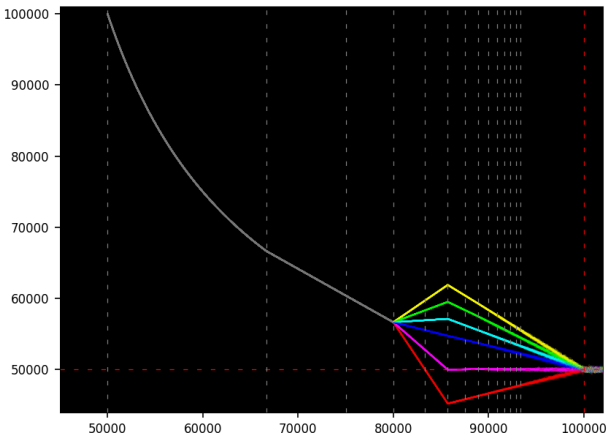
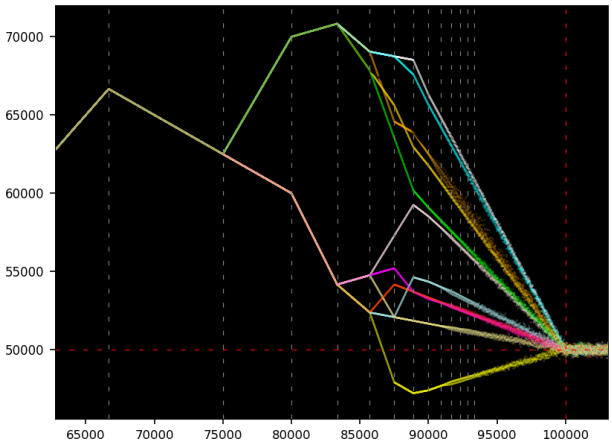
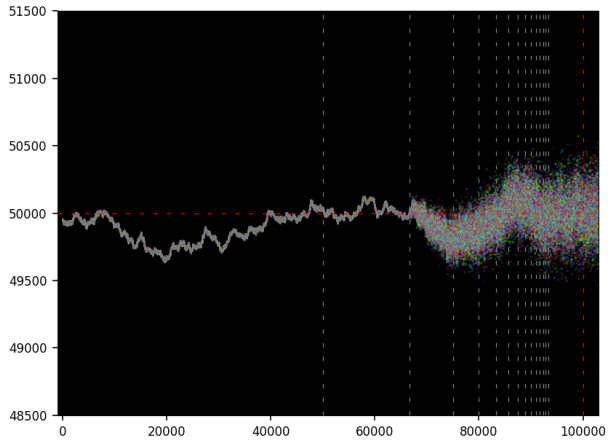
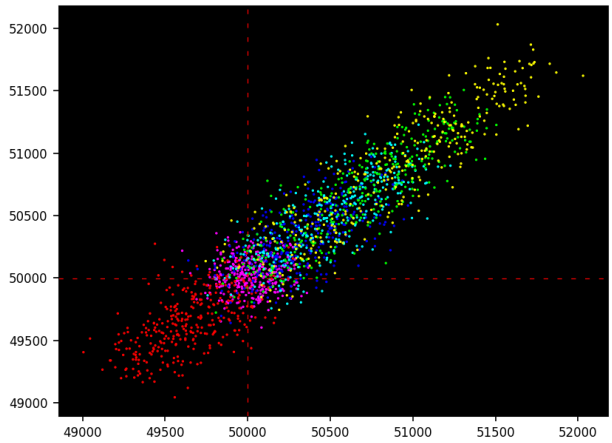
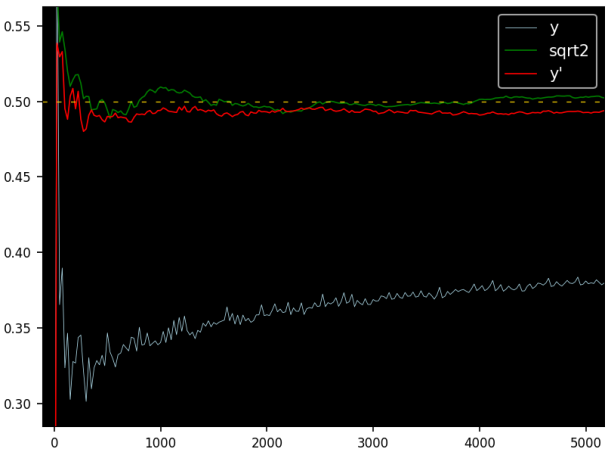
Breakthroughs on the Digit Distribution of Classic Constants

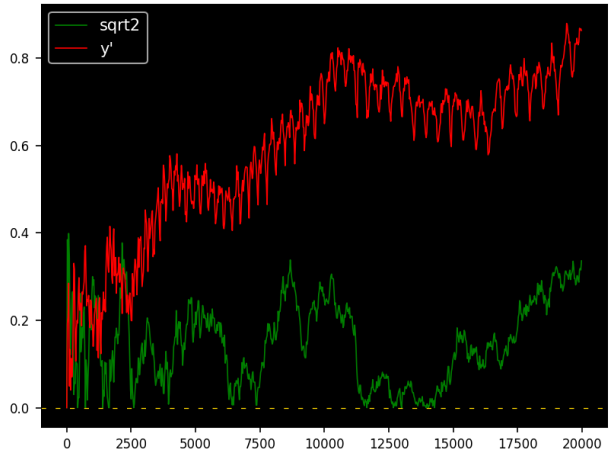












Contents

1 Link Between the Digit Sum Function and Auto-Convolutions **6**
1.1 A new type of string operators . . . 6
1.1.1 String class . . . 6
1.1.2 Truncation, number representation, and convergence . . . 6
1.1.3 String convolution and square root . . . 7
1.1.4 Well-balanced strings . . . 7
1.2 Infinite sequences of iterated auto-convoluted strings . . . 7
1.2.1 Visualizing iterated auto-convoluted string sequences . . . 8
1.2.2 Fundamental result about the number of zeros and ones . . . 9
1.3 Solving one of the greatest mathematical mysteries . . . 12
1.3.1 Testing different seeds . . . 13
1.3.2 Another interesting sequence . . . 15
1.3.3 Surprising, unpublished results about the digit distribution . . . 15
1.3.4 Application to cryptography . . . 15
1.3.5 Python code . . . 16

2 Petabyte Challenge to Help Crack the Famous Math Conjecture **19**
2.1 Introduction . . . 19
2.2 Spectacular behavior of the digit sum function . . . 20
2.2.1 Potential scenarios . . . 20
2.2.2 Dynamics of the unbroken bifurcation process . . . 21
2.3 Case studies . . . 22
2.4 An extreme case . . . 23
2.5 Applications and AI Challenge with petabytes dataset . . . 24
2.5.1 AI challenge . . . 24
2.5.2 The dataset . . . 25
2.6 Python code . . . 26

3 From Digit Sum to Universal Dataset and Benchmarking AI Algorithms **29**
3.1 Introduction . . . 29
3.2 Deep dive into the digit sum function . . . 30
3.2.1 Digit sum function: examples . . . 30
3.2.2 Spectacular behavior of digit sum with primorials . . . 31
3.2.3 Future research . . . 32
3.2.4 References . . . 33
3.2.5 Comparison with standard methodology . . . 33
3.3 Infinite dataset and applications . . . 34
3.4 Python code . . . 34
3.4.1 Forward iterations . . . 35
3.4.2 Backward iterations . . . 38

4 Quantum Dynamics, Logistic Map, and Digit Distribution of Special Constants **41**
4.1 Introduction . . . 41
4.2 Logistic map and the digit sum function . . . 42
4.2.1 Model comparison, with illustrations . . . 42
4.2.2 Normality of special math constants . . . 45
4.2.3 Applications and references . . . 45
4.3 Re-balancing an uneven digit distribution . . . 46
4.3.1 Digit-balancing transforms . . . 46

4.3.2 Digit block balancing . . . 48
4.4 Conclusion . . . 52
4.5 Main Python code . . . 53

5 Test of Normality and Digit Distribution of Algebraic Numbers 56
5.1 Simple normality test with application to PRNGs . . . 56
5.1.1 High-performance computing with Chebyshev polynomials . . . 57
5.1.2 Application with test of randomness and Python code . . . 58
5.1.3 Problem and solution . . . 60
5.2 Another interesting discrete quadratic dynamical system . . . 60
5.2.1 Case with multiple limits . . . 61
5.2.2 Case with single limit . . . 61
5.3 Surprising results about the digit distribution . . . 62
5.3.1 Python code for the computer-assisted proof of the main theorem . . . 64
5.3.2 Python code for the deeper theorem . . . 66
5.4 Strong patterns found in the digits of algebraic numbers . . . 67
5.4.1 Python code to compute the digits . . . 69
5.5 Correlated bit strings: seminal result and applications . . . 70
5.5.1 Autocorrelations in related sequences . . . 72
5.5.2 Python code . . . 72

6 Quantum States and the Riemann Zeta Function 74
6.1 Synthetic primes, quantum states, and the Riemann Hypothesis . . . 74
6.1.1 Definitions . . . 74
6.1.2 Building a Beurling eta function by deletion . . . 75
6.1.3 Building a Beurling eta function by swapping . . . 75
6.1.4 Applications and Python code . . . 76
6.2 Quantum derivatives, GenAI, and the Riemann Hypothesis . . . 82
6.2.1 Cornerstone result to bypass the roadblocks . . . 83
6.2.2 Quantum derivative of functions nowhere differentiable . . . 84
6.2.3 Project and solution . . . 85
6.2.4 Python code . . . 89

7 Quantum, chaotic and fractal types of algorithmic convergence 94
7.1 Digit count generating function and fractal convergence . . . 94
7.2 Other examples of chaotic convergence . . . 96
7.2.1 Smooth convergence but with multiple branches . . . 96
7.2.2 Chaotic convergence with multiple branches . . . 96
7.2.3 Deep dive into the chaotic case . . . 100
7.2.4 Interesting connection between 3^n and the digits of $\sqrt{2}$ . . . 102
7.3 Random polynomials and spectral analysis of digit distributions . . . 105
7.3.1 Orbits of the digit generating function . . . 105
7.3.2 Connection to Littlewood, Shapiro, and Newman Polynomials . . . 107
7.3.3 Spectral signature of even versus uneven bit strings . . . 108
7.3.4 Mesmerizing video featuring 1000 cases, with Python code . . . 110

8 Towards Deterministic AI: The NPG Random Generator 115
8.1 NPG: fast non-periodic pseudo-random number generator . . . 115
8.1.1 NPG non-standard fundamental examples . . . 115
8.1.2 Deterministic vs cryptographically secure NPG vs other PRNGs . . . 116
8.1.3 NPG success vs PCG64 and PCG64DXSM statistical failures . . . 117
8.1.4 Optimum parameters to speed up computations . . . 119
8.2 Compute time and Python code . . . 119
8.2.1 NPG source code . . . 120
8.2.2 PRNG tests of randomness . . . 123

Appendix A The Pi Day xLLM agent 127

Appendix B Convolution, Approximations, and Signal Processing 130
B.1 Approximations to mathematical function . . . 130
B.1.1 Finding the roots of ζ with fast-converging series . . . 130
B.1.2 Approximation based on quantization . . . 131

B.2 Non-causal discrete convolution with Gaussian kernel 132
B.2.1 Problem 132
B.2.2 Solution 133
B.2.3 Python code 134

Bibliography **137**

Index **139**

Introduction

Since the first edition entitled "0 and 1 – From Elemental Math to Quantum AI" and released in early 2025, a lot of progress has been made. Fascinating new results have been uncovered and proved by the author, many still leading to interesting quantum dynamics. In 100 pages, the new material presented here goes far beyond any articles and books published so far on the topic.

This second edition offers a trip deep into the most elusive and fascinating multi-century old conjecture in number theory: are the binary digits of the fundamental math constants evenly distributed? No one even knows if the proportions of '0' and '1' exist, for any of them: it could oscillate indefinitely between 0% and 100%. This new edition includes a new chapter on testing randomness with a much simplified version of Weyl's criterion. It also features a breakthrough result regarding the binary digit distribution, stating that the proportion of 1 must lie between $\frac{5}{16}$ and $\frac{11}{16}$ for a large class of numbers including all the standard mathematical constants such as π, e or $\sqrt{2}$. The details, with a hard, computer-assisted proof, are in the new chapter 5 and published here for the first time. In another example, I use quadratic dynamical systems on a matrix space with Chebyshev polynomials to unearth beautiful results.

This book is written in simple English even when covering advanced topics, avoiding jargon and advanced mathematics when not necessary. It is offered with enterprise-grade Python code for scientific and high-performance computing with the Gmpy2 library, numerous high-quality illustrations, a comprehensive clickable index and bibliography, along with efficient algorithms not taught in any classroom or textbook. The target audience includes professionals in computer science, physics, AI, machine learning, engineering, quantitative finance, and related fields, as well as students and beginners with one year of exposure to college-level mathematics and Python.

The book opens up new fundamental research areas in theoretical and computational number theory, numerical approximation, dynamical systems, quantum dynamics, and the physics of numbers. It has a strong emphasis on applications: automated pattern detection and theorem proving with AI, agent-based modeling, building a universal unbiased pattern-rich synthetic dataset, cryptography (fast, strong random number generators based on irrational numbers), dynamical systems with chaos detection and isolation, computer-intensive simulations, and high performance computing to handle numbers such as $2^n + 1$ at power 2^n with $n = 10^6$.

Each chapter is self-contained and can be read separately from the others. Compared to the first version, this second edition contains significantly more material, including results published here for the first time. In particular, chapters 6 and B are new additions and contain a mix of theory, applications, and off-the-beaten path problems with solution. Quantum states and the Riemann zeta function are central themes in each of them. The section on signal processing and discrete convolution is a very strong, practical introduction to the topic, serving as a cheat sheet for practitioners or as a solid presentation for beginners, summarizing in a few pages material usually spread over several chapters.

About the author

Vincent Granville is a pioneering AI scientist and mathematician, co-founder at DataScienceCentral (acquired by TechTarget in 2020), co-founder and AI lead at Bonding AI, author and patent owner. He worked with Visa, Wells Fargo, eBay, NBC, Microsoft, CNET and several startups. He is also one of top AI influencers working with NVIDIA, and publish a GenAI newsletter with 200,000 subscribers.

Vincent is a former post-doc at University of Cambridge. He published in *Journal of Number Theory*, *Journal of the Royal Statistical Society* (Series B), and *IEEE Transactions on Pattern Analysis and Machine Intelligence*. He is the author of multiple books, available here, including "Synthetic Data and Generative AI" (Elsevier, 2024). Vincent lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, probabilistic and computational number theory.

Chapter 1

Link Between the Digit Sum Function and Auto-Convolutions

The problem in question is one of the oldest and most intriguing in number theory, yet its formulation can be understood by kids in elementary school. It is about the digit distribution of well-known math constants.This chapter sets a significant milestone towards a final solution. It serves as blueprint, featuring the architecture of a highly constructive yet difficult proof, based on new theoretical developments and concepts. I worked with numbers with more than 2^{10000} digits, that is, larger than 2^m where $m = 2^{10000}$, to uncover patterns and the mechanism leading to 50% of '0' and '1' in the binary digits of one of the most well known math constants. I explain these patterns with arguments based on a new theory: the iterated auto-convolutions of infinite strings of characters and their congruence classes, akin to p-adic numbers. There is a strong connection to the deepest aspects of discrete dynamical systems approaching their chaotic regime. Also, there are practical applications to cryptography, and computations are doubled-checked using external libraries.

1.1 A new type of string operators

By string, I mean a sequence of characters as found in any programming language. I focus on binary strings, with an alphabet consisting of two characters, labeled as '0' and '1' for simplicity. Strings with at least two characters must start with '1'. It is tempting to introduce the theory that follows using very abstract concepts, also generalizing to alphabets with more than two characters. However, since the goal is to prove a result in number theory, I will focus only on the minimum needed to make the desired progress, using familiar terminology. Note that strings can have an infinite number of characters, on the right side.

1.1.1 String class

The first concept is that of string class. Two finite strings are said to be equivalent, that is, belonging to a same class, if they are identical after stripping any trailing '0' that follows the last '1' on the right side. For instance, the strings '10110100' and '101101' are equivalent. The version that ends with '1' on the right is called the canonical form. String classes are similar to modulo classes in arithmetic. Whenever I use the word string in the remaining of my discussion, it is to be understood as the associated string class in its entirety. You may use the notation $\{S\}$ to distinguish the string S from its class. We will stick to S in both cases, for simplicity.

1.1.2 Truncation, number representation, and convergence

The concept is trivial to engineers and programmers, but less obvious to mathematicians. In our context, truncation is always performed on the right side, keeping the left part of the string unchanged. From a mathematical viewpoint, a truncated string is similar to a residue in modulo classes, except that arithmetic residues consist of digits on the right, while string residues (the result of truncation) consists of characters on the left. In some sense, string classes have analogies to p-adic numbers; both can be infinite.

I now introduce the concept of number representation. I use it to define string convolution, but it could be avoided at the expense of increased abstraction. A string S represents the number x if the characters in S match the binary digits of x in the same order. The notations $S(x)$ or $S \equiv x$ mean that S represents x. The mapping is one-to-one at the class level: for instance, '1011' or '10110' represents both $13, 26, 6.5, 3.25$ and so on. The correspondence is up to a factor 2^m where m is a positive or negative integer.

Finally, a sequence of strings (S_n) converges to S if the number ρ_n of consecutive identical characters at the beginning (on the left) between S_n and S_{n+1} increases and becomes infinite as $n \to \infty$, matching those of S. If x_n represents S_n with $x_n \to x$, the following notations (shortcuts) may be used: $S_n \to S$ or $S_n \to x$. The notation is abusive since we are dealing both with string and number classes rather than specific instances. For instance, convergence to $\sqrt{2}$ means convergence to any number of the form $2^m\sqrt{2}$ where m is a positive or negative integer, including $m = \infty$.

1.1.3 String convolution and square root

If the string classes S and T represent respectively the numbers x and y, then the convolution is defined as $S * T = x \cdot y$. Again, this is up to a factor 2^m where m is a positive or negative integer. When $S = T$, I use the word auto-convolution. Moving forward, we deal with auto-convolution exclusively. The core mechanism in this product is the carry-over computations that propagate chaos from right to left, but usually with dissipation. This will soon become evident.

The inverse of the auto-convolution is the square root operator. More specifically, S is the square root of T if $S * S = T$. Both '1' are $S(\sqrt{2})$ = '1011010100...' are square roots of '1'. The square roots of '11' are $S(\sqrt{3})$ and $S(\sqrt{6})$. Note that $2\sqrt{3}$ also represents the square root of '11', but $S(2\sqrt{3}) = S(\sqrt{3})$; the two numbers belong to the same class and have identical strings.

In mathematical terms, we have a topological homeomorphism between string classes and number classes. The latter are similar to p-adic numbers. Both form a multiplicative abelian group, with the convolution product for strings (including infinite strings), equivalent to the algebraic product for numbers. The concepts of limit and convergence are well defined in both spaces, with strings inheriting the definition from the homeomorphism.

1.1.4 Well-balanced strings

A formal definition is not needed at this stage. In layman's terms, well-balanced means the absence of strong patterns in the string in question. Many definitions are possible for finite strings. For an infinite string $S(x)$, the classic definition is that x (the class of numbers that it represents) must be normal in base 2. It follows that the set of infinite strings that are not well-balanced have Lebesgue measure zero: they are incredibly rare, yet still as numerous as real numbers based on Cantor's definition of countable.

In practice, we encounter the following situation: $S_1 \subset S_2 \subset \ldots$ is an infinite sequence of embedded strings of increasing lengths. Each finite string is well-balanced (in some sense), making the limit S_∞ well-balanced, that is, free of obvious biases and patterns.

One would think that the auto-convolution of a well-balanced string is well-balanced, as this operator tends to increase or maintain mixing in the resulting sequence. While this is true in most cases, there are infinitely many exceptions. The most notorious counterexample is $S(\sqrt{2}) * S(\sqrt{2})$ = '1'. All mathematicians believe that $S(\sqrt{2})$ is well-balanced. However this fact is still unproved to this day, not even in this paper, even when using the weakest possible definition of well-balanced.

1.2 Infinite sequences of iterated auto-convoluted strings

In this section, I discuss the new machinery leading to the mathematical breakthrough. The link to the number theory conjecture addressed here is established in section 1.3. Readers with a good mathematical intuition may be able to assess the significance of the material shared here, and what it really is all about. If not, the secret is revealed in section 1.3.

For each integer $n > 0$, we create an infinite sequence of iterated auto-convoluted strings, where the initial string S_n^0 consists of $n + 1$ characters, all equal to '0' except the first and last one equal to '1'. In other words,

$$S_n^{k+1} = S_n^k * S_n^k, \quad k = 0, 1, \ldots \tag{1.1}$$

If for instance $n = 1000$, it very quickly leads to titanic strings of incredible size, unmanageable even if you had access to the entire computing power in the universe for trillions of years. To avoid this problem, I truncate the strings, keeping only the first $2n$ characters on the left at each iteration. One can prove that this truncation preserves the first $n - 3$ characters in the following sense:

Theorem 1.2.1 *Let x_n (defined up to a factor 2^m where m can be any positive or negative integer) be the number representing the string S_n^n using truncation to first $2n$ characters at each iteration in recursion (1.1). Then, the first $n - 3$ digits of x_{n+k} are identical regardless of $k = 0, 1, 2$ and so on. This is also true when $k \to \infty$.*

This theorem is easy to prove. I choose m such that $2 \leq x_n < 4$, regardless of n. Thus m depends on n. Also this condition uniquely defines x_n. The reason for this choice will become obvious in section 1.3, but I prefer to keep the secret for now. An obvious consequence of the truncation is the fact that for a fixed n, the sequence (S_n^k) indexed by k eventually becomes periodic, though the period is gigantic. For an empirical verification of Theorem 1.2.1, the Python code also performs a precision check using an external library for computations, based on the exact value of x_∞.

Theorem 1.2.1 can be restated as follows: the string S_n^n converges as $n \to \infty$, gaining about one additional character that will remain unchanged when moving from n to $n+1$, despite the truncation. In a nutshell, the truncation consists of replacing the characters beyond the first $2n$, by an infinite string of '0'. But you could use any arbitrary string instead of zeros, without impacting the first $n-3$ characters in subsequent iterations. The following is a direct consequence of the definition of auto-convolution and its inverse (square root operator).

Theorem 1.2.2 *Let k be a fixed integer and x_∞ be the limit number representing S_n^n as $n \to \infty$, assuming that the sequence (S_n^n) converges. Then (S_n^{n+k}) converges to $(x_\infty)^{2^k}$ as $n \to \infty$. Here, k can be positive or negative. This is true with or without truncation, and regardless of the initial configurations S_n^0 for $n = 1, 2$ and so on. Extra care is needed with $k < 0$ as fractional powers of a string are not uniquely defined.*

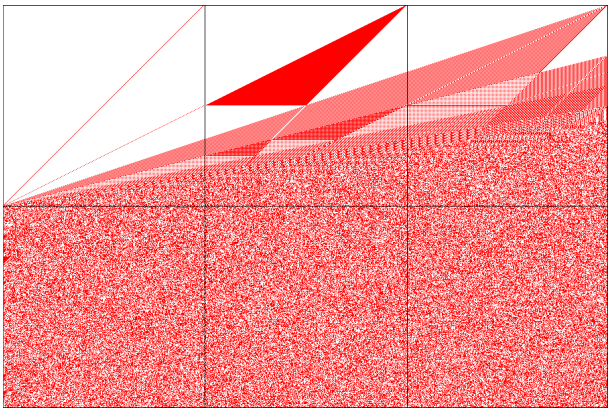


Figure 1.1: First $3n$ characters of S_n^k, for $n = 300$ and $1 \leq k \leq 2n$ (one line per k)

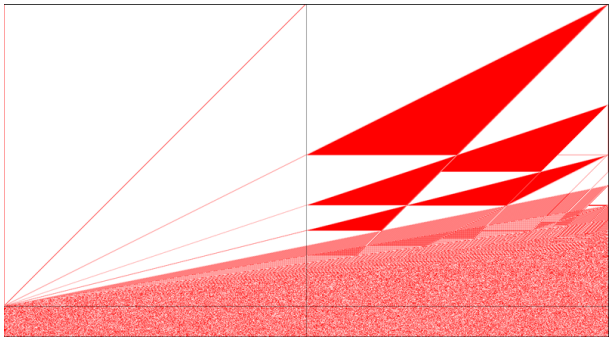


Figure 1.2: Same as top left part of Figure 1.1, but with a different seed $S_n^0 \equiv 2^n + 3$

1.2.1 Visualizing iterated auto-convoluted string sequences

Figure 1.1 shows S_n^k truncated to $3n$ characters, with $n = 300$. Each line in the picture corresponds to a specific value of k, starting with $k = 1$ at the top, and ending with $k = 2n$ at the bottom. Thus the size of the image is 900×600 pixels. A red pixel represents '1', and a white one represents '0'. The initial string S_n^0, also called the seed, consists of $n+1$ characters all zeros except the first and last ones. This explains why there is so much white on the top half, especially on the left side.

As k increases, more and more '1' start to pop up from the right side, and propagate to the left. Patterns become more and more complex until $k = n$. Beyond $k = n$, we are in full chaotic mode. We do not care about the bottom half of the picture (the chaotic zone). Likewise, we do not care about the rightmost n pixels. They could be anything, yet they have no impact on the final conclusion. Indeed, we only need the first $2n$ pixels on the left, and we only care about the top left block ($n \times n$ pixels). However the color of these pixels is impacted by those in the middle block.

The framework is identical to the dynamical systems described in my book about chaos [15]. If we keep $2n$ digits at all times, the precision decreases by about one digit per iteration when incrementing k, thus we can expect to still have n correct digits after n iterations. Beyond that point, the precision continues to deteriorate, and when $k = 2n$, we have lost all precision. The process still continues as k increases, but it is like starting with a new seed every n iterations or so.

Finally, Figure 1.3 shows a more granular view, zooming in on a portion of the top left block in Figure 1.3 to further reveal the patterns. Here I kept only the $2n$ first leftmost digits when truncating (rather than $3n$ in Figure 1.3), yet the pixels are identical.

As a side note, it would be interesting to check if the sequence (S_n^k) indexed by k (with n fixed) is dense, ergodic, and find its invariant measure in the string distribution space. For each k, you first need to choose a unique number to represent S_n^k, (say) the one in $[2, 4[$, and then check if the corresponding sequence of numbers is dense in $[2, 4[$. Here truncation is not allowed, otherwise the answer to the first question is obviously negative.

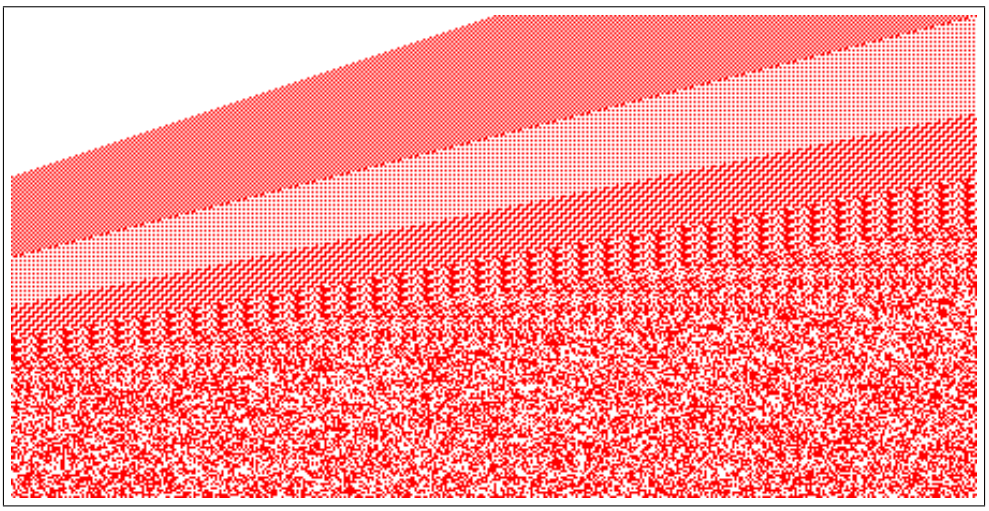


Figure 1.3: Zoom in on bottom right corner of top left block in Figure 1.1

1.2.2 Fundamental result about the number of zeros and ones

I briefly discussed chaotic dynamical systems in section 1.2.1, mentioning that for the string sequence (S_n^k) indexed by k with fixed n, chaos starts at $k = n$. One of the novelties is that we are dealing with strings rather than real numbers or vectors. The choice of the seed S_n^0, in this case the most rudimentary string with $n + 1$ characters, is entirely responsible for the observed behavior. While patterns are obvious in Figure 1.1 and especially Figure 1.3, in this section they are amplified to a whole new level, with deeper connections to dynamical systems. This is possible thanks to working with one of the simplest iterated auto-convolution systems for strings, with truncation that luckily preserves the interesting properties throughout the first n iterations ($k \leq n$, but not thereafter). It leads to spectacular discoveries. Before starting, let me introduce the following constants, linked to the abscissas of the dashed vertical lines in Figure 1.4:

$$\rho_1 = \frac{1}{2},\ \rho_2 = \frac{2}{3},\ \rho_3 = \frac{3}{4},\ \rho_4 = \frac{4}{5},\ \rho_5 = \frac{5}{6},\ \ldots \tag{1.2}$$

In Figure 1.4, if you look at the red dots, the proportion of '1' in the first n characters of the string S_n^k (with n fixed), using the usual truncation mechanism, follows this path:

- It starts at a very low close to 0% (after all, the seed only has two '1'),
- then abruptly starts growing steadily at about $k = \rho_2 n$.
- Around $k = \rho_3 n$, the blue and red curves separate, but the slope stays the same on the red curve until $k = \rho_4 n$. At this point, about 20% of the characters are '1' on the red trajectory.
- Suddenly at $k = \rho_4 n$ until $k = \rho_5 n$, the slope is steeper, bringing the proportion of '1' to about 30% at $k = \rho_5 n$.
- From there, two branches open up, with the proportion of '1' still increasing.
- At $k = \rho_6 n$, more bifurcations appear. Eventually the growth tapers off, more and more branches open up faster and faster,
- and when $k = \rho_\infty n = n$, we enter the fully chaotic zone with the proportion of '1' oscillating around 50% moving forward.

At the same time (not shown in the picture), the proportion of '0' moves in the opposite direction (flipped side), starting at almost 100% to eventually drop and likewise, oscillate around 50% when reaching the chaotic zone.

While not visible on the picture, S_n^k has exactly two '1' between $k = 0$ (the seed) and $k = \rho_1 n$, and exactly three '1' between $k = \rho_1 n$ and $k = \rho_2 n$ making the proportion of '1' virtually zero when n is large and $k < \rho_2 n$. The red dots correspond to even values of k, and the blue dots to odd values. All of this is classic in dynamical systems that start gentle and evolve to chaos. I summarize it in the following theorem.

Theorem 1.2.3 *The string sequence (S_n^k) indexed by k, with n fixed, constitutes a dynamical system in non-chaotic phase when $k < n$. In this phase, the number of ones (or zeros) follows a regime with changing points governed by schedule (2.4). In particular:*

- *Prior to branching ($k < \rho_5 n$ if k is even) the proportion of ones steadily increases as k increases, to reach above 29.1% (approximately), and never drops below that threshold thereafter when $k > \rho_5 n$.*

- *The 29.1% threshold in question is absolute. It holds for any n large enough, including $n = \infty$.*

Determining a precise value for the 29.1% threshold may prove difficult. A proof targeting 20% or 25% is likely to be easier to reach. Table 1.1 displays values of ν_3, ν_4 and ν_5 with n ranging from 1000 to 8000. These values represent the proportion of '1' in the first n digits of S_n^k when k is the closest even integer respectively to $\frac{3}{4}n$, $\frac{4}{5}n$ and $\frac{5}{6}n$. That is, on the red orbit in Figure 1.4, when it crosses respectively the second, third, and fourth vertical dashed lines from the left. The takeaway is that these values are absolute as stated in theorem 1.2.3, not really depending on n. In particular, the 29.1% mentioned earlier corresponds to $\nu_5(n)$.

n	1000	2000	4000	8000
$\nu_3(n)$	0.127	0.126	0.125	0.125
$\nu_4(n)$	0.202	0.201	0.200	0.200
$\nu_5(n)$	0.294	0.291	0.292	0.291

Table 1.1: Sample values of ν_3, ν_4, ν_5

Bumps or dips (if there are any) on the smooth sections of the blue and red paths in Figure 1.4 are of very small amplitude and duration (invisible), essentially having no impact. Also, the only numbers that matter are those attached to the red path. Ignoring the blue path has no impact on the final conclusion, about the guaranteed minimum proportion of '1' when $k = n$: if n is odd, then at the next sequence $n + 1$ is even and on a red path, with S_n^n and S_{n+1}^{n+1} having the same number of '1' in the first n digits.

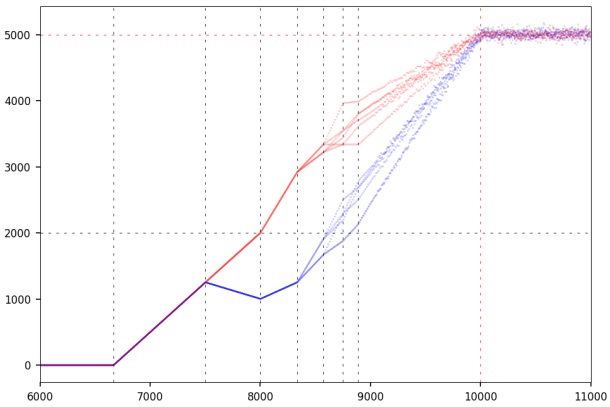


Figure 1.4: Number of '1' in first n chars of S_n^k, with $n = 10^4$ and k on the X-axis

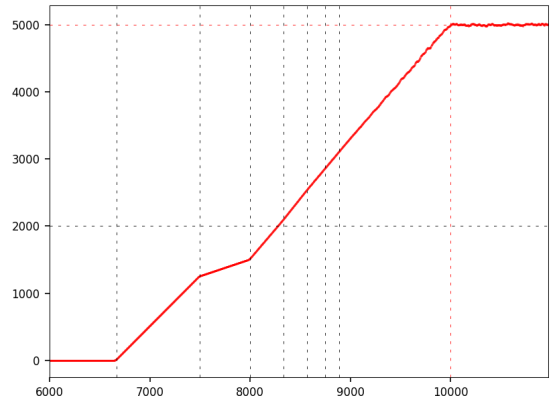


Figure 1.5: Averaging the paths within each band in Figure 1.4

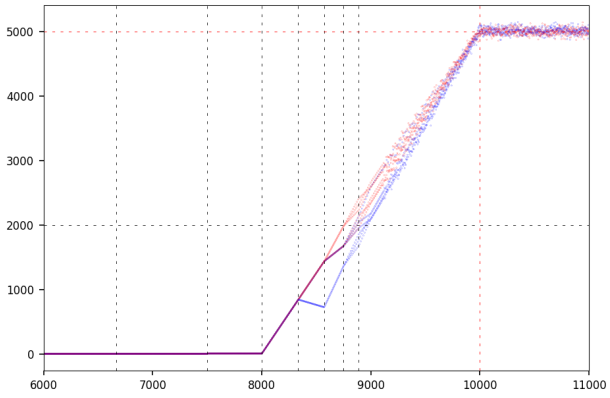


Figure 1.6: Same as Figure 1.4, but with a different seed $S_n^0 \equiv 2^n + 3$

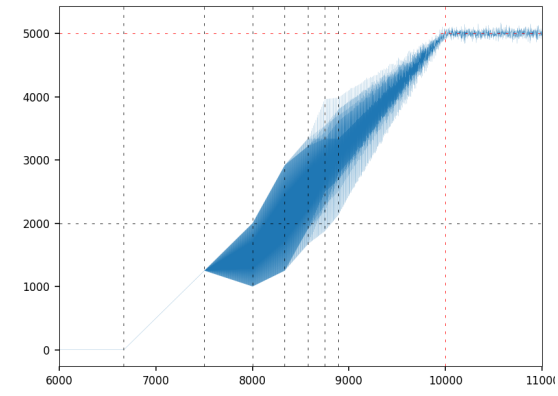


Figure 1.7: Envelope showing the range given k, for the number of '1' in Figure 1.4

1.2.2.1 Mechanics of the bifurcation process

So far I mentioned two paths: the red, and the blue one. They are actually multi-paths, since more and more bifurcations show up as we move to the right in Figure 1.4. Let's focus on the red multi-path (similar arguments

apply to the blue multi-path). At $k = \rho_5 n$, we have a new split, with 2 options: one path steeper than the other one. The steeper one corresponds to $k \equiv 0 \bmod 4$, and the other one to $k \equiv 2 \bmod 4$.

Let us follow the steeper path. The next split, taking place at $k = \rho_6 n$, now offers 3 options: steep climb, moderate climb, or nearly flat (until the next split at $k = \rho_7 n$). Which path to follow depends this time on the value of k modulo 12, with $k \equiv 4 \bmod 12$ for the steeper climb. At the end of that climb, the proportion of '1' jumped from 29.1% to above 33.4%. Moving further to the right, subsequent splits offer more bifurfactions, and the full combinatorial nature of the problem becomes evident (and also discussed in section 1.3.2).

To address the most challenging aspects of the problem in order to get the strongest possible conclusion, one also has to look at paths going down, and how many segments are needed to recover from a decline. Eventually, the width of the chaotic band beyond $k = n$ shrinks and becomes a flat line when $n \to \infty$ (if the plot size is kept constant), because that width is proportional to $\sqrt{n}$ and $\sqrt{n}/n \to 0$. But proving this fact still seems completely out of reach at this point. If proved, it would show that the proportion of '1' at $k = n$ tends to 50%, a conjecture that all mathematicians believe in. Looking at the link between infinite iterated self-convolutions and the limiting Gaussian distribution, could shed more light, with the nearly-Gaussians converging to a singular distribution centered at 50% as $n = \infty$.

1.2.2.2 Interesting variable-length moving averages

The goal here is to get an idea of the general trend in Figure 1.4. I focus on counting the '1's, but it also applies in a mirrored way to the character '0'.

Each path within a band delimited by two consecutive vertical dashed lines in Figure 1.4, is linear (perfectly linear when $n = \infty$). Also, the number of paths in each band, starting on the left and moving to the right, are 1, 1, 2, 2, 4, 12, and so on. Except for the first two values, these are factorials multiplied by 2. Then, based on (2.4), the widths of each band, from left to right, are proportional to

$$\frac{1}{1 \cdot 2}, \quad \frac{1}{2 \cdot 3}, \quad \frac{1}{3 \cdot 4}, \quad \frac{1}{4 \cdot 5}, \quad \cdots$$

The proportionality factor is n. If you average all the paths within each band, and concatenate the resulting segments across all the bands, you get a variable-length moving average that shows how the number of '1's is trending up as k increases. The result is pictured in Figure 1.5. Slight departure from a perfect straight line on the right is due to me using some approximations to produce this plot.

Figure 1.7 was produced with the `plot` command, thus connecting the dots in Figure 1.4 to produce a quantum-like time series similar to Figure 1.2 (bottom part) in [15]. By contrast, Figure 1.4 is a scatterplot. Figure 1.7 has the advantage of showing the bounds for the number of '1' at iteration k, with k between 6000 and 11,000 on the horizontal axis. As discussed earlier, Figure 1.7 shows the average, and $n = 10{,}000$ is fixed. I used the same data in all cases.

1.2.2.3 The inverse system

You might ask the following question. What if everything goes well until S_n^{n-1} but suddenly at S_n^n you lose everything: the well balanced string obtained in the previous iteration suddenly becomes imbalanced in just one step. This is not a theoretical question, it happens in practice. For instance if $S_n^{n-1} \equiv \sqrt{2}$ then $S_n^n \equiv 1$, the most imbalanced of all strings. The question then is how can you end up in a situation like that and how to make sure it is not the case with the seeds that we use?

The situation could be much more subtle than the extreme case just described. To reconstruct the seed of each sequence (S_n^k) with n fixed, assume that $S_n^n \equiv x_n$ and $x_n \to x$. Then $S_1^0 \equiv x^{1/2}$, $S_2^0 \equiv x^{1/4}$, $S_3^0 \equiv x^{1/8}$ and so on results in $S_n^n \to x$. Keep in mind that the square root of a non-zero string always have two values, so there are many solutions. Also, if S_n^n is imbalanced as in our extreme case, S_n^{n-1} may not be, and it converges to $\sqrt{x}$. Note that if $S_n^{n-1} \equiv \sqrt{2}$ and thus $S_n^n \equiv 1$, the seeds are well balanced. In our case, the situation is opposite, with extremely imbalanced seeds consisting only of zeros with a '1' at each end.

Now, it is interesting to make an analogy with the logistic map, another quadratic system, governed by the recursion $s_k = 4s_{k-1}(1 - s_{k-1})$ with a seed s_0 in $[0, 1[$. Our system consists of infinitely many recursions (one for each n), with the n-th recursion being $S_n^k = S_n^{k-1} * S_n^{k-1}$ with each iterate S_n^k, when represented as a real number, constrained to lie (say) in $[2, 4[$. The logistic map also has a binary numeration system attached to it, see section 3.1.1 in [15]. The seed $s_0 = 1/3$, while extremely imbalanced, leads to the well balanced number $(2\pi)^{-1} \arcsin(\sqrt{1/3})$, known to be irrational. The logistic map is homeomorphic to the dyadic map that produces the standard binary digits, and thus, homeomorphic to each of our dynamical systems (one system for each n).

1.2.2.4 Invariant measure

An invariant measure of an ergodic dynamical system, also called equilibrium or attractor distribution, is a probability distribution defined as follows. All dynamical systems are governed by a mapping h that specifies the recursion. In our case, $S_n^{k+1} = h(S_n^k) = S_n^k * S_n^k$. When the string S_n^k is represented by a number lying in $[1, 2[$, the function h is defined as

$$h(x) = \begin{cases} x^2 & \text{if } 1 \le x < \sqrt{2}, \\ \frac{1}{2}x^2 & \text{if } \sqrt{2} \le x < 2. \end{cases}$$

Note that h is the same for all n since all our dynamical systems are identical except for the seed S_n^0. The invariant measure is the distribution F that satisfies $F(x) = F(h(x))$. Finding F involves solving a functional equation, but in our case, $F(x) = \log_2 x$ with $x \in [1, 2[$. This is known as the reciprocal distribution. For the logistic map, the invariant measure is a beta distribution, and for the dyadic map, a uniform distribution.

The standard approach to solve our problem is to use the target number as the seed, show that it is in the attraction domain of the main invariant measure F, and thus conclude that the number in question (the seed) is normal. No one has ever succeeded to prove anything with this technique. To the contrary, our approach uses trivial seeds apparently unrelated to the target number, exhibiting no sign of being in the attraction domain until after k iterations with $k > n$. We don't care if it is or not in the attraction domain, we only care about the behavior when $k \le n$, for each dynamical system (S_n^k) with n fixed. By looking at all n, we make the connection to the target number and have compelling arguments about its digit distribution.

1.3 Solving one of the greatest mathematical mysteries

For any positive integer n, the seed string S_n^0 is equivalent to the number $2^n + 1$. Thus, the string S_n^k obtained via k successive auto-convolutions is equivalent to $2^n + 1$ at power 2^k. Since we work with classes (similar to modulo classes), the magnitude does not matter: we can choose any equivalent number by multiplying by a factor 2^m (with m a positive or negative integer). In particular, we can choose m so that the resulting number always lies in $[2, 4[$, and is thus uniquely defined, without changing the characters in the string S_n^k. You achieve this goal with $m = -\lfloor 2^k \log_2(2^n + 1) \rfloor + 1$, which is equal to $-n \cdot 2^n$ when $k = n$. Here $\lfloor \cdot \rfloor$ stands for the integer part function. From there, we obtain

$$S_n^n \equiv \left(1 + \frac{1}{2^n}\right)^{2^n} \to e = 2.718281828\ldots \quad \text{as } n \to \infty, \tag{1.3}$$

where e is Euler's number. Simple computations using logarithms and Taylor series expansions show that the approximation yields about n correct binary digits, in accordance with earlier claims made in this paper. It is also straightforward to verify that $S_n^{n-1}, S_n^{n+1}, S_n^{n+2}, S_n^{n+3}$ tend respectively to $\sqrt{e}, e^2, e^4$ and e^8 as $n \to \infty$, in accordance with theorem 1.2.2, using the definition of convergence in section 1.1.2.

But what would happen with a different seed S_n^0? With $S_n^0 \equiv 2^n - 1$ and $S_n^0 \equiv 2^n + 3$, the limit constants are respectively e^{-1} and e^3, instead of e. However, there is no guarantee that the observed patterns are identical: it needs to be verified. Then, using $S_n^0 \equiv 2^n + 2 = 2 \cdot (2^{n-1} + 1)$ does not bring anything new as we are staying within the same original string class. Now, I can state the main result – the very first *deep* result about the digit distribution associated to popular math constants.

Theorem 1.3.1 *For n large enough, the proportion p_n of ones in the first n binary digits of $e = 2.7182\ldots$ satisfies $p_n > \mu$, and the proportion q_n of zeros satisfies $q_n < 1 - \mu$, where $\mu > 0$ does not depend on n even as $n \to \infty$.*

The last claim "even as $n \to \infty$" is still an open question. Also, theorem 1.3.1 is stated in a very weak form given all the advances shared in this paper. I present it as a starting point rather than the best that we have obtained so far. Yet, if proved this weak version would a phenomenal result in itself. In particular, the statement that μ stays strictly positive even when n is infinite is the strongest result ever obtained for any major mathematical constant such as $\pi, e, \log 2$ or $\sqrt{2}$, by a long shot. For instance, it has been established that the proportion of ones in the first n binary digits of $\sqrt{2}$ is larger than $\sqrt{2n}$, see [47]. However, since $\sqrt{2n}/n \to 0$ as $n \to \infty$, it leads to $\mu = 0$. Also the proof is rather simple and cannot be improved to get a stronger result.

