Quantum Dynamics, Logistic Map, and Digit Distribution of Special Math Constants

Vincent Granville, *Co-Founder, BondingAI.io*
vincent@bondingai.io | linkedin.com/in/vincentg/
April 2025

***Abstract*—Using the logistic map instead of the base quadratic system as in my previous paper [11], I obtain very similar quantum dynamics, this time for the function $\sin^2(\sqrt{x})$ instead of $\exp(x)$. When x is a small integer or a product of consecutive primes, my framework reveals new insights on the digit distribution of major math constants. I also discuss deep findings about the chaotic nature of dynamical systems and several applications including in AI.**

I. Introduction

Let n be a fixed, large integer, say $n = 10^6$. The dynamical system $S_{k+1} = S_k^2$, starting with $S_0 = 2^n + 1$, that is, a string consisting of $n-1$ zeros with a one at both ends, has this particular property: the first n binary digits of S_n match those of $\exp(1)$, give or take. To make it a true mapping and without changing the conclusions, each S_k is rescaled: it is multiplied by an integer power of 2 (negative or positive), so that it stays in $[1, 2[$ for all $k = 0, 1, 2$ and so on.

If you replace $S_0 = 2^n + 1$ by $S_0 = 2^n + x$ where x is a small integer, say $x = \pm 1$ or $x = 3$, then the first n binary digits of S_n match those of $\exp(x)$. This remains true if each S_k is truncated to a precision of $2n$ bits. I discuss the details in [9], [10] and [11].

I call the dynamical system in question the base quadratic map. Other quadratic dynamical systems include the logistic map $S_{k+1} = 4S_k(1 - S_k)$ and the standard quadratic map $S_{k+1} = S_k^2 + c$ where c is a constant. The latter is defined in the complex plane and leads to the Mandelbrot set featured in Figure 1.

The goal here is to obtain similar results for the logistic map, thus expanding my investigations about the digit sum function in [9]. For the logistic map, starting with the seed string $S_0 = 2^{-2n}$ consisting of a single '1', we get $S_n = \sin^2(1)$ correct to $2n$ binary digits if we keep a $2n$-bit precision at all times. Even better: if $S_0 = x \cdot 2^{-2n}$, then $S_n = \sin^2(\sqrt{x})$ correct to $2n$ bits. The quantum dynamics of the digit sum are very similar to those observed with the base quadratic map. This framework offers new directions to reach our ultimate goal: proving deep results about the digit distribution of special math constants.

II. Logistic map and the digit sum function

For the logistic map $S_{k+1} = 4S_k(1-S_k)$ with $0 \le S_k \le 1$, there is a closed-form expression for the k-th iterate:

$$S_k = \sin^2(2^k \arcsin \sqrt{S_0}). \tag{1}$$

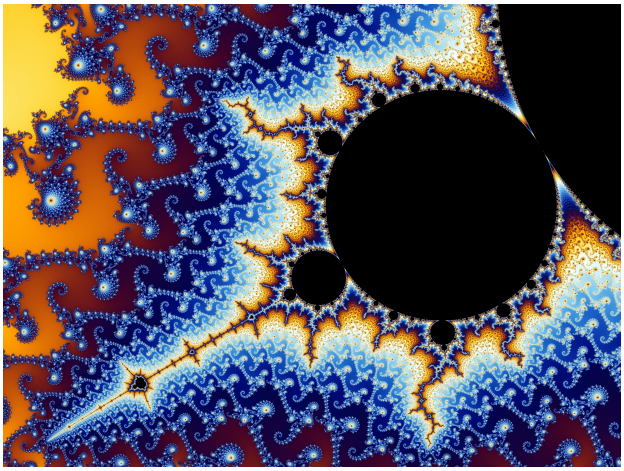

Fig. 1. Mandelbrot set linked to the standard quadratic map

In particular, if $S_0 = x \cdot 2^{-2n}$ with $x > 0$, then using a Taylor expansion for $\arcsin(\sqrt{S_0})$, we get

$$S_k = \sin^2\left(\frac{x^{1/2}}{2^{n-k}} + \frac{1}{6} \cdot \frac{x^{3/2}}{2^{3n-k}} + \cdots\right). \tag{2}$$

Thus, when $k = n$ and $n \to \infty$, we get $S_n \to \sin^2(\sqrt{x})$. By contrast, for the base quadratic map with the seed $S_0 = 2^n + x$ and proper rescaling (multiplication by an integer power of 2), we obtained the asymptotic formula

$$S_k = \left(1 + \frac{x}{2^n}\right)^{2^k} \sim \exp\left(\frac{x}{2^{n-k}}\right) \tag{3}$$

converging to $\exp(x)$ when $k = n$ and $n \to \infty$. The right part in (3) has an accuracy of about $2n - k$ bits if the precision on the left part is kept to $2n$ bits at all times.

A. Model comparison, with illustrations

Here $n = 10^5$ is fixed. The digit sum function ζ_k counts the number of '1' in the first n binary digits of S_k, for $k = 0, 1, 2$ and so on. Since we start with a seed S_0 close to 0 with the proportion of '1' typically increasing over time until reaching about 50% at $k = n$, I use the adjusted digit sum ζ_k' instead. It counts the number of '1' in the first n digits of S_k, starting at position $n - k$ in the digit expansion of S_k.

Let $\rho = k/n$. The behavior of ζ_k' is trivial when $\rho < 0.50$. Up until $\rho = 0.75$, patterns are usually strong and obvious. It starts to get somewhat chaotic as $\rho > 0.90$, and when $\rho \ge 1$, we are in full chaotic phase. Thus I focus on $0.75 < \rho < 1$.

