A Universal Dataset to Test, Enhance and Benchmark AI Algorithms

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

***Abstract*—The infinite dataset presented here is an invaluable tool to test, enhance or benchmark pattern detection algorithms for fraud detection, cybersecurity, and other applications. The methodology relies on string auto-convolutions to discover deep insights about the digit sum function, offering a new perspective towards solving a famous multi-century old conjecture: are the binary digits of e evenly distributed? In this article, I discuss the results obtained so far, both empirical and those formally proved, including several new ones. I also discuss the dataset, its relevancy to modern AI as a fundamental testing system, the incredibly rich and diversified set of patterns that it boasts, as well as connections to large language models (LLMs), quantum dynamics, synthetic data, and cryptography. I also provide very efficient, fast Python code to produce the data, dealing with integer numbers larger than $2^n + 1$ at power 2^n, with n larger than 10^6.**

I. Introduction

This paper is aimed at two distinct types of readers. On one hand, business professionals who want to use the featured dataset for simulation, benchmarking, testing and enhancing AI algorithms. And on the other hand, researchers interested in the most recent advances towards proving a famous multi-century old number theory conjecture. Section II is intended to the latter and also explains how the dataset is built; readers only interested in the applications can skip it and move to section III.

It all started with a seed string S_0 consisting of $2n + 1$ bits, all zeros except a one at both ends, thus representing the integer 2^n+1. The iterated self-convolutions of the seed string, defined as $S_{k+1} = S_k * S_k$, corresponds to taking the square of the integer represented by S_k, at each iteration $k = 0, 1$ and so on. For any fixed n, when $k = n$, the first n bits of S_n match the first binary digits of the number e, give or take. This remains true when n is infinite. This also remains true if for $k = 0, 1, 2$ and so on, S_k is truncated, keeping only the first $2n$ bits on the left, at all times. All this is formally explained in [9].

The next step consists of working with different seed strings, namely $2^n + x$ with x a small integer number, positive or negative, leading to the first n binary digits at $\exp(x)$ when $k = n$. The resulting datasets has $n+1$ rows, one for each S_k ($k = 0, 1, \dots, n$). And each rows has $2n$ bits after truncation, though we are only interested in the first n bits on the left; the rightmost n bits are there to make sure that when $k = n$, the first n binary digits of $\exp(x)$ match those of S_n.

You can multiply S_k by an integer power of 2, positive or negative, so that it represents a real number lying (say) in $[1, 2[$ at all times. Then the sequence (S_k) with fixed n is a quadratic dynamical system homeomorphic both to the logistic map and the dyadic map. Its invariant measure is the reciprocal distribution, defined as $F(z) = \log_2(z)$ for $1 \leq z < 2$.

Finally, rather then starting at $k = 0$, you can start at $k = n$ with $S_n = \exp(x)$ and $2n$-bit precision, and move backward to $k = n-1, n-2$ all the way to $k = 0$. The inverse transform to $S_{k+1} = S_k * S_k$ is $S_k = \sqrt{S_{k+1}}$. However, extra care is needed as the square root operator is a one-to-two mapping. This is illustrated later in this article and also in [9]. Yet, it leads to much faster computations, and also serves to verify the correctness of the results obtained. All the material covered so far is now well established. The novelty here is consists of:

- Using $n = 10^6$ rather than 10^5 in [10] and 10^4 in [9]. This is possible thanks to leveraging the inverse transform. The code is much faster than earlier versions, and more robust.
- Testing many values of x, most not even integers. Some involving product of primes, called primorials, leading to simple and unique patterns when $k \approx n$, getting us one big step closer to understand the digit distribution of numbers related to e. With detailed explanations.

Whether you are interested in the dataset or in proving the famous conjecture (the fact that the binary digits of e are evenly distributed) the hardest part is when k gets very close to n. This is also the part of the dataset that brings the most value.

II. Deep dive into the digit sum function

In my previous article on this topic [10], I use the notations $S(n, k, x)$ and $\zeta_S(n, k, x)$ to represent respectively the k-th iterate in the recursion, and the number of '1' in its first n digits. Since n and x are fixed, I use S_k and ζ_k here, instead. I showed how peculiar the behavior of the digit sum function ζ_k is when $x = \pm 1$ or $x = 3$ and $0 \leq k \leq n$. It looks like a quantum function with values depending on which residue class k belongs to. To the contrary, if the seed S_0 is a random string, then all iterates S_1, S_2 and so on are usually random. Exception are rare but exist, in the same way and for the same reasons that not all reals are normal numbers.