About theorem 1.3.1, "n large enough" may be interpreted as $n > 50$. Also, a low hanging fruit is $\mu = 10\%$, while improving to $\mu = 20\%$ (more challenging), and even $\mu = 30\%$ (significantly more challenging), appears to be within reach based on results in section 1.2.2. In the end, theorem 1.3.1 is a weaker version of theorem 1.2.3, applied to Euler's number instead of the original formulation that deals with auto-convoluted string sequences. Formula (1.3) is the connection between both.

For additional references, see my book on chaos and dynamical systems, discussing a stronger concept of normality in various numeration systems [15]. Andrew Granville and Davig Bailey [5] are also good references on

this topic. The Wolfram entry for the digit count function (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. It is also known as the Hamming weight, with a very fast algorithm described here and a full chapter in [49]. It is particularly relevant to our problem. Regarding recent publications on normal numbers, see Verónica Becher [6] and [2]. For a discussion about the carry digit function (a 2-cocycle) that propagates 1's from right to left in the successive iterations, see [1, 8]. Another application of the digit count function – also called sum-of-digits function, for binary sequences – is featured in [32] in the context of genotype maps, with processes not unlike the dynamical systems discussed in this chapter, and blancmange curves almost identical to Figure 3.3 in my book on numeration systems [15]. The potential connection to quantum maps and quantum cryptography [11, 46] is worth investigating.

Also look for references on infinite iterated self-convolutions of a function, with re-scaling, and convergence to a Gaussian distribution. In our case, the function in question is equivalent to a mapping from $[2, 4[$ onto itself. The implicit non-continuous re-scaling is done at each iteration to keep the iterates in that interval. Related material include discrete dynamical systems, the logistic map in particular, with bifurcations at $\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}$ and so on. In short, the central argument to the proof of theorem 1.3.1 and its stronger version, say with $\mu = 29.1\%$, is to analyze an infinite number of infinite iterated sequences of truncated auto-convoluted binary string classes similar to p-adic numbers, look at the converging diagonal entries in the associated topological space, and use re-scaling. The guaranteed minimum of at least 29.1% of ones in the binary digits of Euler's number, comes as a special constant in a specific dynamical system in its non-chaotic phase just before chaos arises, starting with the simplest seed string of length $n+1$ in each sequence (S_n^k) indexed by k, with fixed n, and then let $n \to \infty$.

1.3.1 Testing different seeds

Here I show the results when using the seed string $S_n^0 \equiv 2^n + 3$ in the n-th dynamical system. Assuming $n \geq 3$, this string consists of $n+1$ characters, all zero except a one at the beginning (left side), and two ones at the end (right side). Now the first $n - \tau$ digits of S_n^n match those of e^3, with $\tau = 7$ in the tests. The maximum value of τ needs to be determined, but it is a small, fixed integer, yet larger than when $S_n^0 \equiv 2^n + 1$. The main differences and similarities, compared to the seed $S_n^0 \equiv 2^n + 1$, are:

- When comparing Figure 1.6 with Figure 1.4: a steeper climb starting later on the right, with the new seed. Of course, the proportion of '1' also starts oscillating around 50% as soon as $k \geq n$, with k on the horizontal axis. For both seeds, I used $n = 10^4$.
- The bifurcation patterns are similar. Quite strikingly, the change points abscissa, that is, the values in Formula (2.4), are identical for both seeds.
- With the new seed, the proportion of '1' evolves in a much narrower range even though the patterns in Figure 1.2 are more complex than those in Figure 1.1. Thus, the new seed might be easier to deal with, when trying to establish formal mathematical proofs.
- The values in Table 1.1 are now different, but the invariance property is preserved: they don't really depend on n (they are asymptotically stable as n increases), although they depend on the seed.

Very few seeds generate the behavior observed in Figures 1.4 and 1.6. With the seed $S_n^0 \equiv 2^{q_n/p_n}$ where p_n, q_n are coprime integers, the sequence (S_n^k) indexed by k (with n fixed) eventually becomes periodic. Thus, this is also true for the proportion of '1' as a function of k. In particular, if $p_n \neq 2$ is a prime, then the length of the period is equal to the multiplicative order of 2 modulo p_n if no truncation is used. In this case, the quantum behavior visible in Figure 1.7 is absent; we are dealing with classic time series.

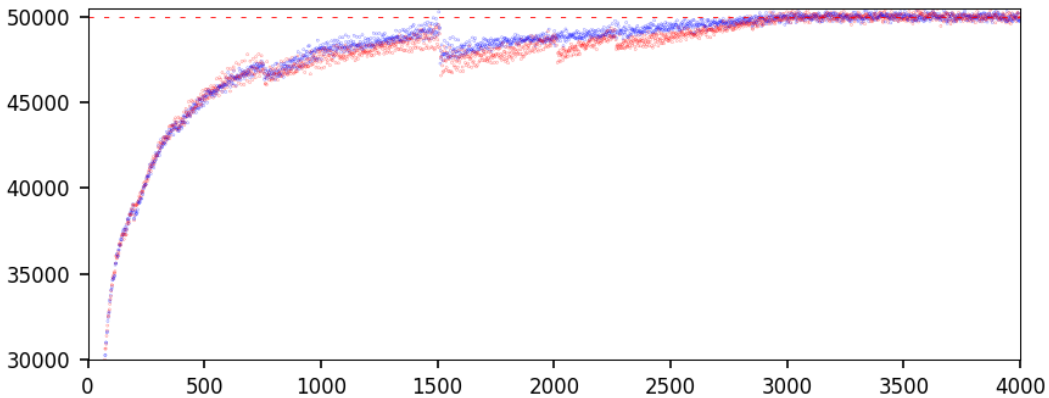


Figure 1.8: Number of '1' (Y-axis) vs k (X-axis) with unbalanced random seed and $n = 10^5$

If the seed if well balanced, consisting of random '1' and '0' in equal proportions, and is not in the previous category, then the behavior is usually identical to the fully chaotic phase in Figure 1.4, that is, the rightmost portion with $k \geq n$ on the horizontal axis. In this case, chaos starts right at $k = 0$ rather than $k = n$: there is no ramp-up phase.

Figure 1.8 features a seed with $n+1$ characters all zeros except five '1' at random locations, with $n = 10^5$. Both axes are truncated to focus on the ramp-up process at the very beginning, moving from five to about 50,000 '1' (the equilibrium with 50% of '0' and '1') in less than 3000 iterations. As in Figure 1.4, the red and blue dots correspond respectively to even and odd values of k on the X-axis. Note the drops in the curve, reminiscent of the bifurcation change points. Full chaos starts at $k \approx 0.03 \times n$, compared to $k = n$ in Figure 1.4. The Python code is listed below and here on GitHub. It is a simplified and faster version of the one listed in section 1.3.5, for random seeds.

```
import random

n = 100000
L = 2*n
H = int(0.04*n)

sprod = '1'
random.seed(334)

for k in range(n+1):
   u = random.uniform(0, 1)
   if u < 0.00004:
      sprod += '1'
   else:
      sprod += '0'

scnt1 = sprod.count('1') # number of 1 in seed
prod = int(sprod, 2) # seed

arr_count1 = []
arr_count0 = []
arr_count1.append(scnt1)
arr_count0.append(n+1-scnt1)

#--- 1. Main

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

   prod = prod*prod
   pstri = bin(prod)
   stri = pstri[0: L+2]
   prod = int(stri, 2)
   lsr = len(stri)
   stri = stri[2: len(stri)]
   estri = stri[0:n] # leftmost n digits
   ecnt0 = estri.count('0')
   ecnt1 = estri.count('1')
   arr_count1.append(ecnt1)
   arr_count0.append(ecnt0)
   print("%3d %3d %3d" %(k, ecnt0, ecnt1))

#--- 2. Create the plots

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

xvalues = np.arange(1, H+2, 1)
arr_color = []

for k in range(H+1):
   if k%2 == 0:
      arr_color.append((1, 0, 0)) # red
   else:
      arr_color.append((0, 0, 1)) # blue

plt.scatter(xvalues[0:H],arr_count1[0:H],s=0.01, c=arr_color[0:H])
```

```
plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.4,dashes=(5,10))
plt.xlim([0, H])
plt.ylim([0.3*n, 0.505*n])
plt.show()

print("\nNumber of 1 in seed:", scnt1)
```

1.3.2 Another interesting sequence

In order to obtain interesting results with my framework, you need to have string convergence. Few mathematical constants fit the bill. It works for e (possibly the simplest case) because e can very easily be approximated by a sequence A_n/B_n where A_n is an integer, and B_n of power of 2. In this case, assuming x is an integer,

$$A_n = (2^n + x)^{2^n},\ B_n = 2^{n \cdot 2^n} \Rightarrow \frac{A_n}{B_n} \rightarrow \exp(x) \text{ as } n \rightarrow \infty.$$

One might think that since A_n can be expanded using the binomial theorem, a simpler case consists of working with just one coefficient – the central one – in the binomial expansion. While there is a way to do it, the solution is probably far more complicated and only works with $x = 1$. The starting point could be this:

$$A_n = n \cdot \binom{2n}{n}^2,\ B_n = 2^{4n} \Rightarrow \frac{A_n}{B_n} \rightarrow \frac{1}{\pi} \text{ as } n \rightarrow \infty.$$

The savvy reader is invited to play with it! If the denominator B_n is a power of b rather than 2, you may get interesting results in base b rather than in the binary numeration system, especially if b is prime or a power of a prime. In short, my scheme allows you to entirely ignore the denominator and focus just on the digits of A_n.

I use the central binomial coefficients, closely related to Catalan numbers [Wiki], in Formula (4.9), leading to quadratic irrationals rather then $1/\pi$. See also Formula (2.5) obtained using AI.

1.3.3 Surprising, unpublished results about the digit distribution

The length of the longest run of '1' starting at position n in the binary digit expansion of $\sqrt{2}$ cannot be larger than n. This generalizes to other quadratic irrationals. The proof is simple but not amenable to any stronger result. It is conjectured that the length of the longest run starting at position n is at most $\log_2 n$, or $\log_b n$ in base b. This would be true if the digits were randomly distributed in equal proportions.

A much stronger result is the following. If two irrational numbers are linearly independent over the set of rationals, then their binary digit sequences are uncorrelated. In particular, the binary digits of $\sqrt{2}$ and $\sqrt{3}$ are uncorrelated. This is due to the fact that the correlation between the digit sequences of $p\alpha$ and $q\alpha$ is $1/(pq)$ if p, q are odd coprime integers and α is irrational. See statement and proof, here. This result is important to build strong PRNGs that rely on a large number of irrationals, see [15]. It guarantees the absence of cross-correlations. Note that the correlation between two infinite sequences of digits is defined as the limit when $n \rightarrow \infty$ of the empirical correlation computed on the first n digits, if it exists. It is discussed in great details with several examples, in [15].

Finally, in base 2, the rational $1/3^n$ has a period P_n of length $2 \cdot 3^{n-1}$ starting after the trailing zeros on the left. The period has the same number of '0' and '1' for $n = 1, 2$ and so on. For any number z in $[1, 2[$, say $z = \sqrt{2}$, it is always possible to find a sub-sequence (n_k) and a strictly increasing integer-valued function g such the first $g(k)$ binary digits of P_{n_k} match those of z. While P_{n_k} has the same proportion of '0' and '1', this is not the case if you look at the first $g(k)$ digits, especially since $g(k)$ is much smaller than the period. Thus, a lot more hard work is needed to make any conclusion regarding the digits of z. Note that if you multiply $1/3^k$ by an integer power of 2 so that it stays in $[1, 2[$ at all times, then the median rescaled value computed over $k = 1, 2, \cdots, n$ converges to $\sqrt{2}$ as $n \rightarrow \infty$. See details here. It is linked to the dyadic map starting with $\log_2 3$, with the k-th iterate being the fractional part of $2^k \log_2 3$.

There is an abundant literature about the dyadic odometer map [Wiki], another word for the dyadic map. Same with the Bernoulli convolution related to our framework. See also the digit sum entry in Wolfram, here. The theory about the generating function of the digit sum [1] is of special interest. I haven't explored all these references yet, and how to leverage them.

1.3.4 Application to cryptography

If you look at Figure 1.1, the bottom half (where the chaos truly starts) seems to produce decent random bits. However, if you use this system to generate pseudo-random numbers, you face the same issues as with congruential random generators. First, the truncation defined in section 1.1.2 is a type of modulo operator, that is,

generating congruence classes of some sort. Then, while the period may be extremely large, these systems have security vulnerabilities if n is not large enough. In addition, the way the bits are computed in my system is not very efficient, as the goal is to solve theoretical problems rather than designing fast apps. Yet, the Python code in section 1.3.5 uses a library to speed up some computations by several orders of magnitude, and I use it to double check my results with external algorithms. You might want to have a look at it.

That said, I designed and tested a few fast, secure and strong random number generators based on irrational numbers. For details, see chapter 13 in [16], entitled "Strong Random Generators for Reproducible AI", as well as chapter 4 in [15], entitled "Random Numbers Based on Quadratic Irrationals". Finally, if you use an arbitrary unbalanced seed instead of $S_n^0 \equiv 2^n + 1$, after a very short ramp-up phase, you obtain well balanced random strings of length $n+1$ after as few as $k = 0.03 \times n$ iterations instead of n iterations for $S_n^0 \equiv 2^n + 1$, as illustrated in Figure 1.8 with $n = 10^5$. Computations may be accelerated using ad-hoc algorithms, or with more random seeds.

1.3.5 Python code

The Python code `number_theory.py` is also on GitHub, here. To analyze the growing complexity of the patterns as n increases, I highly recommend to use AI. These patterns are governed by universal dynamical system thresholds, such as 29.1% mentioned earlier. These thresholds are estimated in section [3] in the code. In section [2], I use the Mpmath library to compute the digits of e and to double check that my algorithm yields at least $n - 3$ correct digits at each n.

```
from PIL import Image, ImageDraw

n = 1000
L = 2*n
H = int(1.1*n)

height,width = (H+1, L+1)
img1 = Image.new( mode = "RGBA", size = (L+3, H+2), color = (0, 0, 0) )
pix1 = img1.load()
draw1 = ImageDraw.Draw(img1,"RGBA")

prod = 2**n + 1
offset_H = 0
arr_count1 = []
arr_count0 = []
arr_count1.append(2)
arr_count0.append(n-2)


#--- [1] --- Main

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

   prod = prod*prod # this is (2^n + 1) at power 2^k (truncated)
   pstri = bin(prod)
   stri = pstri[0: L+2]
   prod = int(stri, 2)
   lsr = len(stri)

   if k == n+1:
      offset_H = 1
      for l in range(L+1):
         pix1[l, k-1] = (0, 0, 0)

   stri = stri[2: len(stri)]
   if k == n:
      e_approx = stri
   rstri = stri[n:2*n] # rightmost n digits in first 2n digits
   rcnt0 = rstri.count('0')
   rcnt1 = rstri.count('1')
   estri = stri[0:n] # leftmost n digits
   ecnt0 = estri.count('0')
   ecnt1 = estri.count('1')
   bnum = 0

   for l in range(n):
      bnum += int(estri[l]) / 2**(l-1)
   offset_L = 0
   for l in range(min(L, len(stri))):
      if l == n+1:
```

```
            pix1[l, k-1] = (0, 0, 0)
            offset_L = 1
        elif l == 2*n+2:
            pix1[l+1, k-1] = (0, 0, 0)
            offset_L = 2
        if stri[l] == '1':
            pix1[l+offset_L, k-1+offset_H] = (255, 0, 0)
        elif stri[l] == '0':
            pix1[l+offset_L, k-1+offset_H] = (255, 255, 255)

    arr_count1.append(ecnt1)
    arr_count0.append(ecnt0)

    print("%3d %3d %3d %3d %3d %f" %(k, ecnt0, ecnt1, rcnt0, rcnt1, bnum))

img1.save("img_1.png")
img2 = img1.crop((n-n/2, n-n/4, n, n)) # left, top, right, bottom
img2.save("img_2.png")


#--- [2] --- Fast computation of binary digits of e

from mpmath import mp
import numpy as np

# Set precision for n binary digits
mp.dps = int(n*np.log2(10))
e_value = mp.e # Get e in decimal

# Convert to binary
e_binary = bin(int(e_value * (2 ** n)))[2:]

k = 0
print()
while e_approx[k] == e_binary[k]:
    k += 1
print("%d correct digits (n = %d)" %(k, n))


#--- [3] --- Compute nu values in Table 1

arr_rho = []
print()
for j in range(1,6):
    threshold = int(n*j/(j+1))
    if threshold % 2 == 1:
        threshold += 1
    p1_even = arr_count1[threshold]/n
    print("Rho %1d: %7.5f" %(j, p1_even))


#--- [4] --- Create the plots

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

xvalues = np.arange(1, H+2, 1)
arr_color = []
for k in range(H+1):
    if k%2 == 0:
        arr_color.append((1,0, 0))
    else:
        arr_color.append((0, 0, 1))

xmin = int(0.6*n)
xmax = int(1.1*n)

plt.scatter(xvalues[xmin:xmax], arr_count1[xmin:xmax], s = 0.01, c = arr_color[xmin:xmax])
plt.axvline(x=2*n/3, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
plt.axvline(x=3*n/4, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
plt.axvline(x=4*n/5, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
plt.axvline(x=5*n/6, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
```

```
127 plt.axvline(x=6*n/7, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
128 plt.axvline(x=7*n/8, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
129 plt.axvline(x=8*n/9, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
130 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.4, dashes=(5, 10))
131 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.4, dashes=(5, 10))
132 plt.axhline(y=n/2, color='red', linestyle='--', linewidth = 0.4, dashes=(5, 10))
133 plt.xlim([xmin, xmax])
134 plt.show()
```

Chapter 2

Petabyte Challenge to Help Crack the Famous Math Conjecture

In Chapter 1, I paved the way to proving a famous multi-century old conjecture: are the digits of major mathematical constant such as $\pi, e, \log 2, \sqrt{2}$ evenly distributed? No one before ever managed to prove even the most basic trivialities, such as whether the proportion of '0' or '1' exists in the binary expansions of any of these constants, or if it oscillates indefinitely between 0% and 100%. Here I provide an overview of the new framework built to uncover deep results about the digit distribution of Euler's number e, discuss the latest developments, share an upgraded version of the code, and feature new potential research areas across multiple fields, arising from my discovery.

2.1 Introduction

My first article on the topic, published in February 2025, is posted here as paper 51. The methodology relies on material discussed in my book "Gentle Introduction to Chaotic Dynamical Systems" [15]. The original article is now part of that book. I used bit strings to represent binary digit sequences, establishing the link to large language models (LLMs). Here, I will stick to integers instead, to simplify the presentation. The main idea is to work with iterated self-convolutions. For fixed integers n and x, iteratively define the sequence $\{S(n,k,x)\}$ with the recursion

$$S(n,k+1,x) = S^2(n,k,x), \quad k = 0,1,2\ldots \tag{2.1}$$

with $S(n,0,x) = 2^n + x$. The initial value $S(n,0,x)$ is called the seed. We are interested in $x = \pm 1$ or $x = 3$. It follows that the first $\tau(n)$ binary digits of $S(n,n,x)$ match those of $\exp(x)$, with $\tau(n) = n + O(1)$. This remains true if at each iteration k, we truncate $S(n,k,x)$ and only keep the first $2n$ binary digits on the left, so that $S(n,k,x)$ is an integer with $2n$ digits at all times.

A more traditional approach consists in multiplying the seed $S(n,0,x)$ by an integer power of 2, positive or negative, so that it lies in the interval $[1,2[$. Then, replace Formula (2.1) by the recursion

$$S(n,k+1,x) = \lambda \cdot S^2(n,k,x) \tag{2.2}$$

with $\lambda = 1$ if $S(n,k,x) < \sqrt{2}$, otherwise $\lambda = \frac{1}{2}$.

Then, the sequence $\{S(n,k,x)\}$ indexed by k, with fixed n and x, is an ergodic quadratic dynamical systems. It is homeomorphic (also called conjugate) both to the logistic map and the dyadic map, with $S(n,k,x) \in [1,2[$ at all times. In short, it is a mapping from $[1,2[$ onto itself. Its invariant measure is the reciprocal distribution, a probability distribution defined as $F(z) = \log_2 z$ for $1 \leq z < 2$. The theory, including the connection to normal numbers, is discussed in chapter 6 in [15].

Whether using (2.1) or (2.2), the end results are the same, as we are only interested in the first n binary digits of $S(n,k,x)$ on the left side, for $k = 0,1,2$ and so on. I now introduce the digit sum function, also known as digit count for binary digits, as $\zeta_S(n,k,x)$. For a fixed n and x, it counts the number of '1' in the first n binary digits of $S(n,k,x)$. The normalized version

$$\zeta_S^*(n,k,x) = \frac{1}{n}\zeta_S(n,k,x) \tag{2.3}$$

takes a value between 0 and 1. It represents the proportion of '1' in the first n digits of $S(n,k,x)$. Therefore, $\zeta_S^*(n,n,x)$ represents the proportion of '1' in the first $\tau(n)$ digits in the binary expansion of $\exp(x)$. It is not

difficult to prove that $\tau(n) = n + O(1)$. Finally, to obtain theoretical bounds on the proportion of '1' in $\exp(x)$, here with $x = \pm 1$ or $x = 3$, you need to study the spectacular behavior of $\zeta_S^*(n, n, x)$ for fixed n and x, and then let $n \to \infty$. The remaining of the article focuses on this problem, and its various applications to quantum dynamics, cryptography, AI and LLM evaluation in particular, dynamical systems, high-performance computing, and so on.

2.2 Spectacular behavior of the digit sum function

Before digging into this problem, it is worth noticing that my approach is radically different from all previous attempts to prove that the binary digits of numbers such as e or π are evenly distributed. They all failed even at proving much weaker results. The main innovations in my framework are:

- Focusing on very few digits on the left side, rather than many digits on the right side and ignoring the leftmost part of the digit expansion. Yet, as $n \to \infty$, 'very few' eventually becomes infinite.
- To uncover patterns, experimenting with numbers as large as $2^n + 1$ at power 2^n, with $n = 10^5$, far beyond the capability of any computer system. This is possible thanks to the truncation mechanism.
- Working with systems exhibiting strong and fascinating patterns that can be precisely quantified, rather than navigating in the dark. It helps visualize the steps necessary to build a solid proof.
- Building on expertise with discrete dynamical systems, thus having a clear idea of what to expect, both in terms of leverage and challenges.
- Working with the most basic seeds leading to a unique behavior. We do not care if the seed is attached to the main invariant measure or not, as we stop at iteration $k = n$ in $S(n, k, x)$, that is, at the very end of the non-chaotic regime of the underlying dynamical system, just before full chaos sets in.

2.2.1 Potential scenarios

Moving forward, I use $n = 10^5$ in all my illustrations. Unless otherwise specified, the integers n and x are fixed. As with all similar dynamical systems (dyadic map, in particular), the main function of interest, in this case the digit sum $\zeta_S(n, k, x)$ as a function of k, can have a wide variety of shapes depending on the seed. Here, the seed corresponds to $S(n, k, x)$ with $k = 0$. That is, $S(n, 0, x) = 2^n + x$.

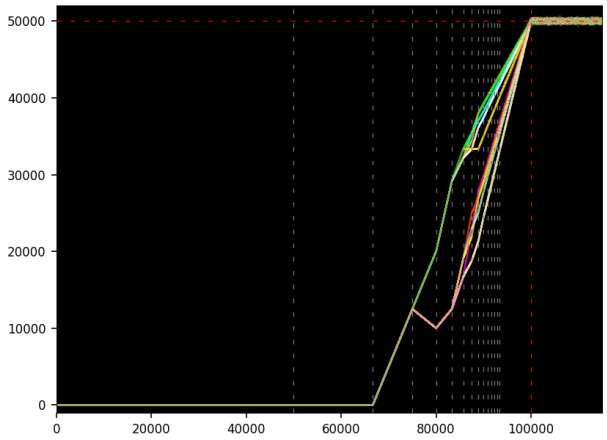


Figure 2.1: Digit sum $\zeta_S(n, k, x)$ with $x = 1$, $n = 10^5$, and k on the X-axis

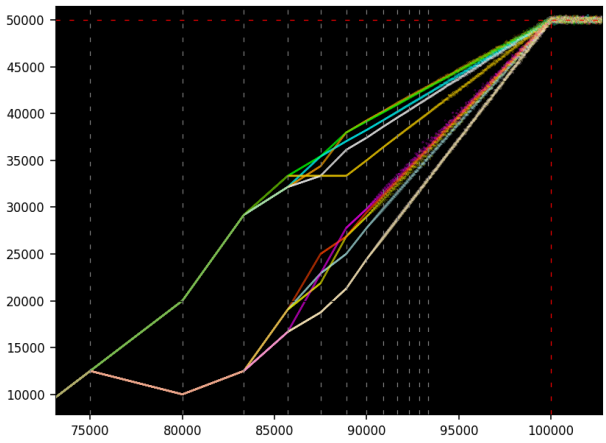


Figure 2.2: Zoom in on the right part of Figure 2.1

Figure 2.1 features an extremely rare behavior, and one of the easiest to predict. Nevertheless, it is the only one of interest to prove deep results about the binary digits of e. The seed corresponds to a string with $n - 1$ consecutive '0's, with a '1' at both ends. I will use this case as a basis to describe alternate, much more common scenarios.

At $k = 3n/4$, the bifurcation phase starts and lasts until $k = n$. And when $k \geq n$, we are in the full chaotic phase, with the proportion of '1' oscillating around 50%. Interestingly, at $k = n$, the width of the horizontal band visible in the top right corner is conjectured to be $O(\sqrt{n \log \log n})$ at most, a consequence of the law of the iterated logarithm, and $O(\sqrt{n})$ on average, a consequence of the central-limit theorem.

The limit of the digit sum function when $n \to \infty$, when plotted on a fixed-size window, is a continuous time process with infinitesimal increments. Yet, unlike a Brownian motion (the limit of a random walk), it is

nowhere continuous, jumping from one state to another as k increases. The finite number of potential states also increases as k increases. Each state has its own color in Figure 2.1, and corresponds to a particular residue of k in a specific congruential class. Despite being nowhere continuous, the graph appears to be much less erratic than a Brownian motion. In particular, the segments are connected with no apparent discontinuity, a property specific to the seed $2^n + x$ with $x = \pm 1$ or $x = 3$, but not true in other cases. Due to the potential states, the multi-branch function is called a quantum function. Such functions, including quantum derivatives, are discussed in section 4.4 in [17], in the context of the Riemann Hypothesis.

I can now share a number of possible scenarios for the behavior of the digit sum function, depending on the seed, for a large variety of seeds. The list below does not cover all the cases, but only those that are either the most common, or related to our discussion. The seed must have at least n bits, starting with '1', and ending with '1' unless its length is infinite.

- The most common case by far is when the seed is a random string with n bits. Then there is no warm-up phase. Instead, we enter the chaotic phase at the very beginning, when $k = 0$, instead of $k = n$ in Figure 2.1.
- If the seed consists mostly of '0', with the proportion of '1' above (say) 10%, then the chaotic phase is reached in very few iterations. This is the most interesting case for cryptography.
- Seeds with $n = 10^5$ bits with only 3 to 5 bits set to '1's at random locations, are useful for research. The chaotic phase can be reached pretty fast, but sometimes with gaps in the warm-up phase. See Figure 2.7.
- If the seed is a real number $2^{q/p}$ with p, q coprime integers, p an odd prime, then there is no chaotic phase. The digit sum function is periodic. The length of the period is the multiplicative order of 2 modulo p.

The seeds $S(n, 0, x)$ that we are working with to establish deep results about the digits of e, are among the very few that require so many iterations ($k = n$) before reaching the chaotic phase.

2.2.2 Dynamics of the unbroken bifurcation process

I conjecture that the seeds $S(n, 0, x) = 2^n + x$ with $x = \pm 1$ or $x = 3$, results in no gap during the non-chaotic phase, unlike Figure 2.7. If gaps were present, proving the desired result would potentially be more difficult. I use the word "unbroken bifurcation process" to describe the absence of gap.

If you look at Figure 2.1, new forks and changes in the slope occur at specific locations k on the X-axis. The vertical dashed lines show these locations. They are identical regardless of n, and whether $x = \pm 1$ or $x = 3$. These locations correspond to $k/n \approx \rho_1$, ρ_2, ρ_3 and so on, with

$$\rho_1 = \frac{1}{2},\ \rho_2 = \frac{2}{3},\ \rho_3 = \frac{3}{4},\ \rho_4 = \frac{4}{5},\ \rho_5 = \frac{5}{6},\ \ldots \tag{2.4}$$

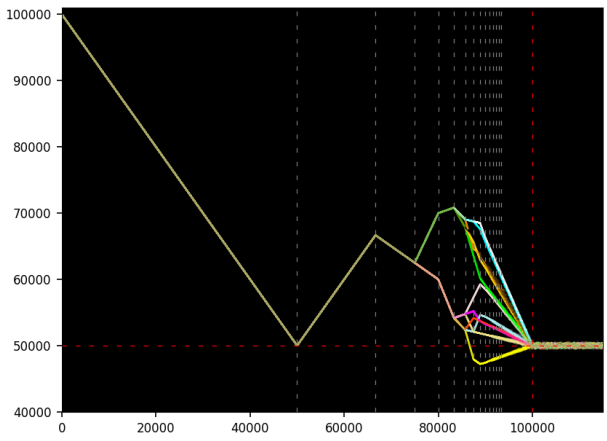


Figure 2.3: Digit sum $\zeta_S(n, k, x)$ with $x = -1$, $n = 10^5$, and k on the X-axis

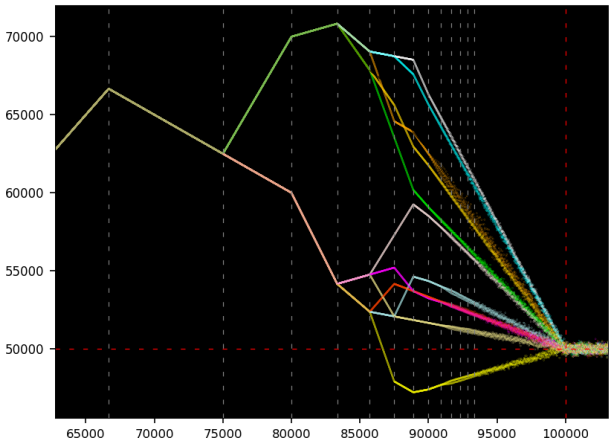


Figure 2.4: Zoom in on the right part of Figure 2.3

These approximated values are extremely accurate. On the right on the X-axis, beyond $k/n = \rho_7$, the number of change points start exploding with more and more segments, shorter and shorter, eventually evolving in a very narrow range just before reaching the chaotic phase with about 50% of '1'. Within each vertical band delimited by consecutive vertical dashed lines, the number of segments, from left to right, is respectively equal to 1, 1, 2, 2, 4, 12 and so on. Except for the first two values, these are factorials multiplied by 2. Since $S(n, k, x) = 2^n + x$

at power 2^k, as k increases this combinatorial explosion is not surprising. Then, based on (2.4), the widths of each band, from left to right, are proportional to

$$\frac{1}{1\cdot 2}, \quad \frac{1}{2\cdot 3}, \quad \frac{1}{3\cdot 4}, \quad \frac{1}{4\cdot 5}, \quad \cdots$$

The proportionality factor is n. Paths are determined by congruential residues. For instance, the green path in Figure 2.2 corresponds to values of k such that $k \equiv 4 \bmod 12$. Finally, the shape of the graph, for a specific x, stays the same regardless of n. But different values of x produce different shapes.

2.3 Case studies

Figures 2.1 and 2.2 feature the case $x = 1$. Here, I show the dynamics of the digit sum function for the cases $x = -1$ and $x = 3$, associated respectively with the binary digits of e^{-1} and e^3. Then Figure 2.7 shows the results when using a seed consisting of a string of length $n = 10^5$, all '0' except three '1' at random locations, plus a '1' at both ends. Finally, I discuss what happens when averaging values within each vertical band delimited by vertical dashed lines in Figures 2.1 and 2.2.

The case $x = -1$ is featured in Figures 2.3 and 2.4. The seed $2^n - 1$ consists of n bits all '1', without any '0'. The graph is more elaborate than the cases $x = 1$ and $x = 3$, starting at $k = 0$ with all '1', with the proportion going down to 50% as we reach $k = n/2$, but then bouncing back up before eventually stabilizing around 50% when $k > n$. Despite the different behavior compared to $x = 1$ or $x = 3$, the vertical dashed lines indicating new bifurcations and slope changes, still have the same abscissas.

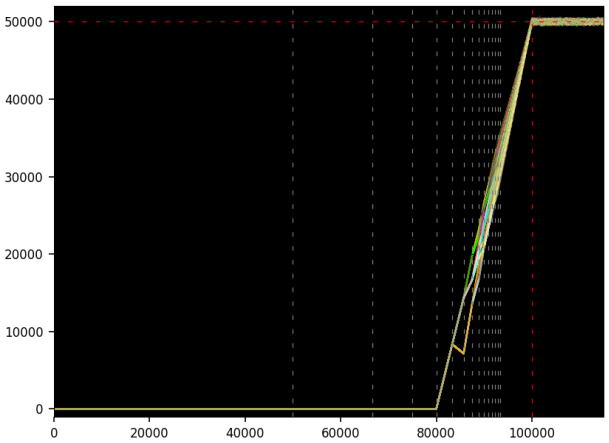


Figure 2.5: Digit sum $\zeta_S(n, k, x)$ with $x = 3$, $n = 10^5$, and k on the X-axis

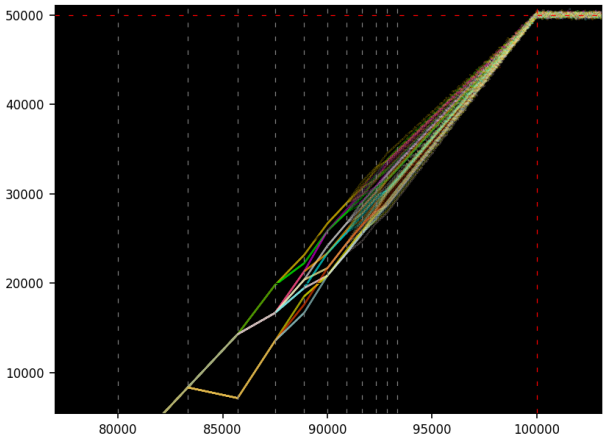


Figure 2.6: Zoom in on the right part of Figure 2.5

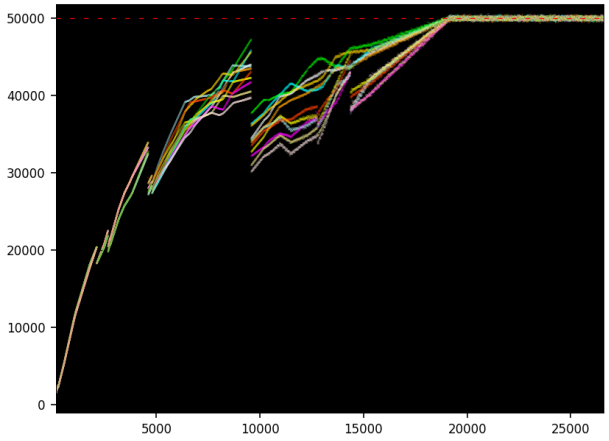


Figure 2.7: Digit sum: seed with $n = 10^5$ bits with five '1' at random locations

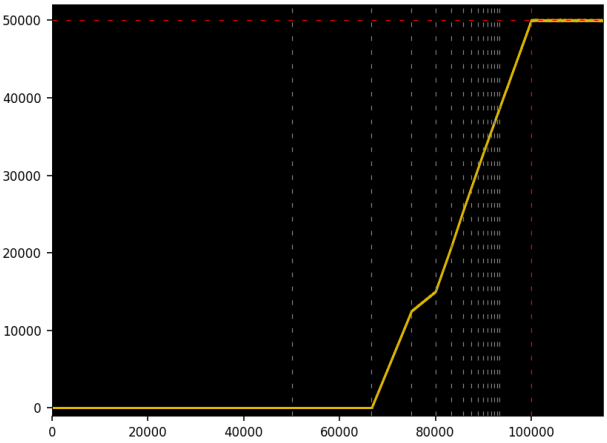


Figure 2.8: Averaging the segments within each vertical band in Figure 2.1

At first glance, it seems more difficult to prove a formal theorem about the digit distribution for this case. However, due to the various segments being more spread out, possibly with fewer hidden features, it certainly makes the patterns easier to quantify, maybe facilitating a proof for the digits of e^{-1}, compared to e or e^3.

Figures 2.5 and 2.6 correspond to the case $x = 3$. The seed has $n+1$ bits, all '0' except the first one on the left and the two rightmost equal to '1'. Despite the seed being apparently more complex than the case $x = 1$, having three '1' rather than two, the bifurcation phase (before chaos) evolves in a much narrower range, mostly with increases in the proportion of '1' over time until about 50% is reached, with very few, moderate setbacks. It could make a proof easier than for the case $x = 1$. However many features may be hidden, making pattern analysis more complicated. Again, the positions of the vertical dashed lines are unchanged.

Figure 2.7 features a seed not related to the digits of any known number. It is not part of a scheme where increasing n results in convergence of $S(n, n, x)$ to some constant. Unlike the previous examples, the locations of the '1's in the seed of length $n = 10^5$ are spread randomly, yet with a '1' at each end. The total number of '1' is five. Thus, depending on the locations of the random '1's, the integer x can be a very large number, possibly much larger than $2^{n/2}$ yet much smaller than 2^n. Nevertheless, this case is very interesting because it shows that gaps sometimes occur. While mild in most cases, in this example they are rather substantial. Note that the X-axis is truncated to magnify the gaps, as they appear early in the process, with the rightmost visible one at $k < n/6$.

Finally, if you average the segments withing each vertical band delimited by consecutive vertical dashed lines in Figure 2.1, you obtain Figure 2.8. Thus, we are approaching a straight line starting at $k = 4n/5$, reaching the chaotic phase at $k = n$ for the number of '1's in the first n binary digits of $2^n + 1$ at power 2^k. Thus, with about 50% of '1' when $n = k$, for the first n digits of e.

To produce Figure 2.8, I actually used a much simpler technique to get an approximation: the curve is a moving average based on a window with 12 consecutive values of k. In addition, all the pictures and results featured here were produced using the Python code in section 3.4. My algorithm is based on formula (2.1) combined with the truncation mechanism. Also, I double-checked the correctness in the digits of e with an external library. The fastest version of the code is on GitHub, here. It runs about twice as fast as the version in section 3.4, by truncating $S(n, k, x)$ to $2n - k$ bits rather than $2n$, and removing the trailing '0's on the right.

2.4 An extreme case

Here $x = \log \mu$ with $\mu = 2$. We need a precision of n bits on $\log \mu$. It is equivalent to using the seed μ^{r_n} with $r_n = 2^{-n}$. Thus, $S(n, n, x) = \mu$ and $S(n, n-1, x) = \sqrt{\mu}$, up to the first n digits. This example shows yet another type of behavior in the digit sum function.

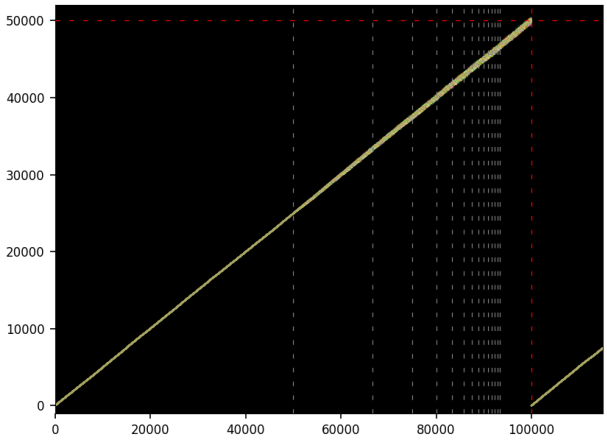


Figure 2.9: Digit sum $\zeta_S(n, k, x)$ with $x = \log 2$, $n = 10^5$, k on the X-axis

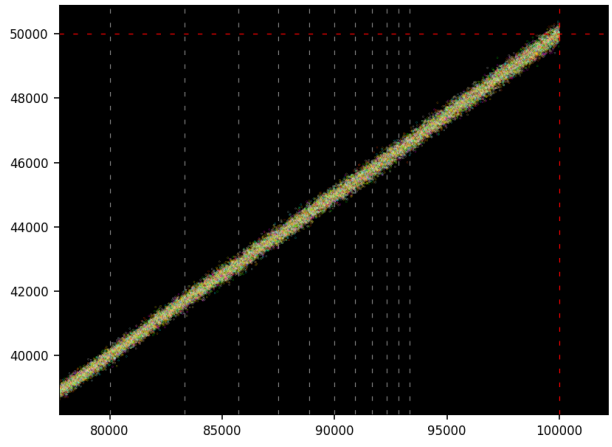


Figure 2.10: Zoom in on the right part of Figure 2.9

The proportion of '1' in the first n digits at $k = n$ is 0% or 100% depending on whether we approach the limit from under or from above, since $2 = 0.11111\ldots = 1.00000\ldots$ in binary form. For a fixed n, formula (2.1) – combined with truncating $S(n, k, x)$ to $2n$ bits at each iteration k – guarantees about n correct digits in $S(n, k, x)$ up to $k = n$. Beyond $k = n$, the precision continues to drop. This explains why the rightmost part of the digit sum function in Figure 2.9, when $k > n$, is not an horizontal line as it should be: it is due to increasing loss in precision as k increases.

Note that in this example, there is no bifurcation process. Also, the coloring scheme is meaningless, and not linked to congruential classes. I used the same code as in section 3.4 to produce Figure 2.9 and 2.10. The only difference is about the computation of the seed, done with the following code:

```
mu = 2
prod = 2**n + gmpy2.log(mu)
prod = int(2**n * prod)
```

The seed is denoted as `prod` in the code, and turned into an integer with a precision of $2n$ bits.

2.5 Applications and AI Challenge with petabytes dataset

In this section, I first provide a quick overview of potential applications, with recent references. Then I discuss an interesting challenge: using large language models (LLMs) to leverage my research and uncover deeper insights, test and compare their mathematical and pattern detection capabilities, based on the digit dataset and via reasoning. After all, the iterates $S(n, k, x)$ are highly correlated strings similar to sentences as in English prose or DNA sequences, here with an alphabet consisting of two letters: '0' and '1'. Let's start with the references:

- The framework presented here relies on discrete quadratic dynamical systems. This family also includes the logistic map and the example discussed in [48]. For additional references, see my book on chaos and dynamical systems [15].
- 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 [5] are good references on this topic. For recent publications on normal numbers, see Verónica Becher [6] 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 [47].
- 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 [49]. 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 $S(n, k, x)$, see [1, 8].
- An interesting application of the digit sum is featured in [32] in the context of genotype maps, with processes not unlike the dynamical systems discussed in this chapter, and blancmange curves almost identical to Figure 3.3 in my book on numeration systems [15].
- There is a connection to quantum maps and quantum cryptography [11, 46]. For PRNGs (pseudo-random generators) based on irrational numbers, see chapter 13 in [16] or chapter 4 in [15]. Finally, if you use an arbitrary seed instead of $S(n, 0, x) = 2^n + 1$, 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 [34, 45, 51]. In our case, the model would be a simple formula that generates the values of the digit sum function, to study its asymptotic properties.

2.5.1 AI challenge

The last item in the bullet list brings an interesting challenge. The idea is to use AI and large language models to get the full picture about the patterns. The goal is to formally prove the deepest possible results that you can get from my framework, about the binary digits of the number e, or related numbers such as e^{-1} or e^3. In particular, we want to fully understand and accurately quantify the bifurcation phase shown in all the figures. Questions to answer include

- The number of segments (referred to as quantum states) in each vertical band delimited by successive vertical dashed lines, for instance in Figures 2.2, 2.4, and 2.6. We want to find a general formula for the number of segments based on the location on the X-axis.
- Which values of k correspond to each segment. We know that the answer depends on the residues of k modulo specific integers linked to factorials. What are these residues and the modulo classes in question? It seems like and exact, simple, and general answer can be obtained. The number of segments within a vertical bands is directly linked to the modulo classes in question.
- The slopes of these segments, and the mean slope when averaged within a vertical band, with a general formula applicable to any location on the X-axis. Show that when the slope is moving in the wrong direction (away from the 50% target in the proportion of '1'), it can only do so for so long.

- Confirm the absence of gaps when $x = \pm 1$ or $x = 3$, by contrast to Figure 2.7. What gap-free cases share in common? Also confirm and extend Formula (2.4) to cover the entire range from $k = 0$ to $k = n$. Finally, what is the exact shape of the envelope pictured in Figure 2.11, showing the minimum and maximum potential values for the digit sum function at any location k, with $0 \leq k \leq n$.

Rather than smart guesses or predictions, we want actual proofs whenever possible. The purpose of using AI is not to get approximations to model parameters, but exact values when possible, and even a generic formula that generates all the values, such as Formula (2.4). AI may also be able to identify other applications underlined by the same dynamical models, providing valuable information. This approach is discussed in a recent paper on data-driven model discovery [33]. The following is an example of result obtained with AI in a different context. Yet it is closely related to the digit distribution of special math constants and in particular, to the material discussed in section 4.3.2 when $x = \frac{1}{16}$.

$$\sum_{k=0}^{\infty} \binom{3k}{k} x^k = \frac{2}{\sqrt{4-27x}} \cos\left[\frac{1}{3}\arcsin\left(\frac{3}{2}\sqrt{3x}\right)\right]. \tag{2.5}$$

Note that $\binom{3k}{k}$ is a binomial coefficient. This formula was obtained with Wolfram's computational intelligence platform, a freely accessible AI agent. You can verify this result here. The series converges if $|x| < 4/27$.