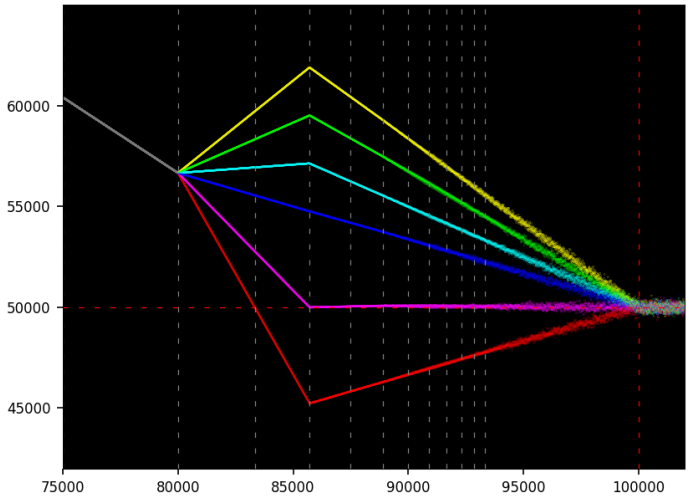

Fig. 2. Logistic map: ζ'_k with $x = 1, n = 10^5$, k on X-axis

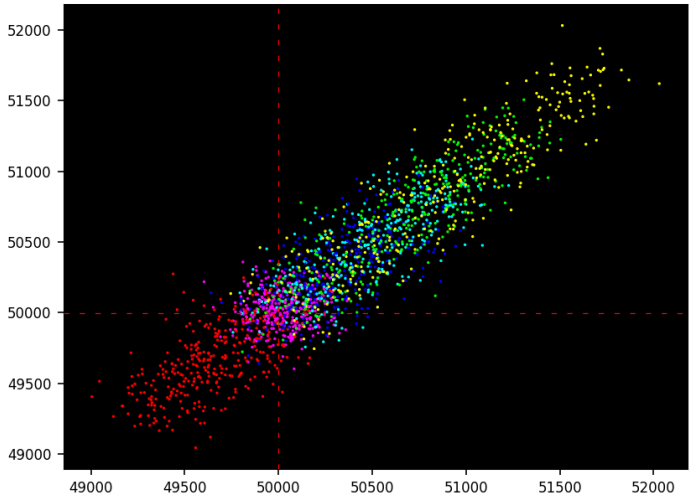

Fig. 5. Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = 1, n = 10^5$

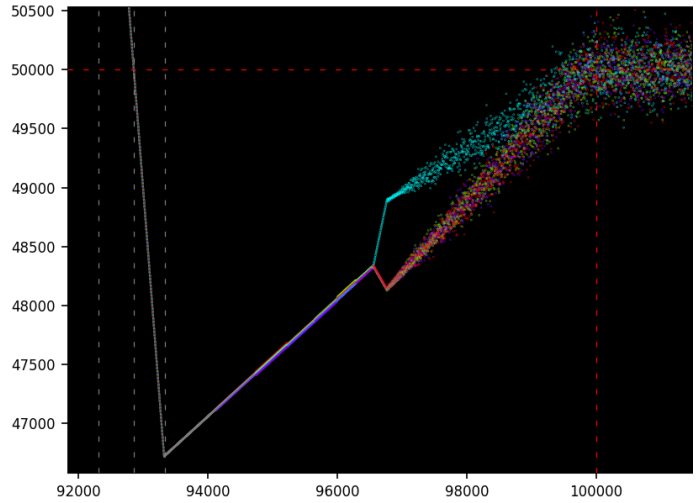

Fig. 3. Logistic map: ζ'_k with $x = \pi_\kappa, n = 10^5 - 1$, k on X-axis

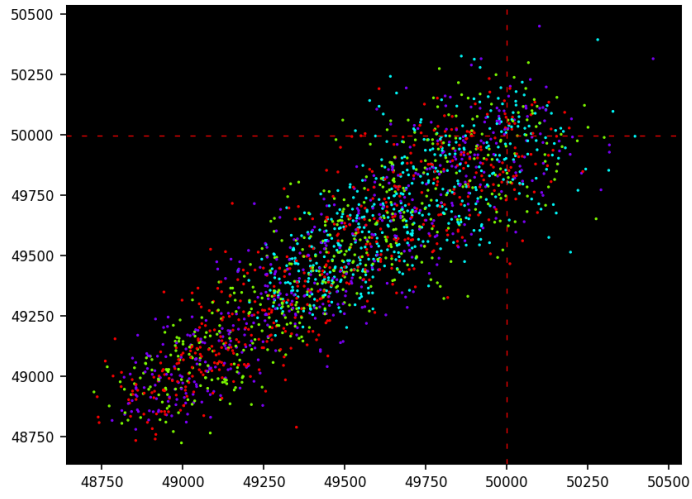

Fig. 6. Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = \pi_\kappa, n = 10^5 - 1$

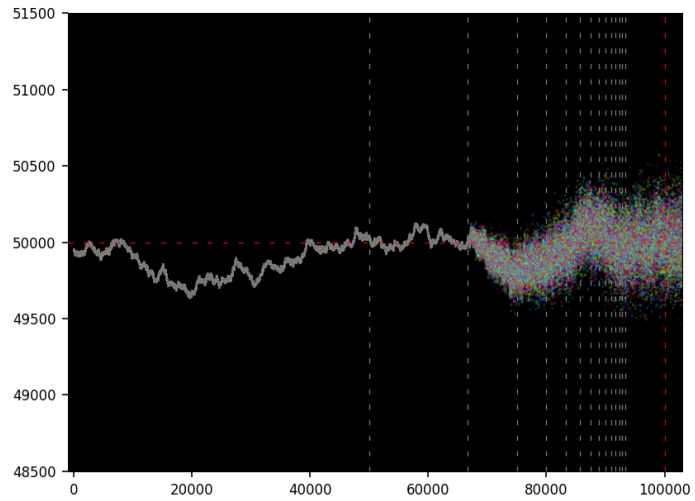

Fig. 4. Logistic map: ζ'_k with $x = \sqrt{2}, n = 10^5$, k on X-axis

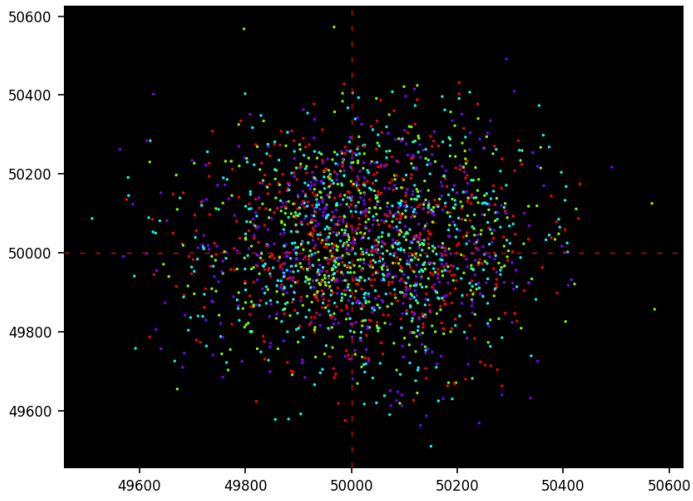

Fig. 7. Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = \sqrt{2}, n = 10^5$

From now on, BQ denotes the base quadratic map with illustrations in [11]. With the seeds $S_0 = x \cdot 2^{-2n}$ for the logistic map and $S_0 = 2^n + x$ rescaled to $1 + x \cdot 2^{-n}$ for the BQ map, we observe the following 3 types of behavior for ζ'_k.

- **Chaotic**. This represents the vast majority of cases, for instance if x is a random number in $[0, 1]$. See Figures 4 and 7.
- **Quantic**. See Figures 2 and 5 for the logistic map, and Figure 8 for the BQ map. It happens when x is a small integer, say $x = 1$. The color indicates the congruential class that k belongs to, modulo 6.
- **Hybrid**. You don't have multiple branches depending on the congruential class that k belongs to as in the quantic case, at least until $k \approx 0.97 \cdot n$. See Figure 3 and 6. Here, $x = \pi_\kappa$ is the κ-th primorial, that is, the product of the first κ primes ($\kappa = 9$).

Zoom in on any figure to see the details. In particular, the vertical dashed lines indicate the abscissa of change points (forking or slope change) in the function ζ'_k. The values seen in the pictures (specific k's on the X-axis) are those of the BQ map, but remain valid for the logistic map. They correspond to $k = \rho n$, with $\rho = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}$ and so on.

The peculiar shape in the quantic and hybrid cases, both for the BQ and logistic maps, is explained by the unusual Taylor series when fully expanding formulas (2) and (3). See section II.B in [11] for details.

Figures 2, 3, and 4 represent time series with quantum states for the digit sum ζ'_k with k on the X-axis and ζ'_k on the Y-axis. By contrast, Figures 5, 6 and 7 are scatterplots representing the vector (ζ'_{k-w}, ζ'_k) for $0.98 \cdot n < k < n$. It shows a spectral view when we are approaching chaos; full chaos starts at $k \geq n$ in the quantic and hybrid cases, and at about $k = 0$ in almost all other cases. The parameter w is called the time lag. An alternative view consists of showing the autocorrelation function computed on the ζ'_k sequence when k is close to n, for various time lag values $w = 1, 2$ and so on. However, in the quantic and hybrid cases, the process is non-stationary until $k \geq n$.

