,.0f}m’.format(x)
   else:
      return ’-${:,.0f}m’.format(abs(x))

mpl.rcParams[’axes.linewidth’] = 0.5
mpl.rc(’xtick’, labelsize=8)
mpl.rc(’ytick’, labelsize=8)

fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize =(8, 3))
```

```
axes[0].tick_params(axis='both', which='major', labelsize=8)
axes[0].tick_params(axis='both', which='minor', labelsize=8)

# plot: y axis in millions of dollars
tick_m = mtick.FuncFormatter(currency_ticks_m)
axes[0].yaxis.set_major_formatter(tick_m)
axes[0].plot(x, y, linewidth=0.3, c='tab:green')

# histogram: x axis in thousands of dollars
tick_k = mtick.FuncFormatter(currency_ticks_k)
axes[1].xaxis.set_major_formatter(tick_k)
plt.locator_params(axis='both', nbins=4)
axes[1].hist(arr_profits, bins = 100, linewidth = 0.5,edgecolor = "red",
         color = 'bisque', stacked=True)
plt.show()

#--- print ROI table

print ("ROI Table:")
for d in range(129):
   print("%3d %8.4f" %(d, ROI_table[d]))
print("Maximum multiplier: ", ROI_table[0])
```

5.4.3 Legal engineering

I do not provide legal advice here. Instead I offer different options about how to set up this type of business. You should discuss them with your lawyer, to find out the best solution to create a platform based on this system, depending on your local regulations. The business model may be perceived as a lottery, even though the winning numbers can – theoretically – be precomputed. It is in fact a mathematical contest that does not involve chance, but one that nobody can win given current computer hardware.

The first step is to avoid keywords such as bet, number guessing, gambling, or wager. On the plus side, conservative ROI tables are not different from playing major indexes on the stock market, and the game is fair. Also, given enough time and trials, all players, even if making random guesses, will regularly end up in positive territory due to mathematical laws pertaining to symmetric random walks. To avoid paying taxes on what tax agencies might consider as disguised wagers, special care must be taken. I now discuss several options.

1. Partner with a company allowed to operate this type of business, to avoid most of the red tape. Use a licensing agreement, where you design and provide the winning numbers.
2. Do it yourself with reliable third-party vendors. The platform that you choose to process transactions (accept online bets and so on) may also help you set up your business to meet legal requirements.
3. Find a trustworthy partner in a gambling-friendly country. Your partner will create a company in that country. Your separate, local company receive payments from this third-party entity. You pay local taxes on profits that your company makes (revenue from the foreign entity minus expenses), as any regularly business does. This strategy will protect you against being accused in your home country of operating an illegal lottery. It is targeted at US citizens in particular.
4. Offer the free version only, with no real money being gambled. On occasions, offer a reward (cash prize) to attract participants, without wagers or transaction fees. This option is interesting if you want to grow a website and make money via different means, for instance consulting or training. It will showcase your expertise.
5. Use a subscription-based model, with a fixed monthly fee, offering data and other services. Allow paid subscribers to use a portion of the subscription fee to participate in the system, without transaction fees or wagers, yet with the same chances of earning the same real money. There is no penalty for losing: the cost is absorbed by the subscription fees. Money paid to winners can be deducted by the operator as consulting expenses, to test the system. This is one way to grow your subscription base.
6. Players use the system for free, yet they can make real money as in the paid version: there are no wagers or transaction fees. Monetization comes from sponsorship, typically in the form of advertising revenue.
7. Make it a private venture. Participation is by invitation only, and approved players are listed as friends.

I presented this system at the Operations Research Society conference (INFORMS), in a session about algorithm biases. See the abstract here.

Bibliography

[1] David Bailey, Jonathan Borwein, and Neil Calkin. *Experimental Mathematics in Action*. A K Peters, 2007. 49

[2] David Bailey and Richard Crandall. Random generators and normal numbers. *Experimental Mathematics*, 11, 2002. Project Euclid [Link]. 78

[3] Rabi Bhattacharya and Edward Waymire. *Random Walk, Brownian Motion, and Martingales*. Springer, 2021. 14

[4] Luis Báez-Duarte et al. Étude de l'autocorrélation multiplicative de la fonction 'partie fractionnaire'. *The Ramanujan Journal*, 78:215–240, 2005. [Link]. 26