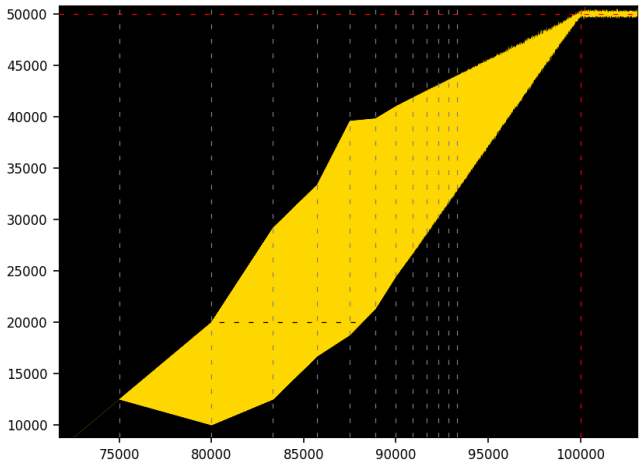


Figure 2.11: Upper/lower bounds for $\zeta_S(n, k, x)$ with $x = 1$, $n = 10^5$, and k on the X-axis

2.5.2 The dataset

Large language models (LLMs) are trained to predict the next tokens given previous tokens, based on a large database of text. In our case, for a fixed n and x, each $S(n, k, x)$ is a string consisting of $2n$ bits. In LLM parlance, it is a block of tokens, each token consisting of (say) 512 bits. We want to predict $S(n, k+1, x), S(n, k+2, x)$ and so on, based on $S(n, k, x), S(n, k-1, x)$ and so on. The training set consists of all $S(n, k, x)$ with $0 \leq k \leq n$. We only care about the first n bits on the left. Indeed are only interested in the digit sum function computed on these first n bits, at any k. The problem is trivial until k gets close to n. The closer to n, the harder. Thus we can restrict training and predictions to $k/n > 0.85$. The predictions must all be exact in this case, and the methodology must use cross-validation.

Detecting the rules that make correct predictions is even more important. Or identifying other well-studied dynamical systems that have a very similar behavior, with known properties. In short, anything that can lead to formally proving a deep result about the digit sum, is highly valuable. Better, an actual proof if some LLMs can come up with one. Perhaps something like this: the proportion of '1' in the binary digits of e is above 10%. This in itself would be a phenomenal result, beating everything obtained previously, by a very long shot. Proving that it is exactly 50% is the best that we could expect.

For that purpose, one can create a dataset consisting of $S(n, k, x)$ strings with $n = 10^7$, for 100 different values of x, some causing gaps and some not in the digit sum function. This requires generating $2 \times 10^7 \times 10^7 \times 10^2$ bits, that is, 2.5 petabytes. LLMs that can detect the patterns with a much smaller dataset, and those that still perform well very close to $k = n$, should be rewarded. The code in section 3.4, with $n = 10^5$, is a starting point to create the dataset. It runs fast (about a minute) on a small laptop. Each value of x can be run in parallel. I uploaded the values of the digit sum function for $x = 1$, $n = 10^5$, and $0 \leq k \leq n$, on GitHub, here.

2.6 Python code

The Python code `number_theory_fast_v2.py` is also on GitHub, here. The variable `p` plays the role of x in the code. If set to 0, it generates a random seed. The variable `H` determines the upper limit for k, that is, the number of iterations. In Part 2 in the code, I use the Mpmath library to compute the digits of e, to double check that my algorithm yields the correct digits as advertised. Finally, the envelope in Figure 2.11 is produced as a plot in Part 3, rather than the default scatterplot. The code to produce Figure 2.8 is in Part 4.

```
n = 100000
H = int(1.15*n)

def assign_color(k):

   if k%12 == 0:
      color = 'lime'
   elif k%12 == 2:
      color = 'white'
   elif k%12 == 4:
      color = 'black'
   elif k%12 == 6:
      color = 'cyan'
   elif k%12 == 8:
      color = 'gold'
   elif k%12 == 10:
      color = 'orange'
   elif k%12 == 1:
      color = 'magenta'
   elif k%12 == 3:
      color = 'paleturquoise'
   elif k%12 == 5:
      color = 'khaki'
   elif k%12 == 7:
      color = 'yellow'
   elif k%12 == 9:
      color = 'mistyrose'
   elif k%12 == 11:
      color = 'orangered'
   return(color)


#--- Part 1. Main

import gmpy2
import numpy as np

kmin = 0.00 * n # compute digit sum if k > kmin
kmax = 1.15 * n # compute digit sum if k < kmax
kmax = min(H, kmax)

# precision set to L bits to keep at least about n correct bits till k=kmax
ctx = gmpy2.get_context()
L = n + int(kmax+1)
ctx.precision = L

# p = 1: for e; ~ n correct bits after n iter
#  seed with n+1 bits, all 0 except rightmost and leftmost
# p = 3: for e^3, ~ n correct bits after n iter
#  seed with n+1 bits, all 0 except 2 rightmost and leftmost
# p = -1: for 1/e, ~ n correct bits after n iter
#  seed with n bits, all 1
# p must be integer != 0; use p=0 for random seed

p = 1 # try -1, 0 (random seed), 1, 3

def create_random_seed(n, cnt1):

   # create random seed of length n-1 with cnt1 '1' at random locations
   # and add a 1' at both ends; cnt1 must be <= n-1

   cnt1 = min(cnt1, n-1)
   numpy_seed = 453 # numpy seed to initiate numpy PRNG, not the model seed
   np.random.seed(numpy_seed)
   random_locations = np.random.choice(np.arange(1, n),size=cnt1,replace=False)
   prod = 2**n + 1 # seed with n+1 '0' except a '1' at both ends
   for position in random_locations:
```

```
        # add the cnt1 random '1's between both ends
        prod += 2**int(position)
    return(prod)


# create seed with n+1 bits if p>=0, or n bits if p<0
if p != 0:
    prod = gmpy2.mpz(2**n + p)
else:
    # random seed with number of '1' in seed to cnt1+2
    # test: set n = 30000; cnt1 = 0, 3, 4, 5, 100 and see what happens!
    cnt1 = 3
    prod = create_random_seed(n, cnt1)

# local variables
arr_count1 = []
arr_colors = []
xvalues = []
ecnt1 = -1

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

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

    prod = prod*prod
    pstri = bin(int(prod))
    stri = pstri[0: L+2]
    prod = int(stri, 2)
    prod = gmpy2.mpz(prod)

    if k > kmin and k < kmax:
        stri = stri[2:]
        if k == n:
            e_approx = stri
        estri = stri[0:n] # leftmost n digits
        ecnt1 = estri.count('1')
        arr_count1.append(ecnt1)
        color = assign_color(k)
        arr_colors.append(color)
        xvalues.append(k)
        OUT.write(str(k) + "\t" + str(ecnt1) + "\n")

    if k%1000 == 0:
        print("%3d %3d" %(k, ecnt1))

OUT.close()


#--- Part 2. Double-check the digits of e

from mpmath import mp

# Set precision to L binary digits
mp.dps = int(L*np.log2(10))
e_value = (mp.e)**abs(p) # Get e^|p| in decimal

if p > 0:
    # Convert e_value to binary string
    e_binary = bin(int(e_value * (2 ** n)))[2:]

elif p < 0:
    e_iapprox = int(e_approx, 2) # convert string e_approx to integer
    e_ivalue = int(2**(2*n) * e_value)
    one = e_iapprox * e_ivalue
    e_approx = bin(one)[2:]
    e_binary = "1" * (2*n) # string of 2n bits, all '1'

if p != 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
    print("\n%d correct digits (n = %d)" %(k, n))


#--- Part 3. Create the main plot
```

```
144
145 import matplotlib.pyplot as plt
146 import matplotlib as mpl
147 import numpy as np
148
149 mpl.rcParams['axes.linewidth'] = 0.5
150 plt.rcParams['xtick.labelsize'] = 8
151 plt.rcParams['ytick.labelsize'] = 8
152 plt.rcParams['axes.facecolor'] = 'black'
153
154 plt.scatter(xvalues, arr_count1,s=0.01, c=arr_colors)
155 # plt.plot(xvalues, arr_count1,linewidth=0.04, c='gold')
156
157 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
158 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
159 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
160
161 for k in range(1,15):
162    plt.axvline(x=k*n/(k+1),c='gray', linestyle='--', linewidth=0.6,dashes=(5, 10))
163
164 if p > 0:
165    # start with about 0% of 1 going up to about 50%
166    ymax = 0.52 * n
167    plt.ylim([-0.01 * n, ymax])
168 elif p < 0:
169    # start with 100% of 1 going down to about 50%
170    ymax = 1.01 * n
171    plt.ylim([0.40 * n, ymax])
172 elif p == 0:
173    ymax = 1.00 * n
174    plt.ylim([-0.01 * n, ymax])
175 plt.xlim([kmin, kmax])
176
177 plt.show()
178
179 #--- Part 4. Create the average plot
180
181 arr_avg = []
182 arr_xval = []
183 arr_count1 = np.array(arr_count1)
184
185 for k in range(0, int(kmax-12)):
186    y_avg = np.average(arr_count1[k:k+12])
187    arr_avg.append(y_avg)
188    arr_xval.append(k)
189
190 plt.scatter(arr_xval,arr_avg,s=0.0002, c='gold')
191 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
192 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
193 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
194 plt.xlim(0.00*n, kmax-12)
195 plt.ylim(-0.01*n, ymax)
196
197 for k in range(1,15):
198    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
199
200 plt.show()
```

Chapter 3

From Digit Sum to Universal Dataset and Benchmarking AI Algorithms

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 chapter, 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.

3.1 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 3.2 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 3.3.

It all started with a seed string S_0 that consists of $n + 1$ bits, all zeros except a one at both ends, thus representing the integer $2^n + 1$. The self-convolution 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 Chapter 1.

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 dataset has $n + 1$ rows, one for each S_k $(k = 0, 1, \ldots, n)$. 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 an ergodic discrete 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 chapter and also in Chapter 1. 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 Chapter 2 and 10^4 in Chapter 1. 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.

3.2 Deep dive into the digit sum function

In Chapter 2, 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. Exceptions 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

$$S_{n-k} = \exp\left(\frac{x}{2^k}\right) = 1 + \frac{1}{1!}\frac{x}{2^k} + \frac{1}{2!}\frac{x^2}{2^{2k}} + \cdots \tag{3.1}$$

for $k = 0, 1$ and so on. Formula (3.1) 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

$$S_k = 1 + \frac{1}{1!}\frac{x}{2^{n-k}} + \frac{1}{2!}\frac{x^2}{2^{2(n-k)}} + \cdots \tag{3.2}$$

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.

3.2.1 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 Chapter 1.

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 $0 \leq k \leq n$.

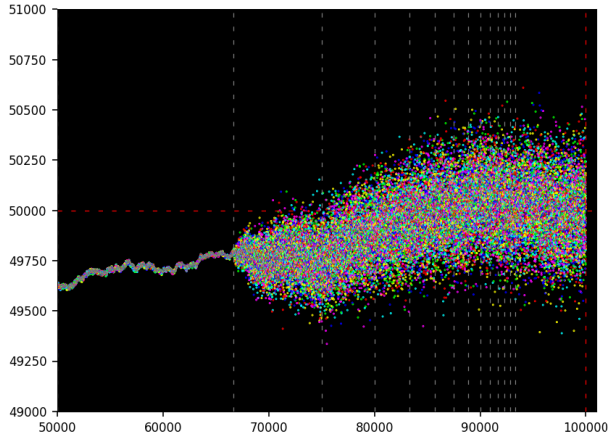


Figure 3.1: Adjusted digit sum ζ'_k, random seed, $n = 10^5$, k on X-axis

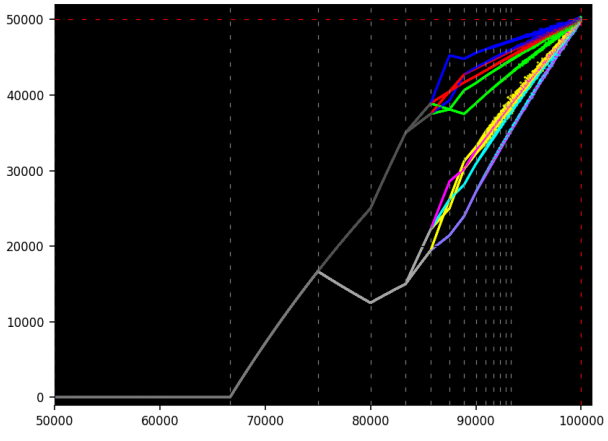


Figure 3.2: Adjusted digit sum ζ'_k, seed with $x = 1$, $n = 10^5$, k on X-axis

Figure 3.1 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 3.2 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 2.1 and 2.2, also based on the same seed but using the non-adjusted digit sum ζ_k instead of ζ'_k. As in Figure 3.1 in this chapter, 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 Chapter 1, 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 3.2.2 goes one step further, in an attempt to reach such a major milestone.

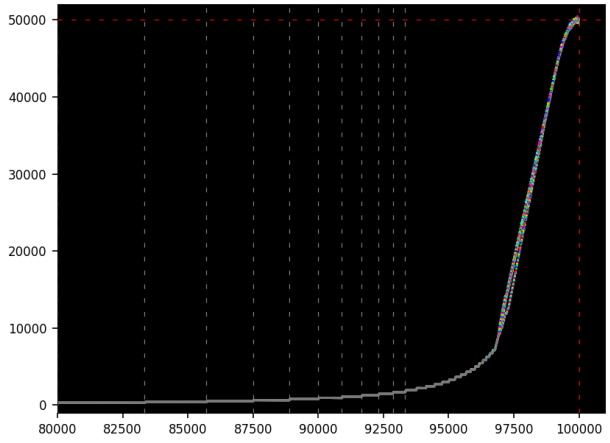


Figure 3.3: Adjusted digit sum ζ'_k with primorial, $n = 10^5$, k on X-axis

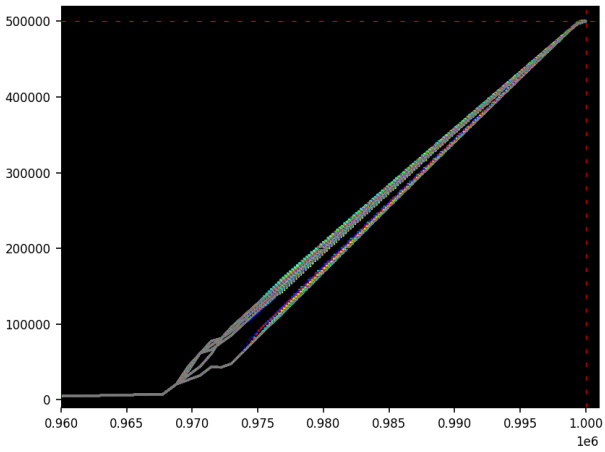


Figure 3.4: Adjusted digit sum ζ'_k with primorial, $n = 10^6$, k on X-axis

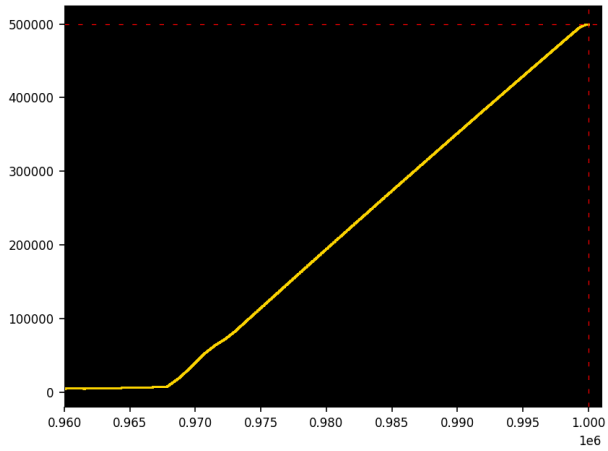


Figure 3.5: Moving average applied to Figure 3.4, window size is 60

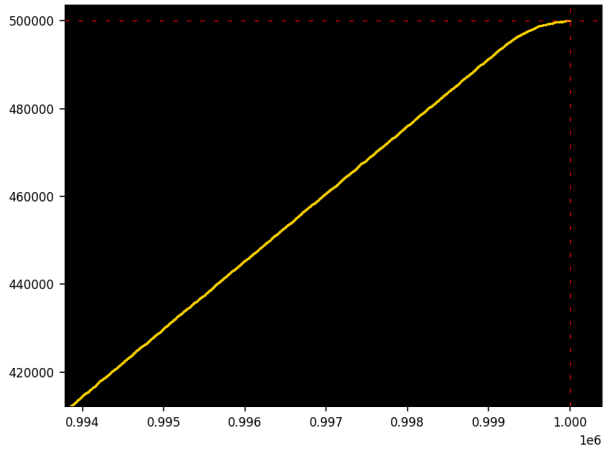


Figure 3.6: Zoom on top right corner in Figure 3.5

3.2.2 Spectacular behavior of digit sum with primorials

Formula (3.2) 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 (3.2) 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

$$x_\kappa = \pi_\kappa \cdot 2^{\nu_\kappa}, \quad \text{with } \nu_\kappa = -\lfloor \log_2 \pi_\kappa \rfloor. \tag{3.3}$$

Figure 3.3 and 3.4 show the substantial reduction in chaos as k gets very close to n, when using $x = x_\kappa$ defined by formula (3.3) 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 3.5 and 3.6 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 (3.3) results in extensive carry-over across multiple terms in formula (3.2), thus increasing chaos.

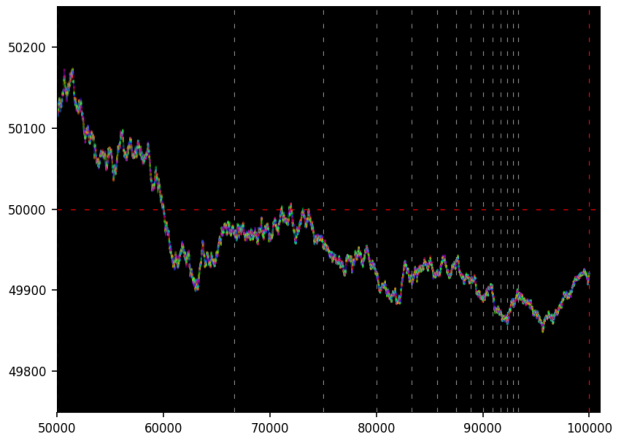


Figure 3.7: Typical digit sum function obtained with traditional method

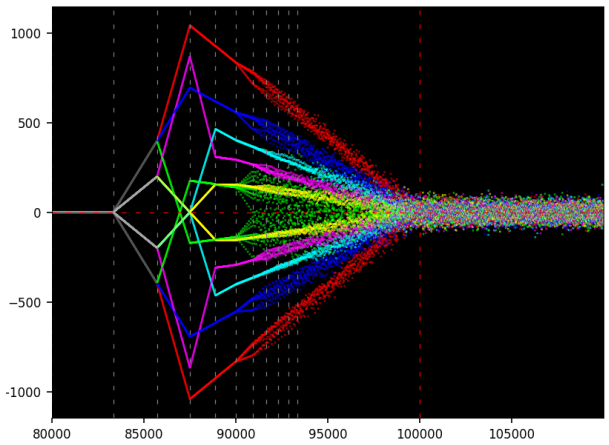


Figure 3.8: $\zeta_{k+60} - \zeta_k$, with $n = 10^5, x = 1$ and k on X-axis

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.

3.2.3 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 3.4.1, 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 3.2 is a first step in that direction, showing ζ'_k's path with a different color based on k mod 6. Likewise and not surprisingly, averaging w consecutive values of k, where w is a number with many divisors ($w = 60$ in Figure 3.5) 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.

Also, the digit sum function has been extensively studied in other contexts, see [30]. 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. Also, $\zeta(y_1 y_2; b) \equiv \zeta(y_1; b)\zeta(y_2; b) \bmod (b - 1)$. In particular,

$$\zeta(S_{k+1}; b) = \zeta(S_k^2; b) \equiv \zeta^2(S_k; b) \bmod (b - 1). \tag{3.4}$$

This applies to the full set of digits, not the first n ones as in ζ_k. Working with different bases that are power of 2, with n also a power of 2, is another topic of interest.

Finally, it would be interesting to look at iterations leading to fixed points that are algebraic numbers. For instance, consider the system $S_{k+1} = 2^{-\nu} P(S_k)$ where $P(z)$ is a polynom with integer coefficients, $S_0 = 0$, and ν is a positive integer. Under the right conditions S_k converges to a solution of $2^{-\nu} P(z) = z$, with quadratic convergence. In addition, all S_k are dyadic rationals, that is, integer numbers divided by a power of 2. The case $P(z) = 1 + z^2$ with $\nu = 3$ leads to $S_\infty = 4 - \sqrt{15}$. The first k binary digits of S_k and S_∞ are identical. Actually, a lot more than the first k.

3.2.4 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 [48]. For additional references, see my book on chaos and dynamical systems [15].
- 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 [5] are good references on this topic. For recent publications on normal numbers, see Verónica Becher [6] 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 [47].
- 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 [49]. 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, 8].
- An interesting application of the digit sum is featured in [32] in the context of genotype maps, with processes not unlike the dynamical systems discussed in this chapter, and blancmange curves almost identical to Figure 3.3 in my book on numeration systems [15].
- There is a connection to quantum maps and quantum cryptography [11, 46]. For PRNGs (pseudo-random generators) based on irrational numbers, see chapter 13 in [16] or chapter 4 in [15]. 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 [34, 45, 51]. 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 [33].

3.2.5 Comparison with standard methodology

For almost every seed S_0, our methodology produces an adjusted digit sum ζ'_k similar to that in Figure 3.1. Exceptions include $S_0 = 2^n + x$ with x a small integer or $x = x_\kappa$, leading to shapes like those respectively in Figure 3.2 and 3.3.

The classic methodology proceeds differently, by looking at the proportions of '1' in the first k digits of the target number $\exp(x)$, for $k = 1, 2$ and so on all the way to ∞. In almost all examples, and for all normal numbers in particular, it behaves as a re-scaled Brownian motion similar to the one featured in Figure 3.7. That picture corresponds to $x = x_\kappa$ with $\kappa = 30$. There is no indication that x_κ is very peculiar when using the standard methodology.

Also, there is no clear path to explain why the binary digits of $\exp(x)$, for specific values such as $x = 1$ or $x = x_\kappa$, are evenly distributed. Evidence on small data (a few trillion digits) suggests that this is true. Yet there is no reason to believe that this is the case for the whole digit sequence, other than the fact it is true for almost all numbers. Still, there are infinitely many exceptions, for instance if $\exp(x)$ is a rational number.

The methodology presented here, based on the iterated self-convolutions of a seed string S_0 with $2n + 1$ bits, shows a path towards formal results. Yet, we are still far from a mathematical proof. For instance, if $S_k = \sqrt{2}$ when $k = n - 1$, then $S_n =$'1', corresponding to a number where all digits are '0' except the first one. But in that case, the seed S_0 is random. The opposite is true with $S_n = \exp(1)$ truncated to n digits: the seed S_0 consists entirely of '0' exept a '1' at both ends.

3.3 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 3.4.1 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-correlation 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 3.8. 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 Chapter 2 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 Chapter 2. 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 3.4. 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.

3.4 Python code

Here I share two different versions of my program: one based on the forward recursion in section 3.4.1, starting with S_0, and the other one based on the backward recursion in section 3.4.2, starting with 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, including the primorial computations (π_κ) discussed in section 3.2.2. Both programs also produce extra plots not discussed in this chapter. In both programs, x is represented by the variable `p`, while S_k is represented by the variable `prod`.

3.4.1 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, and keeping a precision of $2n - k$ bits at iteration k, rather than $2n$. 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
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)

```

```
140
141 #--- 3. Create the main plot
142
143 import matplotlib.pyplot as plt
144 import matplotlib as mpl
145 import numpy as np
146
147 mpl.rcParams['axes.linewidth'] = 0.5
148 plt.rcParams['xtick.labelsize'] = 8
149 plt.rcParams['ytick.labelsize'] = 8
150 plt.rcParams['axes.facecolor'] = 'black'
151
152 plt.scatter(xvalues, arr_count1,s=0.01, c=arr_colors)
153 #plt.plot(xvalues, arr_count1,linewidth=0.1, c='gold')
154
155 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
156 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
157 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
158
159 for k in range(1,15):
160    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
161
162 # we start with about 0% of 1 going up to about 50%
163 ymax = 0.52 * n
164 plt.ylim([-0.01 * n, ymax])
165 #plt.ylim([-0.01 * n, 1.01*n])
166
167 plt.xlim([kmin, kmax])
168 # plt.xlim([0.0*n, kmax])
169 plt.show()
170
171
172 #--- 4. Create the moving average plot
173
174 arr_avg = []
175 arr_xval = []
176 arr_count1 = np.array(arr_count1)
177 w = 6 # moving average window
178
179 for k in range(0, len(arr_count1)-w):
180    y_avg = np.average(arr_count1[k:k+w])
181    arr_avg.append(y_avg)
182    arr_xval.append(k)
183
184 plt.scatter(arr_xval,arr_avg,s=0.0002, c='gold')
185 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
186 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
187 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
188 plt.xlim(0.00*n, kmax-w)
189 plt.ylim(-0.01*n, ymax)
190
191 for k in range(1,15):
192    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
193
194 plt.show()
195
196 nv = len(arr_xval)
197 st = int(4*n/5)
198 arr_delta = np.array(arr_avg[1:nv]) - np.array(arr_avg[0:nv-1])
199 plt.scatter(arr_xval[st+1:nv], arr_delta[st+0:nv-1], s=0.08, c=arr_colors[st+1:nv])
200 for k in range(1,15):
201    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
202 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
203 plt.axhline(y=0,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
204 plt.xlim(st, nv)
205 plt.show()
206
207
208 #--- 5. Create AR scatterplot
209
210 nv = n
211 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])
212 plt.show()
```

3.4.2 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
```

```
144
145 seed_type = "primorial"
146 ncolors = 6 # try number with many divisors: 12, 30, ...
147 colorTable = generate_contrasting_colors(ncolors)
148 z = initialize_reverse_seed(seed_type)
149 (arr_count1, arr_colors, xvalues) = iexp(n, start, iters, ncolors, z, u, v, truncate)
150
151 #--- 6. Create the main plot
152
153 import matplotlib.pyplot as plt
154 import matplotlib as mpl
155 import numpy as np
156
157 mpl.rcParams['axes.linewidth'] = 0.5
158 plt.rcParams['xtick.labelsize'] = 8
159 plt.rcParams['ytick.labelsize'] = 8
160 plt.rcParams['axes.facecolor'] = 'black'
161
162 plt.scatter(xvalues, arr_count1,s=0.2, c=arr_colors)
163 #plt.plot(xvalues, arr_count1,linewidth=0.1, c='gold')
164
165 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
166 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
167 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
168
169 for k in range(1,15):
170    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
171
172 plt.ylim([-0.01 * n, 0.52*n])
173 plt.xlim([0.70 * n, 1.02*n])
174 plt.show()
175
176 #--- 7. Create the moving average plot
177
178 arr_avg = []
179 arr_xval = []
180 arr_count1 = np.array(arr_count1)
181 w = 60
182 for k in range(0, len(arr_count1)-w):
183    #tmp = arr_count1[k:k+w]
184    #print(k, len(tmp)) ##,arr_count1[k])
185    y_avg = np.average(arr_count1[k:k+w])
186    arr_avg.append(y_avg)
187    arr_xval.append(n-start-k)
188
189 plt.scatter(arr_xval,arr_avg,s=0.02, c='gold')
190 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
191 plt.axhline(y=n/5, color='black', linestyle='--', linewidth = 0.6, dashes=(5, 10))
192 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
193 plt.xlim([0.50*n, 1.01*n])
194 for k in range(1,15):
195    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
196 plt.show()
197
198 nv = len(arr_xval)
199 st = 0 ##int(4*n/5)
200 arr_delta = np.array(arr_avg[0:nv-1])-np.array(arr_avg[1:nv])
201 plt.scatter(arr_xval[st+1:nv], arr_delta[st+0:nv-1], s=0.08, c=arr_colors[st+1:nv])
202 for k in range(1,15):
203    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
204 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
205 plt.axhline(y=0,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
206 plt.show()
207
208 #--- 8. One more scatterplot
209
210 nv = len(arr_count1)
211 w = 1
212 arr_delta = np.array(arr_count1[0:nv-w]) - np.array(arr_count1[w:nv])
213 plt.scatter(xvalues[w:nv], arr_delta, s=0.08, c=arr_colors[w:nv])
214 for k in range(1,15):
215    plt.axvline(x=k*n/(k+1),c='gray',linestyle='--', linewidth=0.6,dashes=(5, 10))
216 plt.axvline(x=n, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
217 plt.axhline(y=0,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
218 plt.show()
```

Chapter 4

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

Using the logistic map instead of the base quadratic system as in Chapter 3, 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.

4.1 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 the previous chapters.

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 4.1.

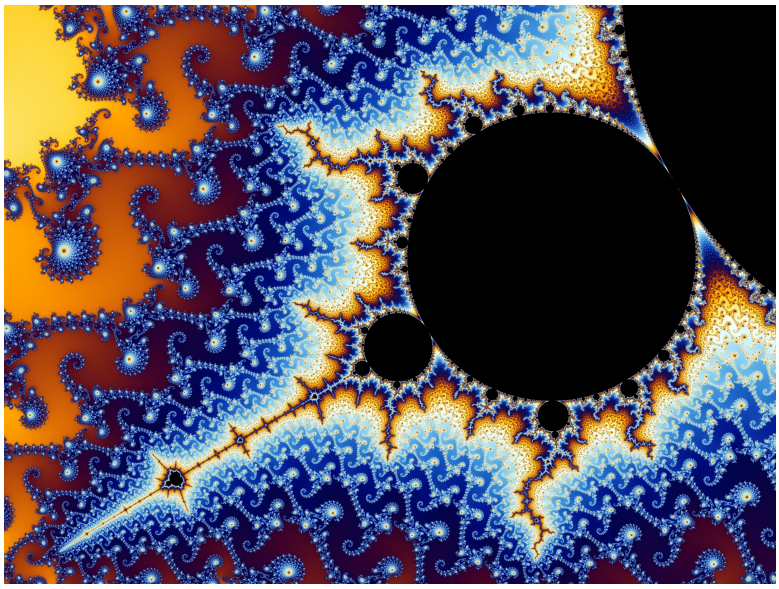


Figure 4.1: Mandelbrot set linked to the standard quadratic map

The goal here is to obtain similar results for the logistic map, thus expanding my investigations about the

digit sum function in chapter 1. 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.

4.2 Logistic map and the digit sum function

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

$$S_k = \sin^2(2^k \arcsin \sqrt{S_0}). \tag{4.1}$$

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

$$S_k = \sin^2\left(\frac{x^{1/2}}{2^{n-k}} + \frac{1}{6} \cdot \frac{x^{3/2}}{2^{3n-k}} + \cdots\right). \tag{4.2}$$

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

$$S_k = \left(1 + \frac{x}{2^n}\right)^{2^k} \sim \exp\left(\frac{x}{2^{n-k}}\right) \tag{4.3}$$

converging to $\exp(x)$ when $k = n$ and $n \to \infty$. The right part in (4.3) has an accuracy of about $2n - k$ bits if the precision on the left part is kept to $2n$ bits at all times.

4.2.1 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 $\rho \geq 1$, we are in full chaotic phase. Thus I focus on $0.75 < \rho < 1$.

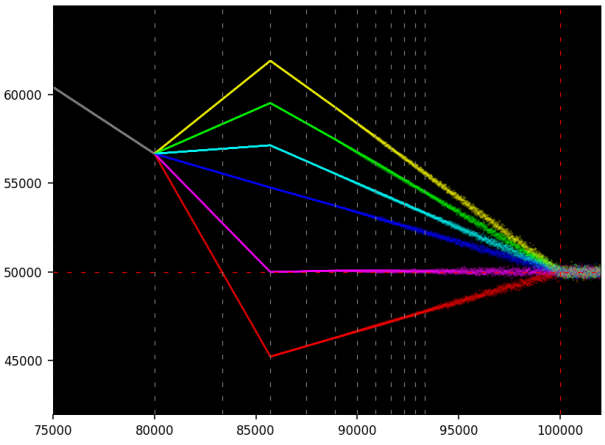


Figure 4.2: Logistic map: ζ'_k with $x = 1, n = 10^5$, k on X-axis

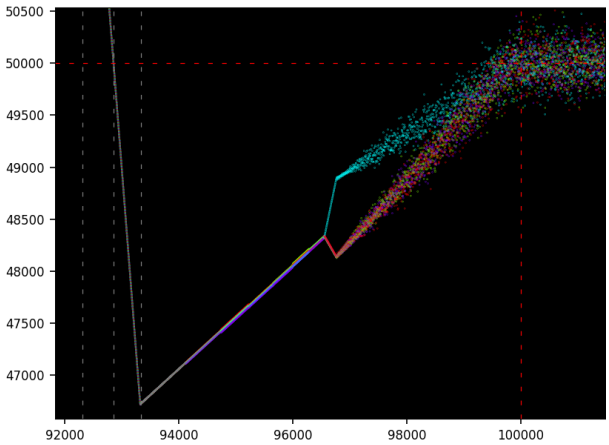


Figure 4.3: Logistic map: ζ'_k with $x = \pi_\kappa, n = 10^5 - 1$, k on X-axis

From now on, BQ denotes the base quadratic map with illustrations in chapter 3. 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 4.4 and 4.7.

- **Quantic**. See Figures 4.2 and 4.5 for the logistic map, and Figure 4.8 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 4.3 and 4.6. Here, $x = \pi_\kappa$ is the κ-th primorial, that is, the product of the first κ primes ($\kappa = 9$).

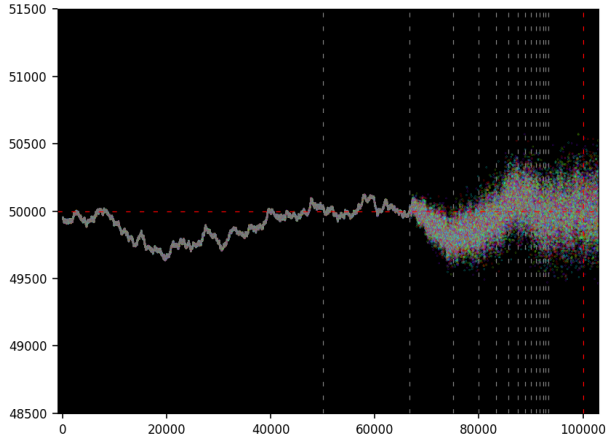


Figure 4.4: Logistic map: ζ'_k with $x = \sqrt{2}, n = 10^5$, k on X-axis

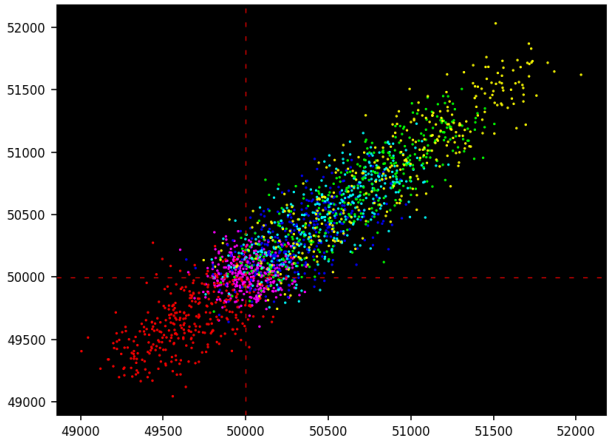


Figure 4.5: Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = 1, n = 10^5$

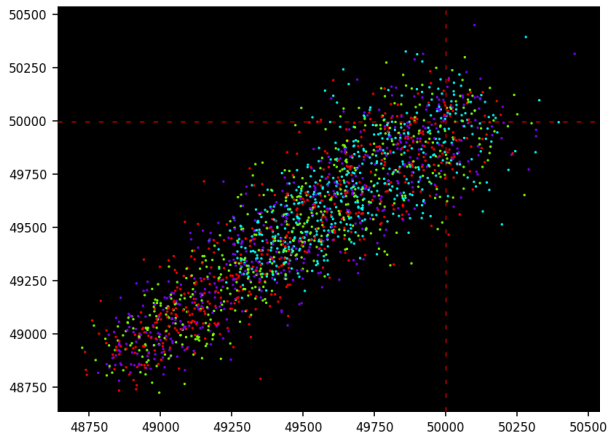


Figure 4.6: Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = \pi_\kappa, n = 10^5 - 1$

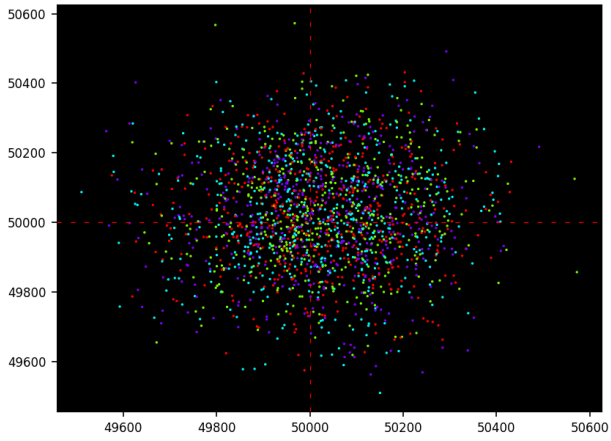


Figure 4.7: Log. map: $(\zeta'_{k-12}, \zeta'_k)$ with $0.98 < \frac{k}{n} < 1$, $x = \sqrt{2}, n = 10^5$

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 (4.2) and (4.3). See section 3.2.2 for details.

Figures 4.2, 4.3, and 4.4 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 4.5, 4.6 and 4.7 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 4.7 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 4.4 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.

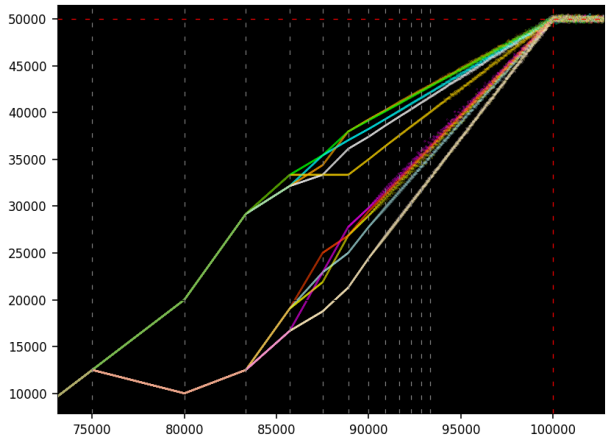


Figure 4.8: BQ map: ζ'_k with $x = 1, n = 10^5$, k on X-axis

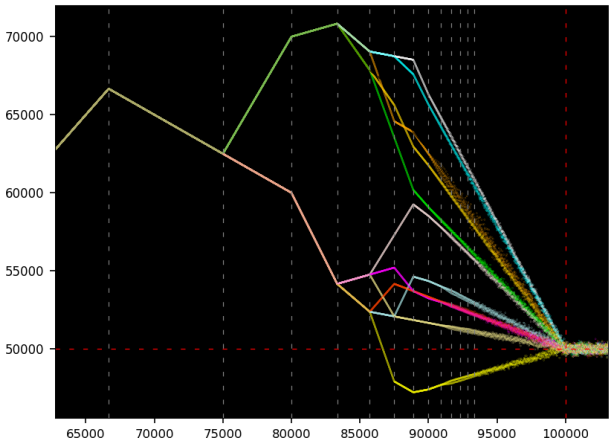


Figure 4.9: BQ map: ζ_k with $x = -1, n = 10^5$, k on X-axis

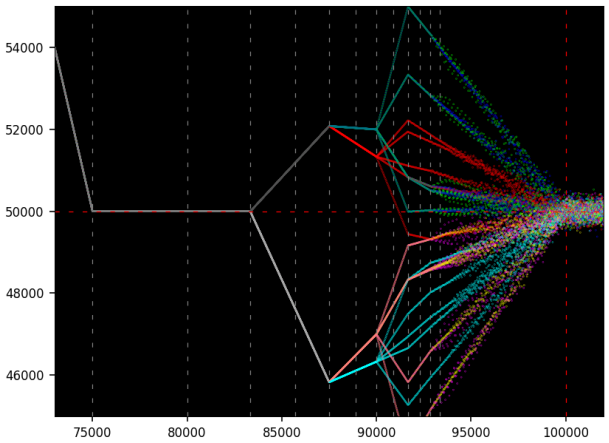


Figure 4.10: Logistic map: ζ_k with $x = 9, n = 10^5 - 1$, k on X-axis

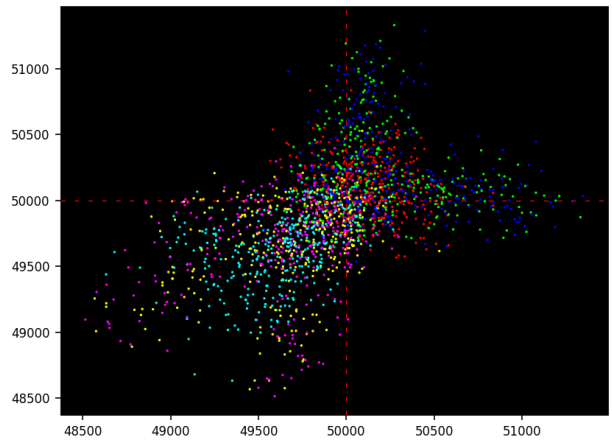


Figure 4.11: Log. map: (ζ_{k-12}, ζ_k) with $0.98 < \frac{k}{n} < 1$, $x = 9, n = 10^5 - 1$

See also Figures 4.8, 4.9, 4.10 and 4.11. 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 4.3 and 4.10 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.

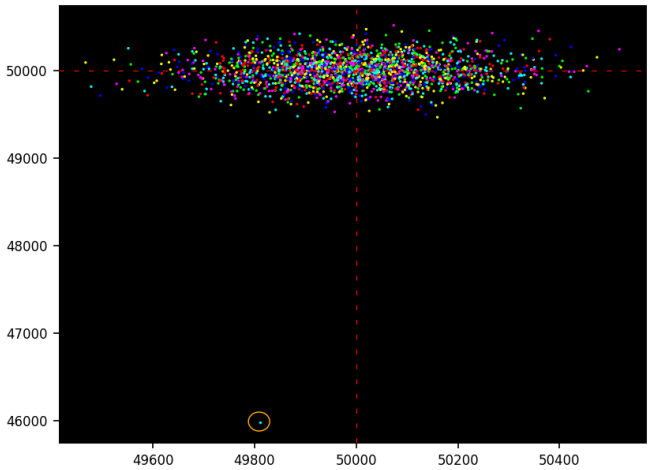


Figure 4.12: Same as Fig. 4.7 but with x leading to $\zeta'_n/n \approx 0.46$ instead of 0.50

Note that Figures 4.10 and 4.11 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. For pictures featuring moving averages that collapse multi-branches into a single one with a constant slope, see Figure 3.5.

Finally, in Chapter 3, 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 4.5, S_k is denoted as `prod`. The link to fractals is visible in Figures 1.1 and 1.3.

4.2.2 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 chapter.

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 4.4, how would the sequence (ζ'_k) look like, for $k = 0, 1$ and so on? The answer: exactly as in Figure 4.4. But the spectral view would be very different from the corresponding Figure 4.7. Instead, it would look like Figure 4.12.

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 4.12, 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 4.2 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 4.4

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

$$S_{n-1} = \frac{1 \pm \sqrt{1 - S_n}}{2} \tag{4.4}$$

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 (4.4). Set the precision to $2n$ bits and proceed as in all other cases ($x = 1, 9$, and so on) using the code in section 4.5 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
```

4.2.3 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 [48]. For additional references, see my book on chaos and dynamical systems [15].

- 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 [5] are good references on this topic. For recent publications on normal numbers, see Verónica Becher [6] and [2, 30]. 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 [47].
- 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 [49]. 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, 8].
- An interesting application of the digit sum is featured in [32] in the context of genotype maps, with processes not unlike the dynamical systems discussed in this chapter, and blancmange curves almost identical to Figure 3.3 in my book on numeration systems [15].
- There is a connection to quantum maps and quantum cryptography [11, 46]. For PRNGs (pseudo-random generators) based on irrational numbers, see chapter 13 in [16] or chapter 4 in [15]. 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 [34, 45, 51]. 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 [33].

4.3 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 (3.2), 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.

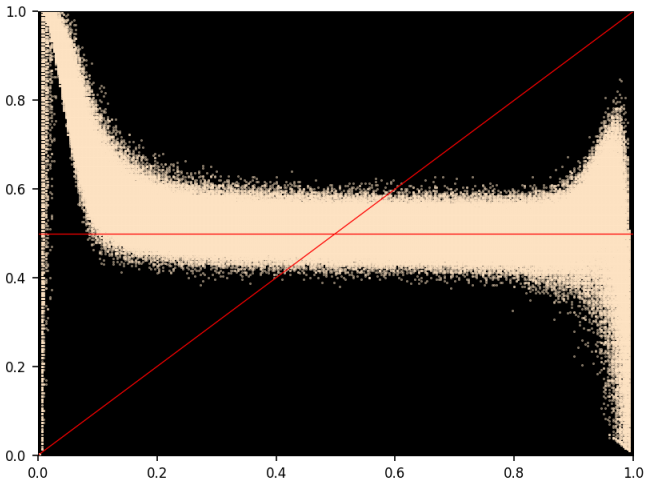


Figure 4.13: Scatterplot: proportion of '1' in y (X-axis) vs. in $y' = y(1-y)$ (Y-axis), with y in $[0, 1]$

4.3.1 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.

Figure 4.13 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 4.13, 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 4.13, 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()
```

4.3.2 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

$$\psi_n(x) = \sum_{k=0}^{n} B_k x^k, \tag{4.5}$$

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

$$y'_n = \Big(1 - \frac{1}{2^\nu}\Big)\psi_n\Big(\frac{1}{4^\nu}\Big) + \frac{1}{2^\nu}\sum_{k=0}^{n}\Big(\frac{1}{4^\nu}\Big)^k M_{k,\nu} \tag{4.6}$$

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

$$y_n = \psi_n\Big(\frac{1}{2^\nu}\Big). \tag{4.7}$$

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 (4.6). 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

$$\frac{1}{2^\nu} = \psi^{-1}(y), \quad \frac{1}{4^\nu} = \Big[\psi^{-1}(y)\Big]^2 \tag{4.8}$$

in formula (4.6).