Note that $S_k = (S_0)^m$ with $m = 2^k$. Conversely, the inverse system starting with $S_n = \exp(x)$ and going backward, leads to

$$S_{n-k} = \exp\left(\frac{x}{2^k}\right) = 1 + \frac{1}{1!}\frac{x}{2^k} + \frac{1}{2!}\frac{x^2}{2^{2k}} + \cdots \qquad (1)$$

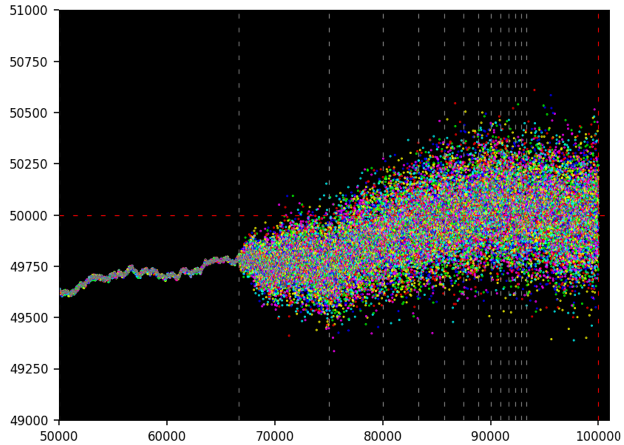

Fig. 1. Adjusted digit sum ζ'_k, random seed, $n = 10^5$, k on X-axis

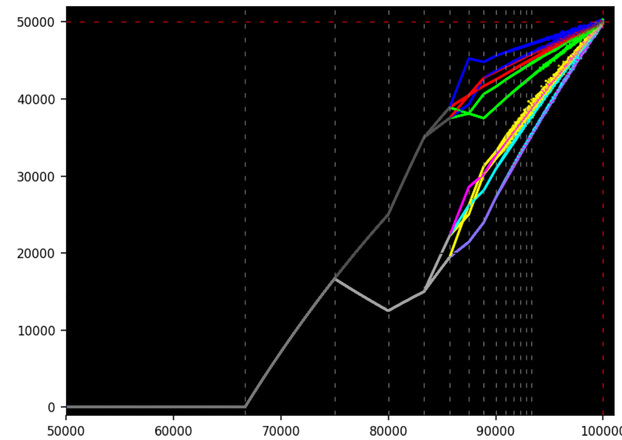

Fig. 2. Adjusted digit sum ζ'_k, seed with $x = 1$, $n = 10^5$, k on X-axis

for $k = 0, 1$ and so on. Formula (1) intuitively explains many of the patterns observed when k is a large integer, say $k = \rho n$ with $0 < \rho < 1$. It can be rewritten as

$$S_k = 1 + \frac{1}{1!}\frac{x}{2^{n-k}} + \frac{1}{2!}\frac{x^2}{2^{2(n-k)}} + \cdots \qquad (2)$$

The starting value $S_n = \exp(x)$ in the inverse system is called the reverse seed. Since S_n is truncated to the first $2n$ bits in the inverse system, barring issues due to the square root not being uniquely defined, one can expect S_0 to be correct up to the first n bits. If instead we start backward at $k = n$ with $3n$ bits of precision, we end up with a precision of $2n$ bits at S_0. Then moving forward with the standard system ($k = 0, 1$ and so on), we end up back at the same S_n when $k = n$, but now with a precision of n bits. This full trip back and forth can help validate the computations.

A. *Digit sum function: examples*

Let $\{\cdot\}$ denotes the fractional part function. One can show that $S_k = 2^{\nu_k + \gamma_k}$ where ν_k is an integer (positive or negative) and $\gamma_k = \{\, 2^k \log_2 S_0 \,\}$. It follows that if S_0 is a random seed, then S_k is almost surely random for all $k > 0$. The converse is not true: if $S_k > 1$, the successive square roots S_{k-1}, S_{k-2} and so on are closer and closer to 1. Typically, if S_k is random, S_0 will start with the digit '1', followed by k zeros. This is due to the square root not being uniquely defined: for instance, '1' and $\sqrt{2}$ are both roots of '1', as explained in [9].

As a result, it makes sense to define an alternate version of the digit sum function. This new function called adjusted digit sum and denoted as ζ'_k, counts the number of '1' in the first n digits of S_k starting at position $n - k$, with $0 \le k \le n$.

Figure 1 shows the behavior of the adjusted digit sum ζ'_k on the Y-axis, with k on the X-axis, when the reverse seed S_n is a random string with $2n$ bits, using the backward iterations. Here $n = 10^5$. The colors mean nothing, and ζ'_k randomly hovers around $n/2$ as expected. For details, zoom in on the picture.

Figure 2 shows the behavior of ζ'_k for the reverse seed $S_n = e$ truncated to $2n$ bits with $n = 10^5$. This corresponds to the seed $S_0 = 2^n + 1$ when using the forward rather than backward iterations. It exhibits the same structure as Figures 1 and 2 in [10], also based on the same seed but using the non-adjusted digit sum ζ_k instead of ζ'_k. As in Figure 1 in this article, the colors represents the congruential classes (specifically, k mod 6) and convey important information this time. The quantum dynamics of the system are also obvious.