With a random seed, the scatterplot in Figure 7 shows a Gaussian distribution centered at $(\frac{n}{2}, \frac{n}{2})$. Curiously, the seed with $x = \pi^2/4$ leads to ζ'_k hovering around $n/2$ as in Figure 4 when $k < n$, but suddenly dropping to 0 at $k = n$, since $S_n \approx \sin^2(\sqrt{x}) = 1$. Keep in mind that ζ'_k counts the '1' in the first n digits of S_k, starting at position $n - k$ in the digit expansion of S_k.

See also Figures 8, 9, 10 and 11. For the BQ map, x is the parameter in the seed $S_0 = 2^n + x$; for the logistic map, it comes from $S_0 = x \cdot 2^{-2n}$. In Figures 3 and 10 featuring the logistic map, x is a multiple of 3; to see the congruential classes in 6 colors, I had to replace $n = 10^5$ by $n = 10^5 - 1$, which is a multiple of 3.

Note that Figures 10 and 11 feature the non-adjusted digit sum ζ_k instead of ζ'_k. Discontinuities in ζ'_k are sometimes observed. Each figure has its own scale: the X- and Y-axis are custom-truncated depending on the case, to provide the best view, in particular near $k = n$ where the real action is taking place. Pictures featuring moving averages that collapse multi-branches into a single one with a constant slope, are posted in [11].

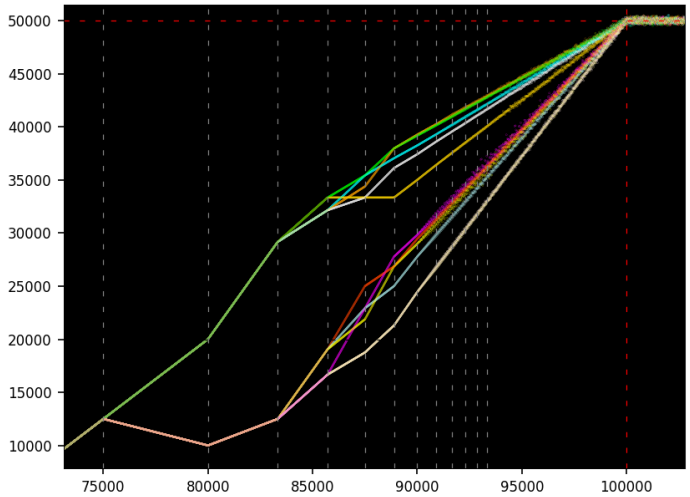

Fig. 8. BQ map: ζ'_k with $x = 1, n = 10^5$, k on X-axis

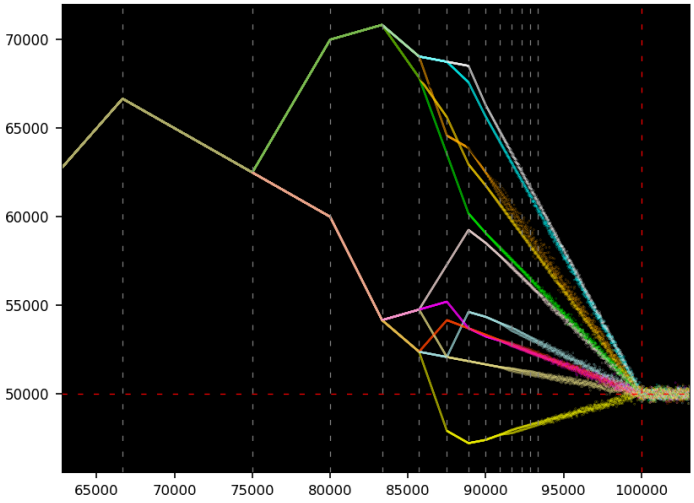

Fig. 9. BQ map: ζ_k with $x = -1, n = 10^5$, k on X-axis

Finally, in [11], I discuss the inverse iterations for the BQ map, moving backward from S_{k+1} to S_k rather than forward from S_k to S_{k+1}. It is possible to do the same for the logistic map. However, in both maps, the inverse mapping S_{k+1} to S_k is not one-to-one, but one-to-two. In the Python code in section V, S_k is denoted as `prod`. The link to fractals is visible in Figures 1 and 2 in [9].

B. Normality of special math constants

My framework was first designed for the BQ map to answer this question: are the binary digits of e evenly distributed? That is, is e simply normal in base 2? This famous multi-century old conjecture is still an open question. For the logistic map, replace e by $\sin^2(1)$. In this section, I share the progress that I made recently thanks to the new methodology discussed in this article.

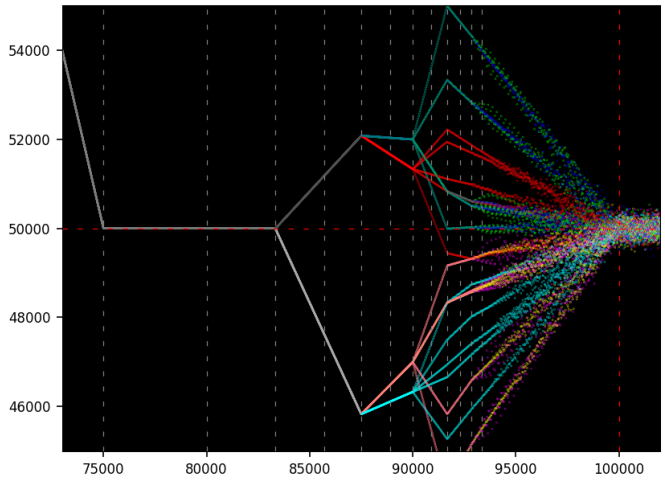

Fig. 10. Logistic map: ζ_k with $x = 9, n = 10^5 - 1$, k on X-axis

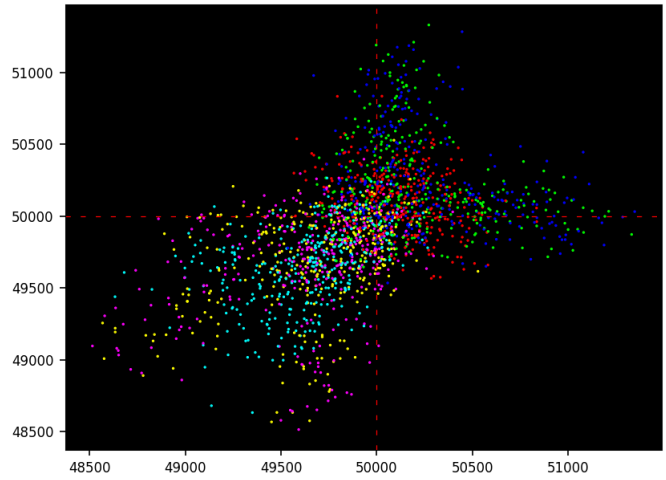

Fig. 11. Log. map: (ζ_{k-12}, ζ_k) with $0.98 < \frac{k}{n} < 1$, $x = 9, n = 10^5 - 1$

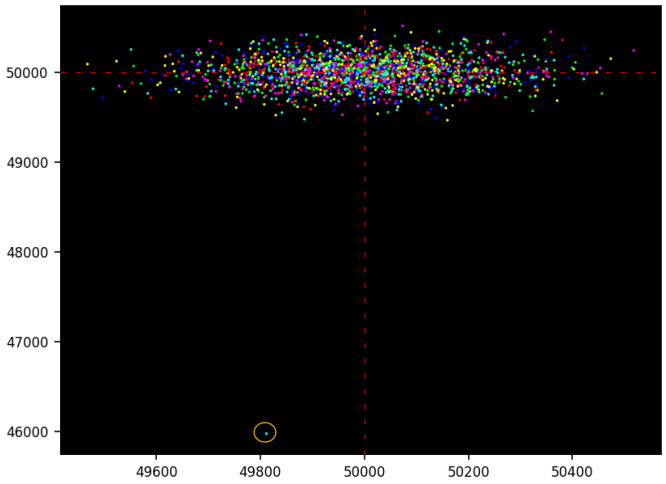

Fig. 12. Same as Fig. 7 but with x leading to $\zeta'_n/n \approx 0.46$ instead of 0.50

Let $n = 10^5$. Using a random S_n in $[0, 1]$ with a proportion $p = 46\%$ of '1' in its binary digit expansion rather than $p = 50\%$ as in Figure 4, how would the sequence (ζ'_k) look like, for $k = 0, 1$ and so on? The answer: exactly as in Figure 4. But the spectral view would be very different from the corresponding Figure 7. Instead, it would look like Figure 12.