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 (4.6), to

$$y' = \frac{2^\nu - 1}{\sqrt{4^\nu - 4}} + \frac{2^\nu}{4^\nu - 4} \tag{4.9}$$

When $\nu = 0$, the series for y and y', based on formulas (4.5), (4.6), and (4.7), diverge. But if each y'_n is multiplied by negative integer power of 2 to stay within $[1, 2[$, then $y' = \frac{4}{3}$.

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

$$\psi(x) := \sum_{k=0}^{\infty} \lfloor g(k) \rfloor x^k = \frac{x}{1-x} \sum_{k=1}^{\infty} x^{\nu_k} \tag{4.10}$$

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 (4.10), 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 (4.10) and (4.7) with $x = \frac{1}{2}$ and $\nu = 1$.

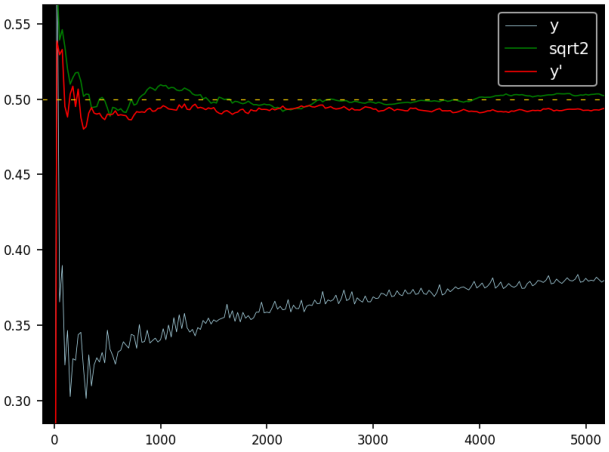


Figure 4.14: Proportion of '1' at iteration k, with k on X-axis, $b = 1.5$

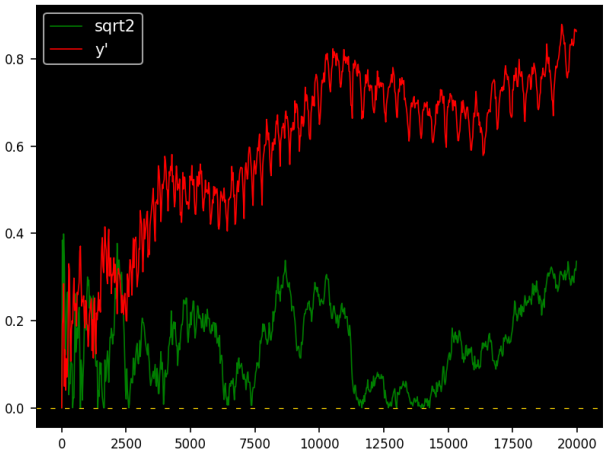


Figure 4.15: Distance to 50% of '1' at iteration k, with k on X-axis, $b = 1, 5$

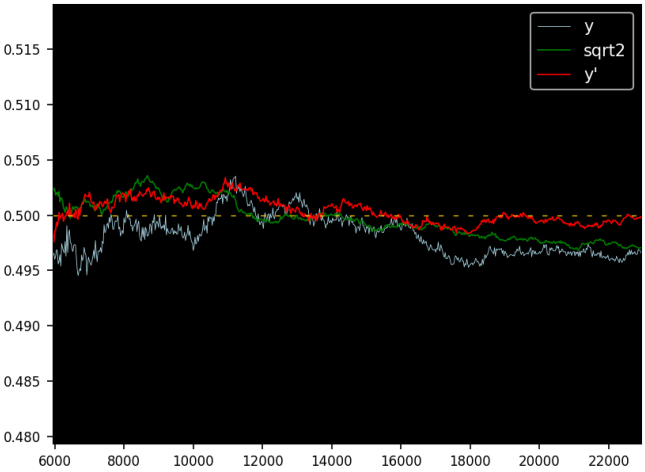


Figure 4.16: Proportion of '1' at iteration k, with k on X-axis, $b = 3.5$

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 (4.9); `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 4.14, 4.15, 4.16 deal with $B_k = \lfloor g(k) \rfloor$ featured in formula (4.10), 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 4.15 displays $|q_k - 0.5|\sqrt{k}$ where q_k is the proportion of '1' obtained at iteration k. As seen in Figure 4.14, 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)
```

```
66          B_k = int(stri, 2)
67          B_max = 2**(nu + dx) - 1
68
69       elif mode == 'SmallGrowth':
70          cx = 3.5
71          bx = 1
72          B_k = int(bx * k**cx)
73          # try: B_k = 3**k >> k # equal to int[(3/2)**k]
74          B_max = 0
75          if B_k > 0:
76             bits = len(bin(B_k)) - 2
77             B_max = 2**(bits+1) # equal to 2^(int(log2 B_k))
78
79       u_k = B_k
80       v_k = B_max - B_k
81       s_k = pow2 * u_k + v_k
82
83       sum_u = pow2**2 * sum_u + u_k
84       sum_v = pow2**2 * sum_v + v_k
85       sum = pow2 * sum_u + sum_v # for y'
86       sum_y = pow2 * sum_y + B_k # for y
87       # equivalenly, sum = pow2**2 * sum + s_k
88
89       if k % 25 == 0:
90          # computations needed to visualize intermediary results
91          stri1 = bin(sum_y)[2:]
92          stri2 = bin(sqrt2)[2:]
93          stri3 = bin(sum)[2:]
94          rlen = -1
95          correct_digits = -1
96          if mode == 'Bernoulli' and nu > 1:
97             # if correct_digits stalls, restart with larger N
98             correct_digits = compute_correct_digits(sum, pow2, n)
99             rlen = correct_digits/len(stri3)
100            stri1 = stri1[0:correct_digits]
101            stri2 = stri2[0:correct_digits]
102            stri3 = stri3[0:correct_digits]
103         else:
104            stri2 = stri2[0:len(stri3)]
105         q1 = stri1.count('1')/len(stri1)
106         q2 = stri2.count('1')/len(stri2)
107         q3 = stri3.count('1')/len(stri3)
108         arr_q1.append(q1)
109         arr_q2.append(q2)
110         arr_q3.append(q3)
111         arr_xval.append(k)
112         arr_rlen.append(rlen)
113         arr_delta2.append(abs(q2-0.5)*np.sqrt(k))
114         arr_delta3.append(abs(q3-0.5)*np.sqrt(k))
115
116         print("%5d %5d %6.4f %6.4f %6.4f %6.4f" %
117             (k, correct_digits, rlen, q1, q2, q3))
118
119 print()
120 print(stri1[-60:]) # last 60 digits of y
121 print(stri3[-60:]) # last 60 digits of y'
122 print()
123
124 #-
125
126 # compute rescaled y and y'; factor is a power of nu
127 # works even when series diverges (Bernoulli with nu in {0, 1})
128 # if Bernoulli with nu = 0, then y' = 4/3
129
130 y = 0
131 y_prime = 0
132 for k in range(60):
133    y+= int(stri1[k])/2**k
134    y_prime += int(stri3[k])/2**k
135 print("y = %14.12f" % (y))
136 print("y' = %14.12f" % (y_prime))
137
138
139 #--- 2. Main plot: % of dgits equal to '1' as k increases
140
141 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()
```

4.4 Conclusion

The previous chapters focus on the simplest quadratic map. Here I showed how to attain similar results with the most well-known dynamical system: the logistic map. It also confirms that orbits in chaotic dynamical systems are very sensitive to the seed. Indeed, 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 , agent based modeling, and more: see Chapter 2 and 3. 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.

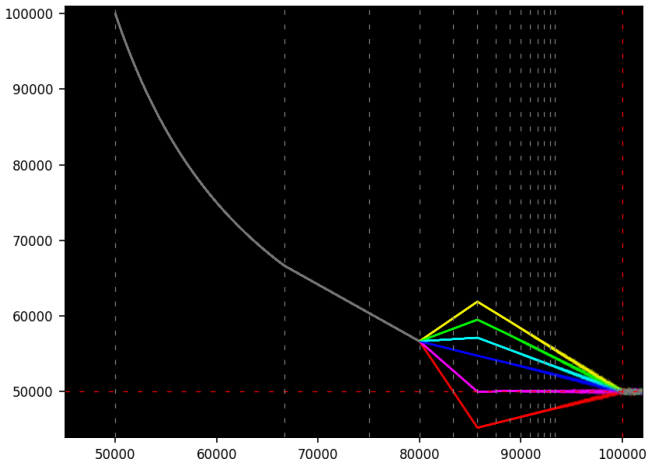


Figure 4.17: Unusual precision parameters producing curve and discontinuity

4.5 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 4.17, 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.

```
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
```

```
131 lag = 12
132 tail = 2000
133 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])
134 # 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])
135 plt.axhline(y=n/2,color='red',linestyle='--', linewidth=0.6,dashes=(5,10))
136 plt.axvline(x=n/2, color='red', linestyle='-', linewidth = 0.6, dashes=(5, 10))
137
138 plt.show()
```

Chapter 5

Test of Normality and Digit Distribution of Algebraic Numbers

The previous chapters feature new discoveries about the binary digit distribution of numbers related to Euler's number e, based on quadratic dynamical systems defined via simple auto-convolutions. In this chapter, the focus is on algebraic numbers, with binary digit sequences generated by a similar mechanism. A new fundamental theorem is introduced with proof, setting a new bar in what qualifies as "deep result" about these digit sequences, moving forward. Finally, in section 5.1, I discuss a test of randomness – more specifically about normality – based on a considerably simplified version of Weyl's criterion. Again, it relies on quadratic dynamical systems, this time with a state space consisting of 2×2 matrices with determinant and spectral radius equal to 1. The problem is strongly connected to asymptotic bounds of Chebyshev polynomials on some subsets of $[0, 1]$.

5.1 Simple normality test with application to PRNGs

The context is testing whether the binary digits of an irrational number, say $x_0 = \pi$, look random enough. This would make its digit sequence a good candidate to generate random bits. The methodology described here is new. It starts with a simplified version of the Weyl criterion for normality, along with introducing $y_0 = \cos 2\pi x_0$ to transform irrational numbers into rational ones via Chebyshev polynomials. Assuming $x_0 \in [0, 1]$, let start with the dyadic map $x_{n+1} = \{2x_n\}$ where the brackets denote the fractional part. The k-th binary digit of x_0 is the integer part of $2x_k$. For non-zero integers τ, the formula $x_{n+1} = \{2x_n\}$ can successively be rewritten as

$$\begin{aligned} x_{k+1} &= 2x_k \bmod 1 \\ 2\pi\tau x_{k+1} &= 2 \cdot (2\pi\tau x_k) \bmod 2\pi \\ \exp(2\pi i\tau x_{k+1}) &= \exp(2 \cdot 2\pi i\tau x_k) \\ \exp(2\pi i\tau x_{k+1}) &= \big(\exp(2\pi i\tau x_k)\big)^2 \\ \exp(2\pi i\tau x_k) &= \big(\exp(2\pi i\tau x_0)\big)^{2^k} \end{aligned}$$

Now, using the notation $z_k = \exp(2\pi i\tau x_k) = z_0^{2^k}$, we have

$$\prod_{k=0}^{n-1}(1 + z_k) = \prod_{k=0}^{n-1}\left(1 + z_0^{2^k}\right) = \sum_{k=0}^{2^n-1} z_0^k = \frac{1 - z_0^{2^n}}{1 - z_0}. \tag{5.1}$$

We say that x_0 is q-normal in base 2 if and only if

$$\lim_{n\to\infty}\left[\prod_{k=0}^{n-1}\left(1 + z_0^{2^k}\right)\right]^{1/n} = 1 \tag{5.2}$$

for all non-zero integer τ. Taking the logarithm, the convergence in (5.2) is equivalent to the following:

$$\lim_{n\to\infty}\frac{1}{n}\sum_{k=0}^{n-1} z_0^{2^k} = 0, \quad \text{that is ,} \quad \lim_{n\to\infty}\frac{1}{n}\sum_{k=0}^{n-1}\exp\left(2\pi i\tau 2^k x_0\right) = 0 \tag{5.3}$$

for all non-zero integer τ. Interestingly, the right part in formula (5.3) is the Weyl criterion for the normality of x_0 in base 2. Thus the concepts of q-normality and normality are identical. Thanks to (5.1), we can go one step further to considerably simplify the Weyl criterion. The result is stated in the following theorem.

Theorem 5.1.1 *A number $x_0 \in [0,1]$ is normal in base 2 if and only if the following condition is satisfied, for all non-zero integer τ:*

$$\lim_{n\to\infty} \frac{1}{n} \log\Big[1 - \cos(2\pi\tau 2^n x_0)\Big] = 0 \tag{5.4}$$

Proof

This is a consequence of the fact that the Weyl criterion for normality is equivalent to (5.2) which itself relies on (5.1). Taking the logarithm of the complex norm in (5.2) and combining with (5.1), the convergence criterion can be rewritten as

$$\frac{1}{n}\log\left|\left|\prod_{k=0}^{n-1}\Big(1+z_0^{2^k}\Big)\right|\right|^2 = \frac{1}{n}\log\left|\left|\frac{1-z_0^{2^n}}{1-z_0}\right|\right|^2 \to 0 \text{ as } n\to\infty.$$

To conclude, note that

$$\frac{1}{n}\log\left|\left|\frac{1-z_0^{2^n}}{1-z_0}\right|\right|^2 = \frac{1}{n}\log\big|\big|1-z_0^{2^n}\big|\big|^2 - \frac{1}{n}\log\big|\big|1-z_0\big|\big|^2$$

with

$$\frac{1}{n}\log\big|\big|1-z_0\big|\big|^2 \to 0, \quad \big|\big|1-z_0^{2^n}\big|\big|^2 = 2\Big(1-\cos(2\pi i\tau 2^n x_0)\Big), \quad \frac{1}{n}\log 2 \to 0.$$

Extra care is needed when taking the n-th root or the logarithm of complex numbers as these are not uniquely defined. The details are beyond the scope of this presentation. The product in (5.2) represents the geometric mean of a set of complex numbers. ∎

A consequence is that a rational number $x_0 = a/b$ cannot be normal. To prove it, use $\tau = b$ in (5.4). Then, the limit is $-\infty$, violating the criterion.

5.1.1 High-performance computing with Chebyshev polynomials

The computation of $\cos(2\pi\tau 2^n x_0)$ in formula (5.4) is not trivial when n is large, say $n = 10^5$. The problem is compounded by the fact that x_0 is irrational, for instance $x_0 = \log 2$. However, we can focus on a class of irrationals x_0 that are a lot easier to deal with. This is the case is $x_0 = (2\pi)^{-1}\arccos y_0$, with $y_0 = p/q$ a rational number and arccos the inverse cosine function also called arc cosine. That is,

$$x_0 = \frac{1}{2\pi}\arccos y_0, \quad y_0 = \cos 2\pi x_0, \quad y_0 = \frac{p}{q} \text{ with } 0 < p < q. \tag{5.5}$$

Here p, q are coprime integers with $q > 2$. Then, x_0 is still an irrational number. We then have

$$\cos(2\pi m x_0) = \cos(m \arccos y_0) = T_m(y_0), \text{ with } m = \tau 2^m. \tag{5.6}$$

Here T_m is the Chebyshev polynomial of degree m, $T_m(y_0)$ is a rational number, and (5.4) can be restated as

$$\lim_{n\to\infty} \frac{1}{n} \log\Big[1 - T_m(y_0)\Big] = 0, \text{ with } m = \tau 2^n. \tag{5.7}$$

Chebyshev polynomials satisfy the recursion $T_{m+1}(y) = 2yT_m(y) - T_{m-1}(y)$ with the initial conditions $T_0(y) = 1$ and $T_1(y) = y$. Now let $V_m(y) = T_{m-1}(y)$, with $V_0(y) = T_{-1}(y) = y$. It is easy to prove that

$$\begin{bmatrix} T_m(y) \\ V_m(y) \end{bmatrix} = A \begin{bmatrix} T_{m-1}(y) \\ V_{m-1}(y) \end{bmatrix} = A^m \begin{bmatrix} T_0(y) \\ V_0(y) \end{bmatrix} = A^m \begin{bmatrix} 1 \\ y \end{bmatrix}, \text{ with } A = \begin{bmatrix} 2y & -1 \\ 1 & 0 \end{bmatrix}. \tag{5.8}$$

Another expression, not used here, is also available:

$$T_m(y) = \frac{1}{2}\Big[\Big(y + i\sqrt{1-y^2}\Big)^m + \Big(y - i\sqrt{1-y^2}\Big)^m\Big]. \tag{5.9}$$

Note that if $0 < y_0 < 1$, then $|T_m(y_0)| \le 1$. To check the normality of x_0 using (5.7), we want to know how close $T_m(y_0)$ can get to 1 when y_0 is a rational number. Too close (for some non-zero integer τ as $n \to \infty$) implies that x_0 is not normal. The converse is true. Again, $m = \tau 2^n$. The topic of asymptotic bounds ($m \to \infty$) for $|T_m(y)|$ when $y \in [-1, 1]$ is studied in the literature [4, 39] and linked to the concept of logarithmic capacity. However, I could not find how this theory helps solve our problem.

Since $m = \tau 2^n$, there is a very efficient way to compute $T_m(y_0)$ based on formula (5.8). First, let $A_0(\tau) = A^\tau$. Then iteratively compute $A_{k+1}(\tau) = A_k^2(\tau)$. We have $A^m = A_n(\tau)$, and $T_m(y_0)$ is the first component of the bivariate vector $A^m \cdot (1, y_0)^T$. See the Python code in section 5.1.2, featuring high-performance computing with the gmpy2 library.

5.1.2 Application with test of randomness and Python code

Figure 5.1 shows, for any n up to $n = 5000$, the upper and lower bounds of $\lambda_n(\tau, y_0)$ over all $\tau \in \{1, \ldots, 100\}$, with n on the X axis and $y_0 = \frac{3}{5}$, based on

$$\lambda_n(\tau, y_0) = \frac{1}{n} \log \Big[1 - T_m(y_0)\Big] = \frac{1}{n} \log \Big[1 - \cos(2\pi m x_0)\Big], \text{ with } m = \tau 2^n. \tag{5.10}$$

The convergence of both curves to zero empirically shows that $x_0 = (2\pi)^{-1} \arccos \frac{3}{5}$ is normal in base 2. It also means that the binary digits of x_0 are equidistributed. This is a weak form or randomness, yet superior to what is currently implemented in standard random number generators based on rational numbers with a finite period. The upper bound (orange curve) is straightforward as $\lambda_n(\tau, y_0) \leq (\log 2)/n$ regardless of τ or y_0. The challenge is with the lower bound (blue curve) for which no asymptotic minimum (lim inf) is guaranteed.

Failure to satisfy normality happens when the cosine term in (5.10) gets too close to 1 as n gets increasingly large, that is, when the fractional part of mx_0 gets either too close to 1 or too close to 0. Therefore, the Weyl criterion for normality of x_0 in base 2 can further be simplified to

$$\lim_{n \to \infty} \frac{1}{n} \log \Big(\big| \{\tau 2^n x_0\} \big| \Big) = 0 \tag{5.11}$$

for all non-zero integer τ. The brackets $\{\cdot\}$ represent the fractional part function. It may still be impossible to verify if x_0 is a number such as π, $\sqrt{2}$ or $\log 2$. Thus our focus on numbers such as $x_0 = (2\pi)^{-1} \arccos \frac{3}{5}$.

For a stronger concept of normality, called strong normality, see chapter 4 in [15]. It better captures strong randomness. Finally, computing $T_m(y_0)$ is based on the recursion $A_{k+1}(\tau) = A_k^2(\tau)$ with $A_0(\tau) = A^\tau$ where A is the 2×2 matrix defined in formula (5.8). The determinant and spectral radius of A are both equal to 1. Thus, this is also true for any integer power of A. The recursion in question, preserving the determinant and spectral radius, corresponds to a quadratic dynamical system also called quadratic map where the state space consists of 2×2 real matrices with determinant and spectral radius equal to 1. There are strong connections to the material presented in chapter 1, where I also use a quadratic map to compute the digits of a special irrational number, using integers only and a truncation mechanism similar to that in the code below, with the same goal of assessing whether or not the binary digits look random.

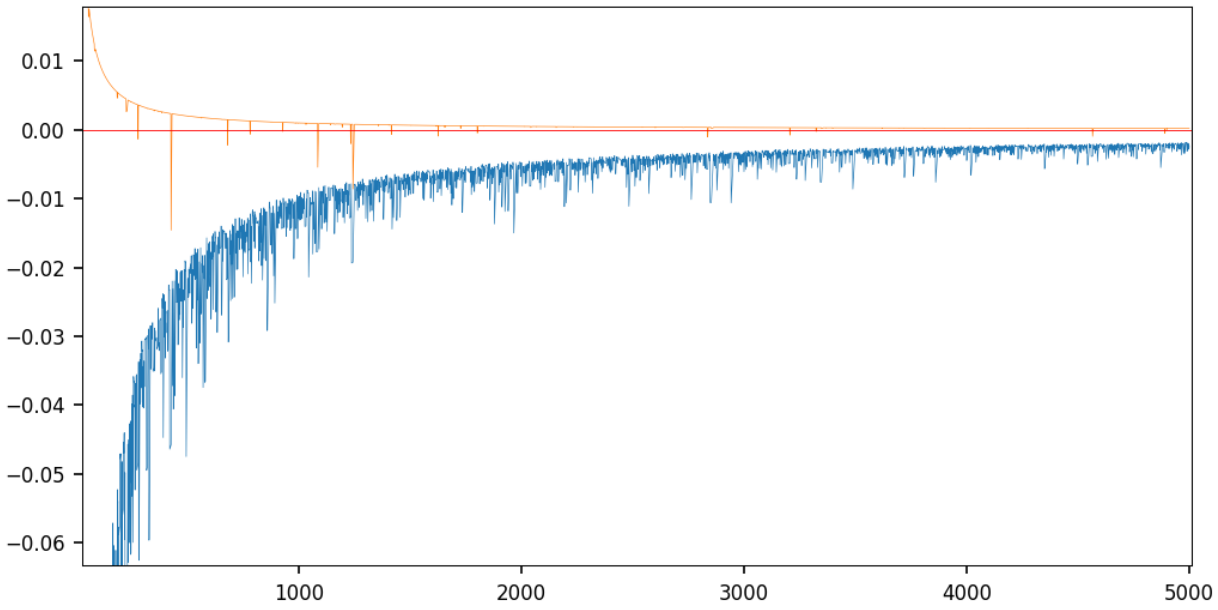


Figure 5.1: Upper and lower bounds for $\lambda_n(\tau)$ on Y axis, with n on X-axis, based on 100 values of τ

About truncation, the Python code below, despite dealing with irrational numbers, manipulates integer numbers only when `mode` is set to `'arccos'`. Yet these numbers become insanely large when n grows to 5000 and we compute A^m with $m = \tau 2^n$ and $\tau = 100$. Thus truncation is unavoidable. Yet, how do we make sure that we still preserve at least 12 digits of accuracy until the very end? In chapter 1, there is a theoretical framework that guarantees the desired precision. But not here. However, the determinant, always equal to 1, plays the role of a checksum. When the precision drops below a certain threshold, it shows in the digits of the determinant. Likewise, when testing with `mode='direct'`, $x_0 = 3/7$ and $\tau = 7$, we must have $T_m(y_0) = 1$ at

all times. When this is no longer true, it means that we have precision issues and must increase the precision parameter `ctx.precision` in the code. By default, its value is 10^4 bits at the starting point ($n = 0$).

The program below named `normal_numbers5.py` is available on my Google drive, here. The rational y_0 corresponds to `y0_1` in the code. As a rule of thumb, you start with full precision in the inner loop when $n = 0$, and you lose about one digit at each iteration as n increases. This is consistent with the similar algorithm in chapter 1. So if the precision is set to 600 bits and `n_max` is set to 400, in the last iteration the precision on $T_m(y_0)$ is about $600 - 400 = 200$ bits. Also, $T_m(y_0)$ is denoted as `T_frac` in the code, with $m = \tau 2^n$ as usual.

```
import numpy as np
import gmpy2

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['legend.fontsize'] = 'x-small'

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

mode = 'arccos' # options: 'direct', 'arccos'
y0_0 = gmpy2.mpfr(1)
if mode == 'direct':
   # x0 must be in ]-1, 1[
   # if x0 = a/b rational and tau=b, lim=0 always unless precision error
   x0 = gmpy2.mpfr(3)/gmpy2.mpfr(7) ## use trancendental number
   pi = gmpy2.const_pi()
   y0_1 = gmpy2.cos(2*pi*x0)
else:
   # 0 < p < q, both integers
   p = gmpy2.mpz(3)
   q = gmpy2.mpz(5)
   qn = gmpy2.mpz(q)
   y0_1 = gmpy2.mpfr(p/q)

A0 = [[gmpy2.mpfr(2*y0_1), gmpy2.mpfr(-1)],
     [gmpy2.mpfr(1), gmpy2.mpfr(0)]]
y0 = [y0_0, y0_1]

n_max = 5000
tau_max = 100
arr_lim = np.zeros((n_max, tau_max))
for tau in range(1,tau_max):
   A = np.linalg.matrix_power(A0, tau)
   lim = 0
   for n in range(0, n_max):
      T = np.matmul(A, y0)
      T_frac = T[0]
      # determinant must always be 1, used as checksum
      det = A[0,0]*A[1,1] - A[0,1]*A[1,0]
      if n > 0:
         #lim = (1 - T_frac)**(1/n)
         lim = gmpy2.log2(1 - T_frac)/n
      arr_lim[n, tau] = lim
      if n % 100 == 0:
         print("n: %5d tau: %3d T_frac: %12.9f lim %12.9f det %12.9f"
               % (n, tau, T_frac, lim, det))
      A = np.matmul(A, A)

xval = []
arr_min = []
arr_max = []
for n in range(n_max):
   tmin = 999999999.99
   tmax = -999999999.99
   for tau in range(1,tau_max):
      lim = arr_lim[n,tau]
      if lim < tmin:
         tmin = lim
      if lim > tmax:
         tmax = lim
   xval.append(n)
```

```
    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:

$$p_{n+1}^* = (p_n + q_n)^2 + \delta_n, \tag{5.12}$$

$$q_{n+1} = 2^\mu \cdot q_n^2, \tag{5.13}$$

$$p_{n+1} = \varphi(p_{n+1}^*, q_n, \nu), \tag{5.14}$$

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:

$$r_{3n+1} \to x_1 = 0.4967593019587711799534532381218225582952600510550467421351 28\ldots \tag{5.16}$$
$$r_{3n+2} \to x_2 = 0.2800360510000134955214242422455185478475023210296033045544 07\ldots \tag{5.17}$$
$$r_{3n} \to x_3 = 0.4096230729649272876269750365153427418450310723487637374203 30\ldots \tag{5.18}$$

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, 15, 47] 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	01010100000000000000	11111111101010101010	5	0.26	1	3	5	7	
19	4	max_00	01010110000000000000	10101010101000000000	13	0.68	1	3	5	6	
19	4	min_01	01010110000000000000	10101010101000000000	6	0.32	1	3	5	6	
19	4	max_01	01010100000000000000	11111111101010101010	14	0.74	1	3	5	7	
19	4	min_10	11111111111101010101	10101010100111111111	6	0.32	12	14	16	18	
19	4	max_10	01010101111111111111	10101010100000000000	14	0.74	0	2	4	6	
19	4	min_11	01010101111111111111	10101010100000000000	5	0.26	0	2	4	6	
19	4	max_11	11111111111101010101	10101010100111111111	13	0.68	12	14	16	18	
18	4	min_00	01010101000000000000	11111111101010101010	5	0.28	1	3	5	7	
18	4	max_00	01010110000000000000	10101010100000000000	13	0.72	1	3	5	6	
18	4	min_01	01010110000000000000	10101010100000000000	5	0.28	1	3	5	6	
18	4	max_01	01010101000000000000	11111111101010101010	13	0.72	1	3	5	7	
18	4	min_10	11111111101010101	10101010011111111111	5	0.28	10	12	14	16	
18	4	max_10	01010101111111111111	10101010010000000000	13	0.72	0	2	4	6	
18	4	min_11	01010101111111111111	10101010010000000000	5	0.28	0	2	4	6	
18	4	max_11	11111111101010101	10101010011111111111	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. ∎

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 $1010110000010011000001 0_2$ 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
```

```
      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:

$$x_n = \{bx_{n-1}\} = \{b^n x_0\}, \quad d_n = \lfloor bx_n \rfloor \tag{5.28}$$

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 [15]. 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 section 5.5.2.

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

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

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 [15] 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.

5.5.1 Autocorrelations in related sequences

The sequence $x_n = \{\beta + x_{n-1}\} = \{\beta n + x_0\}$ where $\beta > 0$ is an irrational number, behaves very differently from that defined by (5.28). Again, the n-th "digit" in base b is defined as $d_n = \lfloor bx_n \rfloor$. Now the invariant measure is unique and applies to all x_0. Again, it is the uniform distribution on $[0, 1]$, but the k-lag autocorrelations are much stronger and do not decay as k increases. Likewise, the digits have strong long-range autocorrelations, a big contrast with (5.28) where they are uncorrelated. This generalizes to the sequence $x_n = \{\beta n^\theta\}$. For details, see [29].

5.5.2 Python code

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)
```

```
43
44
45  def vg_correl(z, p, q):
46
47      # correl b/w binary digits of pz and qz
48      mode = 'constant' # options: 'random', 'constant'
49
50      # local variables
51      zstri = gmpy2.digits(z, 2)[0] # get binary digits of z as a string
52      X, pX, qX = 0, 0, 0
53      d1, d2, e1, e2 = 0, 0, 0, 0
54      prod, count = 0, 0
55      sum1 = 0
56      sum2 = 0
57
58      # loop over digits in reverse order
59      for k in range(kmax):
60
61          # b is a digit of X
62          if mode == 'random':
63              b = np.random.randint(0, 2)
64          else:
65              b = int(zstri[kmax-k-1])
66          X = b + X/2
67
68          c1 = p*b
69          old_d1 = d1
70          old_e1 = e1
71          d1 = (c1 + old_e1//2) %2 # digit of pX
72          e1 = (old_e1//2) + c1 - d1
73          pX = d1 + pX/2
74
75          c2 = q*b
76          old_d2 = d2
77          old_e2 = e2
78          d2 = (c2 + old_e2//2) %2 #digit of qX
79          e2 = (old_e2//2) + c2 - d2
80          qX = d2 + qX/2
81
82          prod += d1*d2
83          count += 1
84          sum1 += d1
85          sum2 += d2
86          mean1 = sum1/count
87          mean2 = sum2/count
88          std1 = (mean1 * (1 - mean1))**0.5
89          std2 = (mean2 * (1 - mean2))**0.5
90          covar = prod/count - mean1*mean2
91          if count > 100:
92              correl = covar/(std1*std2)
93          else:
94              correl = 0
95          #correl = 4*prod/count - 1
96
97          if k% 100000 == 0:
98              checksum = p*q*correl # should be close to 1
99              print("vg>k = %7d, correl = %9.6f checksum = %9.6f" % (k, correl, checksum))
100
101     print("\np = %3d, q = %3d" %(p, q))
102     print("X = %12.9f, pX = %12.9f, qX = %12.9f" % (X, pX, qX))
103     print("X = %12.9f, p*X = %12.9f, q*X = %12.9f" % (X, p*X, q*X))
104     print("Correl = %7.4f, 1/(p*q) = %7.4f" % (correl, 1/(p*q)))
105     return(correl)
106
107 #--- Main
108
109 correl1 = gmpy2_correl(z, p, q)
110 correl2 = vg_correl(z, p, q)
```

Chapter 6

Quantum States and the Riemann Zeta Function

From the beginning, the Riemann Hypothesis has been strongly connected to prime numbers. Both are deeply intertwined, and all attempts to prove this famous conjecture – still unsolved today – heavily rely on analytic number theory where primes play a key role. But what if the connection was indirect, with even a stronger link to synthetic numbers that are not even integers? Could the Riemann Hypothesis be explained by something other than prime numbers? It would open new opportunities to solve this problem, based on generated numbers with the appropriate distribution, possibly moving the discussion from number theory to standard calculus and real analysis. This is the topic of section 6.1.

This chapter discusses state-of-the-art research about the Riemann zeta function, with a very innovative perspective on the subject. Finally, I discuss the connection to quantum systems and sub-quantum states.

6.1 Synthetic primes, quantum states, and the Riemann Hypothesis

Using synthetic numbers – random ones, not even integers – I build a mathematical function with the exact same roots as the Riemann zeta function, on the critical line [Wiki]. This is at the core of the Riemann Hypothesis. It challenges the myth that the theory is regulated by prime numbers. In other words, you can do a deep dive on the topic, maybe even prove the conjecture, without ever discussing prime numbers. My arbitrary numbers form a synthetic multiplicative semigroup [Wiki] and have a specific distribution and properties. Yet, they do not depend on the exact values of the standard prime numbers.

The material presented here is accessible to readers with a modest mathematical background. Knowledge of advanced topics such as analytic continuation, Dirichlet series, or complex number theory, are not needed to gain a deep understanding of the mechanics behind the scenes.

6.1.1 Definitions

Let $\mathscr{P} = \{p_1, p_2, p_3 \dots\}$ be a set of strictly increasing real numbers called Beurling primes, with $p_1 = 2$. Also, let $\log p_1, \log p_2$ and so on be linearly independent over the set of rational numbers. Let $S_{\mathscr{P}}$ be the set of all product combinations

$$\omega = p_1^{a_1} p_2^{a_2} p_3^{a_3} \cdots$$

where $p_1, p_2, \cdots \in \mathscr{P}$ and a_1, a_2 and so on are positive integers, with all but a finite number of them equal to zero. I can now define the Dirichlet eta function [Wiki] attached to $\mathscr{P}$ as

$$\eta_{\mathscr{P}}(s) = \sum_{\omega \in S_{\mathscr{P}}} \delta(\omega) \cdot \frac{1}{\omega^s} = \delta(\omega_1) \cdot \frac{1}{\omega_1^s} + \delta(\omega_2) \cdot \frac{1}{\omega_2^s} + \dots \tag{6.1}$$

where $s = \sigma + it$ is a complex number. Here, $\delta(\omega) \in \{-1, +1\}$. In practice, the series (6.1) is not absolument convergent when $\sigma \leq 1$. Thus the ordering of the terms is critical. I define the order and the δ function later. For now, note that we always have $\omega_1 = 1, \omega_2 = 2, \delta(\omega_1) = 1$ and $\delta(\omega_2) = -1$. From there, one can define the associated Riemann zeta function as follows:

$$\zeta_{\mathscr{P}}(s) = \frac{\eta_{\mathscr{P}}(s)}{1 - 2^{1-s}} = \sum_{w \in S_{\mathscr{P}}} \frac{1}{\omega^s} = \prod_{p \in \mathscr{P}} \frac{1}{1 - p^{-s}} \tag{6.2}$$

where typically, both the series and Euler product [Wiki] in (6.2) diverge when $\sigma \leq 1$.

6.1.2 Building a Beurling eta function by deletion

This is the easiest case. First, you pick up primes numbers $p_{k_1}, p_{k_2}, p_{k_3}$ and so on from the set of standard odd primes $3, 5, 7, 11$ and so on. We must keep $p_1 = 2$ which is responsible for the negative terms in the resulting series (6.1); without it, all terms would be positive and the series would diverge. Then, from the standard eta function

$$\eta(s) = \sum_{k=1}^{\infty} (-1)^{k+1} \cdot \frac{1}{k^s}, \tag{6.3}$$

you remove all terms k^{-s} where k is a multiple of at least one of the p_{k_i}'s. The resulting set $\mathscr{P}$ contains all the standard primes except p_{k_1}, p_{k_2} and so on. Also, $\delta(w)$ in the resulting series (6.1) is equal to -1 if w is even, and to $+1$ otherwise. We have

$$\eta_{\mathscr{P}}(s) = \lambda_{\mathscr{P}} \cdot \eta(s), \quad \text{with } \lambda_{\mathscr{P}} = \left[\prod_i \frac{1}{1 - p_{k_i}^{-s}} \right]^{-1}. \tag{6.4}$$

One would expect that the function $\eta_{\mathscr{P}}$ has the same non-trivial roots as the standard Riemann zeta function, provided the following condition is met:

$$\sum_i \frac{1}{p_{k_i}^s} < \infty. \tag{6.5}$$

In short, condition (6.5) states that it works even if you remove infinitely mamy primes, as long as you don't remove too many of them.

6.1.3 Building a Beurling eta function by swapping

For simplicity, let us assume that we replace each standard odd prime p_k by a strictly positive real number q_k such that $\log q_1, \log q_2$ and so on are linearly independent on the set of rational numbers. Some of the q_k's (a few, none, or infinitely many) can be identical to the corresponding p_k's. You perform the swapping directly in series (6.3) to obtain (6.1), by replacing each k^{-s} in (6.3) with

$$\omega_k = \left(k \cdot \frac{q_1^{a_1} q_2^{a_2} \cdots}{p_1^{a_1} p_2^{a_2} \cdots} \right)^{-s}$$

where a_1, a_2 and so on are the 2-adic valuations of k respectively for p_1, p_2 and so on. Again, $q_1 = p_1 = 2$ is unchanged, to guarantee convergence. The p-adic valuation of k for a prime p is the largest integer a such that p^a divides k. If k is not a multitple of p, then $a = 0$. The resulting set $\mathscr{P}$ consists of q_1, q_2 and so on while $S_{\mathscr{P}}$ contains the ω_k's. Now we have

$$\eta_{\mathscr{P}}(s) = \lambda_{\mathscr{P}} \cdot \eta(s), \quad \text{with } \lambda_{\mathscr{P}} = \prod_k \frac{1 - p_k^{-s}}{1 - q_k^{-s}} = \frac{\zeta_{\mathscr{P}}(s)}{\zeta(s)}. \tag{6.6}$$

This time, we need $0 < \lambda_{\mathscr{P}} < \infty$ for (6.6) to be valid, and to make sure that $\eta_{\mathscr{P}}$ has the same non-trivial roots as the standard Riemann zeta function. In particular, we want this to be true when $\sigma = \Re(s) = \frac{1}{2}$. Finally, this is only a brief sketch explaining the mechanism at play. In particular, the ratio of the two zeta functions on the right-hand side may be $0/0$ (undetermined) depending on s. However, there is less ambiguity if only a finite number of q_k are different from the corresponding p_k.

Also, the Dirichlet series obtained by changing the terms in (6.3) as per the algorithm discussed, switching from $\eta(s)$ to $\eta_{\mathscr{P}}(s)$ without re-ordering, must converge at the target value s. Otherwise, the results may not be valid. The proof of convergence may not be obvious in all cases.

More broadly, two Beurling eta functions, one based on a set $\mathscr{P}$ and the other one on a set $\mathscr{P}'$, have the same non-trivial roots if the ratio

$$\lambda_{\mathscr{P}|\mathscr{P}'} = \prod_k \frac{1 - {q'_k}^{-s}}{1 - q_k^{-s}} = \frac{\zeta_{\mathscr{P}}(s)}{\zeta_{\mathscr{P}'}(s)}, \quad q_k \in \mathscr{P},\ q'_k \in \mathscr{P}'$$

satisfies $0 < \lambda_{\mathscr{P}|\mathscr{P}'} < \infty$ or equivalently, $0 < \lambda_{\mathscr{P}'|\mathscr{P}} < \infty$.

I provide illustrations in sections 6.1.4 and B.1.1. In particular, Figure 6.2 shows that $s_3 = \frac{1}{2} + 25.01085758\, i$, the 3rd zero of ζ, is also a zero of $\eta_{\mathscr{P}}$ when $\mathscr{P}$ consists of all the primes except $3, 5, 7$ (see section 6.1.2). This is true for all zeros of ζ on the critical line. Figure 6.1 shows that replacing primes by other numbers (see section 6.1.3) also preserves the zeros without adding new ones. In this example, primes have been modified as follows: $3 \mapsto 4.051$, $5 \mapsto 7.916$, $7 \mapsto 9.114$. See Part 3 of the code in section 6.1.4.

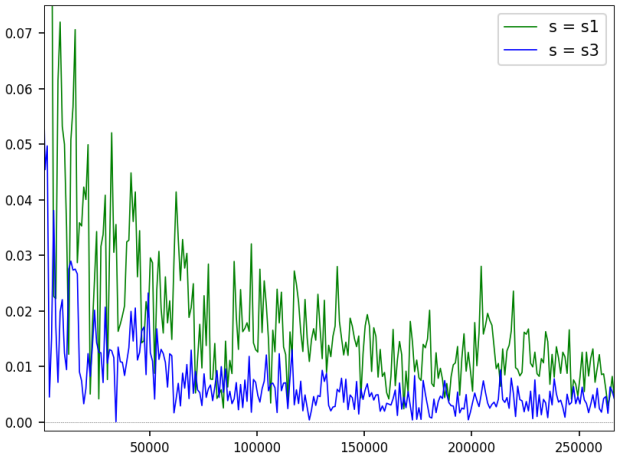


Figure 6.1: $||\eta_{\mathscr{P}}(s)|| \to 0$ as $n \to \infty$; s_1, s_3 are 1st and 3rd roots of ζ, with n on X axis and special $\mathscr{P}$.

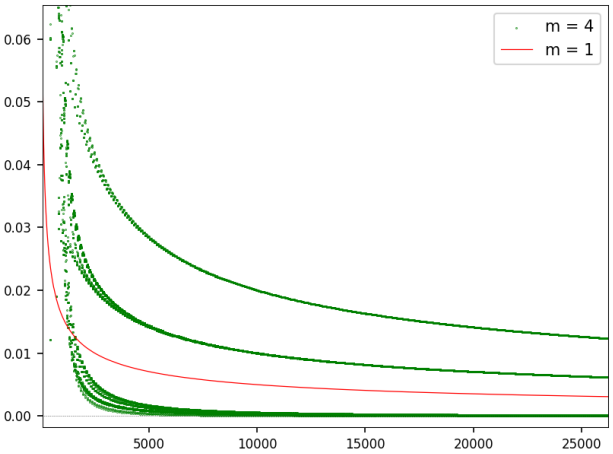


Figure 6.2: Same as figure 6.1, with a different $\mathscr{P}$ and $s = s_3$. The red curve is the standard $\eta(s_3)$.

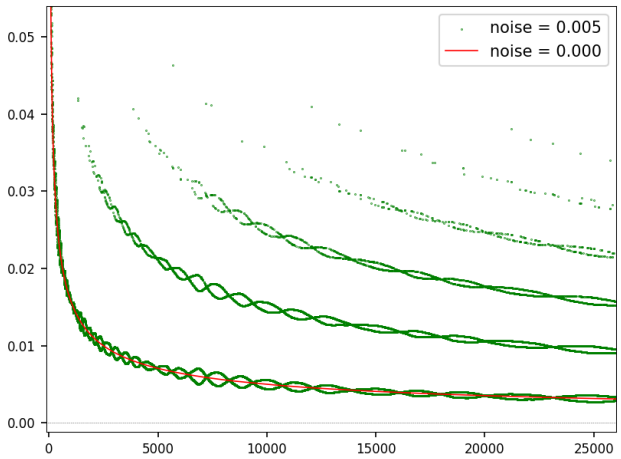


Figure 6.3: $||\eta_{\mathscr{P}}(s_3)|| \to 0$ as $n \to \infty$; p_k is replaced in $\mathscr{P}$ by $q_k = p_k + \lambda_k \cdot k^{1/4}$ if $k > 1$

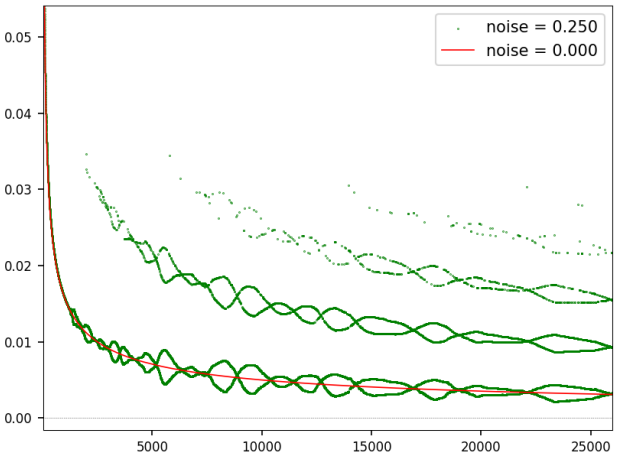


Figure 6.4: Same as Figure 6.3 but with much larger λ_k and $q_k = p_k$ if $k < 200$

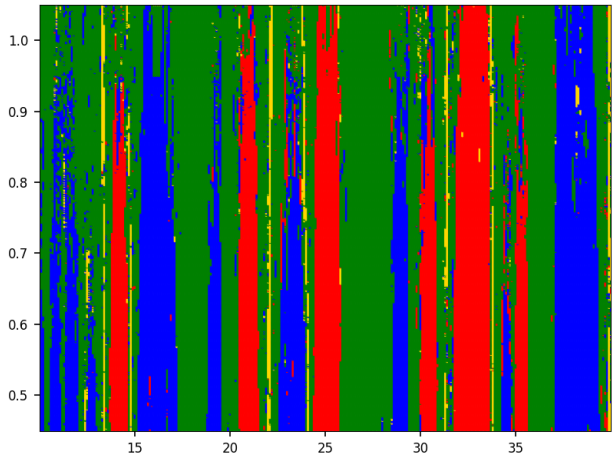


Figure 6.5: $\eta_{\mathscr{P}}(\sigma, t)$ basins of attraction, $10 < t < 40$ (X axis) and $0.45 < \sigma < 1.05$ (Y axis)

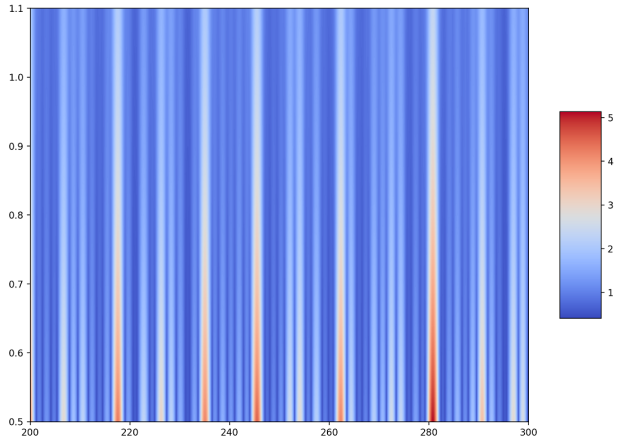


Figure 6.6: $||\zeta(\sigma, t)||$ with $200 < t < 300$ (X axis) and $0.5 < \sigma < 1.1$ (Y axis)

6.1.4 Applications and Python code

Here I discuss applications of (6.1). For a complex-valued function $\eta(s) = f(s) + ig(s)$ with complex argument s, the the notation $||\eta(s)|| = f^2(s) + g^2(s)$ stands for the norm evaluated at s. It is equal to zero if and only if

$\eta(s) = 0$, that is, if s is a root (also called zero) of η. I use the notation $s = \sigma + it$ where σ, t are respectively the real and imaginary parts of s. Also, $s_k = \sigma_k + it_k$ denotes the k-th zero of the Riemann zeta function $\zeta(s)$ with $\sigma_k = \frac{1}{2}$ and $t_k > 0$, for $k = 1, 2$ and so on. The red curve in Figures 6.2, 6.3 and 6.4 represents the standard Dirichlet eta function $\eta(s)$ evaluated at $s = s_3$ using the first n terms in series (6.3), with n on the X axis. The green and blue curves represent transformed versions of η, denoted as $\eta_{\mathscr{P}}(s)$, also supposed to have the exact same roots as $\eta(s)$ or $\zeta(s)$ at $\sigma = \frac{1}{2}$. In particular:

- In Figure 6.1, the set $\mathscr{P}$ consists of the standard primes $2, 3, 5, 7, 11$ and so on except that I replaced $p_2 = 3$ by $q_2 = 4.051$, $p_3 = 5$ by $q_3 = 7.916$, and $p_4 = 7$ by $q_4 = 9.114$. The green and blue curves correspond to $||\eta_{\mathscr{P}}(s)||$ evaluated respectively at $s = s_1$ and $s = s_3$, using the first n terms in (6.1) with n on the X axis.
- In Figure 6.2, for the green curve, the set $\mathscr{P}$ consists of all the standard primes, expect that I removed $p_2 = 3$, $p_3 = 5$ and $p_4 = 7$. Note that as n increases by one unit, $||\eta_{\mathscr{P}}(s_3)||$ (its approximation based on the first n terms) jumps from one level to another. At a given n, the current level is called the quantum state and depends on n mod 3. There are sub-levels that you can only see by zooming in, called sub-quantum states. At the lowest level (closest to the X axis), as n increases, $||\eta_{\mathscr{P}}(s_3)||$ converges to zero faster than the red curve $||\eta(s_3)||$. The word quantum convergence makes sense in this context.
- In Figure 6.3, for the green curve, $\mathscr{P} = \{q_1, q_2, \dots\}$ is a brand new set of random real numbers with $q_1 = 2$ and $q_k = p_k + \lambda_k \cdot k^{\alpha}$ for $k > 1$, where p_2, p_3 and so on are the standard primes and $\alpha = \frac{1}{4}$. Here, the λ_k's are independent uniform deviates on $[-\tau, \tau]$ with $\tau = 0.005$. The parameter τ is called the *noise*. This time, we have infinitely many quantum states though we only see the first few ones. We still have convergence of $||\eta_{\mathscr{P}}(s_3)||$ to zero as n increases.
- Figure 6.4 is identical to Figure 6.3 except that the noise threshold is $\tau = 0.25$ and $q_k = p_k$ if $k < 200$. The latter is accomplished by setting `start=200` in the code.

Convergence of the green curve $||\eta_{\mathscr{P}}(s_3)||$ to zero as n increases in Figures 6.3 and 6.4 is not a trivial problem. It seems like it requires $0 < \lambda_{\mathscr{P}} < \infty$ where $\lambda_{\mathscr{P}}$ is defined in formula (6.6). If this is correct, then we would have convergence when the following condition is met at $s = s_3 = \frac{1}{2} + it_3$:

$$\left| \frac{1}{q_k^s} - \frac{1}{p_k^s} \right| = O\Big(\frac{1}{k^{\beta}}\Big) \text{ as } k \to \infty, \tag{6.7}$$

for some constant $\beta > 1$. Here p_k is the k-th standard prime, thus $p_k \sim k \log k$. Also $q_k = p_k + \lambda_k k^{\alpha}$ with $|\lambda_k|$ bounded. Thus (6.7) is satisfied if $\alpha < \frac{1}{2}$, which is the case both in Figure 6.3 and 6.4.

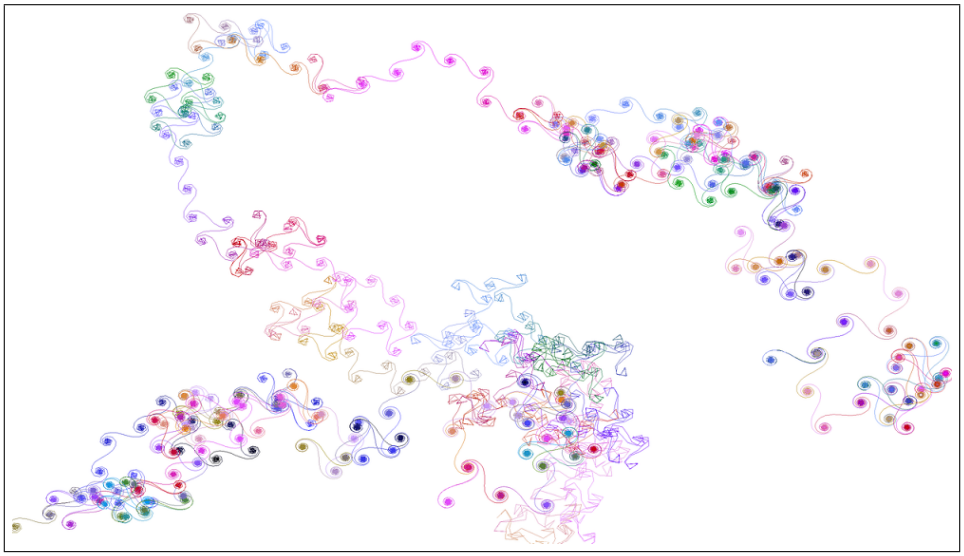


Figure 6.7: Typical convergence path for series (6.3) representing $\eta(s)$ in the complex plane, when the imaginary part t of $s = \sigma + it$ is large

This leads to an interesting question. Let $f(k)$ be a real-valued function defined on the integers $k > 0$ with $f(1) = 0$. Define $q_k = p_k + f(k)$ and $\mathscr{P} = \{q_1, q_2, \cdots\}$. Again, p_1, p_2, p_3 and so on are the standard primes. Is there a function f such that the q_k are chaos-free (unlike the standard primes) with $\eta_{\mathscr{P}}(s)$ and $\eta(s)$ having the

same zeros? By chaos-free, I mean $q_{k+1} - q_k$ smoothly increasing when k is large enough, rather than erratically going up and down infinitely many times as for $p_{k+1} - p_k$. Of course, f would have to be chaotic to remove the chaos in the standard primes. Cramér's conjecture [Wiki] states that $p_{k+1} - p_k < (\log k)^2$ for k large enough. On the other end, $f(k) = k^\alpha$ preserves the roots of η if $\alpha < \frac{1}{2}$. However we need a bigger α to counteract the largest and smallest prime gaps, thus the question remains open.

Another way to somewhat reduce the chaos in standard primes while preserving the roots of η is to choose $q_k = (p_{k-1} + p_k + p_{k+1})/3$ for $k > 200$, with $q_k = p_k$ for $k \leq 200$. The modification can be implemented in the function `build_swap_primes` in the code, along with setting `start=200`. In the code p_k is represented by `arr_primes[k]`. The resulting $||\eta_{\mathscr{P}}(s_3)||$ convergence path to zero is similar to that in Figure 6.4.

The convergence of series (6.1) representing $\eta_{\mathscr{P}}(s)$ is not a trivial topic, as shown by the green curve in Figure 6.4. It seems that the convergence of (6.3) attached to the standard eta function $\eta(s)$ is easier to establish, if you look at the red curve in the same figure. However, this is true only for values of $s = \sigma + it$ where t is small. When t is large and $0 < \sigma < 1$, the red curve also becomes quite chaotic: see Figure 6.8. This is also illustrated in Figure 6.7 where the path starts with the first term in (6.3), and each move corresponds to adding one more term in the series until convergence to $\eta(s)$. See video here showing the path evolving to $\eta(s)$ as the number n of computed terms increases, starting with two different values of s in parallel. Figure 6.7 is described in section 3.4.2 in my book on chaotic systems [15], along with the code to produce the video.

For convergence acceleration techniques applied to this example, see section B.1 and exercise 25 in chapter 5 in [13]. Using synthetic numbers to generate $\eta_{\mathscr{P}}$ functions with desirable properties, is further discussed in chapter 17 in [14].

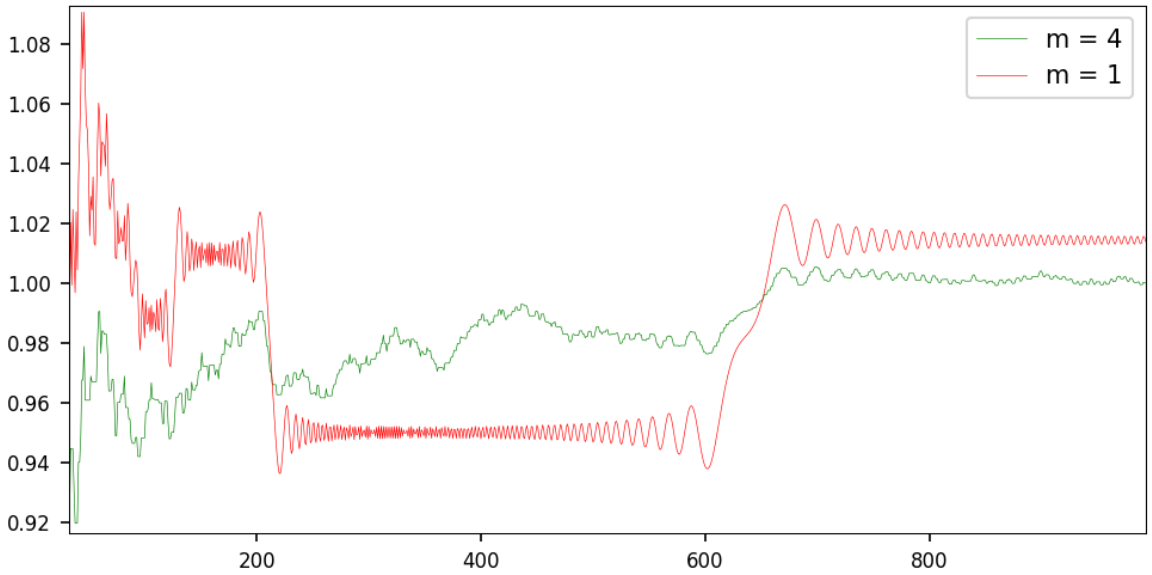


Figure 6.8: Same as Figure 6.2 with $\eta_{\mathscr{P}}(s)$ in green and $\eta(s)$ in red evaluated at $s = 0.95 + it$ with $t = 2000$, rather than at $s = s_3$. The large t causes the jumps in $\eta(s)$ as n increases on the X-axis.

The link to quantum states is further discussed in section 6.2, where I introduce quantum derivatives and remarkable approximations to $\eta(s)$. For articles about Beurling primes, see [7, 40] and chapter 17 in [14] for additional references. Before displaying the Python code, I conclude with two interesting problems:

- **Reordering the terms** in series (6.1). When turning standard primes p_k into new numbers q_k, the terms in the resulting series may no longer be decreasing in absolute value if you keep the original ordering (alternating between positive and negative terms). You may re-order by absolute value – assuming it does not alter the value of $\eta_{\mathscr{P}}(s)$ – but then the series may no longer be alternating: see example (B.3). In the code, this is accomplished with the setting `mode='sorted'`. Still, you should keep the same ratio of positive and negative terms. Question: can you find a set $\mathscr{P}$ such that the sequence of term signs (the + and -) looks random like Bernoulli trials, after the re-ordering in question?

- **Basins of attractions**. In Figure 6.5, each pixel represents a complex number $s_* = \sigma + it$, with σ on the Y-axis and t on the X-axis. For each s_*, I solve $\eta_{\mathscr{P}}(s) = 0$ using an iterative algorithm that starts with $s = s_*$ as the initial guess for the root. Also, I use an approximation of $\eta_{\mathscr{P}}$ based on the first n terms in its series (6.1). Given s_*, the root found is denoted as $g_n(\sigma_*)$. When $n = \infty$, the roots have real part equal either to $\frac{1}{2}$ (red dot) or 1 (green dot), according to the Riemann Hypothesis. Blue dots correspond to other roots, while the yellow color indicates lack of convergence to a solution. As n grows, the areas in

blue shrink to an empty set. Each color determines a basin of attraction in the root system. Such basins are studied in the context of chaotic dynamical systems [15].

Conclusion: even if starting with s_* far away from the critical line $\sigma = \frac{1}{2}$, you can retrieve the roots on that line. Figure 6.6 confirms this fact: even on the line $\sigma = 1$, you can "feel" the existence of a root at $\sigma = \frac{1}{2}$ by looking at $||\zeta(1+it)||$, which exhibit minima at values of t such that $\zeta(\frac{1}{2}+it) = 0$. For details, see section 1.2.2 in [18]. The computation of the roots is done in section 4 in the code below, using `fsolve` from the Scipy library.

The code below is also on GitHub, here. The `pre_compute` function reduces run time. For $s = \sigma + it$ with a very large t, you can improve accuracy by replacing the computation of the `vexp` array in the `eta` and `eta2` functions, with `vexp=np.exp(np.mod(t*alog[1:n],2*np.pi)*1j)`. Since you only need the cosine and sine of the angles in the `t*alog[1:n]` array, taking the values element-wise modulo 2π does not changes the result yet increases accuracy if t is very large.

In Figure 6.8, $\mathscr{P}$ is the set of standard primes without $3, 5, 7$. The green curve represents $||\eta_{\mathscr{P}}(s)||/||\lambda_{\mathscr{P}}||$ based on the Dirichlet series evaluated at $s = 0.95 + 2000\,i$ and truncated to n terms, with n on the X-axis. The normalization by the factor $1/||\lambda_{\mathscr{P}}||$ guarantees that both curves converge to the same value when $n \to \infty$. As per (6.4), $\lambda_{\mathscr{P}} = (1-3^{-s})(1-5^{-s})(1-7^{-s})$. The limit is $||\eta(s)|| = 1.0144236\ldots$ when $n = \infty$, confirmed here by Wolfram. Note that n needs to be much larger than 1000 to see both curves converge to that value.