The vertical dashed lines show change points in the behavior of ζ'_k. As discussed in [9], they occur at specific values $k_\rho = \rho n$ on the X-axis, for $\rho = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}$ and so on. Zoom in to better see them. From the picture, it seems easy to prove that e has about 50% of '1' in its first n digits, by looking at S_k as k approaches n, and then let $n \to \infty$. However, we are still very far from a formal proof. In particular, the behavior of ζ'_k becomes quite chaotic starting around $k \approx 0.88 \cdot n$, and even more so as k gets very close to n.

However, the goal is to prove any result about the binary digits of any major math constant, no matter how weak, as long as it as a deep, ground-breaking result. None are known to this day. An example of weak yet deep result would be this: there is a known rational number x such that the number of '1' in the binary expansion of $\exp(x)$, exceeds 30% in the first n digits, for infinitely many values of n. Section II-B goes one step further, in an attempt to reach such a major milestone.

B. *Spectacular behavior of digit sum with primorials*

Formula (2) is a mix of good and bad news. The well space-out powers of 2 in the denominator of each term contribute to the strong structure observed when $x = 1$. But the factorials contribute to the chaos observed when k gets very close to n. It would seem that if you replace x by a product of consecutive primes – the smallest product that counteract the factorials – things would become nicer. Indeed, this is the case, and the topic of my discussion in this section. However, it is not the magic bullet that will solve all problems.

Let $\pi_\kappa = p_1 \cdot p_2 \cdots p_\kappa$ be the product of the first κ primes also known as the κ-th primorial, with $p_1 = 2$. The nice features in formula (2) are preserved if x is an integer divided by a power of 2, that is, a dyadic rational. Also, in $S_0 = 2^n + x$, we need x to be small, that is $x = O(1)$. Thus, I use

$$x_\kappa = \pi_\kappa \cdot 2^{\nu_\kappa}, \quad \text{with } \nu_\kappa = -\lfloor \log_2 \pi_\kappa \rfloor. \qquad (3)$$

