```
    arr_min.append(tmin)
    arr_max.append(tmax)
    print("n: %5d tmin: %12.9f tmax: %12.9f" % (n, tmin, tmax))

plt.plot(xval, arr_min, linewidth = 0.3, alpha=1)
plt.plot(xval, arr_max, linewidth = 0.3, alpha=1)
plt.axhline(y=0.0, color='r', linewidth=0.4)
plt.show()
```

5.1.3 Problem and solution

Write a version of the Python code that performs exact computations when $y_0 = p/q$ is a rational number with p, q coprime and $p < q$. In this case, $T_m(y_0)$ is also rational number, with numerator and denominator denoted respectively as p_n and q_n, with $p_n < q_n$. Again, $m = \tau 2^n$. Try $(p, q) = (3, 5), (1, 3)$ and $(1, 4)$. Look at $q_n - p_n$ and how close it can get to zero as n grows, depending on τ. If too close to zero for a specific τ and large n, then $x_0 = (2\pi)^{-1} \arccos y_0$ may not be normal. Find patterns in p_n. Note that $q_n = q^m$.

Below is my version for the code. But it only works with small values of n as the number of digits in p_n, q_n grows extremely fast when n increases, quickly eating all the available memory. Don't look at my solution until after you wrote and tested your code. Hopefully, you can write a version that works with bigger n, or a least be able to find patterns in $q_n - p_n$ even if you cannot compute all the digits. Even better, find patterns confirming that x_0 is normal in base 2, after a formal proof. My code normal_numbers.py is posted online, here.

```
import numpy as np
import gmpy2

ctx = gmpy2.get_context()
ctx.precision = 10000

# y0 = p/q
p = gmpy2.mpz(3)
q = gmpy2.mpz(5)
qn = q

# array with 'dtype=object' to store bigint
A = np.array([[2*p, -q], [q,0]], dtype=object)
y0 = np.array([q, p], dtype=object)
tau = 1
A = np.linalg.matrix_power(A, tau)
numlog = 0
denumlog = 0
delta = 0

for n in range(0, 25):
    T = np.matmul(A, y0)
    num = T[0]//q
    denum = qn**tau
    if n > 0:
        numlog = gmpy2.log(abs(num))
        denumlog = gmpy2.log(abs(denum))
    T_frac = gmpy2.mpfr(num/denum)
    if n > 0:
        delta = gmpy2.log2(1-T_frac)/n
    print("n: %5d T_frac: %12.9f delta: %12.9f numlog: %15f denumlog: %15f"
          % (n, T_frac, delta, numlog, denumlog))
    qn = qn * qn
    A = np.matmul(A, A)
```

5.2 Another interesting discrete quadratic dynamical system

Starting with $p_0 = 1$ and $q_0 = 2$, I build a sequence (p_n, q_n) with the three steps below, in that order at each iteration, based on two positive integer parameters μ, ν and an associated sequence of integers (δ_n) discussed later:

$$p^*_{n+1} = (p_n + q_n)^2 + \delta_n, \quad (5.12)$$
$$q_{n+1} = 2^\mu \cdot q_n^2, \quad (5.13)$$
$$p_{n+1} = \varphi(p^*_{n+1}, q_n, \nu), \quad (5.14)$$

The quantities of interest are

$$r_n = \frac{p_n}{q_n} \quad \text{and} \quad x = \lim_{n\to\infty} r_n \tag{5.15}$$

For now, let $\delta_n = 0$. The function $\varphi(p, q, \nu)$ is defined as follows, where // stands for the integer division:

```
def φ(p, q, ν):
    while p · 2^ν > q:
        p = p // 2
    return(p)
```

Each r_n is a dyadic rational. The purpose is to study the distribution of the binary digits of x. I now discuss two cases, with the first one being more difficult.

5.2.1 Case with multiple limits

When $\nu = 1$ and $\mu = 0$, the limit x does not exists; r_n converges to different values depending on n mod 3. The first few values are

$$p_0 = 1,\ p_1 = 2,\ p_2 = 4,\ p_3 = 100,\ p_4 = 31684,\ p_5 = 1181466050$$
$$q_0 = 2,\ q_1 = 4,\ q_2 = 16,\ q_3 = 256,\ q_4 = 65536,\ q_5 = 4294967296$$

with $q_n = 2^{2^n}$ and as $n \to \infty$, for $r_n = p_n/q_n$, we have:

$$r_{3n+1} \to x_1 = 0.49675930195877117995345323812182255829526005105504674213512\,8\ldots \tag{5.16}$$
$$r_{3n+2} \to x_2 = 0.28003605100001349555214242422455185478475023210296033045544\,07\ldots \tag{5.17}$$
$$r_{3n} \to x_3 = 0.40962307296492728762697503651534274184503107234876373742033\,0\ldots \tag{5.18}$$

The limits x_1, x_2, x_3 are the only real solutions in $[0, 1]$, respectively to the following equations:

$$x_1^8 + 8x_1^7 + 60x_1^6 + 248x_1^5 + 1446x_1^4 + 4280x_1^3 + 16124x_1^2 - 237880x_1 + 113569 = 0 \tag{5.19}$$
$$x_2^8 + 8x_2^7 + 44x_2^6 + 152x_2^5 + 537x_2^4 + 1272x_2^3 + 2892x_2^2 - 29208x_2 + 7921 = 0 \tag{5.20}$$
$$x_3^8 + 8x_3^7 + 44x_3^6 + 152x_3^5 + 662x_3^4 + 1784x_3^3 + 4684x_3^2 - 59416x_3 + 23409 = 0 \tag{5.21}$$

These equations sound magical, but they can be re-written in a different form that shows the mechanics behind the scene:

$$x_1 = \frac{1}{4}\Big(1 + \frac{1}{4}\Big[1 + \frac{1}{8}\Big(1 + x_1\Big)^2\Big]^2\Big)^2 \tag{5.22}$$
$$x_2 = \frac{1}{8}\Big(1 + \frac{1}{4}\Big[1 + \frac{1}{4}\Big(1 + x_2\Big)^2\Big]^2\Big)^2 \tag{5.23}$$
$$x_3 = \frac{1}{4}\Big(1 + \frac{1}{8}\Big[1 + \frac{1}{4}\Big(1 + x_3\Big)^2\Big]^2\Big)^2 \tag{5.24}$$

5.2.2 Case with single limit

In many examples where convergence occurs, the limit is unique. This is the case if $\mu = 0, \nu = 10$, or $\mu = 4, \nu = 3$. Convergence also depends on the initial conditions, set here to $p_0 = 1$ and $q_0 = 2$. We then have

$$x = \lim_{n\to\infty} \frac{p_n}{q_n} = 2^\nu - 1 - \sqrt{4^\nu - 2^{\nu+1}}. \tag{5.25}$$

Interestingly, the limit does not depend on μ nor on the initial conditions. Also, x is solution of the quadratic equation

$$x = \frac{1}{2^{\nu+1}}\Big(1 + x\Big)^2 \tag{5.26}$$

where the right side is a simplified version of (5.22), (5.23) and (5.24). On average, at each iteration n, we gain about ν binary digits in approximating the limit (5.25), though the exact number can be as low as zero (but not negative), or rather large. There are three sub-cases, depending on the type of digits gained at each iteration: .