```
from scipy.optimize import fsolve
import numpy as np

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

#--- 1. Precomputations

def pre_compute(swap_primes, del_primes, N, mode):

   p_alog = np.zeros(N)
   p_arr_omega = np.zeros(N)
   p_arr_delta = np.zeros(N)
   flag = 1

   for k in range(1, N):

      if k % 100 == 0:
         print("pre_compute:", k)
      hash_pval = {}
      new_k = k
      for p in swap_primes:
         pval = 0
         while new_k % p == 0:
            new_k = new_k // p
            pval += 1
         hash_pval[p] = pval

      for p in hash_pval:
         new_k *= (swap_primes[p])**hash_pval[p]

      p_alog[k] = np.log(new_k)
      p_arr_omega[k] = new_k
      p_arr_delta[k] = flag
      flag = -flag

      for p in del_primes:
         if k % p == 0:
            p_arr_delta[k] = 0

   if mode == 'sorted':
      ranks = np.abs(p_arr_omega).argsort().argsort()
      alog = np.zeros(N)
      arr_omega = np.zeros(N)
      arr_delta = np.zeros(N)
      for k in range(len(arr_omega)):
         rank = ranks[k]
         alog[rank] = p_alog[k]
```

```
            arr_omega[rank] = p_arr_omega[k]
            arr_delta[rank] = p_arr_delta[k]
        return(alog, arr_omega, arr_delta)
    else:
        return(p_alog, p_arr_omega, p_arr_delta)


def eta2(s, params):

    # t is imaginary part
    sigma = s[0]
    t = s[1]
    n = params[0]
    alog = params[1]
    arr_omega = params[2]
    arr_delta = params[3]
    vexp = np.exp(t*alog[1:n]*1j)
    vpow = arr_delta[1:n] * arr_omega[1:n]**(-sigma)
    sum = np.dot(vexp, vpow)
    norm = np.linalg.norm(sum)
    return(norm)

#--- 2. Removing primes in eta function

sigma = 0.5
t = 25.010858 # 0.5 + it is 3rd root of zeta
s = [sigma, t]
n0 = 4
n1 = 3500
N = 500001
swap_primes = {}
mode = 'unsorted' # options: 'sorted', 'unsorted'

del_primes1 = (3, 5, 7)
m1 = len(del_primes1) + 1
alog, arr_omega, arr_delta = pre_compute(swap_primes, del_primes1, N, mode)
params1 = [0, alog, arr_omega, arr_delta]

del_primes2 = ()
m2 = len(del_primes2) + 1
alog, arr_omega, arr_delta = pre_compute(swap_primes, del_primes2, N, mode)
params2 = [0, alog, arr_omega, arr_delta]

arr_x = []
arr_y1 = []
arr_y2 = []
for n in range(n0, n1):
    params1[0] = n
    params2[0] = n
    arr_x.append(n-1)
    arr_y1.append(eta2(s, params1))
    arr_y2.append(eta2(s, params2))

plt.scatter(arr_x, arr_y1, c='green', s = 0.1, label="m = %d" %m1)
plt.plot(arr_x, arr_y2, c='red', linewidth = 0.6, label="m = %d" %m2)
plt.axhline(y=0.0, color='black', linestyle='dotted', linewidth = 0.3)
plt.legend(loc="upper right")
plt.show()

#--- 3. Changing primes in eta function

def eta(s, params):
    # s[0] = real part, s[1] = imaginary part
    n = params[0]
    alog = params[1]
    arr_omega = params[2]
    arr_delta = params[3]
    vexp = np.exp(s[1]*alog[1:n]*1j)
    vpow = arr_delta[1:n] * arr_omega[1:n]**(-s[0])
    sum = np.dot(vexp, vpow)
    return [sum.real, sum.imag]

swap_primes = {
              3: 4.051,
              5: 7.916,
              7: 9.114
```

```
            }
del_primes = ()
mode = 'unsorted' # options: 'sorted', 'unsorted'
alog, arr_omega, arr_delta = pre_compute(swap_primes, del_primes, N, mode)
arr_x = []
arr_y = []
arr_z = []

sigma = 0.5
t1 = 14.13472514
t3 = 25.01085758
s1 = [sigma, t1]
s3 = [sigma, t3]
params = [0, alog, arr_omega, arr_delta]
for n in range(400, 300000, 1000):
   arr_x.append(n-1)
   params[0] = n
   arr_y.append(eta2(s1, params))
   arr_z.append(eta2(s3, params))

plt.plot(arr_x, arr_y, c='green', linewidth = 0.8, label="s = s1")
plt.plot(arr_x, arr_z, c='blue', linewidth = 0.8, label="s = s3")
plt.axhline(y=0.0, color='black', linestyle='dotted', linewidth = 0.3)
plt.legend(loc="upper right")
plt.show()

#--- 4. Basins of attraction

areal = []
aimag = []
acolor = []
n = 499
found1 = 0
found2 = 0
nfc = 0
eps = 0.05

params = [n, alog, arr_omega, arr_delta]
for real in np.arange(0.45, 1.05, 0.001):
   print("Real: %7.4f" %(real))
   for imag in np.arange(10.00, 40.01, 0.1):
      init = np.array([real, imag])
      root = fsolve(eta, init, params)
      areal.append(real)
      aimag.append(imag)
      sigma = min(1, root[0])
      nfc += 1
      if 0 < abs(root[0] - 0.5) < eps:
         found1 += 1
         acolor.append('red')
      elif 0 < abs(root[0] - 1.0) < eps:
         found2 += 1
         acolor.append('green')
      elif 0 < root[0] < 1:
         acolor.append('blue')
      else:
         acolor.append('gold')

plt.scatter(aimag, areal, s = 0.2, c=acolor)
plt.show()
print()
print("Found: left: %5.3f right: %5.3f" % (found1/nfc, found2/nfc))

#--- 5. Replace all primes by random numbers

from sympy import primerange, isprime

def build_swap_primes(nprimes, start, noise):

   arr_primes = list(primerange(3, nprimes))
   swap_primes = {}
   seed = 503
   np.random.seed(seed)
   N_max = len(arr_primes)
   for k in range(start, N_max):
      prime = arr_primes[k]
```

```
        u = np.random.uniform(-noise, noise)
        swap_primes[prime] = prime + u*(k**0.25)
    print("N_max: ", N_max)
    return(swap_primes, N_max)

del_primes = ()
nprimes = 300000
sigma = 0.5
t = 25.010858 # 0.5 + it is 3rd root of zeta
s = [sigma, t]

noise1 = 0.005
start = 0
swap_primes1, N_max = build_swap_primes(nprimes, start, noise1)
mode = 'sorted' # options: 'sorted', 'unsorted'
N = N_max
n0 = 4
n1 = N_max

alog, arr_omega, arr_delta = pre_compute(swap_primes1, del_primes, N, mode)
params1 = [0, alog, arr_omega, arr_delta]

noise2 = 0.000
swap_primes2, N_max = build_swap_primes(nprimes, start, noise2)
alog, arr_omega, arr_delta = pre_compute(swap_primes2, del_primes, N, mode)
params2 = [0, alog, arr_omega, arr_delta]

arr_x = []
arr_y1 = []
arr_y2 = []
for n in range(n0, n1):
    params1[0] = n
    params2[0] = n
    arr_x.append(n)
    arr_y1.append(eta2(s, params1))
    arr_y2.append(eta2(s, params2))

plt.scatter(arr_x, arr_y1, c='green', label="noise = %5.3f" %noise1, s=0.1) ### linewidth = 0.3)
plt.plot(arr_x, arr_y2, c='red', label="noise = %5.3f" %noise2, linewidth = 0.8)
plt.axhline(y=0.0, color='black', linestyle='dotted', linewidth = 0.3)
plt.legend(loc="upper right")
plt.show()
```

6.2 Quantum derivatives, GenAI, and the Riemann Hypothesis

If you are wondering how close we are to proving the Generalized Riemann Hypothesis (GRH), you should read on. The purpose of this project is to uncover intriguing patterns in prime numbers, and gain new insights on the GRH. I stripped off all the unnecessary math, focusing on the depth and implications of the material discussed here. You will also learn to use the remarkable MPmath library for scientific computing. This is a cool project for people who love math, with the opportunity to work on state-of-the-art research even if you don't have a PhD in number theory.

Many of my discoveries were made possible thanks to pattern detection algorithms (in short, AI) before leading to actual proofs, or disproofs. This data-driven, bottom-up approach is known as experimental math. It contrasts with the top-down, classic theoretical academic framework. The potential of AI and its computing power should not be underestimated to make progress on the most difficult mathematical problems. It offers a big competitive advantage over professional mathematicians focusing on theory exclusively.

My approach is unusual as it is based on the Euler product. The benefit is that you immediately know when the target function, say the Riemann zeta function $\zeta(s)$, has a root or not, wherever the product converges. Also, these products represent analytic function [Wiki] wherever they converge.

I use the standard notation in the complex plane: $s = \sigma + it$, where σ, t are respectively the real and imaginary parts. I focus on the real part only (thus $t = 0$) because of the following result: if for some $s = \sigma_0$, the product converges, then it converges for all $s = \sigma + it$ with $\sigma > \sigma_0$. Now let's define the Euler product. The finite version with n factors is a function of s, namely

$$f(s,n) = \prod_{p \in P_n} \left(1 - \frac{\chi(p)}{p^s}\right)^{-1} = \prod_{k=1}^{n} \left(1 - \frac{\chi(p_k)}{p_k^s}\right)^{-1}.$$

Here $P_n = \{2, 3, 5, 7, 11, \dots\}$ is the set of the first n prime numbers, and p_k denotes the k-th prime with $p_1 = 2$. The function $\chi(p)$ can take on three vales only: $0, -1, +1$. This is not the most generic form, but the one that I will be working with in this section. More general versions are investigated in chapter 17, in [19]. Of course, we are interested in the case $n \to \infty$, where convergence becomes the critical issue. Three particular cases are:

- Rienmann zeta, denoted as $\zeta(s, n)$ or $\zeta(s)$ when $n = \infty$. In this case $\chi(p) = 1$ for all primes p. The resulting product converges only if $\sigma > 1$. Again, σ is the real part of s.
- Dirichlet L-function $L_4(s, n)$ [Wiki] with Dirichlet modular character $\chi = \chi_4$ [Wiki]. Denoted as $L_4(s)$ when $n = \infty$. Here $\chi_4(2) = 0$, $\chi_4(p) = 1$ if $p - 1$ is a multiple of 4, and $\chi_4(p) = -1$ otherwise. The product is absolutely convergent if $\sigma > 1$, but convergence status is unknown if $\frac{1}{2} < \sigma \leq 1$.
- Unnamed function $Q_2(s, n)$, denoted as $Q_2(s)$ when $n = \infty$. Here $\chi(2) = 0$. Otherwise, $\chi(p_k) = 1$ if k is even, and $\chi(p_k) = -1$ if k is odd. Again, p_k is the k-th prime with $p_1 = 2$. The product is absolutely convergent if $\sigma > 1$, and conditionally convergent [Wiki] if $\frac{1}{2} < \sigma \leq 1$.

All these products can be expanded into Dirichlet series [Wiki], and the corresponding χ expanded into multiplicative functions [Wiki] over all positive integers. Also, by construction, Euler products have no zero in their conditional and absolute convergence domains. Most mathematicians believe that the Euler product for $L_4(s)$ conditionally converges when $\frac{1}{2} < \sigma \leq 1$. Proving it would be a massive accomplishment. This would be make L_4 the first example of a function satisfying all the requirements of the Generalized Riemann Hypothesis. The Unnamed function Q_2 actually achieves this goal, with the exception that its associated χ is not periodic. Thus, Q_2 lacks some of the requirements. The Dirichlet series associated to Q_2 (the product expansion as a series) is known to convergence and thus equal to the product if $\sigma > \frac{1}{2}$.

The rest of the discussion is about building the framework to help solve this centuries-old problem. It can probably be generalized to L-functions other than L_4, with one notable exception: the Riemann function itself, which was the one that jump-started all this vast and beautiful mathematical theory.

6.2.1 Cornerstone result to bypass the roadblocks

The goal here is to prove that the Euler product $L_4(s, n)$ converges to some constant $c(s)$ as $n \to \infty$, for some $s = \sigma_0 + it$, with $t = 0$ and some $\sigma_0 < 1$. In turns, it implies that it converges at $s = \sigma + it$, for all t and for all $\sigma > \sigma_0$. It also implies that $c(s) = L_4(s)$, the true value obtained by analytic continuation [Wiki]. Finally, it implies that $L_4(s)$ has no zero if $\sigma > \sigma_0$. This would provide a partial solution to the Generalized Riemann Hypothesis, for L_4 rather than the Riemann zeta function $\zeta(s)$, and not with $\sigma_0 = \frac{1}{2}$ (the conjectured lower bound), but a least for some $\sigma_0 < 1$. This is enough to make countless mathematical theorems true, rather than "true conditionally on the fact that the Riemann Hypothesis is true". It also leads to much more precise results regarding the distribution of primes numbers: results that to this day, are only conjectures. The implications are numerous, well beyond number theory.

The chain of implications that I just mentioned, follows mainly from expanding the Euler product into a Dirichlet-L series. In this case, the expansion is as follows, with $s = \sigma + it$ as usual:

$$\prod_{k=1}^{\infty} \left(1 - \frac{\chi_4(p_k)}{p_k^s}\right)^{-1} = \sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{(2k+1)^s}. \tag{6.8}$$

The series obviously converges when $\sigma > 0$. The product converges for sure when $\sigma > 1$. It is believed that it converges as well when $\sigma > \frac{1}{2}$. The goal here is to establish that it converges when $\sigma > \sigma_0$, for some $\frac{1}{2} < \sigma_0 < 1$. When both converge, they converge to the same value, namely $L_4(s)$ as the series is the analytic continuation of the product, for all $\sigma > 0$. And of course, the product can not be zero when it converges. Thus $L_4(s) \neq 0$ if $\sigma > \sigma_0$.

The big question is how to find a suitable σ_0, and show that it must be strictly smaller than 1. I now focus on this point, leading to some unknown σ_0, very likely in the range $0.85 < \sigma_0 < 0.95$, for a number of reasons. The first step is to approximate the Euler product $L_4(s, n)$ with spectacular accuracy around $\sigma = 0.90$, using statistical techniques and a simple formula. This approximation amounts to denoising the irregularities caused by the prime number distribution, including Chebyshev's bias [Wiki]. After this step, the remaining is standard real analysis, trying to establish a new generic asymptotic result for a specific class of functions, and assuring that it encompasses our framework. The new theorem 6.2.1 in question, albeit independent from number theory, has yet to be precisely stated, let alone proved. The current version is as follows:

Theorem 6.2.1 *Let $A_n = \{a_1, \dots, a_n\}$ and $B_n = \{b_1, \dots, b_n\}$ be two finite sequences of real numbers, with $a_n \to 0$ as $n \to \infty$. Also assume that $b_{n+1} - b_n \to 0$. Now, define ρ_n as the ratio of the standard deviations, respectively computed on A_n (numerator) and B_n (denominator). If ρ_n converges to a non-zero value as $n \to \infty$, then b_n also converges.*

The issue to finalize the theorem is to make sure that it is applicable in our context, and add any additional requirements needed (if any). Is it enough to require $\inf \rho_n > 0$ and $\sup \rho_n < \infty$, rather than the convergence of ρ_n to non-zero? A stronger version, assuming $\sqrt{n} \cdot a_n$ is bounded and $\liminf \rho_n = \rho > 0$, leads to

$$\rho b_n - a_n \sim c + \frac{\alpha}{\sqrt{n}} + \frac{\beta}{\sqrt{n \log n}} + \cdots \tag{6.9}$$

where c, α, β are constants. As a result, $b_n \to c/\rho$. For the term $\beta/\sqrt{n \log n}$ to be valid, additional conditions on the asymptotic behavior of a_n and b_n may be required. Note that a_n and $\alpha/\sqrt{n}$ have the same order of magnitude. As we shall see, a_n captures most of the chaotic part of $L_4(s, n)$, while the term $\beta/\sqrt{n \log n}$ significant improves the approximation.

The following fact is at the very core of the GRH proof that I have in mind. Let us assume that b_n depends *continuously* on some parameter σ. If $\rho_n \to 0$ when $\sigma = \sigma_1$, and $\rho_n \to \infty$ when $\sigma = \sigma_2$, then there most be some σ_0 with $\sigma_1 \leq \sigma_0 \leq \sigma_2$ such that ρ_n converges to non-zero, or at least $\limsup \rho_n < \infty$ and $\liminf \rho_n > 0$ when $\sigma = \sigma_0$. This in turn allows us to use the proposed theoretical framework (results such as theorem 6.2.1) to prove the convergence of $L_4(s, n)$ at $\sigma = \sigma_0$. The challenge in our case is to show that there is such a σ_0, satisfying $\sigma_0 < 1$. However, the difficulty is not caused by crossing the line $\sigma = 1$, and thus unrelated to the prime number distribution. Indeed, most of the interesting action – including crossing our red line – takes place around $\sigma = 0.90$. Thus the problem now appears to be generic, rather than specific to GRH.

Now I establish the connection to the convergence of the Euler product $L_4(s, n)$. First, I introduce two new functions:

$$\delta_n(s) = L_4(s, n) - L_4(s), \quad \Lambda_n = \frac{1}{\varphi(n)} \sum_{k=1}^{n} \chi_4(p_k), \tag{6.10}$$

with $\varphi(n) = n$ for $n = 2, 3, 4$ and so on. An important requirement is that $\Lambda(n) \to 0$. I also tested $\varphi(n) = n \log n$. Then, in formula (6.9), I use the following:

$$a_n = \Lambda_n, \quad b_n = \delta_n(s). \tag{6.11}$$

Here, $L_4(s)$ is obtained via analytic continuation, not as the limit of the Euler product $L_4(s, n)$. The reason is because we don't know if the Euler product converges if $\sigma < 1$, although all evidence suggests that this is the case. Convergence of $\delta_n(s)$ translates to $c = 0$ in formula (6.9). Finally, in the figures, the X-axis represents n.

6.2.2 Quantum derivative of functions nowhere differentiable

Discrete functions such as $L_4(s, n)$, when s is fixed and n is the variable, can be scaled to represent a continuous function. The same principle is used to transform a random walk into a Brownian motion, as discussed in section 1.3 in my book on chaos and dynamical systems [15], and pictured in Figure 6.11. This is true whether the function is deterministic or random. In this context, n represents the time.

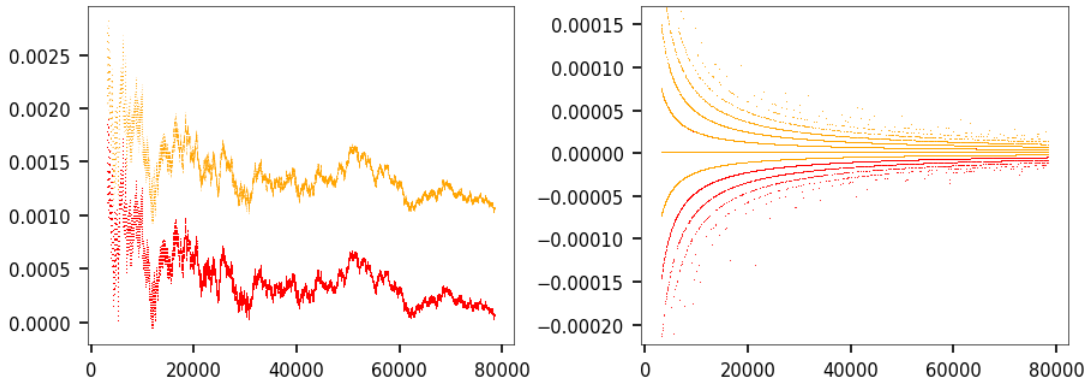


Figure 6.9: Two shifted legs of δ_n (left), and their quantum derivatives (right) [$\sigma = 0.90$]

In general, the resulting function is nowhere differentiable. You can use the integrated function rather than the original one to study the properties, as integration turns chaos into a smooth curve. But what if we could use the derivative instead? The derivative is even more chaotic than the original function, indeed it does not even exist! Yet there is a way to define the derivative and make it particularly useful to discover new insights about the function of interest. This new object called quantum derivative is not a function, but rather a set of points with a specific shape, boundary, and configuration. In some cases, it may consist of multiple curves, or

a dense set with homogeneous or non-homogeneous point density. Two chaotic functions that look identical to the naked eye may have different quantum derivatives.

The goal here is not to formally define the concept of quantum derivative, but to show its potential. For instance, in section 3.3.2.1 of the same book [15], I compute the moments of a cumulative distribution function (CDF) that is nowhere differentiable. For that purpose, I use the density function (PDF), which of course is nowhere defined. Yet, I get the correct values. While transparent to the reader, I implicitly integrated weighted quantum derivatives of the CDF. In short, the quantum derivative of a discrete function $f(n)$ is based on $f(n) - f(n-1)$. If the time-continuous (scaled) version of f is continuous, then the quantum derivative corresponds to the standard derivative. Otherwise, it takes on multiple values, called quantum states [Wiki] in quantum physics.

Now in our context, in Figure 6.9, I show two legs of $\delta_n(s)$: one where $\chi_4(p_n) = +1$, and the other one where $\chi_4(p_n) = -1$. Both give rise to time-continuous functions that are nowhere differentiable, like the Brownian motions in Figure 6.11. Unlike Brownian motions, the variance tends to zero over time. The two functions are almost indistinguishable to the naked eye, so I separated them on the left plot in Figure 6.9. The corresponding quantum derivatives consist of a set of curves (right plot, same figure). They contain a lot of useful information about $L_4(s)$. In particular:

- The left plot in Figure 6.9 shows an asymmetrical distribution of the quantum derivatives around the X-axis. This is caused by the Chebyshev bias, also called prime race: among the first n primes numbers, the difference between the proportion of primes p_k with $\chi_4(p_k) = +1$, and those with $\chi_4(p_k) = -1$, is of the order $1/\sqrt{n}$, in favor of the latter. See [3, 36, 43]. This is known as Littlewood's oscillation theorem [24].
- The various branches in the quantum derivative (same plot) correspond to runs of different lengths in the sequence $\{\chi_4(p_n)\}$: shown as positive or negative depending on the sign of $\chi_4(p_n)$. Each branch has it own point density, asymptotically equal to $2^{-\lambda}$ (a geometric distribution) for the branch featuring runs of length λ, for $\lambda = 1, 2$ and so on. A similar number theory problem with the distribution of run lengths is discussed in section 4.3 in [17], for the binary digits of $\sqrt{2}$.

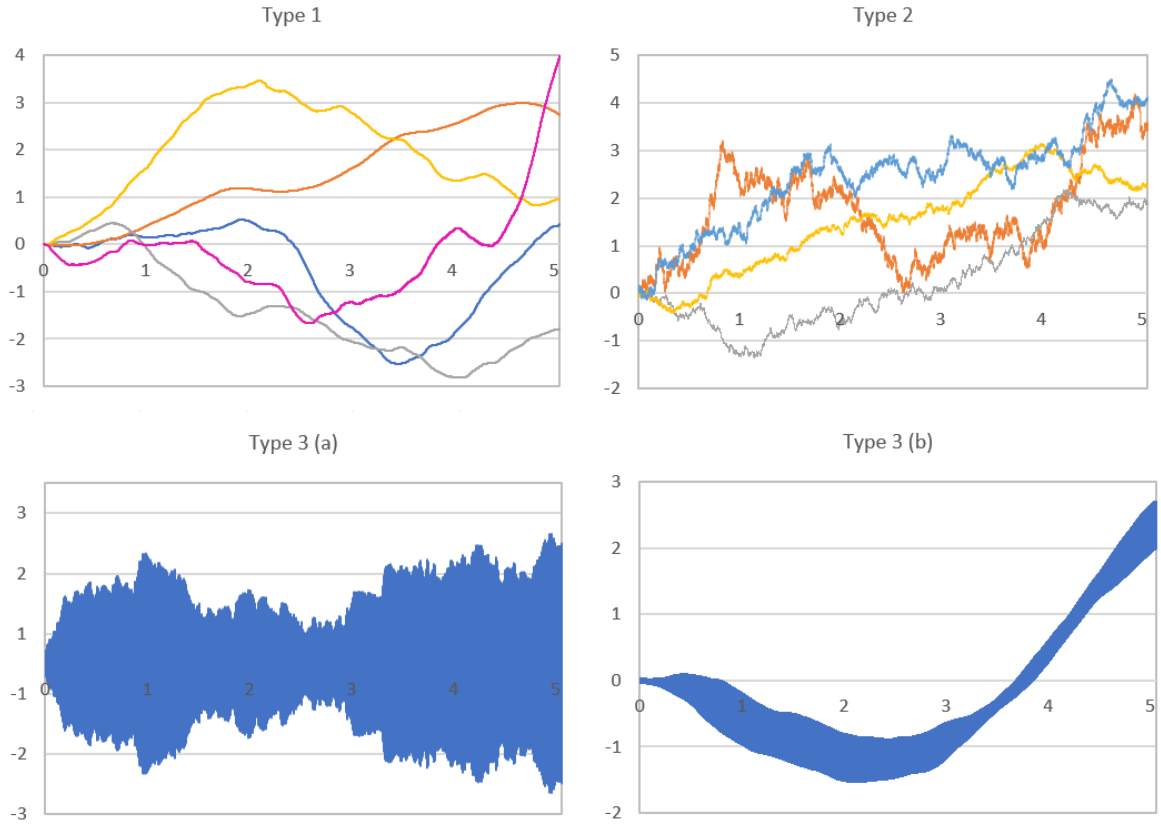


Figure 6.10: Integrated Brownian (top left), Brownian (top right) and quantum derivatives (bottom)

6.2.3 Project and solution

The project consists of checking many of the statements made in section 6.2.1, via computations. Proving the empirical results is beyond the scope of this work. The computations cover not only the case $L_4(s, n)$, but also

other similar functions, including synthetic ones and Rademacher random multiplicative functions [25, 26, 27]. It also includes an empirical verification of theorem 6.2.1, and assessing whether its converse might be true. Finally, you will try different functions φ in formula (6.10) to check the impact on approximation (6.9). In the process, you will get familiar with a few Python libraries:

- MPmath to compute $L_4(s)$ for complex arguments when $\sigma < 1$.
- Primepy to obtain a large list of prime numbers.
- Scipy for curve fitting, when verifying the approximation (6.9).

The curve fitting step is considerably more elaborate than the standard implementation in typical machine learning projects. First, it consists of finding the best c, α, β in (6.9) for a fixed n, and identifying the best model: in this case, the choice of $\sqrt{n}$ and $\sqrt{n \log n}$ for the curve fitting function (best fit). Then, using different n, assess whether or not c, α, β, ρ depend on n or not, and whether $c = 0$ (this would suggest that the Euler product converges).

Finally, you will run the same curve fitting model for random functions, replacing the sequence $\{\chi_4(p_n)\}$ by random sequences of independent $+1$ and -1 evenly distributed, to mimic the behavior of $L_4(s, n)$ and Λ_n. One would expect a better fit when the functions are perfectly random, yet the opposite is true. A possible explanation is the fact that the Chebyshev bias in $L_4(n, s)$ is very well taken care of by the choice of Λ_n, while for random functions, there is no such bias, and thus no correction.

The project consists of the following steps:

Step 1: MPmath library, complex and prime numbers. Compute $L_4(s)$ using the MPmath library. See my code in section 6.2.4. The code is designed to handle complex numbers, even though in the end I only use the real part. Learn how to manipulate complex numbers by playing with the code. Also, see how to create a large list of prime numbers: look at how I use the PrimePy library. Finally, look at the Python `curve_fitting` and `r2_score` functions, to understand how it works, as you will have to use them. The former is from the Scipy library, and the latter (to measure goodness-of-fit) is from the Sklearn library.

Step 2: Curve fitting, part A. Implement Λ_n and the Euler Product $L_4(s, n)$ with n (the number of factors) up to 10^5. My code runs in a few seconds for $n \leq 10^5$. Beyond $n = 10^7$, a distributed architecture may help. In the code, you specify m rather than n, and $p_n \approx m/\log m$ is the largest prime smaller than m. For $s = \sigma + it$, choose $t = 0$, thus avoiding complex numbers, and various values of σ ranging from 0.55 to 1.75. The main step is the curve fitting procedure, similar to a linear regression but for non linear functions. The goal is to approximate $\delta_n(s)$, focusing for now on $\sigma = 0.90$.

The plots of $\delta_n(s)$ and Λ_n at the top in Figure 6.11, with $n = 80{,}000$, look very similar. It seems like there must be some $c_n(s)$ and a strictly positive $\rho_n(s)$ such that $\rho_n(s)\delta_k(s) - \Lambda_k \approx c_n(s)$ for $k = 1, 2, \ldots, n$. Indeed, the scaling factor

$$\rho_n(s) = \frac{\text{Stdev}[\Lambda_1, \ldots, \Lambda_n]}{\text{Stdev}[\delta_1(s), \ldots, \delta_n(s)]},$$

together with $c_n(s) = 0$, work remarkably well. The next step is to refine the linear approximation based on $\rho = \rho_n(s)$, using (6.9) combined with (6.11). This is where the curve fitting takes place; the parameters to estimate are c, α and β, with c close to zero. You can check the spectacular result of the fit, here with $\sigma = 0.90$ and $n \approx 1.25 \times 10^6$, on the bottom left plot in Figure 6.11. Dropping the $\sqrt{n \log n}$ term in (6.9) results in a noticeable drop in performance (test it). For $\rho_n(s)$, also try different options.

Step 3: Curve fitting, part B. Perform the same curve fitting as in Step 2, but this time for different values of n. Keep $s = \sigma + it$ with $t = 0$ and $\sigma = 0.90$. The results should be identical to those in Table 6.1, where $\gamma_n = \sqrt{n} \cdot \Lambda_n$. The coefficients c, α, β and R^2 (the R-squared or quality of the fit) depend on n and s, explaining the notation in the table. Does $\rho_n(s)$ tend to a constant depending only on s, as $n \to \infty$? Or does it stay bounded? What about the other coefficients?

Now do the same with $\sigma = 0.70$ and $\sigma = 1.10$, again with various values of n. Based on your computations, do you think that $\rho_n(s)$ decreases to zero, stays flat, or increases to infinity, depending on whether $s = 0.70$, $s = 0.90$ or $s = 1.10$? If true, what are the potential implications?

Step 4: Comparison with synthetic functions. First, try $\varphi(n) = n \log n$ rather $\varphi(n) = n$, in (6.10). Show that the resulting curve fitting is not as good. Then, replace $\chi_4(p_k)$, both in $L_4(s, n)$ and Λ_n, by independent Rademacher distributions [Wiki], taking the values $+1$ and -1 with the same probability $\frac{1}{2}$. Show that again, the curve fitting is not as good, especially if $n \leq 10^5$. Then, you may even replace p_k (the k-th prime) by $k \log k$. The goal of these substitutions is to compare the results when χ_4 is replaced

by synthetic functions that mimic the behavior of the Dirichlet character modulo 4. Also, you want to assess how much leeway you have in the choice of these functions, for the conclusions to stay valid.

The use of synthetic functions is part of a general approach known as generative AI. If all the results remain valid for such synthetic functions, then the theory developed so far is not dependent on special properties of prime numbers: we isolated that problem, opening the path to an easier proof that the Euler product $L_4(s,n)$ converges to $L_4(s)$ at some location $s = \sigma_0 + it$ with $\sigma_0 < 1$ inside the critical strip.

Step 5: Application outside number theory. Using various pairs of sequences $\{a_n\}$, $\{b_n\}$, empirically verify when the statistical theorem 6.2.1 might be correct, and when it might not.

The Python code in section 6.2.4 allows you to perform all the tasks except Step 5. In particular, for Step 4, set `mode='rn'` in the code. As for the curve fitting plot – the bottom left plot in Figure 6.11 – I multiplied both the target function $\delta_n(s)$ and the fitted curve by $\sqrt{n}$, here with $n = 1.25 \times 10^6$. Both tend to zero, but after multiplication by $\sqrt{n}$, they may or may not tend to a constant strictly above zero. Either way, it seems to indicate that the Euler product converges when $\sigma = 0.90$. What's more, the convergence looks strong, non-chaotic, and the second-order term involving $\sqrt{n \log n}$ in the approximation error, seems to be correct.

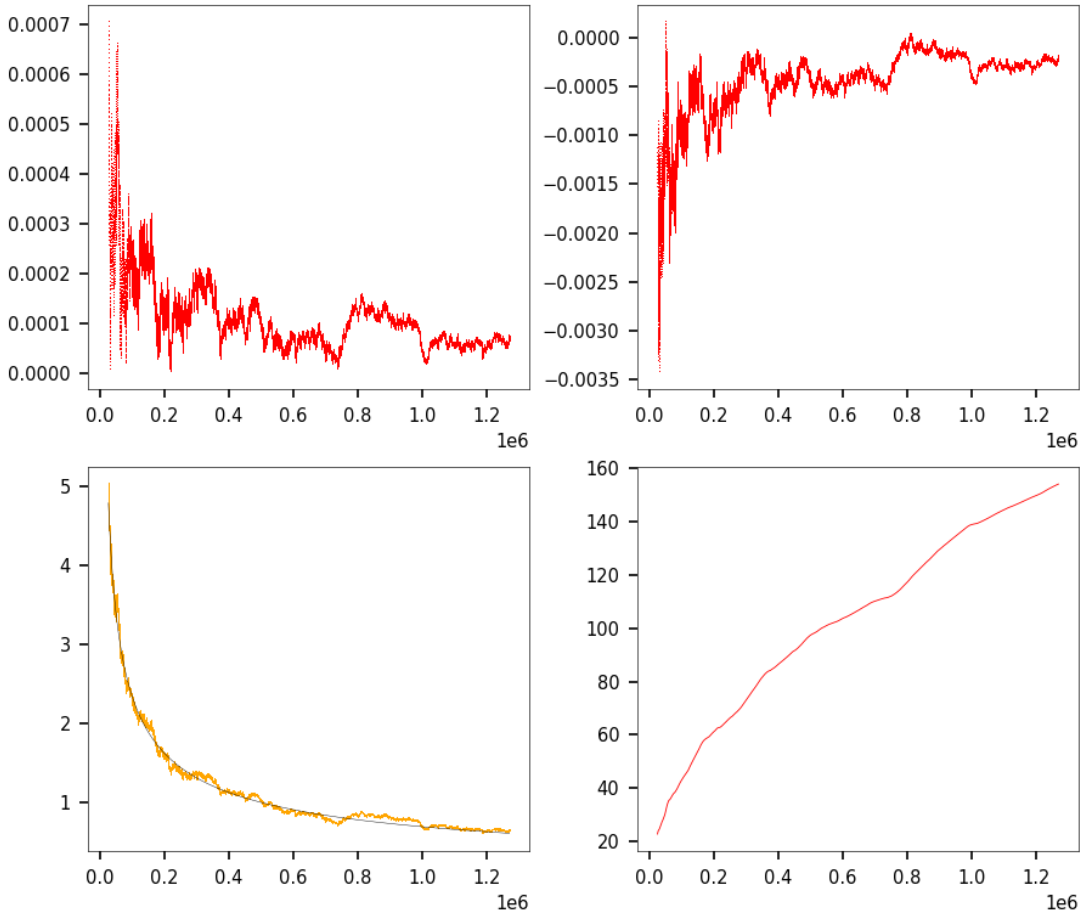


Figure 6.11: Top: δ_n (left), Λ_n (right); bottom: fitting δ_n (left), integrated δ_n (right) [$\sigma = 0.90$]

Regarding Step 3, Table 6.1 is the answer when $\sigma = 0.90$. It seems to indicate that $\rho_n(s)$ convergences (or is at least bounded and strictly above zero) when $\sigma = 0.90$ (remember that $s = \sigma + it$, with $t = 0$). With $\sigma = 0.70$, it seems that $\rho_n(s)$ decreases probably to zero, while with $\sigma = 1.10$, $\rho_n(s)$ is increasing without upper bound. The highest stability is around $\sigma = 0.90$. There, theorem 6.2.1 may apply, which would prove the convergence of the Euler product strictly inside to critical strip. As stated earlier, this would be a huge milestone if it can be proved, partially solving GRH not for $\zeta(s)$, but for the second most famous function of this nature, namely $L_4(s)$. By partial solution, I mean proving it for (say) $\sigma_0 = 0.90 < 1$, but not yet for $\sigma_0 = \frac{1}{2}$.

Unexpectedly, Figure 6.12 shows that the fit is not as good when using a random sequence of $+1$ and -1, evenly distributed, to replace and mimic χ_4. The even distribution is required by the Dirichlet theorem, a generalization of the prime number theorem to arithmetic progressions [Wiki].

Finally, see the short code below as the answer to Step 5. The code is also on GitHub, here. The parameters

p, q play a role similar to σ, and `r` represents ρ_n in theorem 6.2.1. The coefficient ρ_n may decrease to zero, increase to infinity, or converge depending on p and q. Nevertheless, in most cases when p, q are not too small, b_n converges. Applied to $L_4(s, n)$, it means that convergence may occur at s even if $\rho_n(s)$ does not converge.

The existence of some σ_1 for which $\rho_n(s)$ decreases to zero, and some σ_2 for which $\rho_n(s)$ increases to infinity, implies that there must be a σ_0 in the interval $[\sigma_1, \sigma_2]$, for which $\rho_n(s)$ converges or is bounded. This in turn implies that the Euler product $L_4(s, n)$ converges at $s = \sigma + it$ if $\sigma > \sigma_0$. The difficult step is to show that the largest σ_1 resulting in $\rho_n(s)$ decreasing to zero, is < 1. Then, $\sigma_0 < 1$, concluding the proof.

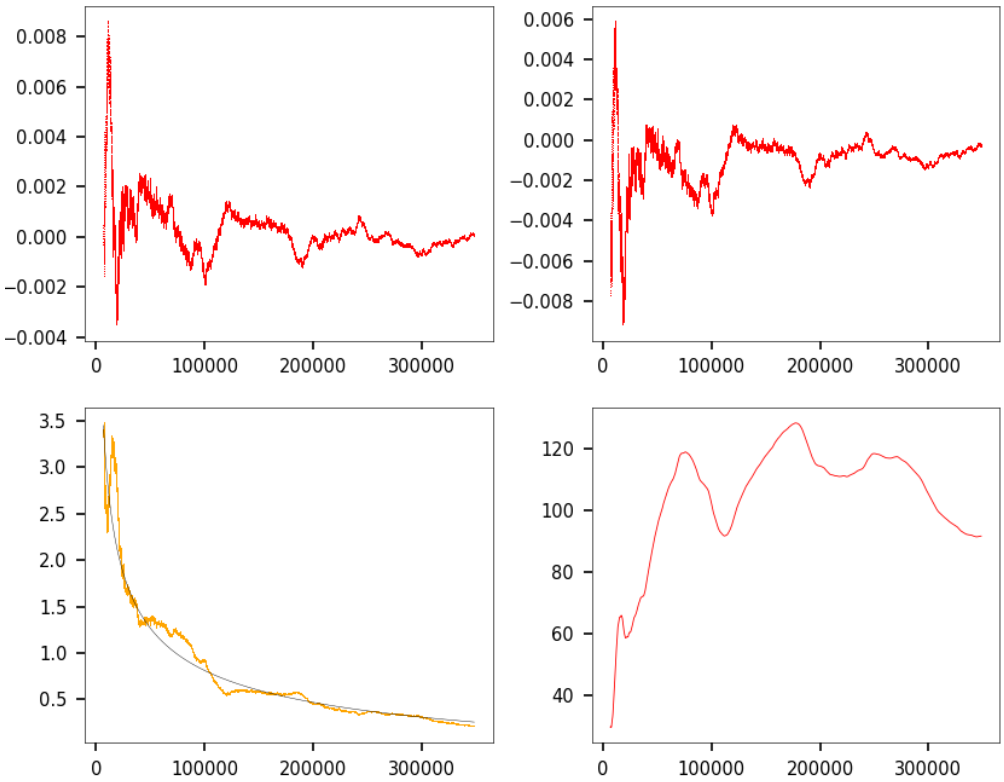


Figure 6.12: Same as Figure 6.11, replacing $\delta_n(s), \Lambda_n$ by synthetic functions

n	γ_n	$c_n(s)$	$\alpha_n(s)$	$\beta_n(s)$	$\rho_n(s)$	R^2
127,040	−0.27	0.00	0.41	0.94	5.122	0.940
254,101	−0.40	0.00	0.39	0.98	5.082	0.976
381,161	−0.35	0.00	0.37	1.03	5.151	0.986
508,222	−0.44	0.00	0.36	1.05	5.149	0.989
635,283	−0.32	0.00	0.36	1.04	5.085	0.989
762,343	−0.22	0.00	0.36	1.03	5.047	0.989
889,404	−0.10	0.00	0.35	1.06	5.123	0.990
1,016,464	−0.44	0.00	0.35	1.07	5.139	0.990
1,143,525	−0.30	0.00	0.34	1.08	5.121	0.991
1,270,586	−0.24	0.00	0.34	1.08	5.111	0.991

Table 6.1: One curve fitting per row, for $\delta_n(s)$ with $\sigma = 0.90$