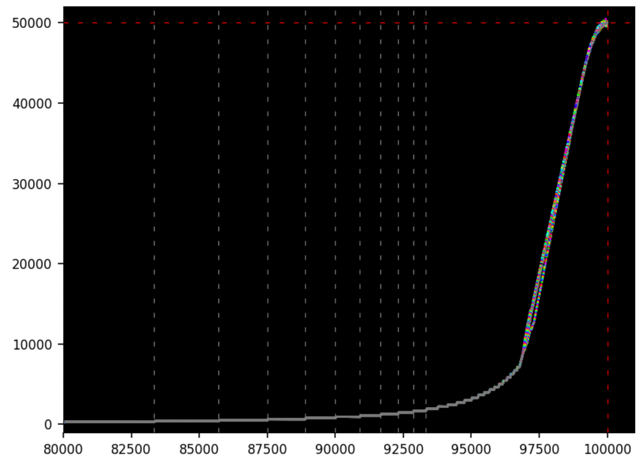

Fig. 3. Adjusted digit sum ζ'_k with primorial, $n = 10^5$, k on X-axis

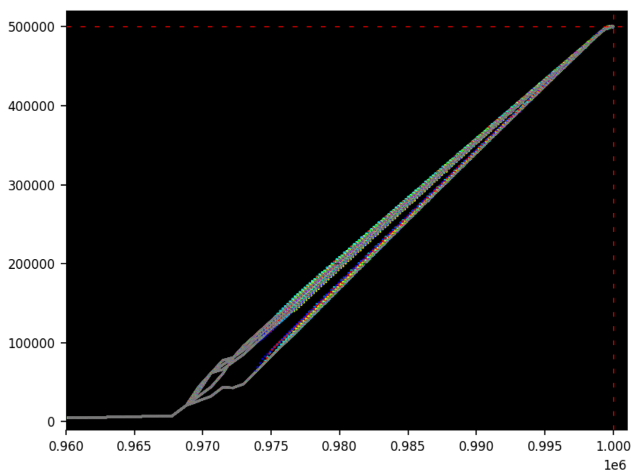

Fig. 4. Adjusted digit sum ζ'_k with primorial, $n = 10^6$, k on X-axis

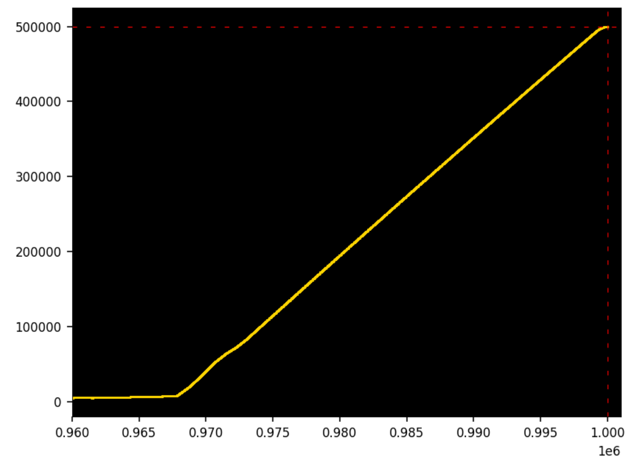

Fig. 5. Moving average applied to Figure 4, window size is 60

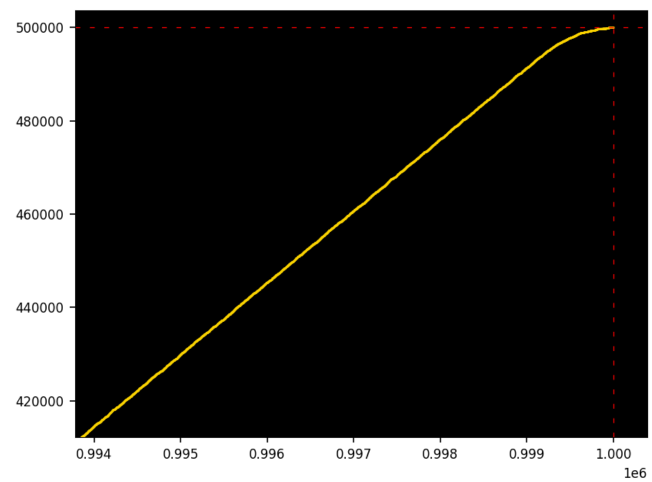

Fig. 6. Zoom on top right corner in Figure 5

Figure 3 and 4 show the substantial reduction in chaos as k gets very close to n, when using $x = x_\kappa$ defined by formula (3) with $\kappa = 30$, instead of $x = 1$. The leftover chaos can still be further reduced, either by showing values of ζ'_k only for (say) $k \equiv 25 \bmod 60$, or by averaging 60 consecutive values in a moving average. Figure 5 and 6 feature the latter, this time with $n = 10^6$. It seems that there is no more chaos left in the last figure, increasing hopes towards proving a famous conjecture. But this is an illusion due to the limited granularity in the picture. Note that I truncated the X- and Y-axis to provide the best possible views, given the fact that real action takes place when $k > 0.90 \cdot n$. Again, zoom in to get better views.

Intuitively, it seems like increasing κ indefinitely is the way to go to eliminate any chaos left. Then use a subsequence κ_1, κ_2 and so on so that x_{κ_j} converges (say) to $x = 1$ when $j \to \infty$. However we can find subsequences converging to any x in $[1, 2[$ because the sequence of logarithm of primorials is dense modulo 1, see here. In other words, you could then use my framework to prove that some non-normal numbers are normal. Clearly that approach can not work and indeed, beyond some rather modest κ, the amount of chaos starts increasing again. This is due to the fact a very large numerator π_κ in $x = x_\kappa$ in formula (3) results in extensive carry-over across multiple terms in formula (2), thus increasing chaos.

At this point, the most spectacular provable result is the unique behavior of the adjusted digit sum ζ'_k when $x = x_\kappa$ and κ is an integer between 10 and 30. Among all possible real numbers x, these few x_κ produce the slowest growing ζ'_k functions when $k < 0.90 \cdot n$ and the least chaotic ones.

Before exploring very deep results, the next step consists in studying the moving average and tail functions, defined as

$$\varphi(\rho) = \lim_{n\to\infty} \frac{1}{(1-\rho)n^2} \sum_{k=\lfloor \rho n\rfloor+1}^{n} \zeta'_k, \quad 0 \le \rho \le 1.$$

$$\psi(\rho) = \lim_{n\to\infty} \frac{1}{2n^{3/2}} \sum_{k=-\sqrt{n}}^{\sqrt{n}} \zeta'_{\lfloor \rho n + k\rfloor}, \quad 0 \le \rho \le 1.$$

As usual, things are very simple when $\rho < 0.50$, and really hard when $\rho > 0.95$. Are the functions φ and ψ well defined? Are they continuous? What about the derivatives? Establishing that $\varphi(1) = \psi(1) = \frac{1}{2}$, for a specific x, does not prove that the binary digits of $\exp(x)$ are evenly distributed. But it is a first step in that direction.

C. Future research

I haven't tried very large values of n yet, say 2^{500} rather than $n = 10^6$ as of now. Then, other than random strings, I barely started to explore integer values of x larger than x_κ with $\kappa = 20{,}000$. Note that when starting with the seed $S_0 = 2^n + x$ with $2n$ bits of precision, you end up with a precision of about $n - \tau$ bits on the target number $\exp(x)$ at iteration $k = n - \tau$, where $\tau = \lfloor \log_2 x \rfloor$. See Python program in section IV-A, where x is denoted as `p`, and τ denoted as `iplog` in the code.

Perhaps the most promising results will come from looking at the behavior of ζ'_k when k belongs to specific residue classes, especially modulo factorial integers of increasing sizes. Figure 2 is a first step in that direction, showing ζ'_k's path with a different color based on $k \bmod 6$. Likewise and not surprisingly, averaging w consecutive values of k, where w is a number with many divisors ($w = 60$ in Figure 5) leads to much smoother paths for the digit sum function. Choosing an optimal w based on n is also a topic of interest. The congruential class to which n belongs may also have an impact.