- **Non-standard**: The extra digits gained are either 1, 11, 100, 101, 110, 111, or a string of digits starting with 1 and containing far more 0's. This is the case if $\mu = 4$ with $\nu = 3$.
- **Standard**: This is the flip side of the non-standard case, with 0, 00, 01, and strings starting with 0 and containing far more 1's, dominating the scene. Example: $\mu = 3$ with $\nu = 3$.

- **Random**: The patterns are much weaker, with 0 and 1 blending in a much less predictable way, resulting in far more diversity in the digit strings added at each iteration. It occurs with larger ν, for instance $\mu = 0$ with $\nu = 10$.

The examples in the first two sub-cases illustrate two different types of convergence to the exact same limit. I discuss the implications regarding the digit distribution in section 5.4. However, before getting into to the details, I present a new theorem that dwarfs everything known so far about the digits of algebraic numbers and other math constants such as π or e. Despite being a multi-century old problem, very little is known to this day, no deep results, not even whether the proportion of 0 or 1 actually exists or if it oscillates without ever converging as the number of digits increase.

The most recent material on this topic is found in [5, 14, 38] and throughout this book. The new theorem 5.3.1 is published here for the first time. While generic, applicable to almost all numbers and relatively easy to prove with mechanical help, it offers a new, spectacular perspective on the subject. It sets a new bar for future discoveries. In particular, to qualify as "deep", any future result would have to be stronger than my theorem.

5.3 Surprising results about the digit distribution

Let $\rho_n(x)$ be the proportion of digits equal to 1 in the first n binary digits of a real number $x \in [0, 1]$, not a dyadic rational. Here I prove that for infinitely many values of n, we asymtotically have $\frac{1}{4} \leq \rho_n(x) \leq \frac{3}{4}$ for at least one of the following two numbers: x or $x' = \lambda + x$ with $\lambda = \frac{2}{3}$. This is the strongest result known to date for standard math constants such as e, π or $\sqrt{2}$. You cannot get tighter upper or lower bounds by choosing a different rational number λ or by refining the methodology proposed in this section. In particular, if for a specific x, the proportion of 0 or 1 actually exists, that proportion must be between $\frac{1}{4}$ and $\frac{3}{4}$, either for x or $\frac{2}{3} + x$, or for both. I now state this result, and provide a computer-assisted proof.

Theorem 5.3.1 *Let $\rho_n(x)$ be the proportion of 1 in the first n binary digits of a real number $x \in [0, 1]$, not a dyadic rational. Also, let $x' = \frac{2}{3} + x$. Then, for infinitely many values of n, we have $\frac{1}{4} \leq \rho_n \leq \frac{3}{4}$ either for x or x', or for both of them. The same is true for the proportion of 0.*

Proof

The idea behind the proof is simple: if a number x, say $\sqrt{2}/8$ has too few 1 in its binary digit expansion, say less than 25% (no one knows), then adding $2/3 = 0.10101010\ldots$ will increase the number of 1 to at least 25%. So either x or $\frac{2}{3} + x$ has at least 25% of 1. But the problem is complicated by the carry over operations which can spread from right to left and annihilate the desired result. To avoid this, one may focus on the first $n + 1$ digits where the rightmost digit in position $n + 1$, is zero. The carry over coming from further on the right side may impact the 0 digit in location $n + 1$, but not the first n digits unless all the digits of 2/3 are 1, which is not the case.

This assumes that the digit 0 appears infinitely many times in the digit expansion of x, even if more and more rarely over time. Then, the 25% threshold occurs at infinitely many locations n before a 0 digit. This is true if x is not a dyadic rational. The same principle applies to all components of the proof, whether x has too few or too many 0 or 1. And the 25% threshold is an absolute bound that cannot be improved. I now formalize the proof. It proceeds by looking, for a fixed n, at all possible digit sequences of length n, containing exactly k ones, for $k = 0, 1, 2$ and so on. Then proving that if the statement is valid for a certain n, it remains true for the next n (proof by recurrence). I also need to do the same analysis by swapping the roles of 0 and 1.

Let $S_1(n, k)$ be a bit string of length n with k ones and $n - k$ zeros. There are $\binom{n}{k}$ such strings, with one of them matching the first n binary digits of x. For ease of discussion, $S_1(n, k)$ is represented as a binary decimal number, for instance '10110100' = 0.10110100_2. Now let $S'_1(n, k) = (\frac{2}{3} + S_1(n, k)) \bmod 1$ truncated to n digits and turned back into a bit string. For instance,

$$\begin{aligned} 2/3 + \text{`10110100'} &= (0.10101010_2 + 0.10110100_2) \bmod 1 \\ &= 1.01011110_2 \bmod 1 \\ &= 0.01011110_2 \\ &= \text{`01011110'}. \end{aligned} \tag{5.27}$$

Given n, let's look at all bit strings $S_1(n, k)$ with $k/n < 25\%$. I now show that for all of them, $S'_1(n, k)$ has at least 25% and at most 75% of 1. Also, I identify which strings $S_1(n, k)$ lead to the lowest and highest proportion of 1 in $S'_1(n, k)$. The Python code below does the computations for any n, and Table 5.1 shows the results. To conclude the proof, one need to show that the result obtained for a given n applies to $n + 1, n + 2$ and so on, using the recurrence principle. We have 4 separate cases, depending on n modulo 4. Also, we need to prove the same result for all strings $S_1(n,k)$ with $k/n > 75\%$. The next step consists of interpreting and generalizing Table 5.1.