```
import numpy as np

N = 10000000
p = 1.00
q = 0.90
stdev = 0.50
seed = 564
np.random.seed(seed)
start = 20
```

```
u = 0
v = 0
a = np.zeros(N)
b = np.zeros(N)

for n in range(2, N):

   u += -0.5 + np.random.randint(0, 2)
   v += np.random.normal(0, stdev)/n**q
   a[n] = u / n**p
   b[n] = v

   if n % 50000 == 0:
      sa = np.std(a[start:n])
      sb = np.std(b[start:n])
      r = sa / sb
      c = r * b[n] - a[n]
      print("n = %7d r =%8.5f an =%8.5f bn =%8.5f c =%8.5f sa =%8.5f sb=%8.5f"
             %(n, r, a[n], b[n], c, sa, sb))
```

Important note. When dealing with the Euler product $L_4(s, n)$, the ratio $\rho_n(s)$ is rather stable (bounded strictly above zero, chaos-free, barely depending on n) and may even converge when $\sigma = 0.90$ and $t = 0$. Again, $s = \sigma + it$. Indeed, both the numerator and denominator appear well-behaved and seemingly chaos-free. Both of them tend to zero as n increases, at the same speed as $1/\sqrt{n}$. The chaos is in $L_4(s, n)$ and Λ_n. This fact can be leveraged to make progress towards proving the convergence of $L_4(s, n)$ at $\sigma = 0.90$. If not at $\sigma = 0.90$, there has to be at least one value $\sigma_0 < 1$ (close to 0.90) for which everything I just wrote, apply.

6.2.4 Python code

The implementation of the quantum derivatives is in section [4] in the following code. The coefficient $\rho_n(s)$ is denoted as `r`, while the parameter γ in table 6.1 is denoted as `mu`. In addition to addressing Step 1 to Step 4, the computation of the Dirichlet-L series and the Q_2 function are in section [2.1]. For Step 5, see the Python code at the end of section 6.2.3. Finally, to use synthetic functions rather than χ_4, set `mode='rn'`. The code is also on GitHub, here.

```
# DirichletL4_EulerProduct.py
# On WolframAlpha: DirichletL[4,2,s], s = sigma + it
#    returns Dirichlet L-function with character modulo k and index j.
#
# References:
#    https://www.maths.nottingham.ac.uk/plp/pmzcw/download/fnt_chap4.pdf
#    https://mpmath.org/doc/current/functions/zeta.html
#    f(s) = dirichlet(s, [0, 1, 0, -1]) in MPmath

import matplotlib.pyplot as plt
import matplotlib as mpl
import mpmath
import numpy as np
from primePy import primes
from scipy.optimize import curve_fit
from sklearn.metrics import r2_score
import warnings
warnings.filterwarnings("ignore")

#--- [1] create tables of prime numbers

m = 1000000 # primes up to m included in Euler product
aprimes = []

for k in range(m):
   if k % 100000 == 0:
      print("Creating prime table up to p <=", k)
   if primes.check(k) and k > 2:
      aprimes.append(k)


#--- [2] Euler product

#--- [2.1] Main function

def L4_Euler_prod(mode = 'L4', sigma = 1.00, t = 0.00):
```

```
37
38      L4 = mpmath.dirichlet(complex(sigma,t), [0, 1, 0, -1])
39      print("\nMPmath lib.: L4(%8.5f + %8.5f i) = %8.5f + %8.5f i"
40           % (sigma, t,L4.real,L4.imag))
41
42      prod = 1.0
43      sum_chi4 = 0
44      sum_delta = 0
45      run_chi4 = 0
46      old_chi4 = 0
47      DLseries = 0
48      flag = 1
49
50      aprod = []
51      adelta = []
52      asum_delta = []
53      achi4 = []
54      arun_chi4 = []
55      asum_chi4 = []
56
57      x1 = []
58      x2 = []
59      error1 = []
60      error2 = []
61      seed = 116 # try 103, 105, 116 & start = 2000 (for mode = 'rn')
62      np.random.seed(seed)
63      eps = 0.000000001
64
65      for k in range(len(aprimes)):
66
67         if mode == 'L4':
68            condition = (aprimes[k] % 4 == 1)
69         elif mode == 'Q2':
70            condition = (k % 2 == 0)
71         elif mode == 'rn':
72            condition = (np.random.uniform(0,1) < 0.5)
73
74         if condition:
75            chi4 = 1
76         else:
77            chi4 = -1
78
79         sum_chi4 += chi4
80         achi4.append(chi4)
81         omega = 1.00 # try 1.00, sigma or 1.10
82         # if omega > 1, asum_chi4[n] --> 0 as n --> infty
83         # asum_chi4.append(sum_chi4/aprimes[k]**omega)
84         asum_chi4.append(sum_chi4/(k+1)**omega)
85         # asum_chi4.append(sum_chi4/(k+1)*(np.log(k+2)))
86
87         if chi4 == old_chi4:
88            run_chi4 += chi4
89         else:
90            run_chi4 = chi4
91         old_chi4 = chi4
92         arun_chi4.append(run_chi4)
93
94         factor = 1 - chi4 * mpmath.power(aprimes[k], -complex(sigma,t))
95         prod *= factor
96         aprod.append(1/prod)
97
98         term = mpmath.power(2*k+1, -complex(sigma,t))
99         DLseries += flag*term
100        flag = -flag
101
102     limit = -eps + 1/prod # full finite product (approx. of the limit)
103     if mode == 'L4':
104        limit = L4 # use exact value instead (infinite product if it converges)
105
106     for k in range(len(aprimes)):
107
108        delta = (aprod[k] - limit).real # use real part
109        adelta.append(delta)
110        sum_delta += delta
111        asum_delta.append(sum_delta)
112        chi4 = achi4[k]
```

```
113
114         if chi4 == 1:
115            x1.append(k)
116            error1.append(delta)
117         elif chi4== -1:
118            x2.append(k)
119            error2.append(delta)
120
121     print("Dirichlet L: DL(%8.5f + %8.5f i) = %8.5f + %8.5f i"
122        % (sigma, t, DLseries.real, DLseries.imag))
123     print("Euler Prod.: %s(%8.5f + %8.5f i) = %8.5f + %8.5f i\n"
124        % (mode, sigma, t, limit.real, limit.imag))
125
126     adelta = np.array(adelta)
127     aprod = np.array(aprod)
128     asum_chi4 = np.array(asum_chi4)
129     asum_delta = np.array(asum_delta)
130     error1 = np.array(error1)
131     error2 = np.array(error2)
132
133     return(limit.real, x1, x2, error1, error2, aprod, adelta, asum_delta,
134            arun_chi4, asum_chi4)
135
136 #--- [2.2] Main part
137
138 mode = 'L4' # options: 'L4', 'Q2', 'rn' (random chi4)
139 (prod, x1, x2, error1, error2, aprod, adelta, asum_delta, arun_chi4,
140     asum_chi4) = L4_Euler_prod(mode, sigma = 0.90, t = 0.00)
141
142
143 #--- [3] Plots (delta is Euler product, minus its limit)
144
145 mpl.rcParams['axes.linewidth'] = 0.3
146 plt.rcParams['xtick.labelsize'] = 7
147 plt.rcParams['ytick.labelsize'] = 7
148
149 #- [3.1] Plot delta and cumulated chi4
150
151 x = np.arange(0, len(aprod), 1)
152
153 # offset < len(aprimes), used to enhance visualizations
154 offset = int(0.02 * len(aprimes))
155
156 # y1 = aprod / prod
157 # plt.plot(x[offset:], y1[offset:], linewidth = 0.1)
158 # plt.show()
159
160 y2 = adelta
161 plt.subplot(2,2,1)
162 plt.plot(x[offset:], y2[offset:], marker=',', markersize=0.1,
163   linestyle='None', c='red')
164
165 y3 = asum_chi4
166 plt.subplot(2,2,2)
167 plt.plot(x[offset:], y3[offset:], marker=',', markersize=0.1,
168   linestyle='None', c='red')
169
170 #- [3.2] Denoising L4, curve fitting
171
172 def objective(x, a, b, c):
173
174     # try c = 0 (actual limit)
175     value = c + a/np.sqrt(x) + b/np.sqrt(x*np.log(x))
176     return value
177
178 def model_fit(x, y2, y3, start, offset, n_max):
179
180     for k in range(n_max):
181
182         n = int(len(y2) * (k + 1) / n_max) - start
183         stdn_y2 = np.std(y2[start:n])
184         stdn_y3 = np.std(y3[start:n])
185         r = stdn_y3 / stdn_y2
186
187         # note: y3 / r ~ mu / sqrt(x) [chaotic part]
188         mu = y3[n] * np.sqrt(n) # tend to a constant ?
```

```
      y4 = y2 * r - y3
      y4_fit = []
      err = -1

      if min(y4[start:]) > 0:
         popt, pcov = curve_fit(objective, x[start:n], y4[start:n],
                    p0=[1, 1, 0], maxfev=5000)
         [a, b, c] = popt
         y4_fit = objective(x, a, b, c)
         err = r2_score(y4[offset:], y4_fit[offset:])
         print("n = %7d mu =%6.2f c =%6.2f a =%5.2f b =%5.2f r =%6.3f err =%6.3f"
                  %(n, mu, c, a, b, r, err))

   return(y4, y4_fit, err, n)

n_max = 10 # testing n_max values of n, equally spaced
start = 20 # use Euler products with at least 'start' factors
if mode == 'rn':
   start = 1000
if start > 0.5 * offset:
   print("Warning: 'start' reduced to 0.5 * offset")
   start = int(0.5 * offset)
(y4, y4_fit, err, n) = model_fit(x, y2, y3, start, offset, n_max)
ns = np.sqrt(n)

if err != -1:
   plt.subplot(2,2,3)
   plt.plot(x[offset:], ns*y4[offset:], marker=',', markersize=0.1,
     linestyle='None', c='orange')
   plt.plot(x[offset:], ns*y4_fit[offset:], linewidth = 0.2, c='black')
else:
   print("Can't fit: some y4 <= 0 (try different seed or increase 'start')")

#--- [3.3] Plot integral of delta

y5 = asum_delta
plt.subplot(2,2,4)
plt.plot(x[offset:], y5[offset:], linewidth = 0.4, c='red')
plt.show()


#--- [4] Quantum derivative

#- [4.1] Function to differentiated: delta, here broken down into 2 legs

plt.subplot(1,2,1)
shift = 0.001
plt.plot(x1[offset:], error1[offset:], marker=',', markersize=0.1,
  linestyle='None', alpha = 1.0, c='red')
plt.plot(x2[offset:], shift + error2[offset:], marker=',', markersize=0.1,
  linestyle='None', alpha = 0.2, c='orange')

#- [4.2] Quantum derivative

def d_error(arr_error):

   diff_error = [] # discrete derivative of the error
   positives = 0
   negatives = 0
   for k in range(len(arr_error)-1):
      diff_error.append(arr_error[k+1] - arr_error[k])
      if arr_error[k+1] - arr_error[k] > 0:
         positives +=1
      else:
         negatives += 1
   return(diff_error, positives, negatives)

(diff_error1, positives1, negatives1) = d_error(error1)
(diff_error2, positives2, negatives2) = d_error(error2)
ymin = 0.5 * float(min(min(diff_error1[offset:]), min(diff_error1[offset:])))
ymax = 0.5 * float(max(max(diff_error1[offset:]), max(diff_error2[offset:])))

plt.subplot(1,2,2)
plt.ylim(ymin, ymax)
plt.plot(x1[offset:len(x1)-1], diff_error1[offset:len(x1)-1], marker=',', markersize=0.1,
       linestyle='None', alpha=0.8, c = 'red')
```

```
plt.plot(x2[offset:len(x2)-1], diff_error2[offset:len(x2)-1], marker=',', markersize=0.1,
       linestyle='None', alpha=0.8, c = 'orange')
plt.show()

print("\nError 1: positives1: %8d negatives1: %8d" % (positives1, negatives1))
print("Error 2: positives2: %8d negatives2: %8d" % (positives2, negatives2))
```

Chapter 7

Quantum, chaotic and fractal types of algorithmic convergence

The problems in this chapter complement the material discussed earlier in the book. Section 7.1 deals with the digit sum, in particular with its generating and cumulative functions. It leads to an interesting fractal convergence behavior featured in Figure 7.1. Section 7.2 discusses linear recurrences with non-fractal, chaotic convergence, illustrated in Figures 7.2 and 7.3.

7.1 Digit count generating function and fractal convergence

Let H_k be the number of 1 in the binary expansion of the positive integer k, also called the hamming weight of k. These weights are listed as the sequence A000120 in the online encyclopedia of integer sequences (OEIS), see here. The Hamming weight normalized cumulative function is pictured in Figure 7.1 and defined as

$$S_H(n) = \frac{1}{n \log_2 n} \sum_{k=0}^{n} H_k, \quad n = 2, 3, 4, \ldots \tag{7.1}$$

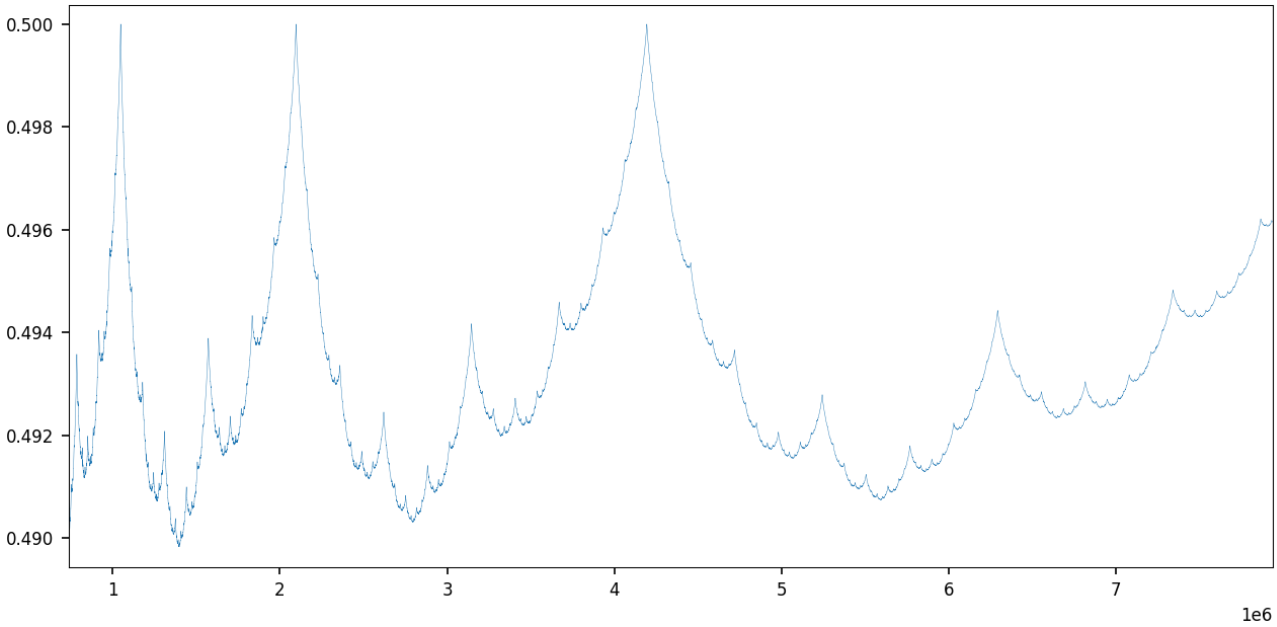


Figure 7.1: Hamming weights normalized cumulative function $S_H(n)$ with $n \leq 8 \times 10^6$ on the X-axis

Clearly, $S_H(n) \to \frac{1}{2}$ as $n \to \infty$ with the peaks occurring when n is a power of 2. The curve is fractal-like: the graph between two successive peaks is identical to that between the two previous peaks, but stretched horizontally by a factor 2, and with minima increasing over time, thus causing some vertical compression as n increases. See below the code for the computations and plot generation. The code is also on GitHub, here.

```
import numpy as np
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['legend.fontsize'] = 'x-small'

def countSetBits(n):
    # compute H(n)
    count = 0
    while(n):
        count += n & 1
        n >>= 1
    return(count)

sum = 0
arr_k = []
arr_sum = []

for k in range(0,8000000):
    d = countSetBits(k)
    sum += d
    arr_k.append(k)
    if k < 2:
        arr_sum.append(0)
    else:
        arr_sum.append(sum/(k*np.log2(k)))
    if k % 10 == 0:
        print("sss", k, d, sum)

plt.plot(arr_k, arr_sum, linewidth = 0.2)
plt.show()
```

Now I discuss some interesting properties related to the generating functions attached to H_n. It is easy to prove that

$$\Lambda_n(\lambda, x) := \prod_{k=0}^{n-1} \left(1 + e^{\lambda} x^{2^k}\right) = \sum_{k=0}^{2^n-1} e^{\lambda H_k} x^k \tag{7.2}$$

Thus $\Lambda_n(0, x) = (1 - x^{2^n})/(1 - x)$. By virtue of (7.2), we also have

$$\Lambda_n(\lambda, 1) = \left(1 + e^{\lambda}\right)^n = \sum_{k=0}^{n} \binom{n}{k} e^{k\lambda} = \sum_{k=0}^{2^n-1} e^{\lambda H_k} \tag{7.3}$$

Now, taking the m-th derivative with respect to λ and then setting $\lambda = 0$, we obtain

$$\sum_{k=0}^{2^n-1} \left(H_k\right)^m = \sum_{k=0}^{n} \binom{n}{k} k^m = 2^n n! \sum_{k=0}^{m} \frac{S(m,k)}{2^k (n-k)!}, \quad m = 0, 1, 2 \ldots \tag{7.4}$$

where $S(\cdot, \cdot)$ denotes the Stirling numbers of the second kind. The left identity in (7.4) follows from (7.3) and is true even if m is not an integer. The expression on the right in (7.4) corresponds to the m-th moment of a Bernoulli distribution of parameters $(n, \frac{1}{2})$, multiplied by 2^n.

The standard form of the Hamming weights generating function, identical to that of binary digit sums, is studied in [38]. See also [41]. The main result is Theorem 1 in [38], stating that

$$\sum_{k=0}^{\infty} H_k x^k = \frac{1}{1-x} \sum_{m=0}^{\infty} \frac{x^{2^m} - 2x^{2^{m+1}} + x^{3 \cdot 2^m}}{(1 - x^{2^m})(1 - x^{2^{m+1}})}. \tag{7.5}$$

The "digit sum" entry in Wolfram also features beautiful results, see here. Among others:

$$\sum_{k=1}^{\infty} \frac{H_k}{k(k+1)} = \log 2, \quad \sum_{k=1}^{\infty} \frac{(2k+1) H_k}{k^2 (k+1)^2} = \frac{\pi^2}{9}$$

$$\sum_{k=1}^{\infty} \frac{H_k + H'_k}{2k(2k+1)} = \gamma, \quad \sum_{k=1}^{\infty} \frac{H_k - H'_k}{2k(2k+1)} = \log \frac{4}{\pi}$$

where γ is the Euler-Mascheroni constant and H'_k is the number of 0 in the binary expansion of k.

The Hamming weights H_n can be computed with the recursion $H_{2n} = H_n$, $H_{2n+1} = H_n + 1$ with $H_0 = 0$. A related sequence with many interesting properties is Stern's diatomic series (A_n) with $A_0 = 0$ and $A_1 = 1$, defined by the recursion $A_{2n} = A_n, A_{2n+1} = A_n + A_{n+1}$. Finally, the integer sequence defined by $C_n = \frac{1}{2}C_{n-1}$ if C_{n-1} even, $C_n = 3C_{n-1} + 1$ otherwise, is presumed to converge to 1 regardless of the initial condition $C_0 > 0$. This is known as the Collatz conjecture.

7.2 Other examples of chaotic convergence

For centuries, mathematicians worked on problems where the convergence is either smooth or does not happen. Now the concept of chaotic convergence is mainstream, popularized by the stochastic gradient descent in deep neural networks, central to LLMs. In this book, most cases also involve various types of chaotic convergence, for instance in Figures 4.16, 5.1, 6.7, 6.8, and 6.11. In some examples such as Figure 7.1, the chaos exhibits a fractal structure. In other examples such as Figures 2.3, 2.7, 6.4, and 6.9, it looks like we have multiple curves converging to a same limit, when in reality there is only one, with jumps from one level to another every millisecond: this pattern is strikingly similar to quantum states. This section features more illustrations falling in these various categories.

For examples of chaotic convergence in the context of explainable deep neural networks, see Figure 4.6 featuring stochastic gradient descent, Figure 4.14 featuring swarm optimization, and Figure 4.2 featuring adaptive loss function all in my book on no-Blackbox LLM architectures [22]. Yet another type is asymptotic periodicity where x_n converges to multiple values depending on the residue class of n modulo some integer. See section 5.2.1.

7.2.1 Smooth convergence but with multiple branches

The first example deals with the recursion $x_{n+3} = 2x_{n+2} - 16x_{n+1} + 4x_n$ with initial conditions $x_0 = 0$, $x_1 = 1$, and $x_2 = 1$. We are interested in the limit

$$\rho := \lim_{n\to\infty} |x_n|^{1/n} \tag{7.6}$$

if it exists. It does in this example. To compute it, proceed as follows:

- Find the roots of the characteristic polynomial, in this case the cubic equation $x^3 = 2x^2 - 16x + 4$.
- Identify the root with the largest norm. Here we have two complex conjugate roots with the same norm, and one real root. The complex roots have the largest norm.
- Then ρ is the square root of the norm in question. More specifically,

$$\rho = \sqrt{\frac{12\omega}{2\omega - \omega^2 + 44}}, \quad \text{with } \omega = \sqrt[3]{82 + 6\sqrt{2553}} \tag{7.7}$$

First, I used AI to find ρ. The LLMs that I tested correctly identify $r \approx 0.2572$ as the real root of the cubic equation, and $\rho \approx 3.9436$ as the solution. Perplexity even mentioned that $\rho = 2\,||r||^{-1/2}$, which is correct. But they all failed when providing the exact values for the roots, thus the exact value for ρ was also incorrect. To be fair, I used the free version of these tools. Then, I used the `solve` function from the SymPy Python library. It is based on symbolic mathematics to find exact solutions. That's how I obtained (7.7).

The most interesting part is how $\rho_n = |x_n|^{1/n}$ converges to ρ as $n \to \infty$, see Figure 7.2 where ρ_n on the Y-axis has a different color depending on n mod 7, with n on the X-axis. We jump from one branch to another each time n is incremented, yet eventually ρ_n converges to n. The associated Python code is listed in section 7.2.2.

7.2.2 Chaotic convergence with multiple branches

I now discuss the case $x_{n+2} = |x_{n+1} - 3x_n|$ with initial conditions $x_0 = 1$ and $x_1 = 1$. Again, we are interested in the same limit ρ defined by (7.6). At first glance, this case seems easier as the characteristic polynomial is now of degree 2 instead of 3. However, the absolute value in the recursion makes it different with a dramatic shift in behavior as seen in Figure 7.3, by contrast to Figure 7.2. Both plots show $|x_n|^{1/n}$ on the Y-axis with n on the X-axis.

It is reminiscent of the problem about random Fibonacci sequences discussed in section 7.4 in [22]. In that example, the equivalent of the ratio $r_n = x_{n+1}/x_n$ asymptotically oscillates between 5 different values. Taking the geometric mean of these values yields the correct, exact value for ρ. However the situation is a lot more complicated here. Even though empirical evidence points to the limit $\rho \approx 1.58010$, it is not obvious to

prove convergence, let alone putting an exact value on ρ. Interestingly, some LLMs erroneously believe that the answer is $\rho = \sqrt{3}$. For a deep dive into this partular case, see setion 7.2.3.

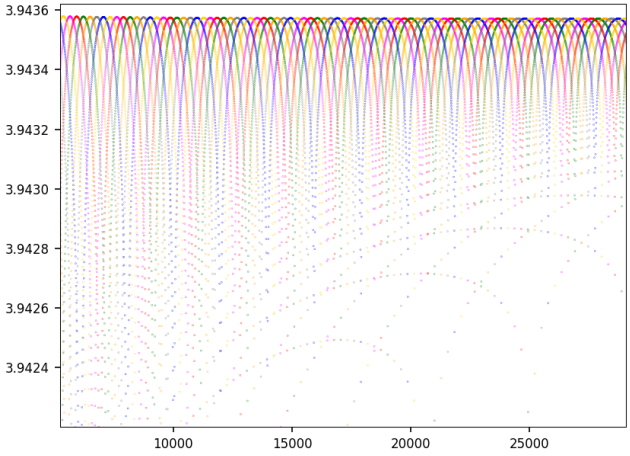


Figure 7.2: Constant jumps in $|x_n|^{1/n}$ gives the illusion of 7 curves but there is only one (section 7.2.1).

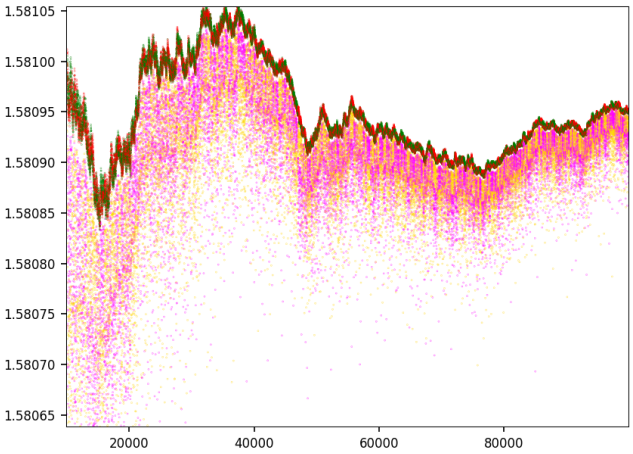


Figure 7.3: 4-colors quantum regime similar to that of Figure 7.2 but now with chaos (section 7.2.2).

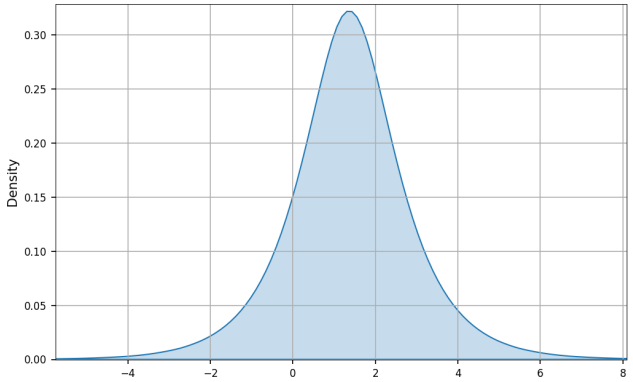


Figure 7.4: Empirical PDF for $\log|x_{n+1}/x_n|$, case featured in Figure 7.2

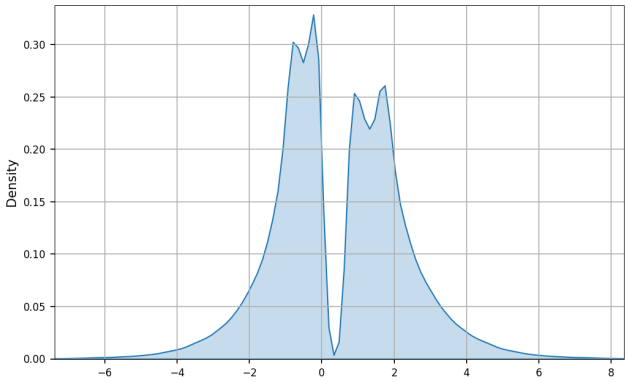


Figure 7.5: Empirical PDF for $\log|x_{n+1}/x_n|$, case featured in Figure 7.3

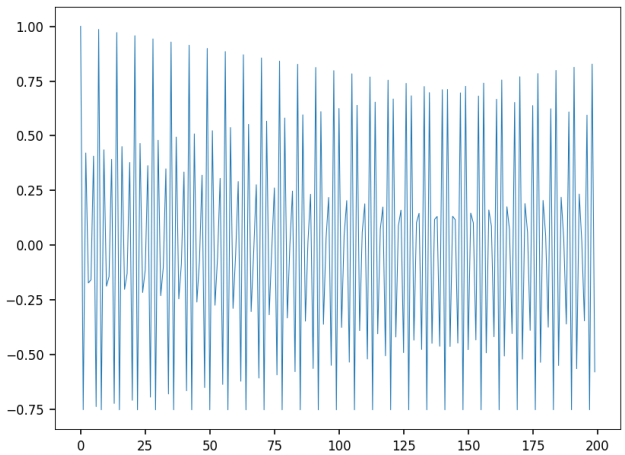


Figure 7.6: Autocorrel function for $\log|x_{n+1}/x_n|$, case featured in Figure 7.2

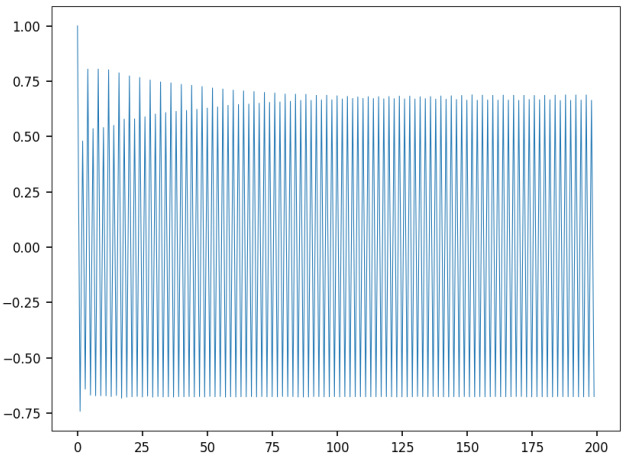


Figure 7.7: Autocorrel function for $\log|x_{n+1}/x_n|$, case featured in Figure 7.3

The curve $\rho_n = |x_n|^{1/n}$ has 4 chaotic bands. For a specific n, the band that ρ_n belongs to depends on n mod 4. The green and red bands are interlaced but distinct. The yellow and pink bands are somewhat more separated. This is further confirmed when looking at the empirical autocorrelation function pictured in Figure 7.7, showing the correlation between the sequences $\log|x_{n+1}/x_n|$ and $\log|x_{n+k+1}/x_{n+k}|$ for $0 \leq k < 200$. A cycle of length 4

is clearly visible. By contrast, the case studied in section 7.2.1 has a cycle of length 7, as pictured in Figure 7.6, which also explains the 7 branches in Figure 7.2.

Finally, I looked at the empirical probability density function (EPDF in Python), computed on the first 10^5 values of $\log|x_{n+1}/x_n|$, skipping the first 10,000 ones. Not surprisingly, the EPDF is smooth and classic for the case featured in Figure 7.2, but bumpy with 4 main peaks and a deep valley for the example discussed here. You can see the contrast by comparing Figures 7.4 and 7.5 respectively. To conclude, the dark upper boundary in Figure 7.3 looks like a Brownian motion to the layman, but it is in fact very different. Ours converges (the variance tends to zero as n increases) while the variance of a Brownian motion keeps growing. However, this is a small issue: rescaling would easily fix it. The main difference is that our curve is significantly more chaotic, and thus suitable as a model when increased chaos is needed. Metrics measuring the amount of chaos are discussed in section 2.3 in my book on chaotic dynamical systems [15].

Below is the Python code used to do the analyses and producing the plots in sections 7.2.1 and 7.2.2. The code is also on GitHub, here. The `mode` parameter allows you to choose between the recursion in section 7.2.1 and that in section 7.2.2.