The explanation is as follows: near to $k = n$ with $n = 10^5$, ζ'_k/n is always close to 50%, with a sudden drop exactly and only at $k = n$, where $\zeta'_n/n \approx 46\%$. Thus, the single outlier in Figure 12, circled in orange. If you increase n from 10^5 to (say) 10^{50}, you would still get an outlier even if increasing $p = 0.46$ to $p = 0.48$.

It sounds as if you cannot get the **quantic** behavior observed in Figure 2 if the digits of S_n follow a Bernoulli process of parameter $p \neq \frac{1}{2}$, for n large enough. Instead, it would have the **chaotic** behavior pictured in Figure 4

That is, the binary digits of $\sin^2(1)$ either do not follow a Bernoulli process, or if they do, then the probability of '1' is $p = \frac{1}{2}$. Of course, almost everyone believe the latter to be true, not the former. There is no formal proof yet, but the progress made here is very encouraging.

A side effect (conjecture) is the following: if the digits of S_n follow a Bernoulli process with $p \neq \frac{1}{2}$, then the digits of

$$S_{n-1} = \frac{1 \pm \sqrt{1 - S_n}}{2} \tag{4}$$

follow a Bernoulli process with $p = \frac{1}{2}$. By "following", I mean that the empirical joint digit distribution converges to that of a Bernoulli process as $n \to \infty$.

Empirical evidence is easy to obtain. Generate $y = S_n$, a number in $[0, 1]$ with digits simulated to follow a Bernoulli process with $p = 0.46$. Since $y = \sin^2(\sqrt{x})$, we have $x = \arcsin^2(\sqrt{y})$. Let's use $S_0 = x \cdot 2^{-2n}$. Note that the inverse map is not one-to-one, see (4). Set the precision to $2n$ bits and proceed as in all other cases ($x = 1, 9$, and so on) using the code in section V to generate S_k and compute ζ'_k for $k = 0, 1$ and so on. I did the test, with the following Python code to generate y and get x.

```
import gmpy2
import numpy as np

np.random.seed(410)
n = 100000
u = np.random.binomial(n=1, p=0.46, size=2*n)
stri = [str(bit) for bit in u]
stri = "".join(stri)
ctx = gmpy2.get_context()
ctx.precision = 2*n
y = gmpy2.mpz(stri, 2)
y /= 2**(2*n)
x = gmpy2.asin(gmpy2.sqrt(y))
x = x*x
```

C. Applications and references

Here I compiled a list of useful references related to the topic, broken down by application, with a focus on literature recently published.

- The framework presented here relies on discrete quadratic dynamical systems. This family also includes the logistic

map and the example discussed in [19]. For additional references, see my book on chaos and dynamical systems [7].

- Showing that the binary digits are evenly distributed is the first step towards proving that e is a normal number. Andrew Granville and Davig Bailey [3] are good references on this topic. For recent publications on normal numbers, see Verónica Becher [4] and [2], [12]. One of best results know for any major math constant is the fact that the proportion of ones in the first n binary digits of $\sqrt{2}$ is larger than $\sqrt{2n}$, see [18].
- The digit sum or digit count functions (both are identical for binary digits) is also known as the Hamming weight, with a fast algorithm described here and a full chapter in [20]. The Wolfram entry for the digit sum (see here) features an exact closed-form formula for the number of digits equal to 1 in the binary expansion of any integer, with more references. For a discussion on the carry digit function (a 2-cocycle) that propagates 1's from right to left in the successive iterations S_k, see [1], [5].
- An interesting application of the digit sum is featured in [13] in the context of genotype maps, with processes not unlike the dynamical systems discussed in this article, and blancmange curves almost identical to Figure 3.3 in my book on numeration systems [7].
- There is a connection to quantum maps and quantum cryptography [6], [17]. For PRNGs (pseudo-random generators) based on irrational numbers, see chapter 13 in [8] or chapter 4 in [7]. Finally, if you use an arbitrary seed S_0 with about 50% of '0' and '1', you obtain strings S_k that look random after very few iterations.
- Deep neural networks have been used to identify the underlying model of dynamical systems, based on available data produced by simulations or from real life observations, see [15], [16], [21]. In our case, the model would be a simple formula that generates the values of the digit sum function, to study its asymptotic properties. See also [14].

III. Re-balancing an uneven digit distribution

Let's pretend that the digits of $y = e$ or $y = \sin^2(1)$ are not evenly distributed. After all, it's not proved yet. Is there a way to apply a transform $y' = \varphi(y)$ so that y' is still a well-known math constant, but with a proportion of '1' closer to 50% in the binary digit expansion? Or is there a way to get rid of the factorials in Formula (2) in [11], to obtain a new formula involving only powers of 2 at the denominator to make the proof easier, possibly for a number other than e as long as it is still a well-known math constant? These are the questions that I address in this section.

A. Digit-balancing transforms

The transform $y' = \varphi(y)$ with $\varphi(y) = \frac{1}{3}$ turns any number y into one with 50% of '1' in the binary digit expansion. But it is totally useless. There are plenty of simple transforms, for instance $\varphi(y) = \sqrt{y}$ or $\varphi(y) = y(1-y)$ that will map *almost* any number with (say) 10% of '1' onto one with a proportion of '1' between 30% and 70%. But they are not full-proof; the nice ones all have exceptions. You need a transform that maps (say) $[0, 1]$ into a subset of Lebesgue measure zero such as the Cantor set. However, these transforms – those that are potentially useful – typically do not turn e into a well-known irrational constant (for those that do, it would be impossible to prove it). Here I show how well-behaved transforms impact the digit distribution, yet without formally solving our problem.

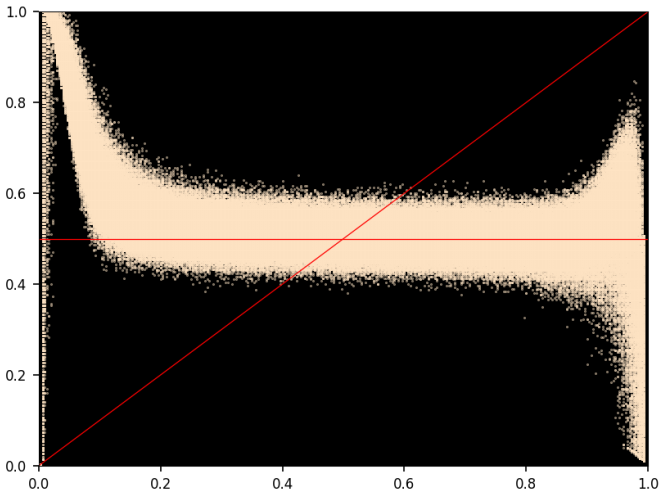

Fig. 13. Scatterplot: proportion of '1' in y (X-axis) vs. in $y' = y(1-y)$ (Y-axis), with y in $[0, 1]$

Figure 13 shows the impact of the map $y \mapsto y' = y(1-x)$ on the proportion of '1' in the binary digit expansion of y', given the proportion of '1' in y. My simulation is as follows. First, set the precision to $n = 300$ bits. Then, for each integer q in $]0, n[$, I generated 2000 random numbers y in $[0, 1]$, each with n bits with q of them set to '1'. Then I counted the '1' in the first n bits of y'. For all y, if seems that if y has about 20% of '1', then y' has anywhere between 40% and 70%.