n	k	Flag	$S_1(n,k)$	$S_1'(n,k)$	k'	ρ'	Pattern				
20	5	min_00	01010101010000000000	11111111111010101010	5	0.25	1	3	5	7	9
20	5	max_00	01010101100000000000	10101010100000000000	15	0.75	1	3	5	7	8
20	5	min_01	01010101100000000000	10101010100000000000	5	0.25	1	3	5	7	8
20	5	max_01	01010101010000000000	11111111111010101010	15	0.75	1	3	5	7	9
20	5	min_10	11111111110101010101	10101010011111111111	5	0.25	10	12	14	16	18
20	5	max_10	01010101011111111111	10101010010000000000	15	0.75	0	2	4	6	8
20	5	min_11	01010101011111111111	10101010010000000000	5	0.25	0	2	4	6	8
20	5	max_11	11111111110101010101	10101010011111111111	15	0.75	10	12	14	16	18
20	4	min_00	01010101000000000000	11111111101010101010	6	0.30	1	3	5	7	
20	4	max_00	01010110000000000000	10101010101000000000	14	0.70	1	3	5	6	
20	4	min_01	01010110000000000000	10101010101000000000	6	0.30	1	3	5	6	
20	4	max_01	01010101000000000000	11111111101010101010	14	0.70	1	3	5	7	
20	4	min_10	11111111111101010101	10101010100111111111	6	0.30	12	14	16	18	
20	4	max_10	01010101111111111111	10101010100100000000	14	0.70	0	2	4	6	
20	4	min_11	01010101111111111111	10101010100100000000	6	0.30	0	2	4	6	
20	4	max_11	11111111111101010101	10101010100111111111	14	0.70	12	14	16	18	
20	3	min_00	01010100000000000000	11111110101010101010	7	0.35	1	3	5		
20	3	max_00	01011000000000000000	10101010101010000000	13	0.65	1	3	4		
20	3	min_01	01011000000000000000	10101010101010000000	7	0.35	1	3	4		
20	3	max_01	01010100000000000000	11111110101010101010	13	0.65	1	3	5		
20	3	min_10	11111111111111010101	10101010101001111111	7	0.35	14	16	18		
20	3	max_10	01010111111111111111	10101010101001000000	13	0.65	0	2	4		
20	3	min_11	01010111111111111111	10101010101001000000	7	0.35	0	2	4		
20	3	max_11	11111111111111010101	10101010101001111111	13	0.65	14	16	18		
19	4	min_00	0101010100000000000	1111111110101010101	5	0.26	1	3	5	7	
19	4	max_00	0101011000000000000	1010101010100000000	13	0.68	1	3	5	6	
19	4	min_01	0101011000000000000	1010101010100000000	6	0.32	1	3	5	6	
19	4	max_01	0101010100000000000	1111111110101010101	14	0.74	1	3	5	7	
19	4	min_10	1111111111110101010	1010101010011111111	6	0.32	12	14	16	18	
19	4	max_10	0101010111111111111	1010101010000000000	14	0.74	0	2	4	6	
19	4	min_11	0101010111111111111	1010101010000000000	5	0.26	0	2	4	6	
19	4	max_11	1111111111110101010	1010101010011111111	13	0.68	12	14	16	18	
18	4	min_00	010101010000000000	111111111010101010	5	0.28	1	3	5	7	
18	4	max_00	010101100000000000	101010101000000000	13	0.72	1	3	5	6	
18	4	min_01	010101100000000000	101010101000000000	5	0.28	1	3	5	6	
18	4	max_01	010101010000000000	111111111010101010	13	0.72	1	3	5	7	
18	4	min_10	111111111101010101	101010100111111111	5	0.28	10	12	14	16	
18	4	max_10	010101011111111111	101010100100000000	13	0.72	0	2	4	6	
18	4	min_11	010101011111111111	101010100100000000	5	0.28	0	2	4	6	
18	4	max_11	111111111101010101	101010100111111111	13	0.72	10	12	14	16	

Table 5.1: Extreme bit strings leading to ratio $\rho' = k'/n$ closest the the bounds 0.25 or 0.75

Table 5.1 covers all the extreme cases for various bit strings $S_1(n,k)$, where the number of 0 in $S_n'(k,n)$ (and by symmetry the number of 1) is either minimum or maximum. Explaining and generalizing all the patterns may be long, but in the end it is a straightforward and purely mechanical, finite process. Here is a summary:

- The most extreme cases are when k/n is closest to 25% or 75%, reaching an absolute minimum or maximum when $n \bmod 4 = 0$.
- In the table, the max_ab and min_ab columns have the following meaning depending on $a, b \in \{0, 1\}$.
  - If $b = 1$, then k' counts the number of 1 in $S_1'(n,k)$. Otherwise, it counts the number of 0.

  - If $a = 1$, the string $S_1(n, k)$ is extreme in terms of the large number of 1 that it contains. Otherwise it is extreme due to its large number of 0.
  - All strings $S_1(n, k)$ contain either less than 25%, or more than 75% of 0. However, the corresponding $S'_1(n, k)$ has ratios of 0 or 1 always in the prescribed range [0.25, 0.75]. This ratio is denoted as ρ' in the table, and equal to k'/n.
- For a fixed n, the smaller k, the less extreme k', thus the less extreme ρ'. While not shown in the table, $\rho' = 50\%$ if $k = 0$ (this is trivial). Conversely, If $\rho = k/n \in]0.25, 0.75[$, then $\rho' \notin [0.25, 0.75]$. So in short, either $S_1(n, k)$ has $\rho \in [0.25, 0.75]$ or $S'_1(n, k)$ has $\rho' \in [0.25, 0.75]$, or both.
- The 'Pattern' column in the table shows the locations of 0 in the string $S(n, k)$ which, given k and n is the most extreme. Here the first position is indexed as position 0. These patterns are straightforward, with the locations being the same as n increases.

Conclusion: At all times, we stay in the 25% to 75% range for $S'_1(n, k)$ with the lower and upper bounds reached exactly when $n \bmod 4 = 0$. Thus this interval cannot be reduced. $\blacksquare$

The code to produce Table 5.1 is listed in section 5.3.1, and also available on GitHub, here. But first, let's see if we can get an even stronger result. Obviously, you need more than the one alternative $\frac{2}{3} + x$ to x, in order to guarantee a proportion of 0 and 1 in a range narrower than $[0.25, 0.75]$. Clearly, in the most extreme case when both x and $x' = \frac{2}{3} + x$ have a proportion of 0 or 1 exactly equal to 25% or 75%, there is a third number, namely $x'' = \frac{1}{3} + x$, for which these ratios are in a narrower interval. This is summarized in the following theorem, stronger than theorem 5.3.1.