```
import numpy as np
import gmpy2
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['legend.fontsize'] = 'x-small'

ndigits = 10000
ctx = gmpy2.get_context()
ctx.precision = ndigits
z = gmpy2.log(2)

# choose N < ndigits/2 to avoid losing precision
N = 5000
mode = 'quantum' # options: 'quantum' or 'chaotic'

x0 = gmpy2.mpfr(0)
x1 = gmpy2.mpfr(1)
x2 = gmpy2.mpfr(1)
arr_k = []
arr_x = []
arr_col = []
arr_log_rho = []

def set_color(k, mode):

   if mode == 'quantum':
      if k % 7 == 0:
         color='red'
      elif k % 7 == 1:
         color='blue'
      elif k % 7 == 2:
         color='green'
      elif k % 7 == 3:
         color='gold'
      elif k % 7 == 4:
         color='orange'
      elif k % 7 == 5:
         color='magenta'
      elif k % 7 == 6:
         color='gray'

   elif mode == 'chaotic':
      if k % 4 == 0:
         color = 'red'
      elif k % 4 == 1:
         color = 'gold'
      elif k % 4 == 2:
         color = 'green'
      elif k % 4 == 3:
         color = 'magenta'
      else:
         color = 'gray'
   else:
```

```
      print("Unsuported mode:", mode)
      exit()
   return(color)


#--- Main

modulus = 4
hash_modulo = {}

for k in range(2,N):
   color = set_color(k, mode)
   if mode == 'quantum':
      x = 2*x2 - 16*x1 + 4*x0
      delta = 0
   elif mode == 'chaotic':
      x = abs(x2 - 3*x1)
   v = gmpy2.log(abs(x))/k
   w = gmpy2.exp(v)
   rho   = float(x/x2)
   if mode == 'chaotic':
      old_rho = float(x2/x1)
      # delta should be 0 for k < ndigits/2
      delta = rho**2 - 1 + 6/old_rho - 9/old_rho**2
   log_rho = np.log(abs(rho))
   if k > N/10:
      print("%6d log_rho: %8.5f %8.5f %f" % (k, log_rho, delta, rho))
      arr_k.append(k)
      arr_x.append(w)
      arr_col.append(color)
      arr_log_rho.append(log_rho)
      residue = k % modulus
      if residue in hash_modulo:
         arr_local = hash_modulo[residue]
         arr_local.append(log_rho)
         hash_modulo[residue] = arr_local
      else:
         hash_modulo[residue] = [log_rho]
   x0 = x1
   x1 = x2
   x2 = x

plt.scatter(arr_k, arr_x, c=arr_col, s=0.02)
plt.show()
plt.scatter(arr_k, arr_log_rho, c=arr_col, s=0.02)
plt.show()
plt.plot(arr_k, arr_log_rho, linewidth=0.4)
plt.show()
print("\nRho[residue] with modulus = %2d" %(modulus))
avg = 1
for residue in range(modulus):
   arr_vals = hash_modulo[residue]
   arr_vals = np.array(arr_vals)
   arr_exp = np.exp(arr_vals)
   iqr_rho = np.quantile(arr_exp,0.75) - np.quantile(arr_exp, 0.25)
   mu = np.average(arr_vals)
   mean_rho = np.exp(mu)
   median = np.std(arr_vals)
   avg *= mean_rho
   print("residue %2d | mu = %8.5f | rho = %8.5f | irq = %8.5f | median = %8.5f"
            % (residue, mu, mean_rho, iqr_rho, median))
avg = avg**(1/modulus)
print("Rho: %8.5f" %(avg))


#--- Plot EPDFs

np_log_rho = np.array(arr_log_rho)
import seaborn as sns
plt.figure(figsize=(8, 5))
# sns.ecdfplot(np_rho)
sns.kdeplot(data=np_log_rho, fill=True)
plt.grid(True)
plt.show()

meanlog = np.mean(np_log_rho)
```

```
medianlog = np.median(np_log_rho)
mean = np.exp(meanlog)
median = np.exp(medianlog)
print("\nRho: %8.5f [median = %8.5f | meanlog = %8.5f]" % (mean, median, meanlog))
print()


#--- Compute autocorrels for log(x/x2)

nobs = len(np_log_rho)
arr_lag = []
arr_autocorrel = []

for lag in range(200):
   mean1 = np.mean(np_log_rho[0: nobs-lag])
   mean2 = np.mean(np_log_rho[lag:nobs])
   std1 = np.std(np_log_rho[0: nobs-lag])
   std2 = np.std(np_log_rho[lag:nobs])
   dotprod = np.dot(np_log_rho[0: nobs-lag], np_log_rho[lag:nobs])
   dotprod /= (nobs-lag)
   autocorrel = (dotprod - mean1*mean2)/(std1*std2)
   arr_lag.append(lag)
   arr_autocorrel.append(autocorrel)
   print("Autcorrel lag %3d: %8.5f" %(lag, autocorrel))

plt.plot(arr_lag, arr_autocorrel, linewidth = 0.5)
plt.show()
```

7.2.3 Deep dive into the chaotic case

Let's $r_n = x_{n+1}/x_n$ and $s_n = \log r_n$, with $r_n \geq 0$ for all n. Then the case $x_{n+2} = |x_{n+1} - 3x_n|$ can be rewritten in two different ways, as follows:

$$r_{n+1} = \left|1 - \frac{3}{r_n}\right| \tag{7.8}$$

$$s_{n+1} = \log\left|1 - 3\exp\left(-s_n\right)\right| \tag{7.9}$$

Also, with $x_0 = x_1 = 1$, we have:

$$x_n = \prod_{k=0}^{n-1} r_k, \qquad \log x_n = \sum_{k=0}^{n-1} s_k, \qquad \rho = \exp\left(\mu\right) \text{ with } \mu = \lim_{n\to\infty} \frac{1}{n}\sum_{k=0}^{n-1} s_k \tag{7.10}$$

Both (7.8) and (7.9) are discrete chaotic dynamical systems. The latter is much better behaved, ergodic, with a non-singular invariant measure and a finite ergodic mean equal to μ in (7.10), with $|x_n|^{1/n} \to \rho = e^{\mu}$. In the code, the number `N` of iterations most be much smaller than the precision (set by `ndigits`) otherwise you quickly get values of x_k that are completely wrong. However, thanks to the ergodicity, you may use a small `ndigits` and a large `N` to massively accelerate the computations without impacting the precision on ρ.

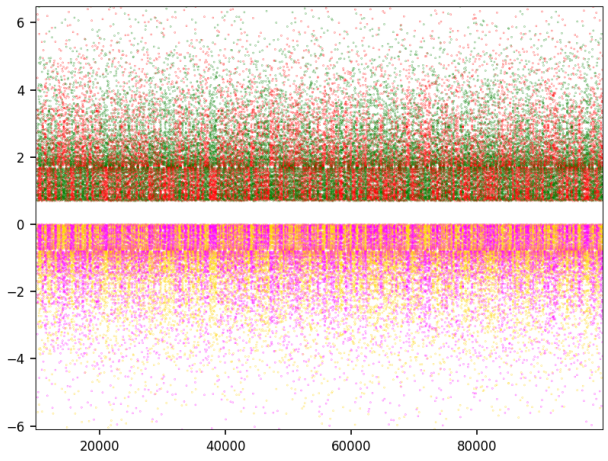


Figure 7.8: Same as Figure 7.3 but now with s_n instead of $|x_n|^{1/n}$ on he Y-axis (n on the X-axis)

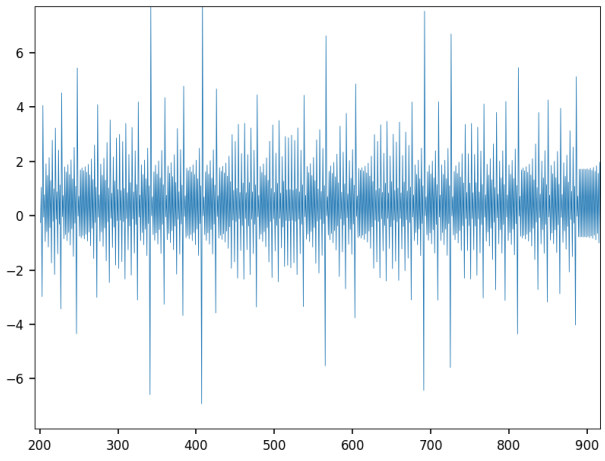


Figure 7.9: Same as Figure 7.8 with standard plot instead of scatterplot, thus missing the empty band

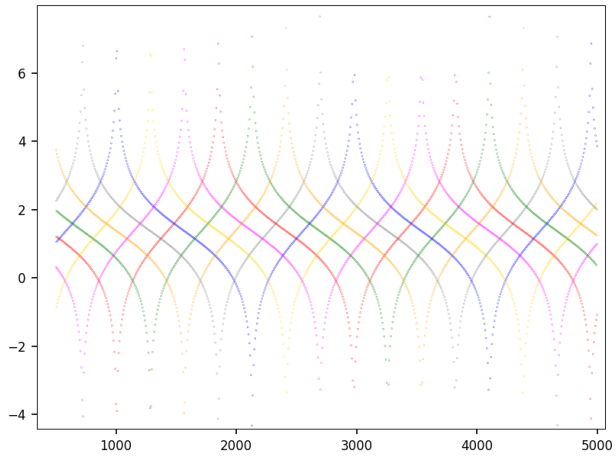


Figure 7.10: Same as Figure 7.2 but now with s_n instead of $|x_n|^{1/n}$ on he Y-axis (n on the X-axis)

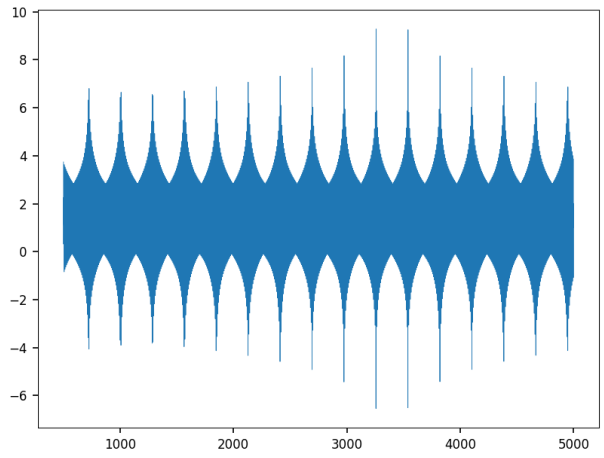


Figure 7.11: Same as Figure 7.10 with standard plot instead of scatterplot. Provided solely as a piece of art

I now state a fundamental result, which is a direct application of the pigeonhole principle. Let the indices $0, 1, 2$ and so on (that is, all the positive integers) be partitioned into m non-overlapping sets $S_0, \ldots, S_{m-1}$, where m may be finite or infinite. Let p_j be the proportion of indices in set S_j, with $p_0 + \cdots + p_{m-1} = 1$. Also, let $S_j(n) = \{i \in S_j \text{ such that } i \leq n\}$, and $|S_j(n)|$ be the number of elements in $S_j(n)$. Then

$$\rho = \exp\left[\sum_{j=0}^{m-1} p_j \mu_j\right], \quad \text{with } \mu_j = \lim_{n\to\infty} \mu_j(n) \text{ and } \mu_j(n) := \frac{1}{|S_j(n)|} \sum_{i\in S_j(n)} s_i. \tag{7.11}$$

I used (7.11) to compute ρ for random Fibonacci sequences discussed in section 7.4 in [22]. In this section, S_j consists of the positive integers congruent to j modulo m. The goal is to find a modulus m for which the ergodic means μ_j $(j = 0, 1, \ldots, m-1)$ are distinct. This is true if $m \in \{2, 4\}$, but not for $m \in \{3, 5\}$. It shows a pattern, pictured with $m = 4$ colors in Figures 7.8 and 7.3. In the Python code in section 7.2.2, $n, j, m, \mu_j(n)$ are denoted respectively as `N`, `residue`, `modulus` and `mu`. This is further summarized in table 7.1.

$m=2$		$m=3$		$m=4$		$m=5$	
j	$\mu_j(n)$	j	$\mu_j(n)$	j	$\mu_j(n)$	j	$\mu_j(n)$
0	1.92912	0	0.46528	0	1.89867	0	0.45958
1	-1.01310	1	0.45667	1	-1.04106	1	0.45089
		2	0.45204	2	1.96136	2	0.46003
				3	-0.98514	3	0.45794
						4	0.46153

Table 7.1: $\mu_j(n)$ for $2 \leq m \leq 5$, with $n = 10^5$

In each column in Table 7.1, the average $\mu_j(n)$ is about 0.458, which is a good approximation to the theoretical μ. But variations between columns are large, pointing to 4 distinct values when $m = 4$. This is reflected in Figure 7.5 with two major peaks, each one being a double summit.

I did not compute the invariant measure attached to (7.9), defined as the CDF or cumulative distribution function $F_Z(z)$ solution to the functional equation

$$F_Z(z) = F_Z\Big(\log\Big|1 - 3\exp\big(-z\big)\Big|\Big). \tag{7.12}$$

In many examples like the logistic map, F_z has a closed-form expression. But this is not the case here, though you can solve (7.12) numerically with an iterative algorithm, to get an approximation: the empirical CDF. An example is discussed in section 2.2.1 in my book on chaotic dynamical systems [15]. If you were able to find a closed-form expression, you can verify it via the Perron-Frobenius operator, also called transfer operator [Wiki]. Also, the theoretical ergodic mean is the expectation $\mathrm{E}[Z]$ attached to F_Z. I asked LLMs to find F_z or $\mu = \mathrm{E}[Z]$ and mostly got erroneous theoretical results yet good approximation for μ. Finally, (7.11) can be rewritten as

$$\rho = e^{\mu}, \quad \mu = \mathrm{E}[Z] = \sum_{j=0}^{m-1} p_j \mu_j. \tag{7.13}$$

Finally, Figures 7.10 and 7.11 are related to the smooth example discussed in section 7.2.1 and should be compared respectively with Figures 7.8 and 7.9 related to the chaotic case discussed in this section. It also illustrates the fact that a standard plot hides some important patterns, thus the preference for a scatterplot.

Technical note: The horizontal empty band in Figure 7.8 is between the values 0 and $\log 2$ on the Y-axis. There are other horizontal bands with low density but non-empty, some visible to the naked eye or by zooming in. These bands are the same regardless of the initial conditions $x_0, x_1 > 0$. The empty band is responsible for the gap between the two main peaks in Figure 7.5. The explanation is simple: if $x_n > \log 2$ then $x_{n+1} < 0$, and if $x_n < 0$ then $x_{n+1} > \log 2$. One way to get rid of the empty band is by splitting the dynamical system into two subsystems $t_n = s_{2n}, t'_n = s_{2n+1}$ with the same recursion $t_{n+1} = g(g(t_n)$, $t'_{n+1} = g(g(t'_n)$, with $g(t) = \log|1 - 3\exp(-t)|$.

7.2.4 Interesting connection between 3^n and the digits of $\sqrt{2}$

The findings in this section apply to the positive integer powers q^n of an integer $q > 1$ not a power of 2. It is not limited to just $q = 3$, although I spent most of my research on the latter. I start with an established theorem.

Theorem 7.2.1 *Let $q > 1$ be an integer, not a power of 2. For $n = 0, 1$ and so on, let ν_n be the dyadic rational*

$$\nu_n = \frac{q^n}{2^{\lfloor n \log_2 q \rfloor}}. \tag{7.14}$$

Let $\nu_{(n)}$ be the median computed on the first $2n+1$ values $\nu_0, \ldots, \nu_{2n}$. Then $\nu_{(n)} \to \sqrt{2}$ as $n \to \infty$.

Proof

The symbols $\lfloor \cdot \rfloor$ and $\{\cdot\}$ denote respectively the integer part and fractional part functions. Also, $q^n = 2^{n \log_2 q}$ where $\log_2$ is the logarithm in base 2. Thus, ν_n is a dyadic rational in $[1, 2]$, equal to

$$\nu_n = 2^{\{n \log_2 q\}}. \tag{7.15}$$

Now, if α is irrational, then the sequence $\{\alpha n\}$ ($n = 0, 1$ and so on) is equidistributed modulo 1 and dense in $[0, 1]$ by virtue of Weyl's equidistribution theorem. Thus the sequence v_n is dense in $[1, 2]$ because $\alpha = \log_2 q$ is irrational. That is, there are integer subsequences $0 \leq \sigma_1 < \sigma_2 < \sigma_3$ and so on such that ν_{σ_n} gets arbitrarily close to any pre-specified constant in $[1, 2]$ including $\sqrt{2}$, as $n \to \infty$..

Finally, due to the equidistribution, the median values $\nu_{(n)}$ tend to the median of 2^U, where U is a random variable with uniform distribution on [0, 1]. Its median is $\sqrt{2}$. ∎

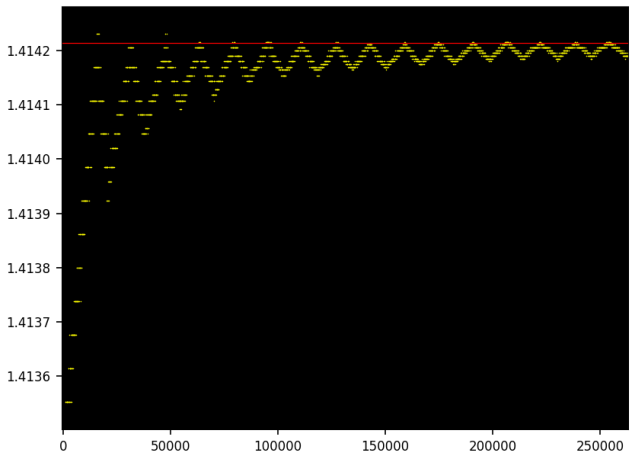


Figure 7.12: $\nu(n) = \text{Median}(\nu_0, \ldots, \nu_{2n})$ on the Y-axis, with abcissa $2n + 1$ on the X-axis

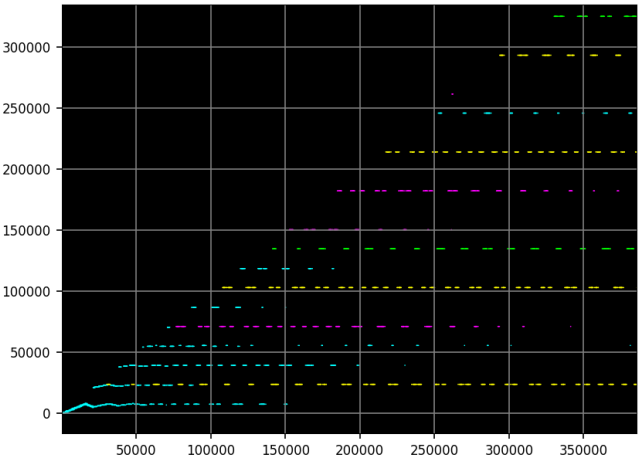


Figure 7.13: Index σ_n satisfying $\nu_{(n)} = \nu_{\sigma_n}$ on the Y-axis, with abcissa $2n + 1$ on the X-axis

Theorem 7.2.1 states that you can build a subsequence of ν_n that convergences to $\sqrt{2}$. It provides an explicit solution based on the medians $\nu_{(n)}$. The binary digits of $\nu_{(n)}$ match those of $\sqrt{2}$ at the limit. And for each n, $\nu_{(n)}$ is one of the values (the median) in $\{\nu_0, \nu_1, \ldots, \nu_{2n}\}$ since this set has an odd number of elements. Let σ_n be the index of the median value $\nu_{(n)}$, uniquely characterized by $\nu_{(n)} = \nu_{\sigma_n}$, with $0 \leq \sigma_n \leq 2n$. Now I discuss some insights, based on $q = 3$.

- The sequence σ_n defined by $\nu_{(n)} = \nu_{\sigma_n}$ goes up and down, jumping from one horizontal level to another as pictured in Figure 7.13. The general trend is up, and eventually $\sigma_\infty = \infty$.

- The colors in Figure 7.13 show how close ν_{σ_n} is to the limit $\sqrt{2}$. Green means that only the first $\kappa = 18$ binary digits of $\nu(n)$ match those of $\sqrt{2}$. Yellow, magenta and cyan correspond respectively to $\kappa = 17$, $\kappa = 16$ and $\kappa \leq 15$.
- Figure 7.12 shows how slowly $\nu_{(n)}$ convergences to the limit $\sqrt{2}$, staying below the limit (the red line) most of the time. However, it periodically gets close and hits the red line, when n is a multiple of 8,000 (approximately). If you replace $q = 3$ by $q = 5$, the pattern is quite different.
- If you replace the median $\nu_{(n)}$ by an average, then convergence is towards $1/\log 2$ instead of $\sqrt{2}$. Indeed,

$$\frac{1}{n}\sum_{k=1}^{n}\nu_k = \frac{1}{n}\sum_{k=1}^{n}\frac{q^k}{2^{\lfloor k\log_2 q\rfloor}} = \frac{1}{n}\sum_{k=1}^{n}2^{\{k\log_2 q\}} \to \frac{1}{\log 2} \text{ as } n\to\infty. \tag{7.16}$$

  This result is particularly interesting when n is a power of 2. Then all the denominators $2^{\lfloor k\log_2 q\rfloor}$ and n in (7.16) are a power of 2.

For each n, the binary digits of ν_n start with 1, followed by a decimal point. The digits on the right starting after the decimal point are called leading digits. One would think that for any digit string of length κ, its frequency is $2^{-\kappa}$. However, this is not the case. For instance, the leading digit '0' has frequency $\log_2(3/2) \approx 58\%$, different from 50%. Similarly, the string '0000' appears about twice as frequently as '1111' in the leading digits of ν_n. Leading digits frequently follow the Benford's law [Wiki]. But this is not the case here.

Finally, for a fixed $\kappa > 0$ and $q = 3$, let $\varphi(\kappa)$ be the minimum n such that the first κ leading binary digits of ν_n match those of $\sqrt{2}$. That is, the number of binary digits of $q^{\varphi(\kappa)}$ is $\lfloor\varphi(\kappa)\log_2 q\rfloor + 1$, of which only the first κ match those of $\sqrt{2}$. Table 7.2 shows how $\varphi(\kappa)$ grows exponentially fast as a function of κ.

κ	$\varphi(\kappa)$	κ	$\varphi(\kappa)$	κ	$\varphi(\kappa)$	κ	$\varphi(\kappa)$
6	6	11	512	17	23,734	21	4,247,415
7	47	13	6,803	18	134,936	24	5,200,100
9	206	15	8,133	20	2,342,045	27	5,390,637

Table 7.2: Exponential growth of $\varphi(\kappa)$ as a function of κ

A common mistake when proving that (say) $\sqrt{2}$ is a normal number is as follows. First, one proves that the binary digits of 3^n are equidistributed when $n \to \infty$. Let us pretend that this fact has been proved. Also, for each n, there is an integer σ_n such that $\nu_{(n)}$ and 3^{σ_n} have the same digits. And $\nu_{(n)}$ converges to $\sqrt{2}$. Thus $\sqrt{2}$ is simply normal in base 2. However this argument is fallacious for the following reasons:

- The digits of 3^{σ_n} and $\nu_{(n)}$ are identical. Thus, if the former has about 50% of 1, this is also true for the latter. But what if most of the 1's are on the right half in the digit expansion? At best, the first $O(\log n)$ digits on the left match those of $\sqrt{2}$ while 3^{σ_n} has $O(n)$ digits.
- It is possible to find many infinite subsequences of ν_n that converge to any constant ξ in $[1, 2]$. For instance to $\xi = \frac{3}{2}$. Despite the 50% of 1 guaranteed in 3^n as $n \to \infty$, the limit ξ has a very different proportion.
- Actually, no one knows if the digits of 3^n are equidistributed. It is a major open problem, see [10, 37].

The Python code below performs the computations and generates the plots used in section 7.2.4. While I work with large numbers such as 3^n with $n = 10^7$, I only need the first $O(\log n)$ leading digits. Thus the `ndigits` parameter is set to a small value (50), far smaller than n. The latter is represented by `N` in the code. In modern AI, this truncation process is known as quantization. Here, it tremendously accelerates the computations. The code is also on GitHub, here.

```
import numpy as np
import gmpy2
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['legend.fontsize'] = 'x-small'

ndigits = 50 # maximum possible match
ctx = gmpy2.get_context()
ctx.precision = ndigits
N = 200000
```

```
string_tests = ('0', '1', '00', '01', '10', '11', '0000', '1111')

power3 = gmpy2.mpz(1)
isqrt2 = gmpy2.isqrt(2 * 2**(2*ndigits))
rescaled2 = isqrt2/ 2**int(gmpy2.log2(isqrt2))
sqrt2 = bin(isqrt2)[2:]

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

def match_strings(bin3, sqrt2):
   # match = number of consecutive identical chars on the left
   match = 0
   Flag = True
   while match < min(len(bin3)-1, len(sqrt2)-1) and Flag:
      if bin3[match] == sqrt2[match]:
         match += 1
      else:
         Flag = False
   return(match)

arr_match = []
arr_nu = []
hash_strings = {}
hash_match = {}
q = 3
lg23 = np.log2(q)
lg2 = np.log(2)
lg3 = np.log(q)
e2 = 2**ndigits


for k in range(N):

   rescaled3 = gmpy2.exp(k*lg3 - int(k*lg23)*lg2)
   arr_nu.append(float(rescaled3))
   power3 = gmpy2.mpz(e2 * rescaled3)
   bin3 = bin(power3)[2:]
   for string in string_tests:
      if bin3[1:len(string)+1] == string:
         update_hash(hash_strings, string, 1)
   match = match_strings(bin3, sqrt2)
   if match not in hash_match:
      hash_match[match] = k
   arr_match.append(match)

#--- Show frequency of some leading digit strings

print()
for string in hash_strings:
   count = hash_strings[string]
   print("String frequency %5s: %8.6f" % (string, count/N))

#--- Show phi(kappa)

print()
for kappa in hash_match:
   print("phi(%3d) = %6d" %(kappa, hash_match[kappa]))

#--- Plot convergence of median to sqrt(2)

arr_median_k = []
arr_median_idx = []
arr_median_match = []
arr_median_value = []
arr_median_color = []

for k in range(len(arr_nu)//20, len(arr_nu), 100):
   # compute nu((k-1)/2)
   arr = np.array(arr_nu[0: k])
   median_idx = np.argpartition(arr, len(arr) // 2)[len(arr) // 2]
   match = arr_match[median_idx]
```

```
    if match == 18:
       arr_median_color.append('lime')
    elif match == 17:
       arr_median_color.append('yellow')
    elif match == 16:
       arr_median_color.append('magenta')
    else:
       arr_median_color.append('cyan')
    arr_median_k.append(k)
    arr_median_idx.append(median_idx)
    arr_median_match.append(match)
    arr_median_value.append(arr[median_idx])
    if k % 10000 == 1:
       print("Convergence: %6d %6d %8.5f %8.5f"
             %(k, median_idx, arr[median_idx], arr_match[median_idx]))

#--- Plots

plt.rcParams['axes.facecolor'] = 'black'
plt.scatter(arr_median_k, arr_median_value,s=0.6, c='yellow', linewidths=0)
plt.axhline(y=np.sqrt(2), color='r', linewidth = 0.6)
plt.show()
plt.scatter(arr_median_k, arr_median_idx, s=0.6, linewidths=0, c = arr_median_color)
plt.grid(color='gray')
plt.show()
```

7.3 Random polynomials and spectral analysis of digit distributions

For a real number $\xi > 1$ with binary digits $d_k(\xi)$, $k = 0, 1, \ldots$ with $d_0(\xi) = 1$, let us define the digit generating function of complex argument z as

$$G(z, \xi) = \sum_{k=0}^{\infty} d_k(\xi) z^k, \quad d_k(\xi) \in \{0, 1\}, \quad \|z\| < 1. \tag{7.17}$$

Let $\xi_* = 2^{-\mu} G\big(\frac{1}{2}, \xi\big)$ where $\mu = \lfloor \log_2 \xi \rfloor$ is an integer power of 2. Then ξ and ξ_* have the same binary digits: ξ_* is the rescaled version of ξ. For convenience, define the polynomial G_n of degree at most n as

$$G_n(z, \xi) = \sum_{k=0}^{n} d_k(\xi) z^k \to G(z, \xi) \text{ as } n \to \infty. \tag{7.18}$$

Now for fixed n and ξ, let $\rho_k(n)$, $k = 0, 1, \ldots$, denote the roots of $G_n(z, \xi)$, sorted in descending order based on the distance $d(\|\rho_k(n)\|, 1)$. Then,

$$G_n(z, \xi) = \pm \prod_k \Big(\rho_k(n) - z\Big). \tag{7.19}$$

The sign in (7.19) is chosen so that $G_n(0, \xi) = +1$. In particular, $G_n(1, \xi)$ is the number of 1 in the first n binary digits of ξ, that is, the digit sum function associated to ξ.

7.3.1 Orbits of the digit generating function

I now define the orbit of G at radius $\rho < 1$ as the set of complex values

$$\Gamma_{\rho,\xi} = \{\, G(z, \xi) \text{ such that } ||z|| = \rho \,\}.$$

See the case $\rho = 0.80$ in Figures 7.16 and 7.17, respectively for $\xi = \pi$ and $\xi = \frac{24}{37}$. By contrast, the curve is a simple oval when $\xi = \frac{4}{7}$. For $\rho = 0.99$, see Figures 7.14 and 7.15, respectively for $\xi = \pi$ and $\xi = \frac{24}{37}$. The precise definition of the orbit at ρ is as follows. Let $z_t = \rho \cdot (\cos 2\pi t + i \sin 2\pi t)$. Then $\Gamma_{\rho,\xi}$ is the parametric curve

$$x_t = \Re\big[G(z_t, \xi)\big], \quad y_t = \Im\big[G(z_t, \xi)\big], \quad \text{with } 0 \le t \le 1. \tag{7.20}$$

The color on the orbit plots changes from blue to red as t increases from 0 to 1, with green for $t \approx 0.5$. Here $\Re, \Im$ denote respectively the real and imaginary parts of a complex number. In the code, I used the approximation G_n to G, with n set to `ndigits=1000` and with 200,000 evenly spaced values of t. As ρ gets closer to 1, you need to increase `ndigits` accordingly. For most ξ, there is no convergence when $\rho \to 1$. And unlike the Riemann zeta function rescaled by the factor $1 - 2^{1-z}$ to give rise to the Dirichlet eta function that converges on

the unit circle, there is no such generic scaling factor applicable to $G(z, \xi)$ to make it converge there. Notable exceptions are integers and dyadic rationals ξ for which $G(z, \xi)$ always exists regardless of z's modulus.

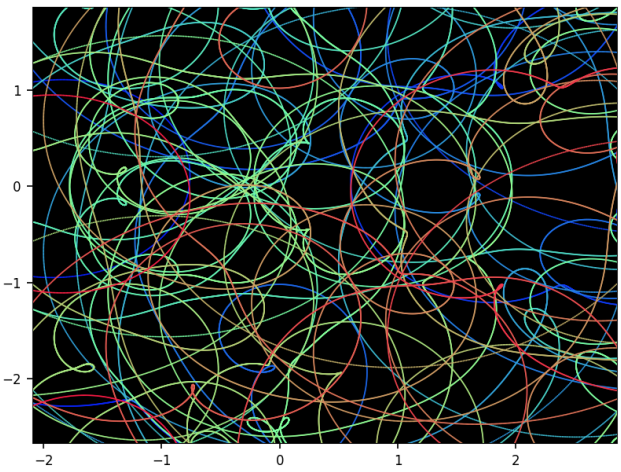


Figure 7.14: $\Gamma_{\rho,\xi}$ with $\rho = 0.99$, $\xi = \pi$

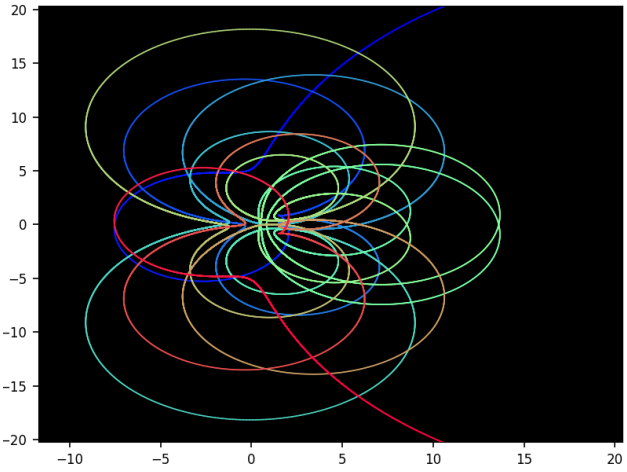


Figure 7.15: $\Gamma_{\rho,\xi}$ with $\rho = 0.99$, $\xi = \frac{24}{37}$

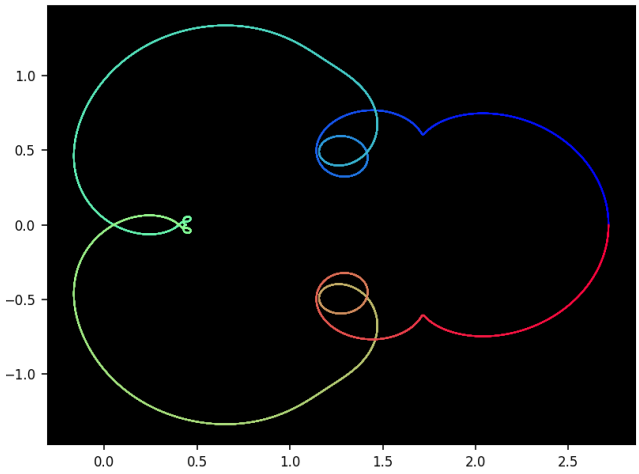


Figure 7.16: $\Gamma_{\rho,\xi}$ with $\rho = 0.80$, $\xi = \pi$

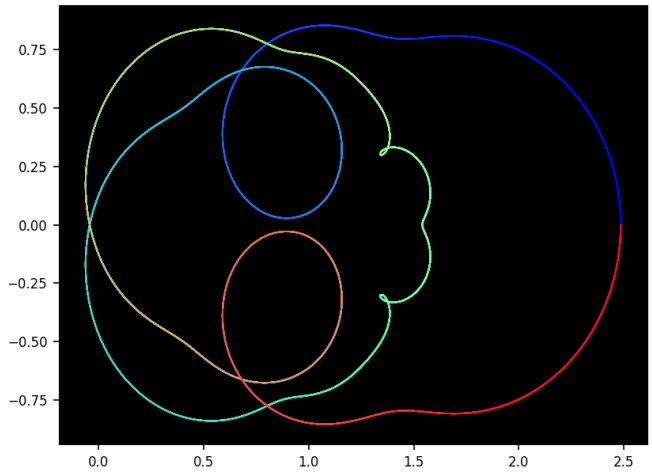


Figure 7.17: $\Gamma_{\rho,\xi}$ with $\rho = 0.80$, $\xi = \frac{24}{37}$

Another way to characterize a digit distribution specific to a number ξ, is pictured in Figure 7.23 representing $G(z, \xi)$ for 4 different ξ, this time with z real in $[-0.75, 0.75]$. These pictures convey more information about the digits than standard analyses, where normal numbers such as π or $\sqrt{2}$ seem indistinguishable when comparing their raw digit distributions. In particular, when ρ approaches 1, the shape of the orbit is an indicator of how scrambled the digits of ξ are.

See below the code to produce the plots just discussed. It contains special characters such as π and ξ, yet they do not cause problems when running Python. Also on GitHub, here.

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import gmpy2

ndigits = 1000
ctx = gmpy2.get_context()
ctx.precision = ndigits

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 'x-small'
plt.figure(facecolor='black')
plt.gcf().set_facecolor("white") # outside the plot
plt.gca().set_facecolor("black") # plot area
```

```
xi1 = gmpy2.const_pi()
xi2 = gmpy2.exp(1)
xi3 = gmpy2.log(2)
xi4 = gmpy2.mpfr(24)/37
arr_xi = [(xi1, 'π'), (xi2, 'exp(1)'), (xi3, 'log(2)'), (xi4, '4/7')]

for m in range(len(arr_xi)):
   value = arr_xi[m]
   xi = value[0]
   xi = gmpy2.mpz(2**(2*ndigits) * xi)
   xi_bin = bin(xi)[2:ndigits+2]
   text = value[1]
   arr_z = []
   arr_G = []
   for x in np.arange(-0.75, 0.75, 0.001):
      sum = 0
      for k in range(ndigits):
         sum += int(xi_bin[k]) * x**k
      arr_z.append(x)
      arr_G.append(sum)
   plt.plot(arr_z, arr_G, linewidth=1.4, label = f"ξ = {text}")
plt.grid(True, color = (0.3, 0.3, 0.3))
plt.legend(loc='upper left', shadow=True)
plt.show()


#--- G(z) orbit on the unit circle

plt.gcf().set_facecolor("white") # outside the plot
plt.gca().set_facecolor("black") # plot area

xi = gmpy2.mpz(2**(2*ndigits) * xi1)
xi_bin = bin(xi)[2:ndigits+2]

## arr_z = []
arr_G_re = []
arr_G_im = []
arr_colors = []
npoints = 200000 # points generated on the circle
radius = 0.99
for t in range(npoints):
   tc = t/npoints
   color = [tc, 4*tc*(1-tc), 0.25+0.75*(1-tc)]
   z_re = radius * np.cos(2*np.pi*tc)
   z_im = radius * np.sin(2*np.pi*tc)
   if t % 1000 == 0:
      print("...",t)
   sum = 0
   for k in range(ndigits):
      sum += int(xi_bin[k]) * complex(z_re, z_im)**k
   arr_G_re.append(sum.real)
   arr_G_im.append(sum.imag)
   arr_colors.append(color)

plt.scatter(arr_G_re, arr_G_im, s=0.8, color = arr_colors, edgecolors = 'none')
plt.show()
```

7.3.2 Connection to Littlewood, Shapiro, and Newman Polynomials

Whether series (7.17) is finite or infinite, for all practical purposes, I work with a finite number of digits. The truncated version defined by (7.18) is a Newman polynomial of degree at most n: one with coefficients in $\{0, 1\}$. They are related to Littlewood polynomials with coefficients in $\{1, -1\}$, and special roots called Salem and Pisot numbers. Special examples include the Shapiro polynomials defined by a simple recursion [Wiki]. The infinite Thue-Morse sequence consisting of 0 and 1, and its generating function [Wiki], also satisfy a simple recurrence. It is used in fractal music. All these polynomials have numerous applications, in particular in signal processing, and are well studied. There are many articles about their complex roots, known to be concentrated around the unit circle. Those with little amplitude on the unit circle are called flat polynomials and involved in the study of the Riemann Hypothesis.

Much of my research focuses on the same topics: the behavior of $G_n(z, \xi)$ when $||z|| = 1$ in section 7.3.1 via the orbit $\Gamma_{\rho,\xi}$. And then its roots in section 7.3.3, to establish a link to normal numbers. Building a set of roots

that leads to random-like digits $d_k(\xi)$ with a known function $G_\infty(z, \xi)$, would be a spectacular accomplishment, and the reverse approach to proving that some known constant ξ is normal. However, it seems out of reach. Even finding a Taylor series with bounded integer coefficients, representing a well-known non-rational function that can be evaluated in closed form at $z = \frac{1}{2}$, seems impossible.

I also looked at what happens when you replace the binary digits 0 and 1 in (7.18) by two arbitrary values not necessarily integer or rational, and most of the properties I am interested in are preserved. An interesting example with digits in $\{-1, 0, 1\}$ showing up in uneven proportions, is the Möbius function [Wiki]. Finally, my techniques to compare rational and irrational numbers, offer an alternative to the irrationality measure [Wiki].

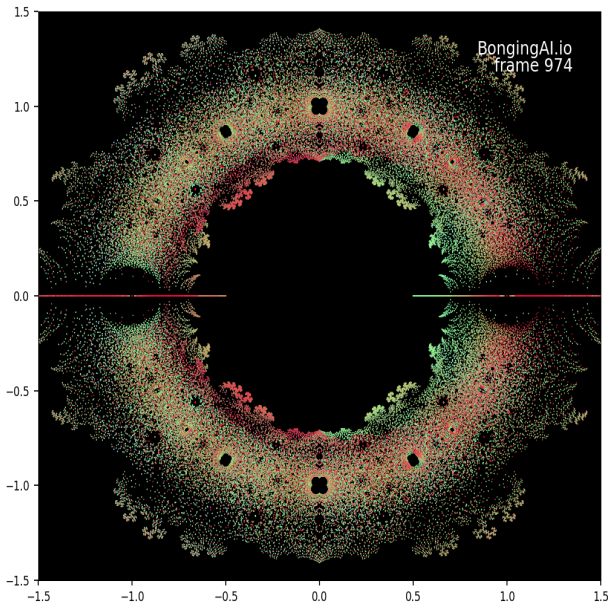


Figure 7.18: Roots of all Littlewood polynomials of degree $n \leq 14$

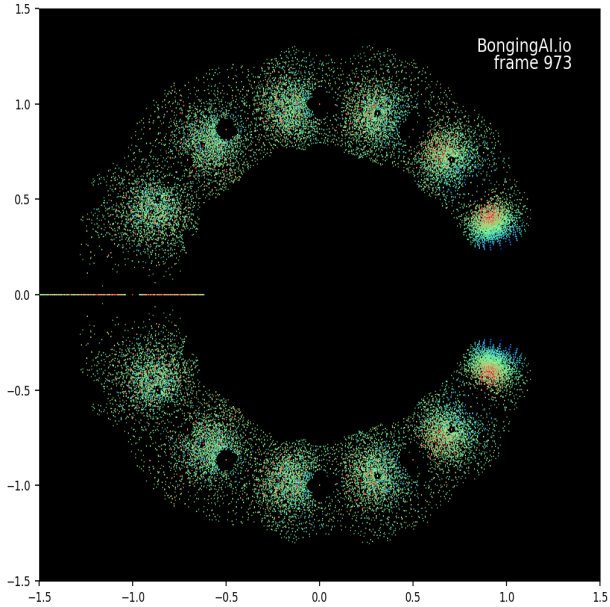


Figure 7.19: Roots of sampled Newman polynomials of degree $n = 14$

7.3.3 Spectral signature of even versus uneven bit strings

Roots of random polynomials have been well studied for decades. Our interest is when the coefficients $d_k(\xi)$ in the polynomial $G_n(z, \xi)$ are distributed as i.id. Bernoulli variables on $\{0, 1\}$ with parameter p. Here $0 < p < 1$ represents the proportion of 1, and the polynomial has degree n: the first and last digits are 1, the other ones are random. The coefficients in question are the binary digits of ξ. We are concerned with the case $n \to \infty$.

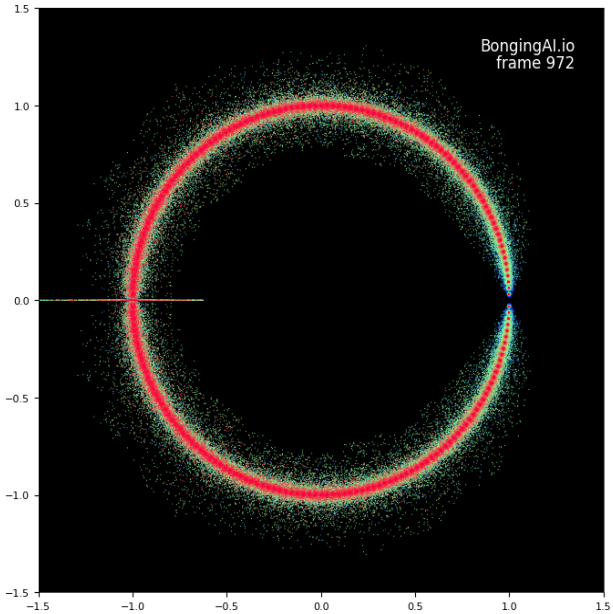


Figure 7.20: Roots of $G_n(z, \xi)$ for 2000 ξ's. p ranges from 0 (blue) to 1 (red); $n = 200$.

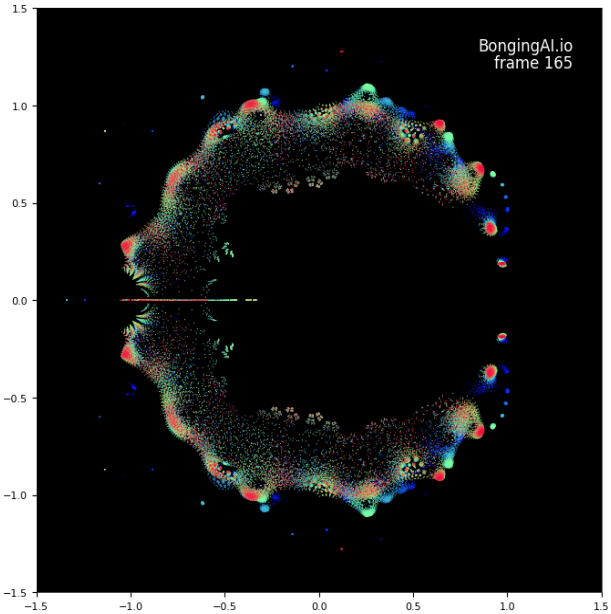


Figure 7.21: Roots of $G_n(z, \xi)$ for 2000 ξ's based on params in frame #165, $n = 35$.

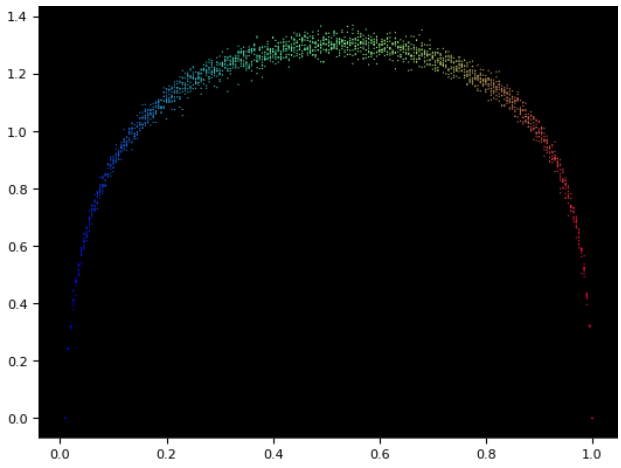


Figure 7.22: $\Delta_\xi(p)$ with p on the X-axis, showing a peak when the proportion of 1 is around $p = \frac{1}{2}$.

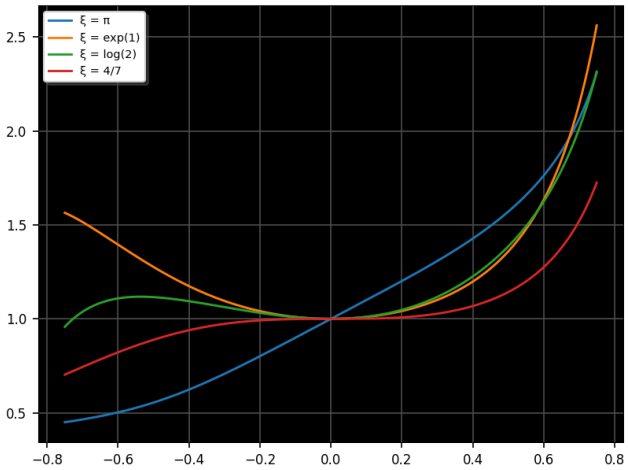


Figure 7.23: $G(z, \xi)$ with $-0.75 < z < 0.75$, for $\xi = \pi, e, \log 2$ and $4/7$, and z on X-axis

The idea is as follows: if ξ is not a normal number in base 2, more specifically if $p \neq \frac{1}{2}$ and departure from normality is strong enough to be detectable in the roots, then it should show up in the root distribution of the digit generating function $G(z, \xi)$ and its polynomial approximation $G_n(z, \xi)$. The literature on random polynomials focuses on coefficients – the binary digits in our case – distributed as Gaussian with zero mean or Bernoulli(p) on $\{-1, 1\}$, but rarely on $\{0, 1\}$. Many results are distribution-independent but some assume that the coefficients have zero expectation. When the expected proportion of 1 is $p = \frac{1}{2}$, the following is known:

- At the limit as $n \to \infty$, the complex roots are uniformly distributed on the unit circle, see [28, 42, 44].
- The proportion of real roots (among all n roots) tends to zero as $n \to \infty$, see [50].
- Complex roots repel each other: the chance of a multiple root is zero, see [31].

Based on simulations (no proof), I show that the situation is much different when p is close to 0 or 1. It provides a new mechanism to test whether a number ξ satisfies some of the requirements for normality in base 2. There are many potential indicators in the root distribution to flag non-normality. The one I chose as the most striking is the scaled average absolute distance

$$\Delta_\xi = \lim_{n\to\infty} \frac{\lambda(n)}{n} \sum_{k=0}^{n-1} \Big|1 - ||\rho_k(n)||\Big| \quad \text{with } \lambda(n) = \sqrt{n}\log n, \tag{7.21}$$

computed on the n roots $\rho_0(n), \ldots, \rho_{n-1}(n)$ in (7.19) and looking at the limit $n \to \infty$. Here $||\cdot||$ represents the modulus of a complex number. Also, the scaling factor $\lambda(n)$ counteracts the fact that the roots drift towards the unit circle as n increases. While leading to a correct conclusion in all my tests, it is still subject to refinement and theoretical investigation. Now I visually show the link between the roots of $G(z, \xi)$ and the proportion of 1 in the digits of the number ξ. The associated Python code is in section 7.3.4.

- For a random number ξ generated with a proportion p of 1 in its binary digit expansion, let's replace the notation Δ_ξ by $\Delta_\xi(p)$ to put the emphasis on p. Figure 7.22 shows $\Delta_\xi(p)$ with p on the X-axis. I tested 2000 values of p evenly spaced between 0 and 1, with one random ξ for each p (that is, one ξ with a proportion p of 1). Each ξ has n digits, thus the degree of the polynomials is n, here with $n = 200$ specified by the parameter `N` in the code. Clearly, $\Delta_\xi(p)$ is maximum when $p = \frac{1}{2}$. Thus Δ_ξ is an indicator of normality (or lack of, to be precise). In other words, the roots behave very differently if too few or too many polynomial coefficients are 1.
- Figure 7.20 shows the combined roots of all the K polynomials for the same set of ξ with the same degree n. The number of polynomials is set to `K=2000` in the code. Thus, there is a total of $n \times K$ roots in the plot, each represented by a pixel. The few real roots are on the horizontal segment. The color ranges from red when the proportion p of 1 is close to 1, to blue when p is close to 0. Green corresponds to middle values closer to $p = \frac{1}{2}$. Green roots are more spread out compared to red ones, thus with a higher $\Delta_\xi(p)$ in Figure 7.22. Blue roots are more concentrated near $(1, 0)$ while red roots are distributed in a narrow annulus close to the unit circle. Thus, both blue and red roots have a lower $\Delta_\xi(p)$ compared to the green ones, reflected in Figure 7.22.