It seems to imply the following: if the proportion of '1' in the number e is around 20%, then it must be between 40% and 70% for the number $e(4-e)$. This would be a spectacular result if proved. However, although dots are very rare outside the band in Figure 13, they eventually cover the entire area as the sample size increases more and more. The band does not have hard boundaries; it represents the region where the density is not very close to zero.

The Python program `nt_digit_transform.py` to perform the simulations on GitHub, here, and listed below. To produce Figure 13, see section "Main Plot" in the code.

```
import numpy as np
import random
import gmpy2

random.seed(410)

def randomize(n, q):

    # q is the number of digits set to 1, among the
      n digits
```

```
    u = random.sample(range(1,n-1), q-1)
    v = np.zeros(n-1)

    for k in range(q-1):
        index = u[k]
        v[index] = 1

    stri = [str(int(bit)) for bit in v]
    stri = "".join(stri)
    stri = '1' + stri
    y = gmpy2.mpz(stri, 2)

    return(y, stri)


#--- 1. Main

n = 300
ctx = gmpy2.get_context()
ctx.precision = 4*n
samples = 2000

arr_dc = np.zeros(n+1)
arr_min = np.zeros(n)
arr_x = []
arr_y = []

low = 1
high = n-1
for q in range(low, high):
    if q % 10 == 0:
        print("q=",q)
    min_dc = n+1

    for k in range(samples):
        (y, stri) = randomize(n, q)
        y2 = (2**n * y*(2**(n) - y))
        stri2 = bin(y2)[2:n+2]
        dc = stri2.count('1')
        if dc < min_dc:
            min_dc=dc
            min_stri2 = stri2
            min_stri = stri
        arr_dc[dc]+=1
        arr_x.append(q)
        arr_y.append(dc)

    arr_min[q] = min_dc


#--- 2. Main plot

arr_x = np.array(arr_x)/n
arr_y = np.array(arr_y)/n

import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['axes.facecolor'] = 'black'

plt.scatter(arr_x, arr_y, s=0.2, c='bisque', alpha
    = 0.6)
plt.plot((0, 1), (0, 1), c='red', linewidth=0.6)
    ##, marker = 'o')
plt.plot((0, 1), (0.5, 0.5), c='red',
    linewidth=0.6) ##, marker = 'o')
plt.ylim((0.00, 1.00))
plt.xlim((0.00, 1.00))
plt.show()


#--- 3. Other plots

arr_dc_cdf = np.zeros(n)
arr_dc_cdf[0] = arr_dc[0]
for k in range(1, n):
    arr_dc_cdf[k] = arr_dc[k] + arr_dc_cdf[k-1]

xval = range(low, high)
plt.plot(xval, arr_min[low:high])
plt.show()
plt.plot(xval, arr_dc_cdf[low:high])
plt.show()
```

B. Digit block balancing

There are many ways to turn a number y into another one y' that has 50% of '1' in its binary digit expansion, whether or not y satisfy this property. Here I share one such method. The difficult part, if y is a well-known irrational constant, is to obtain another well-known irrational constant for y'. First, let ν be a fixed integer and $M_{k,\nu} = 2^\nu - 1$. Note that for now, $M_{k,\nu}$ depends on ν but not on k. This notation will be useful moving forward. The first step is to choose a series

$$\psi_n(x) = \sum_{k=0}^{n} B_k x^k, \tag{5}$$

where $0 \le B_k \le M_{k,\nu}$ is an integer for all k. Let $B'_k = M_{k,\nu} - B_k$. Each string B_k or B'_k consists of ν bits.Then, the concatenated string $B_0 B'_0 \dots B_n B'_n$ has 50% of '1' regardless of n. Its binary digits match those of

$$y'_n = \Big(1 - \frac{1}{2^\nu}\Big)\psi_n\Big(\frac{1}{4^\nu}\Big) + \frac{1}{2^\nu}\sum_{k=0}^{n}\Big(\frac{1}{4^\nu}\Big)^k M_{k,\nu} \tag{6}$$

while the concatenated string $B_0 B_1 \dots B_n$ corresponds to

$$y_n = \psi_n\Big(\frac{1}{2^\nu}\Big). \tag{7}$$

However, since y'_n has exactly 50% of '1' for all n, it cannot converge to a well-known irrational constant when $n \to \infty$. To overcome this problem, the blocks B_k must be more than ν bits long to allow for limited carry-over in successive terms when computing (6). To achieve this goal, one can choose a slow growth integer sequence for (B_k) with known generating function $\psi(x)$ and modify $M_{k,\nu}$ accordingly so that $M_{k,\nu} \ge B_k$ for all k. From now on, n is infinite and $\psi_n(x), y_n, y'_n$ are denoted respectively as $\psi(x), y, y'$. If ψ is invertible, you can express y' as a function of y and conversely. For instance, with the replacements

$$\frac{1}{2^\nu} = \psi^{-1}(y), \quad \frac{1}{4^\nu} = \Big[\psi^{-1}(y)\Big]^2 \tag{8}$$

in formula (6).

I tested the method with the central binomial coefficient $B_k = \binom{2k}{k}$ with $M_{k,\nu} = 4^k$. It is a fast growing sequence, but if $\nu > 1$, it leads to $\psi(x) = 1/\sqrt{1-4x}$ and via (6), to

$$y' = \frac{2^\nu - 1}{\sqrt{4^\nu - 4}} + \frac{2^\nu}{4^\nu - 4} \tag{9}$$

When $\nu = 0$, the series for y and y', based on formulas (5), (6), and (7), diverge. But if each y'_n is multiplied by negative integer power of 2 to stay within $[1, 2[$, then $y' = \frac{4}{3}$.