[5] Keith Conrad. *L-functions and the Riemann Hypothesis*. 2018. 2018 CTNT Summer School [Link]. 65

[6] D.J. Daley and D. Vere-Jones. *An Introduction to the Theory of Point Processes*. Springer, second edition, 2002. Volume 1 – Elementary Theory and Methods. 20

[7] D.J. Daley and D. Vere-Jones. *An Introduction to the Theory of Point Processes*. Springer, second edition, 2014. Volume 2 – General Theory and Structure. 20

[8] Harold G. Diamond and Wen-Bin Zhang. *Beurling Generalized Numbers*. American Mathematical Society, 2016. Mathematical Surveys and Monographs, Volume 213 [Link]. 65

[9] Benjamin Epstein. Some applications of the Mellin transform in statistics. *Annals of Mathematical Statistics*, 19:370–370, 1948. [Link]. 53

[10] P. A. Van Der Geest. The binomial distribution with dependent Bernoulli trials. *Journal of Statistical Computation and Simulation*, pages 141–154, 2004. [Link]. 15

[11] B.V. Gnedenko and A. N. Kolmogorov. *Limit Distributions for Sums of Independent Random Variables*. Addison-Wesley, 1954. 21

[12] Manuel González-Navarrete and Rodrigo Lambert. Non-markovian random walks with memory lapses. *Preprint*, pages 1–14, 2018. arXiv [Link]. 14

[13] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLTechniques.com, 2022. [Link]. 21, 36

[14] Vincent Granville. *Synthetic Data and Generative AI*. MLTechniques.com, 2022. [Link]. 8, 10, 15, 16, 29, 57, 65

[15] Charles M. Grinstead and Laurie Snell. *Introduction to Probability*. American Mathematical Society, second edition, 1997. [Link]. 93

[16] Nasr-Eddine Hamri and Yamina Soula. Basins and critical curves generated by a family of two-dimensional sine maps. *Electronic J. of Theoretical Physics*, 24:139–150, 2010. [Link]. 36

[17] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 62

[18] Adam J. Harper. Moments of random multiplicative functions, I: Low moments, better than squareroot cancellation, and critical multiplicative chaos. *Forum of Mathematics, Pi*, 8:1–95, 2020. [Link]. 62, 63

[19] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 64

[20] T. W. Hilberdink and M. L. Lapidus. Beurling Zeta functions, generalised primes, and fractal membranes. *Preprint*, pages 1–31, 2004. arXiv [Link]. 65

[21] Dixon J. Jones. A chronology of continued square roots and other continued compositions, through the year 2016. *Preprint*, pages 1–98, 2017. arXiv:1707.06139 [Link]. 47

[22] R. Kannan, A. K. Lenstra, and L. Lovász. Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers. *Mathematics of Computation*, 50:335–250, 1988. [Link]. 85

[23] Yuk-Kam Lau, Gerald Tenenbaum, and Jie Wu. On mean values of random multiplicative functions. *Proceedings of the American Mathematical Society*, 142(2):409–420, 2013. [Link]. 62, 63

[24] D. Lenz. Spectral theory of dynamical systems as diffraction theory of sampling functions. *Monatshefte für Mathematik*, 192:625–649, 2020. [Link]. 36

[25] Corina Macovei et al. The autocorrelation function of the logistic map chaotic signal in relation with the statistical independence issue. *IEEE 13th International Conference on Communications*, pages 25–30, 2020. [Link]. 43

[26] Peter Mörters and Yuval Peres. *Brownian Motion*. Cambridge University Press, 2010. Cambridge Series in Statistical and Probabilistic Mathematics, Volume 30 [Link]. 14, 20

[27] John Noonan and Doron Zeilberger. The Goulden-Jackson cluster method: extensions, applications and implementations. *Journal of Difference Equations and Applications*, 5:355–377, 1999. [Link]. 87

[28] Ying-Hui Shao et al. Comparing the performance of FA, DFA and DMA using different synthetic long-range correlated time series. *Preprint*, pages 1–9, 2018. arXiv:1208.4158 [Link]. 34

[29] Grzegorz Sikora. Statistical test for fractional Brownian motion based on detrending moving average algorithm. *Chaos, Solitons & Fractals*, 116:54–62, 2018. [Link]. 34

[30] E.C. Titchmarsh and D.R. Heath-Brown. *The Theory of the Riemann Zeta-Function*. Oxford Science Publications, second edition, 1987. 65

[31] Maistrenko V et al. Chaotic synchronization and antisynchronization in coupled sine maps. *International Journal of Bifurcation and Chaos*, 15:2161–2177, 2005. [Link]. 36

[32] Luyao Wang and Hai Cheng. Pseudo-random number generator based on logistic chaotic system. *Entropy*, 21, 2019. [Link]. 78

[33] David Williams. *Probability with Martingales*. Cambridge University Press, 1991. 93

[34] Lan Wua, Yongcheng Qi, and Jingping Yang. Asymptotics for dependent Bernoulli random variables. *Statistics and Probability Letters*, pages 455–463, 2012. [Link]. 14

[35] Gordan Žitkovic. *Introduction to Stochastic Processes*. University of Texas, 2019. Lecture notes [Link]. 93

Index