Finally, the digit sum function has been extensively studied in other contexts, see [11]. Some of its properties are simple. For instance, $\zeta(y; b) \equiv y \bmod b - 1$, where $\zeta(y, b)$ is the sum of the digits of the integer y in base b. Working with different bases that are power of 2, with n also a power of 2, is another topic of interest.

D. 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 [18]. 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]. 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 [17].
- 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 [19]. 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 [12] 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], [16]. 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 instead of $S_0 = 2^n + 1$, you obtain strings 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 [14], [15], [20]. 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 [13].

III. Infinite Dataset and Applications

For a fixed n and x, the dataset consists of successive bit strings of length $2n$. Each string corresponds to a specific S_k with $0 \leq k \leq n$, though the code in section IV-A also generates S_k for $k > n$. Let $\rho = k/n$. When $\rho < 0.50$, the patterns are trivial. The patterns become more and more complex as ρ increases. They are extremely hard to describe and detect when $\rho > 0.98$. When $\rho > 1$, we are in full chaotic mode, with no pattern. A pattern detection algorithm fails if it detects patterns when $\rho > 1$. One that correctly identifies the patterns at $\rho = 0.95$ is superior to one that cannot find any beyond $\rho = 0.92$.

Patterns are found within each string S_k, but also across successive strings S_k and S_{k+1}, which are highly correlated, although less and less as k increases, and not at all beyond $k = n$. Thus, we have autocorrelations within a string and cross-correlations between strings, both short and long range. Strings can be split into words, either short to emulate categorical features, or long for numerical features, to mimic enterprise datasets.

In addition, the structure in the dataset allows you to test clustering algorithms: the various strings S_k can be clustered, see the colors in Figure 7. Each color represents a cluster related to the congruential class (unknown to your classifier) that k belongs to. As k increases, the number of clusters also increases, with the structure becoming more fuzzy as we approach $k = n$.

The dataset also allows you to test predictive algorithms. In particular, predicting the next strings based on historical data (the previous strings). The example discussed in [10] is related to large language models (LLMs). The length of strings can range from 10^3 to 10^7 (or more) bits. Each value of x generates a specific set of strings, that is, a particular table, thus mimicking a database system with multiple tables or time series. It can be used as generic, very versatile type of synthetic data, or to create synthetic data. The digit sum function plays the role of a response, summary, or aggregate feature; also, it can be computed in bases other than 2.

Finally, the iterated self-convolution $S_{k+1} = S_k * S_k$ or its inverse – the iterated square root of a string – is useful to design efficient, fast pseudo-random number generators (PRNGs) linked to pattern-free transcendental numbers with infinite period (such as e), and thus with much better randomness properties than classical congruential generators. For details, see [10]. The connection to dynamical systems and quantum dynamics can be exploited for simulations, modeling purposes, and agent-based modeling.

Rather than sharing the dataset, I share the code to generate it, in section IV. Given x, the corresponding full dataset is infinite since n can be as large as you want, and there are infinitely many values of x to play with, each generating its own table. For customization based on your enterprise needs, help with data generation, interpretation, sample size, simulations, feature generation, and any other questions about building your own enterprise version to address your priorities, contact the author.

IV. Python Code

Here I share two different versions of my program: one based on the forward recursion in section IV-A, starting with S_0, and the other one based on the backward recursion in section IV-B, starting wit S_n. The latter is faster despite using square roots rather than squares. But what makes it much faster is that we are mostly interested in S_k with k/n between 0.90 and 1.00. Thus, the backward recursion eliminates 90% of the iterations between $k = 0$ and $k = n$.

The core is quite small and simple: it consists of part 1 in the code (called `main`) for the forward recursion, and part 3 (the `iexp` function) for the backward recursion. I use the gmpy2 library to process very large numbers with arbitrary precision. The code for the backward recursion is more recent and integrates the new enhancements and functionalities. Both programs also produce extra plots not discussed in this article. In both programs, x is represented by the variable `p`, while S_k is represented by the variable `prod`.

A. Forward iterations

There is a mechanism to accelerate computations by a factor at least 2, using the function `rstrip_zeros` which removes useless trailing zeros on the right in all strings S_k. Also, I use alternate computations to double-check that the first n binary digits of S_n (give or take) match those of $\exp(x)$. See part 2 in the code. The code is also on GitHub, here.

```
# Faster version than number_theory_fast_v2.py
#   - at iteration k, keep only 2n-k digits in S(n,
    k, x) instead of 2n
#   - also remove the trainling 0 on the right, in
    S(n, k, x)
#   - drawback: I get 19985 correct digits instead
    of 19998 if n = 20000

from primePy import primes
import gmpy2
```