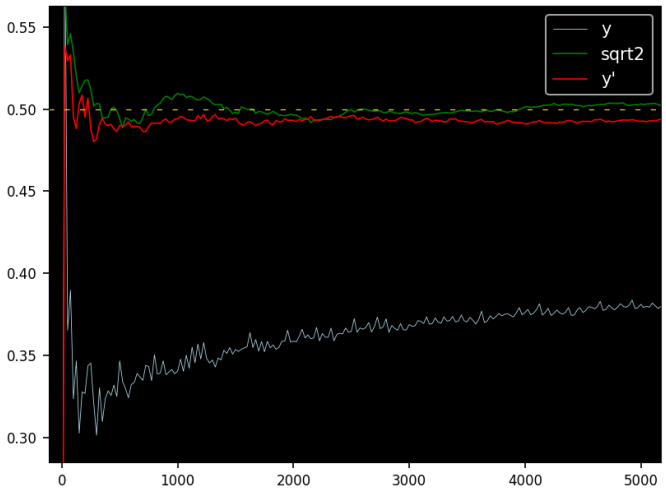

Fig. 14. Proportion of '1' at iteration k, with k on X-axis, $b = 1.5$

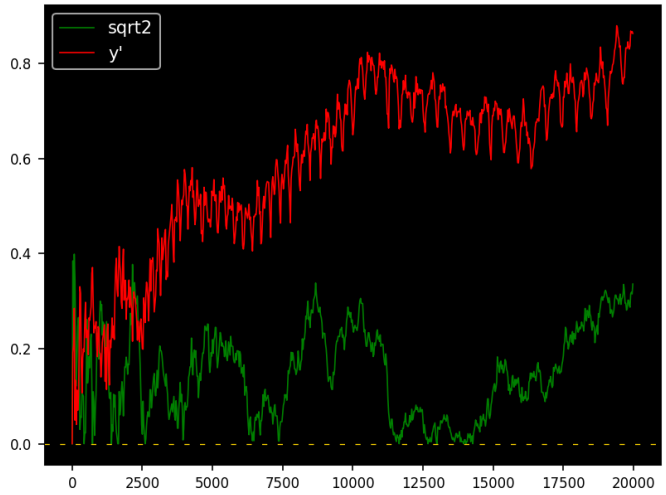

Fig. 15. Distance to 50% of '1' at iteration k, with k on X-axis, $b = 1,5$

The next step consists of looking at slow growth sequences, say $B_k = \lfloor g(k) \rfloor$ where g is strictly increasing with $g(0) = 0$. It is easy to prove that for $0 \leq x < 1$, we have

$$\psi(x) := \sum_{k=0}^{\infty} \lfloor g(k) \rfloor x^k = \frac{x}{1-x} \sum_{k=1}^{\infty} x^{\nu_k} \qquad (10)$$

where $\nu_k = \lfloor g^{-1}(k) \rfloor$. For instance, $g(z) = az^b$ with $a, b > 0$. If g grows too slowly then the binary expansion of $y = \psi(\frac{1}{2})$ has larger and larger gaps as k increases in (10), resulting in a proportion of '1' asymptotically equal to zero, failing to make y a well known irrational constant. The growth rate of $g(z)$ must be at least $O(z)$. However, this applies to y, not to y'. If $g(z) \sim o(z)$, then y cannot be a normal number in base 2, as a direct consequence of (10) and (7) with $x = \frac{1}{2}$ and $\nu = 1$.

The Python program `nt_digit_blocks.py` listed below tests the methodology discussed in this section. The code is also on GitHub, here. Set `mode` to `'Bernoulli'` to use $B_k = \binom{2k}{k}$. The variable `correct_digits` counts the number of correct binary digits in y'_k obtained at each iteration k, based on the theoretical limit y established in formula (9);

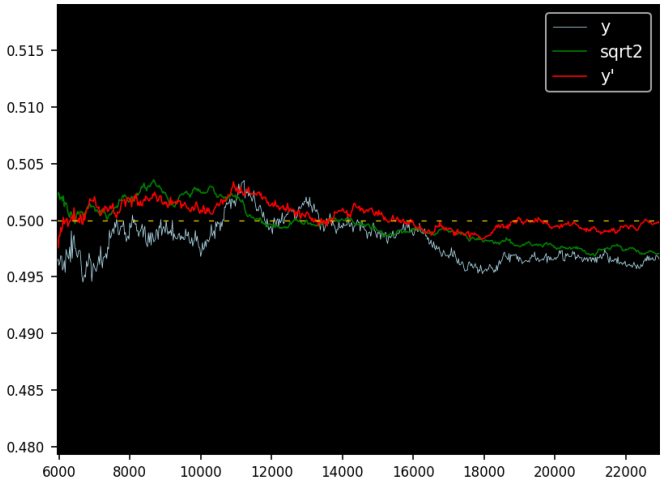

Fig. 16. Proportion of '1' at iteration k, with k on X-axis, $b = 3.5$

`rlen` is `correct_digits` divided by the total number of digits in y'_k. In my tests, it ranges from 50% to 90% depending on k and ν.

Finally, Figures 14, 15, 16 deal with $B_k = \lfloor g(k) \rfloor$ featured in formula (10), with $g(z) = az^b, a = 1, x = \frac{1}{2}$, and $\nu = 1$. Set `mode` to `SmallGrowth` in the code, to test it. The green curve features the number $\sqrt{2}$ supposed to mimic a perfectly random behavior. for comparison purposes. Figure 15 displays $|q_k - 0.5|\sqrt{k}$ where q_k is the proportion of '1' obtained at iteration k. As seen in Figure 14, y' transforms y into a number with 50% of '1'. Also the number y produced with $b = 1.5$ is not normal, at least if you look at the first 20,000 digits. But with $b = 3.5$, it looks normal. From the theory, we know that y cannot be normal if $b < 1$.

```
import gmpy2
import numpy as np

np.random.seed(454)

def compute_correct_digits(sum, pow2, n):

    # to double-check our digits match those of
        theoretical limit
    bsum = bin(sum)[2:]
    bsum_u_exact =
        gmpy2.mpfr((pow2)/gmpy2.sqrt(pow2*pow2-4))
    bsum_main_exact = bsum_u_exact -
        bsum_u_exact/pow2
    bsum_constant_exact =
        pow2/gmpy2.mpfr(pow2*pow2-4)
    bsum_exact = bsum_main_exact +
        bsum_constant_exact
    bsum_exact = bin(gmpy2.mpz(pow2**(2*n) *
        bsum_exact))[2:]

    k = 0
    while k < len(bsum) and bsum[k] == bsum_exact[k]:
       k = k+1
    return(k)


#--- 1. Main

ctx = gmpy2.get_context()
n = 10000
nu = 1
```

```
N = 100*n # precision
pow2 = 2**nu
ctx.precision = N
mode = 'SmallGrowth' # options: 'Bernoulli',
    'Fixed', 'SmallGrowth'

sum = 0
sum_y = 0
sum_u = 0
sum_v = 0
arr_q1 = []
arr_q2 = []
arr_q3 = []
arr_xval = []
arr_rlen = []
arr_delta2 = []
arr_delta3 = []

# sqrt2 is test number to compare digit distrib.
    with those of y and y'
sqrt2 = gmpy2.sqrt(2) # this number used for
    comparison purpose
sqrt2 = gmpy2.mpz(2**(N) * sqrt2) # turn sqrt2 into
    integer with N bits

for k in range(n):

   # B_max corresponds to M_{k, nu} in the paper

   if mode == 'Bernoulli':
      B_k = gmpy2.bincoef(2*k, k)
      B_max = 4**k # known fact: B_k < B_max

   elif mode == 'Fixed':
      # Generate B_k with (nu + dx) bits, each bit
          is '1' with proba p
      # if dx > 0, there is carry over (more if dx
          large compared to nu)
      # dx can be < 0, but (nu + dx) most be >= 0
      # proportion of '1' in y is p only if dx=0
      dx = 40
      p = 0.25
      u = np.random.binomial(n=1, p=p, size=nu+dx)
      stri = [str(bit) for bit in u]
      stri = "".join(stri)
      B_k = int(stri, 2)
      B_max = 2**(nu + dx) - 1

   elif mode == 'SmallGrowth':
      cx = 3.5
      bx = 1
      B_k = int(bx * k**cx)
      # try: B_k = 3**k >> k # equal to
          int[(3/2)**k]
      B_max = 0
      if B_k > 0:
         bits = len(bin(B_k)) - 2
         B_max = 2**(bits+1) # equal to 2^(int(log2
             B_k))

   u_k = B_k
   v_k = B_max - B_k
   s_k = pow2 * u_k + v_k

   sum_u = pow2**2 * sum_u + u_k
   sum_v = pow2**2 * sum_v + v_k
   sum = pow2 * sum_u + sum_v # for y'
   sum_y = pow2 * sum_y + B_k # for y
   # equivalenly, sum = pow2**2 * sum + s_k

   if k % 25 == 0:
      # computations needed to visualize
          intermediary results
      stri1 = bin(sum_y)[2:]
      stri2 = bin(sqrt2)[2:]
      stri3 = bin(sum)[2:]
      rlen = -1
      correct_digits = -1
      if mode == 'Bernoulli' and nu > 1:
         # if correct_digits stalls, restart with
             larger N
         correct_digits =
             compute_correct_digits(sum, pow2, n)
         rlen = correct_digits/len(stri3)
         stri1 = stri1[0:correct_digits]
         stri2 = stri2[0:correct_digits]
         stri3 = stri3[0:correct_digits]
      else:
         stri2 = stri2[0:len(stri3)]
      q1 = stri1.count('1')/len(stri1)
      q2 = stri2.count('1')/len(stri2)
      q3 = stri3.count('1')/len(stri3)
      arr_q1.append(q1)
      arr_q2.append(q2)
      arr_q3.append(q3)
      arr_xval.append(k)
      arr_rlen.append(rlen)
      arr_delta2.append(abs(q2-0.5)*np.sqrt(k))
      arr_delta3.append(abs(q3-0.5)*np.sqrt(k))

      print("%5d %5d %6.4f %6.4f %6.4f %6.4f" %
          (k, correct_digits, rlen, q1, q2, q3))

print()
print(stri1[-60:]) # last 60 digits of y
print(stri3[-60:]) # last 60 digits of y'
print()

#-

# compute rescaled y and y'; factor is a power of nu
# works even when series diverges (Bernoulli with
    nu in {0, 1})
# if Bernoulli with nu = 0, then y' = 4/3

y = 0
y_prime = 0
for k in range(60):
   y+= int(stri1[k])/2**k
   y_prime += int(stri3[k])/2**k
print("y = %14.12f" % (y))
print("y' = %14.12f" % (y_prime))


#--- 2. Main plot: % of dgits equal to '1' as k
    increases

import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['axes.facecolor'] = 'black'

plt.plot(arr_xval, arr_q1, linewidth = 0.4,
    c='lightblue') # for y
plt.plot(arr_xval, arr_q2, linewidth = 0.8,
    c='green') # for sqrt2
plt.plot(arr_xval, arr_q3, linewidth = 0.8,
    c='red') # for y'
plt.axhline(y=0.50,color='gold',linestyle='--',
    linewidth=0.6,dashes=(5,10))

legend = plt.legend(["y","sqrt2","y'"])
plt.setp(legend.get_texts(), color='white')
# plt.ylim(0.48, 0.52)
# plt.xlim(1000, n)
plt.show()


#--- 3. Other plots
```

```
if mode == 'Bernoulli':
   # show proportion of correct digits obtained
       after k iter
   plt.plot(arr_xval, arr_rlen, linewidth = 0.8)
   plt.show()

# show how far y' is to having 50% of '1' at iter k
plt.plot(arr_xval, arr_delta2, linewidth = 0.8,
    c='green')
plt.plot(arr_xval, arr_delta3, linewidth = 0.8,
    c='red')
plt.axhline(y=0.0, color='gold',linestyle='--',
    linewidth=0.6,dashes=(5,10))
legend = plt.legend(["sqrt2","y'"])
plt.setp(legend.get_texts(), color='white')
#plt.xlim(1000, n)
plt.show()
```