Theorem 5.3.2 *Let $\rho_n(x)$ be the proportion of 1 in the first n binary digits of a real number $x \in [0, 1]$, not a dyadic rational. Let $x' = \frac{2}{3} + x$, and $x'' = \frac{1}{3} + x$. Then, for infinitely many values of n, we have $\frac{5}{16} \leq \rho_n \leq \frac{11}{16}$ for at least one of the numbers x, x' or x''. The same is true for the proportion of 0.*

I am still working on a proof, similar to that of theorem 5.3.1 but longer, this time assisted with the Python code in section 5.3.2. The bounds $\frac{5}{16}$ and $\frac{11}{16}$ cannot be improved. They are attained when $n = 16$ and $k = 5$. You can get a minor improvement if x is a specific number not matching the patterns associated with the most extreme $S_1(n, k)$. But for any fundamental math constant such as $x = \pi$ or $x = \sqrt{2}/2$, showing the absence of such patterns in the binary digit expansion would be extremely difficult. The most extreme cases include (among several dozens) rational numbers x with the period $101011000001001100000010_2$ and irrational numbers that asymptotically have an identical digit distribution to x.

5.3.1 Python code for the computer-assisted proof of the main theorem

This is the code used to establish Theorem 5.3.1, produce Table 5.1 and search for the extreme strings in all $\binom{n}{k}$ combinations of n-bit strings. The code is also on GitHub, here.

```
import numpy as np
import gmpy2

def combinations_lexicographic_list(n, k):

   # Return a list with all k-combinations of range(n) in lexicographic order.
   # Each combination is a tuple of indices (0..n-1).

   if k < 0 or k > n:
      return []

   # initial combination: [0, 1, ..., k-1]
   c = list(range(k))
   cnt_00_min = 2*n
   cnt_00_max = -1
   cnt_01_min = 2*n
   cnt_01_max = -1
   cnt_10_min = 2*n
   cnt_10_max = -1
   cnt_11_min = 2*n
   cnt_11_max = -1
   flag = ''
   hash = {}
   print("\nFinding extremes, exhaustive search... \n")

   while True:

      str_0 = ''
```

```
        str_1 = ''
        for idx in range(n):
            if idx in c:
                str_0 += '1'
                str_1 += '0'
            else:
                str_0 += '0'
                str_1 += '1'

        x0 = two_third + gmpy2.mpz(str_0, 2)/2**n
        if x0 >= 1:
            x0 -= 1
        x0_bin = gmpy2.digits(x0, 2)[0]
        x0_bin = x0_bin[0:n]

        x1 = two_third + gmpy2.mpz(str_1, 2)/2**n
        if x1 >= 1:
            x1 -= 1
        x1_bin = gmpy2.digits(x1, 2)[0]
        x1_bin = x1_bin[0:n]

        cnt_00 = x0_bin.count('0')
        cnt_01 = x0_bin.count('1')
        cnt_10 = x1_bin.count('0')
        cnt_11 = x1_bin.count('1')

        combo = np.copy(c)

        if cnt_00 < cnt_00_min:
            cnt_00_min = cnt_00
            hash['min_00'] = (cnt_00, str_0, x0_bin, combo)
            flag += 'a'
        if cnt_00 > cnt_00_max:
            cnt_00_max = cnt_00
            hash['max_00'] = (cnt_00, str_0, x0_bin, combo)
            flag += 'b'
        if cnt_01 < cnt_01_min:
            cnt_01_min = cnt_01
            hash['min_01'] = (cnt_01, str_0, x0_bin, combo)
            flag += 'c'
        if cnt_01 > cnt_01_max:
            cnt_01_max = cnt_01
            hash['max_01'] = (cnt_01, str_0, x0_bin, combo)
            flag += 'd'

        if cnt_10 < cnt_10_min:
            cnt_10_min = cnt_10
            hash['min_10'] = (cnt_10, str_1, x1_bin, combo)
            flag += 'A'
        if cnt_10 > cnt_10_max:
            cnt_10_max = cnt_10
            hash['max_10'] = (cnt_10, str_1, x1_bin, combo)
            flag += 'B'
        if cnt_11 < cnt_11_min:
            cnt_11_min = cnt_11
            hash['min_11'] = (cnt_11, str_1, x1_bin, combo)
            flag += 'C'
        if cnt_11 > cnt_11_max:
            cnt_11_max = cnt_11
            hash['max_11'] = (cnt_11, str_1, x1_bin, combo)
            flag += 'D'

        if flag != '':
            print(c, str_0, x0_bin, cnt_00, cnt_01,
                     str_1, x1_bin, cnt_10, cnt_11, flag)
            flag = ''

        # find rightmost element that can be incremented
        i = k - 1
        while i >= 0 and c[i] == i + (n - k):
            i -= 1

        if i < 0:
            # all combinations generated
            break
```

```
            # increment this element
            c[i] += 1

            # reset the tail to the minimal increasing sequence
            for j in range(i + 1, k):
                c[j] = c[j - 1] + 1

    return(hash)


#--- Main

n = 18
k = 4
str = ''
for idx in range(n):
    if idx % 4 in (0,2):
        str += '1'
    else:
        str += '0'

ctx = gmpy2.get_context()
ctx.precision = 2*n
two_third = gmpy2.mpz(str, 2)/2**n
str2 = gmpy2.digits(two_third, 2)[0]
str2 = str2[0:n]
print("2/3, truncated string:",str2)

hash = combinations_lexicographic_list(n, k)
print("\nSummary\n")

for key in hash:

    value = hash[key]
    count = value[0]
    before = value[1]
    after = value[2]
    combo = value[3]
    combo = [f"{t}" for t in combo]
    combo = ' '.join(combo)
    rho = count/n
    print("%3d %3d | %s %s %s %3d %4.2f| %s" %(n, k, key, before, after, count, rho, combo))
```

5.3.2 Python code for the deeper theorem

This code is also available on GitHub, here. It is used in connection to theorem 5.3.2.

```
import gmpy2

def combinations_lexicographic_list(n, k, thresh, digit):

    # Return a list with all k-combinations of range(n) in lexicographic order.
    # Each combination is a tuple of indices (0..n-1).

    # initial combination: [0, 1, ..., k-1]
    c = list(range(k))
    print("\nFinding extremes, exhaustive search... \n")

    while True:

        stri = ''
        for idx in range(n):
            if idx in c:
                stri += str(digit) # '0' or '1'
            else:
                stri += str(1-digit)

        x0 = two_third + gmpy2.mpz(stri, 2)/2**n
        if x0 >= 1:
            x0 -= 1
        x0_bin = gmpy2.digits(x0, 2)[0]
        x0_bin = x0_bin[0:n]

        y0 = one_third + gmpy2.mpz(stri, 2)/2**n
```

```
        if y0 >= 1:
           y0 -= 1
        y0_bin = gmpy2.digits(y0, 2)[0]
        y0_bin = y0_bin[0:n]

        ratio_1 = k / n
        ratio_2 = x0_bin.count('0') / n
        ratio_3 = y0_bin.count('0') / n
        flag_1 = False
        flag_2 = False
        flag_3 = False

        if thresh < ratio_1 < 1 - thresh:
           flag_1 = True
        if not thresh < ratio_2 < 1 - thresh:
           flag_2 = True
        if not thresh < ratio_3 < 1 - thresh:
           flag_3 = True

        if not flag_1 and flag_2 and flag_3:
           print(stri, x0_bin, y0_bin, ratio_1, ratio_2, ratio_3)

        # find rightmost element that can be incremented
        i = k - 1
        while i >= 0 and c[i] == i + (n - k):
           i -= 1

        if i < 0:
           # all combinations generated
           break

        # increment this element
        c[i] += 1

        # reset the tail to the minimal increasing sequence
        for j in range(i + 1, k):
           c[j] = c[j - 1] + 1

     return()

#--- Main

n = 16
k = 5
stri = ''
xtri = ''
for idx in range(n):
   if idx % 4 in (0,2):
      stri += '1'
      xtri += '0'
   else:
      stri += '0'
      xtri += '1'

ctx = gmpy2.get_context()
ctx.precision = 2*n
two_third = gmpy2.mpz(stri, 2)/2**n
one_third = gmpy2.mpz(xtri, 2)/2**n
thresh = 0.35
digit = 1 # options: 0 or 1
```