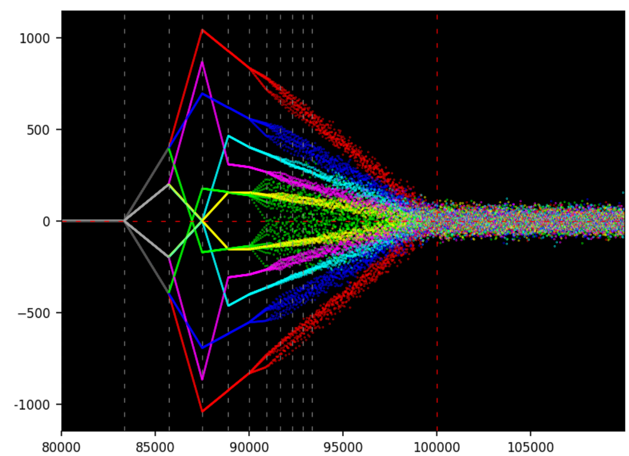

Fig. 7. $\zeta_{k+60} - \zeta_k$, with $n = 10^5, x = 1$ and k on X-axis

```
import numpy as np

n = 100000
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 # try number with many divisors: 12,
    30, ...
colorTable = generate_contrasting_colors(ncolors)


def rstrip_zeros(string):

   # remove '0' on the right after last '1'

   newstring = string
   if string[-1] == '0':
      k = -1
      while string[k] == '0':
         k -= 1
      newstring = string[:k+1]
   return(newstring)


#--- 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
```

```
# p = 2*3*5*7*11*13*17*19*23*29
p = 1 # denoted as x in the paper

# first n binary digits at iteration k=n are those
    of exp(p)
# if p irrational, seed = 2^(2n) + int(2^n * p)
# if p integer, seed = 2^n + p

iplog = 0

if p != int(p):
   # for p irrational, like p = sqrt(2)
   p = gmpy2.floor((2**n)*gmpy2.mpfr(p))
   prod = gmpy2.floor(2**(2*n) + p)
else:
   # for integer, small or large
   iplog = gmpy2.floor(gmpy2.log2(abs(p)))
   prod = gmpy2.floor(2**n + p)

# 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 = prod*prod
   pstri = bin(gmpy2.mpz(prod)) # mpz is round to
       integer, not floor
   stri = pstri[0:2*n-k]  # faster than pstri[0:
       L+2] in older version
   stri = rstrip_zeros(stri) # new to this version
       (faster)
   prod = int(stri, 2)
   prod = gmpy2.floor(prod)

   if k > kmin and k < kmax:
      stri = stri[2:]
      lstri = len(stri)
      if k == n-iplog:
         e_approx = stri
      estri = stri[0:n] # leftmost n digits
      ecnt1 = estri.count('1')
      ecnt1f = stri.count('1')
      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)+"\t"+str(ecnt1f)+"\n")

      if k%1000 == 0:
         print("%6d %6d %6d %6d" %(k, ecnt1,
             lstri,ecnt1f))
   if stri[-1] == '0':
      print(k, stri[-10:])

OUT.close()


#--- 2. Compute bits of e and count correct bits in
    my computation

# Set precision to L binary digits
gmpy2.get_context().precision = 4*n
if p == int(p):
   e_value = gmpy2.exp(p/2**iplog)
else:
   e_value = gmpy2.exp(p)

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

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

e_approx_decimal = 0
for k in range(80):
   e_approx_decimal += int(e_approx[k])/(2**k)
print("e_approx, up to power of 2:",
    e_approx_decimal)


#--- 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=0.01,
    c=arr_colors)
#plt.plot(xvalues, arr_count1,linewidth=0.1,
    c='gold')

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%
ymax = 0.52 * n
plt.ylim([-0.01 * n, ymax])
#plt.ylim([-0.01 * n, 1.01*n])

plt.xlim([kmin, kmax])
# plt.xlim([0.0*n, kmax])
plt.show()


#--- 4. Create the moving average plot

arr_avg = []
arr_xval = []
arr_count1 = np.array(arr_count1)
w = 6 # moving average window

for k in range(0, len(arr_count1)-w):
   y_avg = np.average(arr_count1[k:k+w])
   arr_avg.append(y_avg)
   arr_xval.append(k)

plt.scatter(arr_xval,arr_avg,s=0.0002, c='gold')
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))
plt.xlim(0.00*n, kmax-w)
plt.ylim(-0.01*n, ymax)
```

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

plt.show()

nv = len(arr_xval)
st = int(4*n/5)
arr_delta = np.array(arr_avg[1:nv]) -
     np.array(arr_avg[0:nv-1])
plt.scatter(arr_xval[st+1:nv],
     arr_delta[st+0:nv-1], s=0.08,
     c=arr_colors[st+1:nv])
for k in range(1,15):
   plt.axvline(x=k*n/(k+1),c='gray',linestyle='--',
       linewidth=0.6,dashes=(5, 10))
plt.axvline(x=n, color='red', linestyle='-',
     linewidth = 0.6, dashes=(5, 10))
plt.axhline(y=0,color='red',linestyle='--',
     linewidth=0.6,dashes=(5,10))
plt.xlim(st, nv)
plt.show()