IV. Conclusion

This article is the fourth in this series, preceded by [9], [10] and [11]. The previous articles focus on the simplest quadratic map, while this one shows how we can attain similar results with the most well-know dynamical system: the logistic map.

Orbits in chaotic dynamical systems are very sensitive to the seed. Starting with close seeds $S_0 = x \cdot 2^{-2n}$ and $S'_0 = 0$ I computed $\Delta_k = |S_k - S'_k|$ and showed that $\Delta_n \approx \sin^2(\sqrt{x})$ with a precision of about $2n$ bits, also getting good approximations to Δ_k when $k \leq n$. In the logistic map, S'_0 is a fixed point.

I also discussed the unique quantic behavior of the digit sum function when using special values of x. This opens up new directions to study the digit distribution of special math constants, e in particular. Most importantly, it leads to many applications including cryptography, synthetic data, pattern detection or proof automation with LLMs [10], agent based modeling, and more.

The material presented here can also be used to complement a course on dynamical systems, for scientific research, or to start a PhD thesis on the subject.

V. Main Python code

In the code, the variable `prod` represents S_k. I also use external functions to check how many of the binary digits of S_n match those of $\sin^2(\sqrt{x})$. The code is on GitHub, here.

The fact that $0.111\ldots = 1.000\ldots$ in base 2 can have side effects on the values of ζ_k and ζ'_k. Whether the rightmost digits of S_k are '1000' or '0111' has no real impact on the accuracy. However is has an impact – usually small – on the digit sum, but potentially large especially when k is small. It remains to be seen if it contributes to the increased chaos near $k = n$.

A spectacular manifestation of this side effect is as follows. To compute ζ'_k, I look at the first n digits, starting at position $2n - k$, of the integer number $\lfloor 2^{2n} S_k \rfloor$. The result is shown in Figure 17, with k on the X-axis and ζ'_k on the Y-axis, with $n = 10^5$ and starting with the seed $S_0 = x \cdot 2^{-2n}$ with $x = 1$.

When $k < \frac{n}{2}$, $\zeta'_k = 0$. However at $k = \frac{n}{2}$, $\zeta'_k \approx n$. So there is a discontinuity at $k = \frac{n}{2}$. And then, the path from $k = \frac{n}{2}$ to $k = \frac{2}{3}n$ is not a straight line, but a curve. I did not expect that kind of behavior. To the contrary, if you use $\lfloor 2^{4n} S_k \rfloor$ rather than $\lfloor 2^{2n} S_k \rfloor$ by setting `pow` to 4 rather than 2 in the code, the behavior is different until $k = \frac{2}{3}n$, though not thereafter: when $k < \frac{n}{2}$, $\zeta'_k \approx n$ and there is no discontinuity at $k = \frac{n}{2}$. Also the curvy section is replaced by a straight line. Both behaviors are correct.