5.4 Strong patterns found in the digits of algebraic numbers

Now, let's get back to the sequence (p_n, q_n) defined recursively in section 5.2 using a quadratic map (discrete dynamical system) with formulas (5.12), (5.13) and (5.14), along with the initial conditions $p_0 = 1, q_0 = 2$ and parameters μ, ν. In all cases, q_n is a power of 2. Thus $r_n = p_n/q_n$ is a dyadic rational and p_n is an integer whose binary digits match those of a known algebraic number as $n \to \infty$.

The case $\nu = 1, \mu = 2$ is interesting in the sense that the ratio r_n converges not just to one but actually three different algebraic numbers (5.16), (5.17), and (5.18) depending on n mod 3, jumping from one to another in a circular loop as n increases. The new digits gained at each iteration, and the patterns attached to them, as well as patterns across the three digit sequences, will be published in an upcoming paper.

		$\mu = 3$				$\mu = 4$	
c_0	c_1	freq.	digits	c_0	c_1	freq.	digits
766	0	383	00	0	360	360	1
1120	0	354	0	0	360	351	
1457	337	337	01	341	701	341	10
1457	337	325		341	1353	326	11
1779	981	322	011	977	1671	318	100
2321	1252	271	010	1271	2259	294	101
2741	1462	210	001	1856	2454	195	1000
2903	1948	162	0111	2023	2788	167	110
3173	2218	135	0110	2335	3100	156	1001
3280	2646	107	01111	2335	3448	116	111
3583	2646	101	000	2771	3557	109	10000
3781	2844	99	0101	2971	3757	100	1010
3913	3042	66	01110	3166	3887	65	10001
4105	3106	64	0100	3212	4025	46	1011
4211	3265	53	01101	3347	4115	45	10010
4256	3490	45	011111	3572	4160	45	100000
4355	3556	33	01100	3740	4244	42	100001
4417	3680	31	011110	3852	4300	28	100010
4475	3738	29	0011	3908	4356	28	1100
4523	3858	24	0111110	4064	4382	26	1000000
4589	3924	22	011100	4106	4445	21	10011
4609	4044	20	0111111	4201	4483	19	1000001
4645	4098	18	01011	4252	4534	17	100011
4677	4162	16	011101	4357	4549	15	10000000
4703	4227	13	0111101	4422	4575	13	1000010
4736	4271	11	0111100	4455	4597	11	10100
4747	4348	11	01111111	4487	4613	8	100100
4755	4412	8	011111111	4519	4637	8	1000011
4767	4448	6	01111101	4554	4658	7	10000011
4785	4460	6	01010	4603	4672	7	100000001
4795	4495	5	011111101	4645	4686	7	10000010
4805	4525	5	01111110	4701	4693	7	100000000
4815	4550	5	0111011	4707	4711	6	1101
4825	4570	5	011011	4731	4719	4	10000001
4833	4602	4	0111111110	4755	4725	3	1000000001
4845	4606	4	0010	4770	4731	3	1000100
4851	4627	3	011111110	4779	4740	3	100101
4854	4657	3	01111111111	4799	4744	2	100000000001
4863	4672	3	01111100	4821	4746	2	100000000000
4869	4678	2	011010	4825	4752	2	10101
4871	4680	1	1001	4845	4754	2	10000000000
4874	4682	1	01001	4847	4754	1	00
4875	4691	1	0111111111	4851	4757	1	1000101
4878	4695	1	0111010	4855	4760	1	1000110
4879	4706	1	011111111111	4861	4763	1	100000011
4881	4718	1	01111111111110	4869	4765	1	1000000100
4882	4730	1	0111111111111	4876	4767	1	100000100
4885	4734	1	0111001	4888	4769	1	10000000000001
4887	4743	1	01111111101	4894	4771	1	10000100
4889	4755	1	01111111111011				
4891	4762	1	011111011				
4893	4772	1	011111111101				

Table 5.2: New digit blocks added at each iteration, ordered by frequency

However, at this stage, the main interest is about the case $\nu = 3$ with two sub-cases: $\mu = 3$ and $\mu = 4$. Both feature intriguing and rare patterns about how new digits are being added as precision increases, but wildly different depending on μ. In both sub-cases, r_n converges to the single limit $7 - 4\sqrt{3}$, gaining on average about

$\nu = 3$ new binary digits at each iteration. The binary digits were computed with the code listed in section 5.4.1. The findings are summarized in Table 5.2.

The columns c_0 and c_1 in Table 5.2 show the cumulative number of binary digits, respectively 0 and 1, when aggregated over all the digit blocks ordered by frequency, for $\mu = 3$ (left) and $\mu = 3$ (right). A digit block is a set a new digits matching those of $7 - 4\sqrt{3}$, uncovered when increasing n to $n + 1$ in the iterative computation of $r_n = p_n/q_n$. The rows where the digits column is empty represent iterations that did not result in increasing the precision. The total number of correct digits after 3,300 iterations is about 10,000. The exact number is $c_0 + c_1$ computed on the bottom row of the table. Interestingly, the cases $\mu = 3$ and $\mu = 4$ lead to different mechanisms to sequentially generate the digit blocks, but in the end they both produce the same digit sequence representing the same constant.

There is a strong and seemingly permanent imbalance or bias in the digit block production. Yet in the end everything balances out to produce a digit sequence (the binary digits of $7 - 4\sqrt{3}$) that looks perfectly random. The bias in question can be leveraged to find bounds on the proportion of 0 and 1 in the digits attached to that number. Let's focus on $\mu = 4$ to illustrate.