#--- 5. Create AR scatterplot

nv = n
plt.scatter(arr_count1[nv-2000-1:nv-1],
     arr_count1[nv-2000:nv], s=0.04,
     c=arr_colors[nv-2000-1:nv-1])
plt.show()
```

B. Backward iterations

In the code, the reverse seed $S_n = \exp(x)$ with a precision of $2n$ bits is denoted as `z`, while x is denoted as p. In the main section (part 3), when `truncate` is set to `True`, the leftmost $n-k$ digits are ignored in S_k as they are usually all zeros except for the first one. Instead, I use the next n digits to compute the digit sum function ζ'_k. To avoid confusion, I call it the adjusted digit sum. The standard digit sum is denoted as ζ_k. The code is also on GitHub, here.

```
import gmpy2
from gmpy2 import mpfr
import colorsys

#--- 1. Create table of contrasted colors

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

#--- 2. Functions related to primorials

def update_q(q, k, file):

   q = gmpy2.mpz(k*q)
   iq = 2**gmpy2.floor(gmpy2.log2(q)) # use floor,
       not mpz (mpz = round)
   f = str(gmpy2.mpfr(q/iq))[0:20]
   file.write(str(k) + "\t" + f + "\n")
   return(q)


def primorial(kappa, mode="primorial"):

   # mode = "primorial" --> return p = #kappa with
       correct precision
   # mode = "factorial" --> return p = kappa! with
       correct precision

   from primePy import primes
   ctx = gmpy2.get_context()
   old_precision = ctx.precision
   ctx.precision = 2*kappa
   q = gmpy2.mpz(1)

   OUT = open("primorials.txt", "w")
   # values of r = q/2^int(log2 q) distributed in
       [1, 2] like F(r) = log2 r
   # this is called the reciprocal distribution
   for k in range(2, kappa+1):
      if mode == "primorial":
         if primes.check(k):
            q = update_q(q, k, OUT)
      elif type == "factorials":
         q = update_q(q, k, OUT)
   OUT.close()

   iq = int(gmpy2.floor(gmpy2.log2(q)))
   print("Primorial precision: %d bits | min
       needed: %d bits" %(ctx.precision, iq))
   print()
   p = q
   ctx.precision = old_precision
   return(p)

#--- 3. Main function: the backwars iterations

def iexp(n, start, iters, ncolors, z, u, v,
    truncate):

   arr_count1 = []
   arr_colors = []
   xvalues = []
   ecnt1 = -1

   pow2 = 2**(start)
   z = gmpy2.exp(gmpy2.log(z)/pow2) # z =
       exp[p^(1/2^start)]

   for k in range(n-start, n-start-iters, -1):

      iz = gmpy2.mpz(gmpy2.mpfr(2**(n+5) * z)) ###
          why n+5 ??

      if k % u == v:
         if truncate:
            # strip 1 and first n-k digits (zeros)
                on the left
            stri = bin(iz)[2+n-k:2*n-k+2+1]
            # stri = bin(iz)[2+n-2*k:2*n-k+2+1]
            ecnt1 = stri.count('1') * n/len(stri)
         else:
            stri = bin(iz)[2:n+2+1]
            ecnt1 = stri.count('1')

         arr_count1.append(ecnt1)
         color = colorTable[k % ncolors]
         arr_colors.append(color)
         xvalues.append(k)

      if k%1000 == 0:
         print(k, ecnt1)
      z = gmpy2.sqrt(z)

   return(arr_count1, arr_colors, xvalues)

#--- 4. Function to create reverse seed z
```

```
# initialize seed z

def initialize_reverse_seed(seed_type):

   if seed_type == "primorial":
      kappa = 30 # try 3, 10, 15, 30, 300, 3000
      p = primorial(kappa)
      iplog = int(gmpy2.log2(p))
      p = gmpy2.mpfr(p/(2**iplog))
      # try replacing p by -p
      z = gmpy2.exp(p)

   elif seed_type == "random":
      import numpy as np
      np_seed = 6696
      stri =""
      np.random.seed(np_seed)
      for k in range(2*n+1):
         d = np.random.randint(2)
         stri += str(d)
      p = gmpy2.mpz(int(stri, 2)) ###
      iplog = int(gmpy2.log2(p))
      p = gmpy2.mpfr(p/(2**iplog))
      z = gmpy2.exp(p)

   elif seed_type == "integer":
      # also try -1 (backward/forward algo show
          different paths)
      z = gmpy2.exp(1)

   elif seed_type == "misc":
      z = gmpy2.exp(gmpy2.sqrt(2))
   return(z)