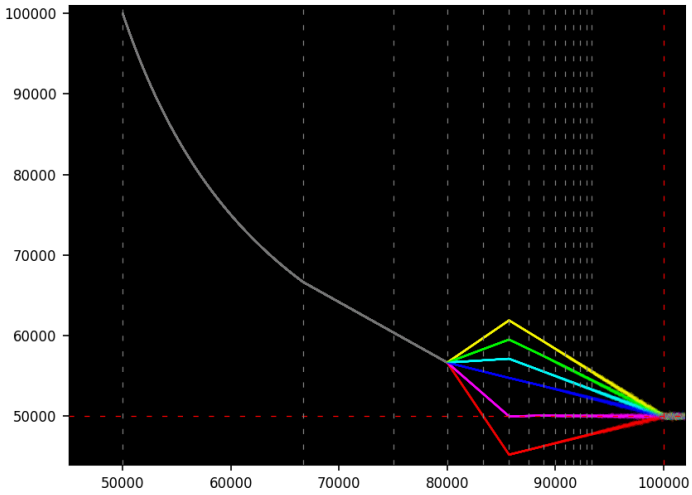

Fig. 17. Unusual precision parameters producing curve and discontinuity

```
import gmpy2
import numpy as np

n = 99999 # choose n divisible by 3 if x = 9 and
    ncolors = 6
H = int(1.1*n)

import colorsys

def hsv_to_rgb(h, s, v):
   return tuple(round(i * 255) for i in
       colorsys.hsv_to_rgb(h, s, v))

def generate_contrasting_colors(ncolors):
   colors = []
   for i in range(ncolors):
      hue = i / ncolors
      col = hsv_to_rgb(hue, 1.0, 1.0)
      color = (col[0]/255, col[1]/255, col[2]/255)
      colors.append(color)
   return colors

ncolors = 6 # 4 colors for hybrid case, 6 for
    quantic
colorTable = generate_contrasting_colors(ncolors)


#--- 1. Main

import gmpy2
import numpy as np

kmin = 0.00 * n # do not compute digit count if k
    <= kmin
kmax = 1.15 * n # do not compute digit count if k
    >= kmax
kmax = min(H, kmax)

# precision set to L bits to keep at least about n
    correct bits till k=n
ctx = gmpy2.get_context()
ctx.precision = 2*n

# x = gmpy2.mpz(2*3*5*7*11*13*19*23*29)
x = gmpy2.mpz(1)
```

```
prod = gmpy2.mpfr(x/2**(2*n))

# local variables
arr_count1 = []
arr_colors = []
xvalues = []
ecnt1 = -1
e_approx = "N/A"

OUT = open("digit_sum.txt", "w")

for k in range(1, H+1):

   prod = 4*prod*(1 - prod)
   pow = 2*n # 2n --> 0.1111.. | 4n --> 1.0000..
   pstri = bin(gmpy2.mpz(2**(pow) * prod))
   stri = pstri[0:2*n]

   if k > kmin and k < kmax:
      stri = stri[2:]
      lstri = len(stri)
      if k == n:
         e_approx = stri
      # estri = stri[0:n]      # for digit sum
      estri = stri[max(0,n-k):2*n-k] # for adjusted
          digit sum
      ecnt1 = estri.count('1') * n / (1+len(estri))
      arr_count1.append(ecnt1)
      color = colorTable[k % ncolors]

      arr_colors.append(color)
      xvalues.append(k)
      OUT.write(str(k)+"\t"+str(ecnt1)+"\t"
          +str(lstri)+"\n")

      if k%1000 == 0:
         print("%6d %6d %6d" %(k, ecnt1, lstri))

OUT.close()


#--- 2. Compute bits of sin^2(sqrt(x_) and count
     correct bits in my computation

# Set precision to L binary digits
gmpy2.get_context().precision = 4*n
e_value = gmpy2.sin(gmpy2.sqrt(x))
e_value = e_value * e_value

# Convert e_value to binary string
e_binary = gmpy2.digits(e_value, 2)[0]

k = 0
while k < len(e_approx) and e_approx[k] ==
     e_binary[k]:
       k += 1
# e_binary should be equal to e_approx up to about
     n bits
print("\n%d correct digits (n = %d)" %(k, n))

e_approx_decimal = 0
for k in range(0,80):
   e_approx_decimal += int(e_approx[k])/(2**(k+1))
print("e_exact : %16.14f" % (e_value))
print("e_approx: %16.14f" % (e_approx_decimal))
print("Up to factor 2 at integer power of 2.")

#--- 3. Create the main plot

import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['axes.facecolor'] = 'black'

plt.scatter(xvalues, arr_count1,s=0.005,
     c=arr_colors)
plt.axhline(y=n/2,color='red',linestyle='--',
     linewidth=0.6,dashes=(5,10))
plt.axhline(y=n/5, color='black', linestyle='--',
     linewidth = 0.6, dashes=(5, 10))
plt.axvline(x=n, color='red', linestyle='-',
     linewidth = 0.6, dashes=(5, 10))

for k in range(1,15):
   plt.axvline(x=k*n/(k+1),c='gray',linestyle='--',
       linewidth=0.6,dashes=(5, 10))

# we start with about 0% of 1 going up to about 50%
plt.ylim([0.44*n, 1.01*n])
plt.xlim([0.45*n, 1.02*n])
plt.show()


#--- 4. Create AR scatterplot

nv = n
lag = 12
tail = 2000
plt.scatter(arr_count1[n-tail-lag:n-lag],
     arr_count1[n-tail:n], s=0.4,
     c=arr_colors[n-tail-lag:n-lag])
# plt.plot(arr_count1[n-tail-lag:n-lag],
     arr_count1[n-tail:n], linewidth=0.6) ##,
     c=arr_colors[n-tail-lag:n-lag])
plt.axhline(y=n/2,color='red',linestyle='--',
     linewidth=0.6,dashes=(5,10))
plt.axvline(x=n/2, color='red', linestyle='-',
     linewidth = 0.6, dashes=(5, 10))

plt.show()
```

References

[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]. 5

[2] Christoph Aistleitner et al. Normal numbers: Arithmetic, computational and probabilistic aspects. 2016. Workshop [Link]. 5

[3] David Bailey, Jonathan Borwein, and Neil Calkin. *Experimental Mathematics in Action*. A K Peters, 2007. 5

[4] Verónica Becher, A. Marchionna, and G. Tenenbaum. Simply normal numbers with digit dependencies. *Mathematika*, 69:988–991, 2023. arXiv:2304.06850 [Link]. 5

[5] James Dolan. Carrying is a 2-cocycle. *Preprint*, pages 1–9, 2023. [Link]. 5

[6] 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]. 5

[7] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [Link]. 5

[8] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLTechniques.com, 2024. [Link]. 5

[9] Vincent Granville. Cracking a famous multi-century old math conjecture. *Preprint*, 2025. MLTechniques.com [Link]. 1, 3, 9

[10] Vincent Granville. LLM challenge with petabytes of data to prove famous number theory conjecture. *Preprint*, 2025. MLTechniques.com [Link]. 1, 9

[11] Vincent Granville. A universal dataset to test, enhance and benchmark AI algorithms. *Preprint*, 2025. MLTechniques.com [Link]. 1, 3, 5, 9

[12] 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. MLTechniques.com [Link]. 5

[13] 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]. 5

[14] Mohammadamin Moradi et al. Data-driven model discovery with Kolmogorov-Arnold networks. *Preprint*, pages 1–6, 2024. arXiv:2409.15167 [Link]. 5

[15] 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]. 5

[16] Yury V. Tiumentsev and Mikhail V. Egorchev. *Neural Network Modeling and Identification of Dynamical Systems*. Elsevier, 2019. 5

[17] Chukwudubem Umeano and Oleksandr Kyriienko. Ground state-based quantum feature maps. *Preprint*, pages 1–8, 2024. arXiv:2024.07174 [Link]. 5

[18] Joseph Vandehey. On the binary digits of $\sqrt{2}$. *Preprint*, pages 1–6, 2017. arXiv:1711.01722 [Link]. 5

[19] Troy Vasiga and Jeffrey Shallit. On the iteration of certain quadratic maps over GF(p). *Discrete Mathematics*, 277:219–240, 2004. [Link]. 5

[20] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, second edition, 2012. 5

[21] 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]. 5

Vincent Granville Vincent Granville is a pioneering GenAI scientist, co-founder at BondingAI.io, the LLM 2.0 platform for hallucination-free, secure, in-house, lightning-fast Enterprise AI at scale with zero weight and no GPU. He is also author (Elsevier, Wiley), publisher, and successful entrepreneur with multi-million-dollar exit. Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET. He completed a post-doc in computational statistics at University of Cambridge.