- There are 2274 digit blocks of length up to 3, and 1059 with length ≥ 4.
- The small blocks contain aggregated totals of 2941 ones and 1440 zeros.
- The large blocks contain aggregated totals of 1830 ones and 3454 zeros.
- Out of 9665 digits, 4381 come from the small blocks, and 5384 from the large ones.

So, if we could prove that about 50% of the digits come from the large blocks, with about 2/3 of them being 0, then it would prove that at least 1/3 of the digits of $7 - 4\sqrt{3}$ are zero. A similar argument holds for the proportion of 1, by looking at $\mu = 3$. This would be a deep result customized to $7 - 4\sqrt{3}$, and stronger than theorem 5.3.1 applicable to all numbers, but weaker than theorem 5.3.2 also applicable to all numbers. Finally, the next steps consists in looking at pairs of consecutive blocks in the binary digit expansion, and check whether the pairs are randomly distributed or not. Likewise, departure from randomness could help us find leverages to prove deeper results about the digit distribution.

5.4.1 Python code to compute the digits

The code in this section features the non-standard sub-case in section 5.2.2. It is also on GitHub, here. As in previous chapters, it relies on truncations to keep the minimum amount of digits that guarantees the desired level of precision.

```
import numpy as np
import gmpy2

ctx = gmpy2.get_context()
ctx.precision = 10000
ndigits = ctx.precision
nmatch = 0
newdigits = ''
hash_newdigits = {}
hash_pairs = {}

p = gmpy2.mpz(1)
q = gmpy2.mpz(2)
mu = 4
nu = 3
N = ndigits // nu

lim = 2**nu - 1 - gmpy2.sqrt(4**nu - 2**(nu+1))
str_lim = gmpy2.digits(lim, 2)[0]
str_lim = str_lim[0:ndigits]

def update_hash(hash, key, count):
   if key in hash:
      hash[key] += count
   else:
      hash[key] = count
   return()

def count_matching_prefix_chars(str1, str2):
   count = 0
   for char1, char2 in zip(str1, str2):
      if char1 == char2:
         count += 1
```

```python
        else:
            break
    return(count)

def phi(p, q, nu):
    while p * 2**nu > q:
        p = p // 2
    return(p)

#--- 1. Main

for k in range(N):

    p = (p+q)**2
    q = 2**mu *q**2

    str_p = gmpy2.digits(p, 2)
    str_p = str_p[0:ndigits]
    p = gmpy2.mpz(str_p, 2)

    str_q = gmpy2.digits(q, 2)
    str_q = str_q[0:ndigits]
    q = gmpy2.mpz(str_q, 2)

    old_p = p
    p = phi(p, q, nu)

    old_nmatch = nmatch
    nmatch = count_matching_prefix_chars(str_p, str_lim)
    old_newdigits = newdigits
    newdigits = str_p[old_nmatch:nmatch]
    pair = (old_newdigits, newdigits)
    update_hash(hash_pairs, pair, 1)
    x = p/q
    update_hash(hash_newdigits, newdigits, 1)
    if k % 1000 == 0:
        print("New digits:", k, nmatch, newdigits)

#--- 2. Summary results

print("\n\n")
hash_newdigits = dict(sorted(hash_newdigits.items(), key=lambda item: item[1], reverse=True))
c1 = 0 # counts digits equal to 1
c0 = 0 # counts digits equal to 0
for key in hash_newdigits:
    noccur = hash_newdigits[key]
    c1 += noccur * key.count('1')
    c0 += noccur * key.count('0')
    print("New digits hash summary:",c0, c1, noccur, key)

print("\n\n")
hash_pairs = dict(sorted(hash_pairs.items(), key=lambda item: item[1], reverse=True))
for pair in hash_pairs:
    cnt = hash_pairs[pair]
    if cnt > 5:
        print("Pair:", cnt, pair)
```

5.5 Correlated bit strings: seminal result and applications

So far, I looked at individual digit sequences separately. Now I focus on comparing sequences, and more specifically, identifying cross-correlations (or their absense) to build other tests of randomness, and with cryptographic applications in mind. Let $x \in [0,1]$ be a real number. Its digits d_0, d_1 and so on in integer base $b > 1$ are obtained using the following recursion:

$$x_n = \{bx_{n-1}\}, d_n = \lfloor bx_n \rfloor \tag{5.28}$$

where $x_0 = x$. The brackets $\{\cdot\}$ denote the fractional part while $\lfloor \cdot \rfloor$ denotes the integer part function. If x is a normal number, the lag-k autocorrelation in the sequence (x_n), that is, the correlation between the sequences (x_n) and (x_{n+k}), is equal to b^{-k}. However, the autocorrelations of any lag in the digit sequence (d_n) are all zero. See section 3.2 entitled "Probabilistic properties of numeration systems" in [14]. Correlation should be interpreted as the limit of the empirical correlation based on the first n terms in the sequence, as $n \to \infty$. The limit exists if x is a normal number. For the exact formulation and computation, see the code in this section.

A less well-known result is the following.

Theorem 5.5.1 *If $x > 0$ is a normal in base 2 and p, q are odd coprime positive integers, then px, qx are also normal in base 2. The correlation between the binary digits of px and qx is equal to*

$$\rho\Big(px, qx\Big) = \frac{1}{pq} = \rho\Big(x, \frac{qx}{p}\Big) = \rho\Big(x, \frac{px}{q}\Big) \tag{5.29}$$

Proof
As a starting point, it is easy to show that for two normal numbers x, y, the correlation between their binary digit sequences $(d_k(x))$ and $(d_k(y))$ satisfies

$$\rho_n(x, y) := -1 + \frac{4}{n}\sum_{k=0}^{n-1} d_k(x)d_k(y) \;\rightarrow\; \rho(x, y), \quad \text{as } n \to \infty. \tag{5.30}$$

Also, a normal number multiplied by a non-zero rational is normal in the same base (Wall's theorem, 1949; see also [9]). Thus, the binary digit distributions of x, px and qx have the same mean $\frac{1}{2}$ and same variance $\frac{1}{4}$. Not all the equalities in (5.29) need to be proved separately, as we have trivial equivalences, using a change of variables preserving normality, and the fact that $\rho(\cdot\,,\cdot)$ is symmetric. For instance, thanks to the substitution $x \mapsto x/p$, we have

$$\rho\Big(px, qx\Big) = \frac{1}{pq} \implies \rho\Big(x, \frac{qx}{p}\Big) = \frac{1}{pq}.$$