#--- 5. Main

# set u=1, v=0 to show all k from k=n-start down to
    k=n-start-iters
# to show results only for k=v mod u, try u=60, v=25
u = 1 # 60 (integer)
v = 0 # 25 (residue modulo u)
# n = 3*7*11*13*u + v # choose n such that n = u
    mod v
n = 100000
truncate = True
start = 0
iters = 50000
iters = min(n-start, iters)
ctx = gmpy2.get_context()
ctx.precision = 2*n

seed_type = "primorial"

ncolors = 6 # try number with many divisors: 12,
    30, ...

colorTable = generate_contrasting_colors(ncolors)
z = initialize_reverse_seed(seed_type)
(arr_count1, arr_colors, xvalues) = iexp(n, start,
    iters, ncolors, z, u, v, truncate)


#--- 6. 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=0.2, c=arr_colors)
#plt.plot(xvalues, arr_count1,linewidth=0.1,
    c='gold')
```

```
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))

plt.ylim([-0.01 * n, 0.52*n])
plt.xlim([0.70 * n, 1.02*n])
plt.show()


#--- 7. Create the moving average plot

arr_avg = []
arr_xval = []
arr_count1 = np.array(arr_count1)
w = 60
for k in range(0, len(arr_count1)-w):
   #tmp = arr_count1[k:k+w]
   #print(k, len(tmp)) ##,arr_count1[k])
   y_avg = np.average(arr_count1[k:k+w])
   arr_avg.append(y_avg)
   arr_xval.append(n-start-k)

plt.scatter(arr_xval,arr_avg,s=0.02, c='gold')
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))
plt.xlim([0.50*n, 1.01*n])
for k in range(1,15):
   plt.axvline(x=k*n/(k+1),c='gray',linestyle='--',
       linewidth=0.6,dashes=(5, 10))
plt.show()

nv = len(arr_xval)
st = 0 ##int(4*n/5)
arr_delta =
    np.array(arr_avg[0:nv-1])-np.array(arr_avg[1:nv])
plt.scatter(arr_xval[st+1:nv],
    arr_delta[st+0:nv-1], s=0.08,
    c=arr_colors[st+1:nv])
for k in range(1,15):
   plt.axvline(x=k*n/(k+1),c='gray',linestyle='--',
       linewidth=0.6,dashes=(5, 10))
plt.axvline(x=n, color='red', linestyle='-',
    linewidth = 0.6, dashes=(5, 10))
plt.axhline(y=0,color='red',linestyle='--',
    linewidth=0.6,dashes=(5,10))
plt.show()


#--- 8. One more scatterplot

nv = len(arr_count1)
w = 1
arr_delta = np.array(arr_count1[0:nv-w]) -
    np.array(arr_count1[w:nv])
plt.scatter(xvalues[w:nv], arr_delta, s=0.08,
    c=arr_colors[w:nv])
for k in range(1,15):
   plt.axvline(x=k*n/(k+1),c='gray',linestyle='--',
       linewidth=0.6,dashes=(5, 10))
plt.axvline(x=n, color='red', linestyle='-',
    linewidth = 0.6, dashes=(5, 10))
plt.axhline(y=0,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]. 4

[2] Christoph Aistleitner et al. Normal numbers: Arithmetic, computational and probabilistic aspects. 2016. Workshop [Link]. 4

[3] David Bailey, Jonathan Borwein, and Neil Calkin. *Experimental Mathematics in Action*. A K Peters, 2007. 4

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

[5] James Dolan. Carrying is a 2-cocycle. *Preprint*, pages 1–9, 2023. [Link]. 4

[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]. 4

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

[8] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLTechniques.com, 2024. [Link]. 4

[9] Vincent Granville. Cracking a famous multi-century old math conjecture. *Preprint*, 2025. MLTechniques.com [Link]. 1, 2

[10] Vincent Granville. Llm challenge with petabytes of data to prove famous number theory conjecture. *Preprint*, 2025. MLTechniques.com [Link]. 1, 2, 4, 5

[11] 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]. 4

[12] 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]. 4

[13] Mohammadamin Moradi et al. Data-driven model discovery with Kolmogorov-Arnold networks. *Preprint*, pages 1–6, 2024. arXiv:2409.15167 [Link]. 4

[14] 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]. 4

[15] Yury V. Tiumentsev and Mikhail V. Egorchev. *Neural Network Modeling and Identification of Dynamical Systems*. Elsevier, 2019. 4

[16] Chukwudubem Umeano and Oleksandr Kyriienko. Ground state-based quantum feature maps. *Preprint*, pages 1–8, 2024. arXiv:2024.07174 [Link]. 4

[17] Joseph Vandehey. On the binary digits of $\sqrt{2}$. *Preprint*, pages 1–6, 2017. arXiv:1711.01722 [Link]. 4

[18] Troy Vasiga and Jeffrey Shallit. On the iteration of certain quadratic maps over GF(p). *Discrete Mathematics*, 277:219–240, 2004. [Link]. 4

[19] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, second edition, 2012. 4

[20] 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]. 4

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.