Also, by a symmetry argument, swapping p and q, we have:

$$\rho\Big(x, \frac{px}{q}\Big) = \frac{1}{pq} \iff \rho\Big(x, \frac{qx}{p}\Big) = \frac{1}{pq}.$$

A proof that $\rho(x, px/q) = (pq)^{-1}$ was first published by William Huber on CrossValidated.com in 2019, see here and here. The proof assumes that the binary digits of x are randomly distributed as an infinite Bernoulli trial with 50% of 0 and 1, a stronger assumption than normality in base 2 ∎

Now, if x is a normal number, $\alpha \neq \beta$ are positive integers and p, q are odd coprime positive integers, then we have the following (the proof is left as an exercise):

$$\rho\Big(2^\alpha px, 2^\beta qx\Big) = 0. \tag{5.31}$$

I now can state and prove the following deep result with important implications, for instance the fact that the binary digit sequences of $\sqrt{2}$ and $\sqrt{3}$ are uncorrelated, that is $\rho(\sqrt{2}, \sqrt{3}) = 0$, if both are normal numbers.

Theorem 5.5.2 *Let x, y be normal numbers in base 2, linearly independent over $\mathbb{Q}$. Then $\rho(x, y) = 0$.*

Proof
Linear independence over $\mathbb{Q}$ means that if $\alpha x = \beta y$ for some integers α, β, we must have $\alpha = \beta = 0$. There are infinitely many pairs of sequences $(\alpha_t), (\beta_t)$ such that $y_t = \alpha_t x/\beta_t \to y$ as $t \to \infty$, with α_t, β_t being positive odd coprimes for all t. By virtue of theorem 5.5.1, we have

$$\rho(x, y_t) = \frac{1}{\alpha_t\beta_t}.$$

As t increases, the number of identical digits on the left in x and y_t increases, and thus $\rho(x, y_t) \to \rho(x, y)$. At the same time, $\alpha_t, \beta_t \to \infty$ because there is no rational number r such that $x = ry$ due to x, y being linearly independent over $\mathbb{Q}$. Thus, $\rho(x, y) = 0$. ∎

Formulas (5.29) and (5.31) lead to a new test of randomness. For instance, to check if the binary digits of a number x are random enough, you first compute $\lambda_n(p, q) = \rho_n(px, qx)$, the empirical correlation on the first n digits, for various values of n, p, q. Then run N simulations, replacing the digits of x by N sequences of random bits. Now let $L_n(p, q)$ and $U_n(p, q)$ be the lower and upper correlations computed over the N samples with fixed p, q. If $\lambda_n(p, q) \notin [L_n(p, q), U_n(p, q]$ far more often than expected by chance, then the digits of x are presumed non-random. Also, if for some p, q, the value $(pq)^{-1}$ is not within these two bounds, it means that your random number generator is defective.

Theorem 5.5.2 is particularly useful in the context of strong PRNGs (pseudo-random number generators), with applications to cryptography where replicability is mandatory, or to test the strength of other PRNGs. In particular, the PRNG discussed in section 4.4 in [14] relies on a large number of quadratic irrationals: the square roots of square-free integers. Random bits are generated by

- choosing (say) 10^6 such numbers,
- for each of them extracting 10^4 binary digits starting at a random location in the digit expansion,
- then concatenate all the collected digits to produce 10^{10} random bits.

This PRNG offers up to 10^6 distinct seed pairs. Each pair consists of (1) a square-free integer and (2) the location or index where the binary digit sequence must start in the associated quadratic irrational. Then theorem 5.5.2 guarantees that the 10^6 digit sequences are uncorrelated, if the underlying square root numbers are normal.

Now I share my Python code to compute the correlation between the binary digits of px and qx, where x is a real number in $[0, 1]$ and p, q are positive integers, coprime or not, odd or even. The function `vg_correl` computes the digits backward with carry-over and returns the correlation, while `gmpy2_correl` uses the gmpy2 library to compute the correlation between the digts of x and those of px/q. The code is also on GitHub, here.

The current version of `gmpy2_correl` has a bug caused by `w_offset` not correct when $p > q$. For instance, `gmpy2_correl(z,p,1)` and `vg_correl(z,p,1)` should obviously return the same correlation equal to $1/p$, but only the latter does, due to digits misalignment in the former when $p > q$ (in this example, $q = 1$).

```
# 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 grammar school multiplication

import numpy as np
import gmpy2

kmax = 10000000
ctx = gmpy2.get_context()
ctx.precision = kmax
ndigits = ctx.precision
z = gmpy2.sqrt(2) # in the article, z is denoted as x

# main parameters
seed = 195
np.random.seed(seed)
# p, q odd integers, coprime
p = 3
q = 5

def gmpy2_correl(z, p, q):

   # correl b/w binary digits of z and pz/q (needs p < q)
   zstri = gmpy2.digits(z, 2)[0] # get binary digits of z as a string
   zoff = gmpy2.digits(z, 2)[1]

   w = gmpy2.mpfr(z*p)/gmpy2.mpz(q)
   woff = gmpy2.digits(w, 2)[1]
   w_offset = '0' * (zoff - woff) # works only if p < q
   wstri = w_offset + gmpy2.digits(w, 2)[0]

   prod = 0
   for k in range(kmax):
      d1 = int(zstri[k])
      d2 = int(wstri[k])
      prod += d1*d2
      correl = 4*prod/(k+1) - 1
      if k % 100000 == 0 and k > 100:
         checksum = correl * p * q # should be close to 1
         print("gmpy2> k: %7d correl: %9.7f check: %9.7f" %(k, correl, checksum))
   return(correl)


def vg_correl(z, p, q):

   # correl b/w binary digits of pz and qz
   mode = 'constant' # options: 'random', 'constant'

   # local variables
   zstri = gmpy2.digits(z, 2)[0] # get binary digits of z as a string
   X, pX, qX = 0, 0, 0
   d1, d2, e1, e2 = 0, 0, 0, 0
   prod, count = 0, 0
   sum1 = 0
   sum2 = 0

```

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

      # b is a digit of X
      if mode == 'random':
         b = np.random.randint(0, 2)
      else:
         b = int(zstri[kmax-k-1])
      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
      sum1 += d1
      sum2 += d2
      mean1 = sum1/count
      mean2 = sum2/count
      std1 = (mean1 * (1 - mean1))**0.5
      std2 = (mean2 * (1 - mean2))**0.5
      covar = prod/count - mean1*mean2
      if count > 100:
         correl = covar/(std1*std2)
      else:
         correl = 0
      #correl = 4*prod/count - 1

      if k% 100000 == 0:
         checksum = p*q*correl # should be close to 1
         print("vg>k = %7d, correl = %9.6f checksum = %9.6f" % (k, correl, checksum))

   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)))
   return(correl)

#--- Main

correl1 = gmpy2_correl(z, p, q)
correl2 = vg_correl(z, p, q)
```

Bibliography

[1] Franklin T. Adams-Watters and Frank Ruskey. Generating functions for the digital sum and other digit counting sequences. *Journal of Integer Sequences*, 12:1–9, 2009. [Link]. 12, 14, 23, 32, 45

[2] Christoph Aistleitner et al. Normal numbers: Arithmetic, computational and probabilistic aspects. 2016. Workshop [Link]. 12, 23, 32, 45

[3] Adel Alamadhi, Michel Planat, and Patrick Solé. Chebyshev's bias and generalized Riemann hypothesis. *Preprint*, pages 1–9, 2011. arXiv:1112.2398 [Link]. 84

[4] Gökalp Alpan and Maxim Zinchenko. Lower bounds for weighted Chebyshev and orthogonal polynomials. *Preprint*, 2024. arXiv:2408.11496v [Link]. 56

[5] David Bailey, Jonathan Borwein, and Neil Calkin. *Experimental Mathematics in Action*. A K Peters, 2007. 11, 23, 32, 45, 61

[6] Verónica Becher, A. Marchionna, and G. Tenenbaum. Simply normal numbers with digit dependencies. *Mathematika*, 69:988–991, 2023. arXiv:2304.06850 [Link]. 12, 23, 32, 45

[7] Frederik Broucke. On zero-density estimates for beurling zeta functions. *Preprint*, pages 1–24, 2024. arXiv:2409:1051v1 [Link]. 77

[8] James Dolan. Carrying is a 2-cocycle. *Preprint*, pages 1–9, 2023. [Link]. 12, 23, 32, 45

[9] David Doty, Jack H. Lutz, and Satyadev Nandakumar. Finite-state dimension and real arithmetic. *Information and Computation*, 205:1640–1651, 2007. arXiv:cs/0602032 [Link]. 70

[10] Faiza Firdousi, Syeda Iram Batool, and Muhammad Amin. A novel construction scheme for nonlinear component based on quantum map. *International Journal of Theoretical Physics*, 58:3871–3898, 2019. [Link]. 12, 23, 32, 45

[11] P. M. Gauthier. Approximating the Riemann zeta-function by polynomials with restricted zeros. *Canadian Mathematical Bulletin*, 62(3):475–478, 2018. [Link]. 95

[12] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLT, 2022. [Link]. 77, 94

[13] Vincent Granville. *Synthetic Data and Generative AI*. MLT, 2022. [Link]. 77

[14] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLT, 2023. [Link]. 7, 10, 11, 12, 14, 15, 18, 23, 32, 44, 45, 57, 61, 69, 70, 77, 78, 83, 84

[15] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLT, 2024. [Link]. 15, 23, 32, 45, 100

[16] Vincent Granville. *State of the Art GenAI & LLMs, Creative Projects & Solutions*. MLT, 2024. [Link]. 20, 84

[17] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLT, 2024. [Link]. 78

[18] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 82

[19] Vincent Granville. *Blueprint: Next-Gen Enterprise RAG & LLM 2.0 – Nvidia PDFs Use Case*. 2025. MLT [Link]. 100

[20] Vincent Granville. Simple, efficient, secure, accurate enterprise AI xLLM 2.0 architecture & operating system. 2025. BondingAI internal report `bdai-scores.pdf`, July 2025. 100

[21] Vincent Granville. *No-Blackbox, Secure, Efficient AI and xLLM Solutions*. MLT, 2026. [Link]. 95, 100

[22] Vincent Granville and Richard L Smith. Disaggregation of rainfall time series via Gibbs sampling. *NISS Technical Report*, pages 1–21, 1996. [Link]. 94

[23] Emil Grosswald. Oscillation theorems of arithmetical functions. *Transactions of the American Mathematical Society*, 126:1–28, 1967. [Link]. 84

[24] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 85

[25] 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]. 85

[26] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 85

[27] M. Madritsch and J. Thuswaldner. The level of distribution of the sum-of-digits function of linear recurrence number systems. *Journal de Théorie des Nombres de Bordeaux*, 34:449–482, 2022. MLT [Link]. 32, 45

[28] Vaibhav Mohanty et al. Maximum mutational robustness in genotype–phenotype maps follows a self-similar blancmange-like curve. *The Royal Society Publishing*, pages 1–16, 2023. [Link]. 12, 23, 32, 45

[29] Mohammadamin Moradi et al. Data-driven model discovery with Kolmogorov-Arnold networks. *Preprint*, pages 1–6, 2024. arXiv:2409.15167 [Link]. 24, 32, 45

[30] K.S. Narendra and K. Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1:4–27, 1990. [Link]. 23, 32, 45

[31] Alan Oppenheim and Ronald W. Schafer. *Discrete–Time Signal Processing*. Prentice Hall, 1999. 97

[32] Michel Planat and Patrick Solé. Efficient prime counting and the Chebyshev primes. *Preprint*, pages 1–15, 2011. arXiv:1109.6489 [Link]. 84

[33] Klaus Schiefermayr and Maxim Zinchenko. Norm estimates for Chebyshev polynomials, i. *Journal of Approximation Theory*, 265, 2021. [Link]. 56

[34] Jan-Christoph Schlage-Putcha and Jasson Vindas. The prime number theorem for Beurlings generalized numbers – new cases. pages 1–26, 2011. [Link]. 77

[35] Terence Tao. Biases between consecutive primes. *Tao's blog*, 2016. [Link]. 84

[36] Yury V. Tiumentsev and Mikhail V. Egorchev. *Neural Network Modeling and Identification of Dynamical Systems*. Elsevier, 2019. 23, 32, 45

[37] Chukwudubem Umeano and Oleksandr Kyriienko. Ground state-based quantum feature maps. *Preprint*, pages 1–8, 2024. arXiv:2024.07174 [Link]. 12, 23, 32, 45

[38] Joseph Vandehey. On the binary digits of $\sqrt{2}$. *Preprint*, pages 1–6, 2017. arXiv:1711.01722 [Link]. 11, 23, 32, 45, 61

[39] Troy Vasiga and Jeffrey Shallit. On the iteration of certain quadratic maps over GF(p). *Discrete Mathematics*, 277:219–240, 2004. [Link]. 23, 32, 44

[40] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, second edition, 2012. 12, 23, 32, 45

[41] Rose Yu and Rui Wang. Learning dynamical systems from data: An introduction to physics-guided deep learning. *Proceedings of the National Academy of Sciences of the United States of America*, 121, 2024. [Link]. 23, 32, 45