The roots of Newman polynomials of degree n mostly lie in n small discs with centers evenly spread across the unit circle. This well-known phenomenon (roots repelling each other) is visible when $n \leq 20$. You can

see it in Figure 7.19. By contrast, the classic representation [Wiki] lacks this feature because it shows all polynomials instead of sampled ones. Figure 7.18 is a replica of the iconic Wikipedia picture. The color shows the corresponding ξ. Because Figure 7.19 is based on coefficients in $\{0, 1\}$ instead of $\{1, -1\}$, it misses the disc at $(1, 0)$ and the positive real roots found in Figure 7.18. To produce Figure 7.18, set `mode='full'` in the code along with `K=2**N` (2 at power `N`), and finally `a=2` and `b=-1` if you want the coefficients to be in $\{1, -1\}$ rather than $\{0, 1\}$. The code is listed in section 7.3.4.

7.3.4 Mesmerizing video featuring 1000 cases, with Python code

Figures 7.18, 7.19, 7.20, 7.21 featuring the roots and Figure 7.22 for the spectrum are produced by the code in section 7.3.4 (this section) while Figures 7.14, 7.15, 7.16, 7.17 and 7.23 linked to the orbits are generated by the code in section 7.3.1. The code in this section also produces a spectacular video with 975 frames numbered from 0 to 974. Each frame shows a root configuration for a specific set of ξ. Also the binary digits of ξ are the coefficients of the associated polynomial. You can watch the video on YouTube, here.

The special frames 974, 973 and 972 are displayed respectively in Figures 7.18, 7.19, 7.20 and discussed in section 7.3.2. All the other ones are based on setting `mode='deterministic'`, including frame 165 shown in Figure 7.21. Each one features the combined roots of K polynomials $G_n(z, \xi_1), \ldots, G_n(z, \xi_K)$ with $\xi_k = f(k; \theta)$. In the code, the function f is denoted as `f_bin` and the bivariate parameter $\theta = (\theta_1, \theta_2)$ is denoted as `options`; $f(k, \theta)$ returns the first n binary digits of ξ_k, which are also the coefficients of $G_n(z, \xi_k)$. Finally, ξ_k is built as follow:

- First, θ_1 allows you to select a number $\xi \in \{\pi, e, \log 2, \frac{4}{7}\}$. For instance, if $\theta_1 = 0$ then $\xi = \pi$.
- Then, $\xi_k = \xi + k$, $\xi + k^2$, $\xi + \frac{1}{k}$ or ξk depending on whether $\theta_2 = 0, 1, 2$ or 3.

The digits of ξ and ξ_k are quite similar but shifted to the left of right with some perturbation at the beginning or at the end depending on the operation, except for the multiplication $\xi_k = \xi k$ which introduces significant scrambling. As a result, in a given video frame, the roots of $G_n(z, \xi_k)$ for $k = 1, \ldots, K$ are somewhat correlated, producing a spectacular distribution as in Figure 7.21. It consists of a fractal-like border loosely following the unit circle, with most roots peppered either inward or outward depending on θ_2. Unless $\theta_2 = 3$ ($\xi_k = k\xi$) where the scrambling produces a dull image.

./...	k	p	Checksum	Digits of ξ_k
0.050	100	0.543	1.00000	11001001011000101110010000101111111
0.100	200	0.514	1.00000	11001000101100010111001000010111111
0.150	300	0.514	1.00000	10010110010110001011100100001011111
0.200	400	0.486	1.00000	11001000010110001011100100001011111
0.250	500	0.571	1.00000	11111010010110001011100100001011111
0.300	600	0.486	1.00000	10010110001011000101110010000101111
0.350	700	0.543	1.00000	10101111001011000101110010000101111
0.400	800	0.457	1.00000	11001000001011000101110010000101111
0.450	900	0.486	1.00000	11100001001011000101110010000101111
0.500	1000	0.543	1.00000	11111010001011000101110010000101111
0.550	1100	0.457	1.00000	10001001100101100010111001000010111
0.600	1200	0.457	1.00000	10010110000101100010111001000010111
0.650	1300	0.457	1.00000	10100010100101100010111001000010111
0.700	1400	0.514	1.00000	10101111000101100010111001000010111
0.750	1500	0.543	1.00000	10111011100101100010111001000010111
0.800	1600	0.429	1.00000	11001000000101100010111001000010111
0.850	1700	0.486	1.00000	11010100100101100010111001000010111
0.900	1800	0.457	1.00000	11100001000101100010111001000010111
0.950	1900	0.543	1.00000	11101101100101100010111001000010111

Table 7.3: Prop. p of 1 in digits of 20 sampled ξ_k, video frame 165

The colors range from blue ($k = 0$) to red ($k = K$) with green for middle values. The fact that each frame has areas with strongly distinct colors – as opposed to colors randomly distributed – means that close polynomials, that is $G_n(z, \xi_k)$ and $G_n(z, \xi_l)$ with $|k - l|$ small, have at least some roots that are also close. Again, this is not true if $\theta_2 = 3$; in that case, index proximity does not translate into coefficients proximity due to the scrambling induced by the multiplication operator.

Besides the `f_bin` options θ_1, θ_2 applicable only to `mode='deterministic'`, there are two other parameters a, b available in all modes. The default is $a = 1, b = 0$. Any other values, integer or not, positive or not, change the binary digit $d \in \{0, 1\}$ into $d^* = ad + b \in \{a + b, b\}$. Or equivalently, it changes the polynomial

$G_n(z,\xi)$ with coefficients $d_k(\xi), k = 0,\ldots,n$ into the polynomial $G_n^*(z,\xi)$ with coefficients $d_k^*(\xi) = ad_k(\xi) + b$. The following result is straightforward:

$$G_n^*(z,\xi) = a\,G_n(z,\xi) + b(1 + z + \cdots + z^n). \tag{7.22}$$

Roots of $G_n(z,\xi)$ and $G_n^*(z,\xi)$ are connected as follows. Let u, v be two roots of $G_n(z,\xi)$, and u_*, v_* two roots of $G_n^*(z,\xi)$. Then, for any fixed ξ, by virtue of (7.22) we have

$$\frac{G_n^*(u,\xi)}{1+u+\cdots+u^n} = \frac{-a\,G_n(u_*,\xi)}{1+u_*+\cdots+u_*^n} \tag{7.23}$$

$$\frac{G_n^*(u,\xi)}{1+u+\cdots+u^n} = \frac{G_n^*(v,\xi)}{1+v+\cdots+v^n} \tag{7.24}$$

$$\frac{G_n(u_*,\xi)}{1+u_*+\cdots+u_*^n} = \frac{G_n(v_*,\xi)}{1+v_*+\cdots+v_*^n}. \tag{7.25}$$

Note that (7.23) does not depend on b. Then, the formulas (7.24) and (7.25) do not depend on either a or b, leading to universal rather than model-specific properties.

The Python code displays a statistical summary after each generated frame. It consists of the parameters $N, K, a, b, \theta_1, \theta_2$ attached to the frame in question, as well as Table 7.3, in this example for frame 165 shown in Figure 7.21. The `checksum` value should always be 1 unless the polynomial coefficient attached to z^n is zero. A value not close enough to 1 indicates lack of accuracy when computing the roots. To fix it, increase the precision or check other parameters (number of steps and so on) in the root–finding algorithm `polynomial.polyroots` from the Numpy library. With my parameter set, none of the generated frames faces this problem. To speed up the computations, process the polynomials in parallel. Also, the code is designed to resume without losing previously generated images if it crashes at any time for any reason. All the results and illustrations are fully replicable. The code is also on GitHub, here.

This research is part of a project at BondingAI.io to build an AI agent specializing in scientific and high-performance computing, as well as AI–generated art. Also, one of the goals is to share beautiful mathematical results with the public at large, thanks to the video and the connection to polynomial roots. The latter is familiar to most high school students but the classroom material is usually dry and lacks exciting experiments. I hope that my contribution fills this gap and will boost the interest in mathematics, probability, and engineering, as these fields are typically considered as unwelcoming by many.

Finally, my work raises an interesting question: when you automatically generate thousands of examples (frames here,) can AI select which ones are the most beautiful and enriching, to include them in the final video? The answer is yes if you train AI, but can it do it well with little training and detect on its own a set of diversified images that are universally considered spectacular or enlightening by everyone? Finally, if you are interested in adding an AI-generated mathematical soundtrack (music) to my video, matching the frames, please contact me at vincent@bondingAI.io.

```
import random
import numpy as np
import gmpy2
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['legend.fontsize'] = 'x-small'

def extract_roots(N, xbin, a, b):

    arr_re = []
    arr_im = []
    delta = 0
    prop = xbin.count('1')/N
    arr_coeffs = []
    for k in range(N):
       digit = int(xbin[k])
       arr_coeffs.append(a*digit+b)
    # coefficients in that order: 1 + 3x + 5x^2 --> [1, 3, 5]
    roots = np.polynomial.polynomial.polyroots(arr_coeffs)

    checksum = 1
    for r in roots:
       re = r.real
```

```
        im = r.imag
        norm = (re**2 + im**2)**0.5
        arr_re.append(re)
        arr_im.append(im)
        delta += abs(norm-1)
        checksum *= norm
    if len(roots) == 0:
        delta = 0
    else:
        delta /= len(roots)
    return(arr_re, arr_im, delta, checksum)


def generate_bit_string(N, prop):

    num_ones = int(0.5 + (N-2) * prop)
    num_zeros = (N-2) - num_ones
    bits = ['1'] * num_ones + ['0'] * num_zeros
    random.shuffle(bits)
    xbin = "".join(bits)
    xbin = '1' + xbin + '1'
    return(xbin)


def f_bin(k, N, options):

    constant = options[0]
    formula = options[1]

    if constant == 0:
        cst = gmpy2.const_pi()
    elif constant == 1:
        cst = gmpy2.exp(1)
    elif constant == 2:
        cst = gmpy2.log(2)
    elif constant == 3:
        cst = gmpy2.mpfr(4)/gmpy2.mpfr(7)

    if formula == 0:
        eta = cst + k
    elif formula == 1:
        eta = cst + k*k
    elif formula == 2:
        eta = cst + gmpy2.mpfr(1)/k
    elif formula == 3:
        eta = cst * k

    x = gmpy2.mpz(2**(2*ndigits) * eta)
    xbin = bin(x)[2:N+2]
    return(xbin)


#--- Core function

def set_of_polynomials(K, N, a, b, mode, f_options):

    arr_delta = []
    arr_prop = []
    arr_col = []

    for k in np.arange(1, K):

        p = k/K
        color = [p, 4*p*(1-p), 0.25+0.75*(1-p)]
        if mode == 'random':
            xbin = generate_bit_string(N, p)
        elif mode == 'deterministic':
            xbin = f_bin(k, N, f_options)
        elif mode == 'full':
            string = bin(k)[2:]
            xbin = string + "0" * (N - len(string))

        p1 = xbin.count('1')/N # should be close to prop
        arr_re, arr_im, delta, checksum = extract_roots(N, xbin, a, b)

        # checksum (product of all roots) must be equal to 1
```

```
104         if k % 100 == 0:
105            string = xbin
106            if len(string) > 60:
107               string = string[:60] + "..."
108            print("%5.3f %5d %5.3f %9.7f %s" % (p, k, p1, checksum, string))
109
110         arr_prop.append(p1)
111         arr_delta.append(np.sqrt(N)*np.log(N)*delta)
112         arr_col.append(color)
113         ax.scatter(arr_re, arr_im, s=0.6, color=color, edgecolors='none')
114
115      return(arr_prop, arr_delta, arr_col)
116
117
118   #--- Main
119
120   width_px, height_px = 800, 800
121   dpi = 100
122   width_in = width_px / dpi
123   height_in = height_px / dpi
124   random.seed(42)
125
126   show_axes = False
127   show_spectrum = False
128   save_spectrum = True
129   show_roots = False
130   save_roots = True
131   saved_frames = []
132
133   """
134   param entry: [N, K, a, b, mode, frame_ID, f_options]
135
136       • N: number of digits (polynomial degree < N)
137       • K: number of polynomials used in a set or video frame
138       • a, b: binary digit replaced by a*digit+b
139       • mode: 'deterministic' or 'random'
140       • frame_ID: index number attached to video frame
141       • f_options: parameters (u, v) used in function f_bin:
142          • u: 0 for Pi, 1 for exp(1), 2 for log(2), 3 for 4/7
143          • v: 0 for u+k, 1 for u+k*k, 2 for u+(1/k), 3 for u*k,
144   """
145
146   params = []
147   frame_ID = 0
148   mode = 'deterministic'
149   K = 2000
150   a = 4
151
152   for b in range(-7, 2, 1):
153      for u in range(0, 4):
154         for v in range(0, 3):
155            for N in range(20, 61, 5):
156               f_options = (u, v)
157               param = [N, K, a, b, mode, frame_ID, f_options]
158               params.append(param)
159               frame_ID += 1
160
161   N = 200
162   K = 2000
163   param = [N, K, 1, 0, 'random', frame_ID, (0,0)]
164   params.append(param)
165   frame_ID += 1
166
167   N = 14
168   K = 12000
169   param = [N, K, 1, 0, 'random', frame_ID, (0,0)]
170   params.append(param)
171   frame_ID+=1
172
173   N = 14
174   K = 2**N
175   param = [N, K, 2, -1, 'full', frame_ID, (0,0)]
176   params.append(param)
177   frame_ID+=1
178
179   nframes = len(params)
```

```
print(nframes, "frames")

for param in params:

   N = param[0]
   K = param[1]
   a = param[2]
   b = param[3]
   mode = param[4]
   frame_ID = param[5]
   f_options = param[6]

   # ndigits (precision) must be larger than N
   ndigits = 5*N // 4
   ctx = gmpy2.get_context()
   ctx.precision = ndigits

   if frame_ID in (0, 165, nframes-3, nframes-2, nframes-1):

      print("\n----------------\nFrame:", frame_ID, "/", nframes)
      print("\ndegree: N = %d\nnumber of ξ's: K = %d\na: %7.4f\nb: %7.4f\nmode: %s\nf_options: %s"
          %(N, K, a, b, mode,str(f_options)))
      print("\n./... k  p   checksum digits of ξ_k")
      if not show_axes:
         plt.style.use('dark_background')
      fig, ax = plt.subplots(figsize=(width_in, height_in))
      arr_prop, arr_delta, arr_col = set_of_polynomials(K, N, a, b, mode, f_options)
      ax.text(0.95, 0.95, 'BongingAI.io', transform=ax.transAxes, ha='right', va='top',
          fontsize=12, color='white')
      ax.text(0.95, 0.92, 'frame ' + str(frame_ID), transform=ax.transAxes, ha='right', va='top',
          fontsize=12, color='white')

      plt.xlim(-1.5, 1.5)
      plt.ylim(-1.5, 1.5)
      if not show_axes:
         plt.axis('off')
         plt.margins(0)
         plt.subplots_adjust(top=1, bottom=0, right=1, left=0, hspace=0, wspace=0)
      else:
         ax.set_facecolor('black')
      if save_roots:
         filename = 'nroots' + str(frame_ID) + '.png'
         plt.savefig(filename, bbox_inches='tight', pad_inches=0, dpi=dpi)
         saved_frames.append(filename)
      if show_roots:
         plt.show()
      else:
         plt.close()

      plt.rcParams['axes.facecolor'] = 'black'
      plt.scatter(arr_prop, arr_delta, s=0.8, c = arr_col, edgecolors = 'none')
      if save_spectrum:
         filename = 'nroots_spectrum' + str(frame_ID) + '.png'
         plt.savefig(filename, bbox_inches='tight', pad_inches=0, dpi=dpi)
      if show_spectrum:
         plt.show()
      else:
         plt.close()


#--- Produce video

import moviepy.video.io.ImageSequenceClip
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(saved_frames, fps=4)
clip.write_videofile('nroots_v1.mp4')
```

Chapter 8

Towards Deterministic AI: The NPG Random Generator

The term deterministic AI has gained considerable traction recently, especially in circles where reproducible results are a necessity. It does not mean the absence of algorithmic randomness, but instead, full control over the stochastic components, to allow for replicability. Yet, at the same time, quantum random generators go in the opposite direction, guaranteeing the lack of replicability as a key benefit.

Anyone who ever debugged code knows how hard it is to fix bugs when getting different results after running the same program twice without any changes. It takes time, makes fine-tuning really hard, and thus it is costly for your company. In addition, non-replicable scientific research published in top journals leads to mistrust by the general public. Thus the growing need to better control randomness.

Here I propose a simple, efficient and very fast random number generator (PRNG) to solve this problem. It is derived from the material discussed earlier in this book, and fits in the same category as PRNGs based on irrational numbers (see section 5.5) but much faster. It is also much faster, simpler and stronger than standards PRNGs found in most libraries. Aloso, as long as you use the same seeds, the output does not change if you install a different version of Numpy or Python, or work with a different programming language on any system.

8.1 NPG: fast non-periodic pseudo-random number generator

Many if not most PRNGs rely on a seed x_0 and an iterative formula $x_{n+1} = f(x_n)$ to generate the random bit strings x_1, x_2 and so on. The function f is usually static: it does not depend on n. Even if these components are multi-dimensional, it leads to periodicity and other patterns. By constrast, my system is based on a function f_n that depends on n, not involving any congruence or modulus, and specified by

$$x_n = a2^{\lambda_n} x_{n-1} + bx_{n-1} + cx_{n-2} \quad \text{(here, } c = 0\text{)}. \tag{8.1}$$

From now on, unless otherwise specified, $a, b \geq 0$ are distinct odd primes, $c = 0$, $x_0 = 1$, and $\lambda_n = n$. A related function is

$$G_n(z) = \prod_{k=1}^{n} \left(az^{\lambda_k} + b\right). \tag{8.2}$$

Clearly, $x_n = x_0 \cdot G_n(2)$. If $||z|| < 1$ and $b = 1$, then $G_n(z)$ converges as $n \to \infty$. This is true if $z = \frac{1}{2}$. In the remainder of this section, I focus on the binary digits of $G_n(2)$. For specific a, b values, my random bit string is obtained by concatenating the binary digit sequences of x_1, x_2 and so on, with some minor twists. I named my new system NPG: it stands for non-periodic generator.

8.1.1 NPG non-standard fundamental examples

One of the most basic and interesting cases is when $a = b = 1$. I call it non-standard because a, b are not prime numbers. Unlike the polynomials $G_n(z, \xi)$ in section 7.3, when $a = b = 1$ those defined by (8.2) have all their complex roots on the unit circle. Also, the binary digits of $G_n(2)$ and $G_n(\frac{1}{2})$ are now identical. Thus, in some sense, both converge to a same value as $n \to \infty$, with $G(2)$ being the p-adic representation of $G(\frac{1}{2})$ with $p = 2$. That is, with infinitely many digits on the left rather than on the right. Here $G(z)$ is defined as

$$G(z) = \lim_{n \to \infty} G_n(z).$$

As a side effect, the first n digits of $G_n(2)$ and $G_{n+1}(2)$ are always identical. The same is true for the last n digits. This, combined with theorem 5.5.1, creates small cross-correlations between the digits of two consecutive iterates x_n and $_{n+1}$. These correlations quickly decay as n increases, and there is no obvious symmetry in the digit sequences.

Because of that, for each n, you want to skip both the first n and last n bits of x_n when concatenating the binary digits of x_1, x_2 and so on. And also drop the first 500 iterates $x_1, \cdots, x_{500}$. Note that x_n has about $n^2/2$ digits. Thus if you concatenate $x_1, \cdots, x_N$, dropping the first 500 iterates amounts to ignoring a proportion $(500/N)^3 = 0.80\%$ of all the generated digits when $N = 2500$. Also, skipping $2n$ digits for each iterate x_n results in losing a proportion $6/N = 0.24\%$ of all digits in our example. So very little is lost, and the proportion drops to zero as N increases. Even without removing any digit, the concatenated bit strings pass the standard tests of randomness, a proof that these tests can be fooled. You remove the digits in question primarily to increase randomness.

The case $a = b = 1$ leads to very fast computations, as formula (8.1) simplifies to `x+=x<<n` in Python. In short, just one addition and a bit shift. On my computer, it runs 3x faster than the PCG64 generator from the Numpy.rng library, thanks to using Gmpy2 for high-performance computing. Finally, the corresponding $G(z)$ is the well-known partition function [Wiki].

Before moving to the general case, let me share another nonstandard example: $a = 3$, $b = 0$, this time with $\lambda(n) = 0$. It leads to $x_n = 3^n$. Everyone who studied its binary digits is convinced that they look perfectly random when n is large, except again for a few digits at the beginning and at the end. Whether they are random or not is a subject of debate, and it remains one of the major unanswered mathematical conjectures. Of course, they are unfit for cryptography, for other reasons. And you cannot concatenate the x_n's because the digits of x_n and x_{n+1} are cross-correlated. The correlation is $\frac{1}{3}$ as $n \to \infty$, a direct consequence of theorem 5.5.1. Then, if you replace 3^n by (say) 983^n, the correlation would be $\frac{1}{983}$. It would be interesting to see if statistical tests of randomness can detect such a weak pattern. More about the digits of these numbers in section 7.2.4.

8.1.2 Deterministic vs cryptographically secure NPG vs other PRNGs

In the general model, $a, b > 2$ are prime numbers. Also, $x_0 = 1$ and $\lambda(n) = n$. These specifications allow you to run multiple sequences in parallel, with a guarantee that they are free of cross-correlations. For each sequence (x_n), the parameters a and b play the role of a bivariate seed. The main differences with standard PRNGs are as follows:

- At iteration n, the state consists of a, b, n and x_n. This is what you need to generate x_{n+1}. By contrast, in standard PRNGs, n is not part of the state. This fundamental difference results in better randomness and infinite period. Also the initial condition – called seed in other PRNGs – is set to $x_0 = 1$ here, and thus static and not part of the seed. The actual "seed" is the vector (a, b) and part of the state. In classic PRNGs, the seed is not an element of the state.
- The size of x_n grows quadratically: x_n has about $n^2/2$ binary digits. In standard PRNGs, x_n has a fixed number of digits and thus, the state has a fixed size. Also, a, b can be large integers with more than 200 binary digits, without impacting the speed. All the arithmetic is performed in big integers, well beyond 64 or even 1024 bits, with the Gmpy2 library. Typically, $n \leq 2500$. The total number of bits produced after n iterations is about $n^3/6$.
- There is no modulo operations involved, no integer division either. Only one addition, one bit shift, and at most two multiplications. Also, almost all the generated digits are used. This explains the speed (much faster than PCG64) despite working with extremely large integers.
- You must exclude the first and last $2n$ bits of each x_n when returning bits to the user. Also, do not return bits from the first iterates $x_1, \ldots, x_{500}$. Even if you ignore the first requirement, the generated bits still pass all my tests while PCG64 failed on a few occasions. This shows that statistical tests of randomness can be fooled and should not be the sole criterion to determine the quality of a PRNG. The total number of bits created over n iterations is of the order $O(n^3)$, while the number of discarded bits is $O(n^2)$.
- If a user needs fixed-size random integers coming one at a time, it is easy to build a wrapper that will hide the complexity, extracting the desired number of bits from x_n or generating the next x_{n+1} as needed, transparently to the user. In this case, the number of bits currently retrieved from x_n is part of the PRNG state. A new x_{n+1} is generated only after exhausting all the bits in x_n. Then x_n is deleted from memory.

My system is designed for industrial consumption where you need from billions to quadrillions of high quality, replicable random bits. For instance in LLMs or heavy simulations. But is my system suitable for cryptography? The answer is no because it is probably the easiest to inverse: finding all past and future x_n very quickly if you

are able to access the PRNG state at any time. Indeed, inversion is trivial with the formula

$$x_{n-1} = \frac{x_n}{a2^{\lambda(n)} + b}. \tag{8.3}$$

This may be a quality in some contexts, but not in cybersecurity! Of course, if you remove the required bits on the left and right as required in the specifications, retrieving x_{n-1} from x_n is a little more difficult, but not enough to guarantee minimum security. And anyway, you remove these bits to increase randomness, but not for security reasons. To significantly increase security without impacting speed or statistical randomness, proceed as follows.

- Rather than fixed a and b, use sequences $A = [a_1, a_2, \dots]$ and $B = [b_1, b_2, \dots]$ where a_n, b_n randomly changes at each iteration n, while still being prime numbers. You can also apply some small randomization to the function λ, renamed λ_* after the change. Let's $\Lambda = [\lambda_*(1), \lambda_*(2), \dots]$.
- The randomization can be weak, low quality. After all, it works well with no randomization! But it is important to not include $a_n, b_n, \lambda_*(n)$ in the current PRNG state, and to not save these values right after the one-time use. This will make the system non-replicable, but it will be impossible to crack it.
- The source of randomization should be non-reproducible, based on external data such as the lowest bits of time stamps, memory/disk usage at the time they are generated, or even random signals from outside your environment. Better: use a combination of all these elements.

With these changes, we have

$$x_n = \prod_{k=1}^{n} \left(a_k 2^{\lambda_*(k)} + b_k\right) = x_1 x_2 \cdots x_{n-1} \times \left(a_n 2^{\lambda_*(n)} + b_n\right). \tag{8.4}$$

It is still a basic product, albeit with factors that are 10^5 to 10^6 bit integers and typically not prime. The latter makes it harder to break. Also, x_n has at least 500 missing bits on both ends since $x_1, \cdots, x_{500}$ are ignored and nowhere stored. In the deterministic version, the parameters are fixed, thus the factors in (8.4) are ordered by size, making it easy to crack. But that is not true in the secure version. Even if you manage to factor x_n, get the correct factors out of many possibilities, you still need to get them in the correct order, a problem of combinatorial complexity. If each x_k has the requested digit sequence truncation, it is even harder. Guessing past or future iterates is just as hard. This makes NPD a strong PRNG and cryptographically secure.

I am now working on a different version where $\lambda_*(n) = 0$ and $b_n = 0$ with (say) 256 bits for a_n. All the x_n's will be truncated to a same size while keeping as many bits as possible to preserve good speed. Formula (8.1) will reduce to a single product: $x_{n+1} = a_n x_n$ along with very fast truncation left and right (no modulo). This is a work in progress.

8.1.3 NPG success vs PCG64 and PCG64DXSM statistical failures

I tested both PCG64 (the new PRNG integrated into Numpy and much software) and my system, for comparison purposes, using a number of statistical tests of randomness. In my case, the bit strings tested consist of the last two iterates x_{N-1} and x_N, for various a, b. It passed all the tests even when not removing the required digits on the left and right. Indeed, the concatenated bits of 3^{N-1} and 3^N also passed these tests (barely) even though the correlation between the two is about $\frac{1}{3}$. By contrast, I experienced a few failures both with PCG64 and its upgraded version PCG64DXSM for parallel architecture. I describe the tests and failures in this section, with the Python code in section 8.2. For the story, PCG64 replaces the Mersenne Twister in Numpy.

Seed	a	b	Bits	Failed_Tests	Strings	PRNG
49	53	31	1,005,414	spectral	B_1	PCG64DXSM
49	53	31	1,005,414	max_hits	B_7	PCG64
42	1	1	994,001	spectral	B_7	PCG64
43	1	1	994,001	spectral	B_5	PCG64
47	1	1	994,001	spectral	B_5, B_6	PCG64
42	1	1991	994,115	records	B_2	PCG64
42	1	1991	994,115	spectral	B_4	PCG64
42	1	1991	994,115	next_bit	B_2	PCG64

Table 8.1: Statistical failures with seed in the 42–50 range

Let ν be the length of the bit string x_{N-1} concatenated with x_N. It depends on a, b and N. I generated $M = 8$ bit strings $B_0, \ldots, B_7$ of length ν, for K different combinations of a and b. For a specific a, b, the first one B_0 comes from my system, the other ones are generated sequentially by PCG64 with the Numpy seed set by `rng_seed`. For now, assume that $K = 1$ and $N = 1000$. In the code, K is implicitly detected as the number of elements (parameter sets a, b) in the `params` array. Also, the first and last few bits of each bit string are discarded, by setting `skip_flag=True`. Of course, a, b have no direct impact on the success or not of PCG64, but ν (a function of a, b) does. The failures are listed in Table 8.1, with ν denoted as "Bits" and `rng_seed` denoted as "Seed". The latter is used for PCG64, not in my system. The "seed" in my system (in the classical sense) is $x_0 = 1$, the lowest possible value, and not listed in the table. Instead, a, b play that role.

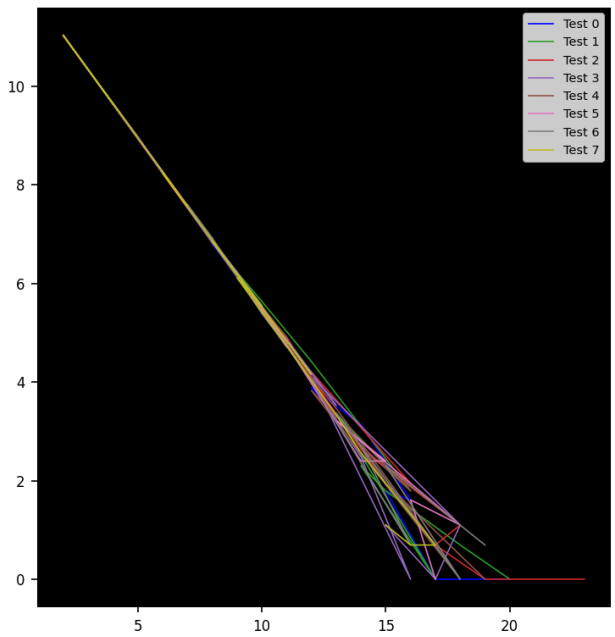


Figure 8.1: Runs of length L; L on X-axis, count (log scale) on Y-axis. Blue for NPG, non-blue for PCG64.

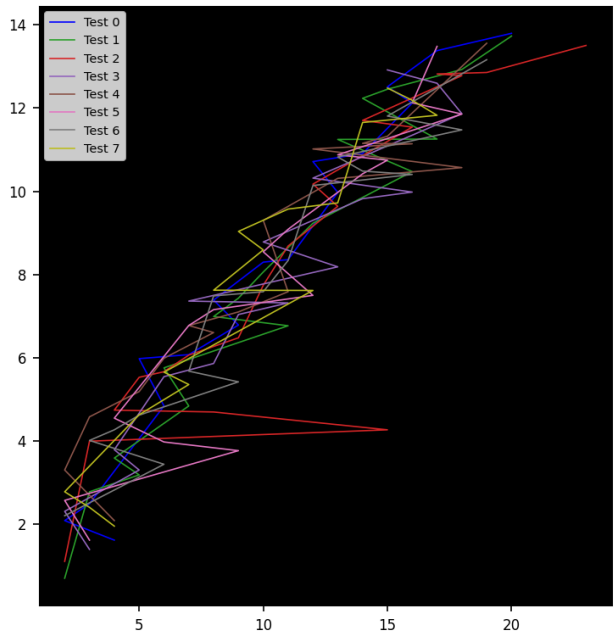


Figure 8.2: Same as Fig. 8.1 but with first arrival of run of length L on Y-axis (log scale), L on X-axis

Failures, while relatively rare, happen frequently enough to be a concern for PCG64. But no failure was found for my system (the absent B_0 string in the table). The Python code in section 8.2 allows for full replication. In Table 8.1, the 'max_hits' issue is part of the collision test while 'records' (Figure 8.2) is part of the run test. Both are described below.

The statistical tests of randomness used here cover a small portion of battery suites such as BigCrush, DieHarder, PractRand or NIST. I selected them for their generality, covering multiple tests at once while being relatively independent. I also developed several strong tests not found in these suites, for instance the Weyl test described in chapter 5. Now, here are the tests that I chose to produce Table 8.1, where only failures are reported:

- The frequency test computes the proportion of any substring of length at most `Lmax` in a bit string. Thus it acts as a test of independence in dimensions up to `Lmax`. It is highly correlated to the entropy test. Note that in the string '10001', the number of '00' is 2 according to my test (correct), while the default Python count is 1. This test, in its form, is also known as the equidistribution or block test.
- The autocorrelation test computes the maximum of the absolute value of all autocorrelations up to lag `maxLag`, in any bit string. It summarizes the autocorrelation function and is thus a spectral test. However, I also included the NIST spectral test based on Fourier transforms. PCG64 failed that test a few times.
- The compression test compresses a bit string with the powerful LZMA compressor. It did not manage to compress any of the bit strings tested, whether coming from PCG64 or NPG. It shows that those strings have high entropy, compatible with good randomness.
- The collision test computes the number of substrings of length `L` occurring more than once in a bit string, especially when `L` is large and few collisions are expected. It also returns max_hits, the maximum number of times one of those rare substrings occurs. That's where PCG64 showed some failures.
- The run test computes all the runs of 0 of various lengths (Figure 8.1) and their first arrival in the bit string (Figure 8.2). Test k ($k = 0, \ldots, 7$) corresponds to the bit string B_k in Table 8.1, with B_0 for NPG and the other indices for PCG64, here with $a = 1$, $b = 1991$ and the PCG64 seed set to 42. In most cases,

no anomaly was found, neither for NPG nor PCG64. The case featured here is an exception with B_2 (in red, PCG64) out of range. A future test could compute the correlation between the lengths of successive runs.

- The next-bit test checks if you can predict the next bit in a bit sequence, fast enough while outperforming pure chance. I use deep neural networks (DNNs) for this task, in a way similar to how they are used to predict the next token in large language models. In my tests, DNNs cannot do better than random guessing for bit streams generated by NPG. By contrast PCG64 shows weaknesses on occasions depending on the test configuration

It is easy to build a bit sequence that not only passes all equidistribution tests [Wiki], but also mathematically proved to do so. This is the case for the Mersenne Twister PRNG, a precursor to PCG64, itself a precursor to my NPG system. It is proved to be 623-dimensional equidistributed up to 32 bits accuracy. Even worse, the Champernowne constant [Wiki] is proved to be equidistributed in any dimension for any number of bits. Yet its base-10 digits sequence is 12345678910111213... and it is a normal number in base 10. The equivalent exists in base 2. In base 2, it would fail the pair-correlation test. In base 2 or 10, it fails the Kolmogorov complexity test [Wiki], which has no implementation because it is undecidable. The Copeland–Erdős constant [Wiki] where the numbers 1, 2, 3 and so on are replaced by the prime numbers, is also fully equidistributed. Yet, it would make for a terrible PRNG. In short, the fact that a PRNG is proved to be equidistributed, has little value.

Finally, some claim that the binary digits of π have no consecutive 0 between positions n and $8n$ for n large enough, and thus its digits do not mimic randomness, see here. This argument is wrong: any random bit sequence has no consecutive 0 between positions n and $n+\log_2 n$. In fact, I proved (and I am not the only one) that the binary digits of $\sqrt{2}$ cannot have consecutive 0 between positions n and $2n$. You expect that from a random sequence when n is large enough. And if not satisfied, it would prove that the sequence is not random. Much of the pseudo-science linked to random bits is spread by people who know little about the topic, despite advanced degrees in computer science and similar fields.

8.1.4 Optimum parameters to speed up computations

Formula (8.1), here $x_n = (a2^{\lambda(n)}+b)x_{n-1}$ with $\lambda(n) = n$, can be computed very quickly without multiplication. First, choose 4 integers $k_1, \ldots, k_4 > 0$. Then, let $a = 2^{k_2}(2^{k_1}+1)+1$ and $b = 2^{k_4}(2^{k_3}+1)+1$. Choose the k_i's so that a, b are distinct primes. You can now compute $x_n = E + E'$ with 5 additions and 5 bit shifts, as follows:

$$\begin{aligned} A &= 2^{\lambda(n)}x_{n-1} & A' &= x_{n-1} \\ B &= 2^{k_1}A & B' &= 2^{k_3}A' \\ C &= B + A & C' &= B' + A' \\ D &= 2^{k_2}C & D' &= 2^{k_4}C' \\ E &= D + A & E' &= D' + A' \end{aligned}$$

Multiplying a number by an integer power of 2 is a called bit shift, a very fast operation. The fastest way to compute the sum of two arbitrarily large integers with ν bits is using parallel prefix addition or carry-lookahead algorithms. These approaches break numbers into smaller chunks, process the carries in parallel, and yield a time complexity of $O(\log \nu)$, bypassing the sequential bottleneck of standard digit-by-digit addition. See details here.

8.2 Compute time and Python code

The compute time is summarized in Table 8.2. A good benchmark is when $N = 2000$, generating about 10^9 bits in 0.10 seconds with PCG64, a standard also reported elsewhere. Unlike PCG64 (linear in time), there is an optimum for NPG. On my laptop, producing 8 bit strings each with $N = 2000$ is much faster than producing a single string with $N = 4000$ even though the total number of bits generated is the same in both cases. Also, I optimized PCG64 for speed, as follows:

```
size = total_bits
block_size = 32
nsamples = 20
start = time.perf_counter()
chunk_size = size // (nsamples * block_size)
for sample in range(nsamples):
   bits = rng.integers(0, 2**block_size, size = chunk_size, dtype=np.uint32)
end = time.perf_counter()
```

I know discuss the code. First, if you perform a large number of tests, some will lead to false positives, just by chance: a truly random sequence erroneously flagged as abnormal. The issue comes from using the same high p-value too many times. This phenomenon is described in most statistical textbooks, along with solutions to address it. Anyway, this is not the case here.

N	Total bits	a	b	NPG	PCG64
1000	166,725,233	1	1991	0.005550	0.016639
1000	166,668,496	1	1	0.003406	0.013890
1000	169,528,867	53	31	0.007111	0.017117
2000	1,333,336,996	1	1	0.022635	0.106265
3000	4,500,005,496	1	1	0.115812	0.338585

Table 8.2: Compute time in seconds, NPG vs PCG64

It is worth checking if the PCG64 failures are linked to the odd number of bits in Table 8.1, dictated by the choice of a and b. Those who built PCG64 probably tested it mostly with more standard values. The spectral test is also quite sensitive to that value. With this cleared, I can now list the NPG code in section 8.2.1, and the testing code in section 8.2.2.

8.2.1 NPG source code

The code produces the iterates $x_1, \ldots, x_N$ given a parameter set a, b and $x_0 = 1$. The `'detailed'` mode is quite slow, but is shows the first 60 digits of x_n and x_{n-1} at each iteration n, along with the cross-correlation between both bit strings, computed for various lags. Set `mode='fast'` to not show these details and massively accelerate computations. To produce a grand total of about ν bits across all iterates, set N to $\lfloor (6\nu)^{1/3} \rfloor$. The parameters a, b are stored in the `params` array initialized at line 106 in the code. You can loop over multiple parameter sets, each one being an entry in `params`. A parameter set consists of 4 elements:

- The first one x_0 is set to 1, but you are free to change it to any positive integer, small or large.
- The second and third ones are a and b. You can try $b = 0$ if you want, although it is a non-standard value.
- The last one specifies the model. If set to `'large'` (default) then $\lambda(n) = n$ in (8.2). Otherwise $\lambda(n) = 0$.

In addition, the code computes the time, both for PCG64 and NPG. You can choose PCG64DXSM instead of PCG64, via the option `numpy_PRNG`, and specify its seed with `rng_seed`.

Finally, the function `python_rnd_bits` creates an array of `M` bit strings. The first one is denoted as B_0 in Table 8.1. It comes from NPG and consists of the concatenation of the last two truncated iterates x_{N-1} and x_N. The next ones come from PCG64. They all have the same size, matching that of B_0. These strings, stored in the `rnd_bits` array, are used as input to the `test_battery` function, to test randomness. To run NPG without the required left and right bit string truncation, set `skip_flag=False` and see if it still passes the randomness tests.

```
# NPG.py
import time
import numpy as np
from numpy.random import Generator, PCG64DXSM
import gmpy2


def count_bits(xbin_old, xbin, match_left, mach_right):

    nbits = len(xbin)
    nbits_old = len(xbin_old)

    match_left = 0
    while xbin[match_left] == xbin_old[match_left] and match_left < nbits_old-1:
       match_left += 1
    match_right = 0
    while xbin[nbits-match_right-1] == xbin_old[nbits_old-match_right-1] and match_right <
        nbits_old-1:
       match_right += 1

    auto_correl = []
    max_ac = 0
    maxLag = 4
```

```
    length = min(len(xbin_old)-maxLag, 200000)

    if length > 3:

       for lag in range(maxLag):
          arr1 = np.array([int(b) for b in xbin_old[0:length]])
          arr2 = np.array([int(b) for b in xbin[lag:lag+length]])
          correl1 = np.corrcoef(arr1, arr2)[0, 1]
          arr1 = np.array([int(b) for b in xbin_old[lag:lag+length]])
          arr2 = np.array([int(b) for b in xbin[0:length]])
          correl2 = np.corrcoef(arr1, arr2)[0, 1]
          auto_correl.append(correl1)
          auto_correl.append(correl2)
       auto_correl = np.array(auto_correl)
       max_ac = np.max(np.abs(auto_correl))

    return(match_left, match_right, max_ac)


#--- Core function

def vg_random(mode, N, x0, a, b, model):

    x = gmpy2.xmpz(x0)
    xbin = '1'
    nbits = 0
    if mode == 'detailed':
       match_left = 0
       match_right = 0

    for k in range(1, N):

       x_old = x
       if model == 'large':
          if a == 1 and b == 1:
             # speed up computations
             x = (x << k) + x
          elif a == 1:
             x = (x << k) + b*x
          elif b == 1:
             x = a*(x << k) + x
          else:
             x = a*(x << k) + b*x
       elif model == 'small':
          x = a*x + b
       nbits += len(x)

       if mode == 'detailed' and k % 250 == 0:
          xbin_old = x_old.digits(2)[2:]
          xbin = x.digits(2)[2:]
          match_left, match_right, auto_correl = count_bits(xbin_old, xbin, match_left, match_right)
          print("k: %4d | bits(xbin): %9d | left: %4d | right: %4d | correl: %8.5f | nbits: %d" %
                      (k, len(x), match_left, match_right, auto_correl, nbits))
          stri_new = xbin
          stri_old = xbin_old
          if len(xbin) > 60:
             stri_new = xbin[0:60] + "..."
             stri_old = xbin_old[0:60] + "..."
          print("Old xbin:", stri_old)
          print("New xbin:", stri_new,"\n")

    return(x, x_old, nbits)


#--- Main ----------------------------------------------------------------

N = 1000
ndigits = N*(N+1) // 2
ctx = gmpy2.get_context()
ctx.precision = ndigits
mode = 'fast' # options: 'fast', 'detailed'

numpy_PRNG = 'PCG64' # options: 'PCG64' or 'PCG64DXSM'
rng_seed = 42
if numpy_PRNG == 'PCG64DXSM':
    bg = PCG64DXSM(seed=rng_seed)
```

```
    rng = Generator(bg)
else:
    rng = np.random.default_rng(rng_seed)

start = time.perf_counter()

total_bits = 0
params = [
         #(1, 3, 5, 'large'),
         #(1, 3, 1, 'small'),
         #(1, 1, 1, 'large'),
         #(1, 3, 0, 'small'),
         #(1, 3, 1, 'large'),
         #(1, 53, 31, 'large'),
         (1, 1, 1991, 'large'),
        ]
nsamples = len(params)


def main(params):

    hash_output = {}
    total_bits = 0
    for param in params:
        x0 = param[0]
        a = param[1]
        b = param[2]
        model = param[3]
        print("\n-------------------------------------")
        print("Tests on last xbin_old + xbin for Param = ", param)
        x, x_old, nbits = vg_random(mode, N, x0, a, b, model)
        total_bits += nbits
        print("bits(last xbin): %8d | Total nbits: %12d" % (len(x), nbits))
        hash_output[param] = (x, x_old)
    return(hash_output, total_bits)

results, total_bits = main(params)


print("\n-------------------------------------")
end = time.perf_counter()
print(f"\nElapsed time Random vg: {end - start:.6f} seconds")
check = (nsamples * N**3 / 6) / total_bits
print("Check should be 1: %8.5f" % (check))


#--- Speed test for Python random library

size = total_bits
block_size = 32
nsamples = 20
start = time.perf_counter()
chunk_size = size // (nsamples * block_size)
for sample in range(nsamples):
    bits = rng.integers(0, 2**block_size, size = chunk_size, dtype=np.uint32)
end = time.perf_counter()
print(f"Elapsed time Random Python: {end - start:.6f} seconds")
print("bits generated: %8d" %(total_bits))


#--- Tests of randomness -------------------------------------------

def python_rnd_bits(M, x, x_old, skip_flag):

    if skip_flag:
        len1 = len(x.digits(2)[2:])
        skip = 1 + int(np.sqrt(2*len1))
        xbin = x_old.digits(2)[2+skip:-skip] + x.digits(2)[2+skip:-skip]
    else:
        xbin = x_old.digits(2)[2:] + x.digits(2)[2:]
    nbits = len(xbin)
    print("nbits: %8d | skip_flag: %s | seed: %3d | PRNG: %s | N: %5d\n" %
                 (nbits, skip_flag, rng_seed, numpy_PRNG, N))

    rnd_bits = []
    rnd_bits.append(xbin)
```

```
    for m in range(M):
       bits = rng.integers(0, 2, size = nbits)
       bits = "".join(map(str, bits))
       rnd_bits.append(bits)
    return(rnd_bits)


M = 8          # compare xbin to M-1 similar bit streams from Python/random
Lmax = 3       # count frequencies for all string of length <= Lmax
maxLag = 4     # max. lag in auto- and cross-correlations
L = 20         # collisions for strings of length L
skip_flag = True # ignore problematic rightmost and leftmost bits


def test_battery(M, Lmax, maxLag, L, skip_flag, results):

    for param in results:
       x, x_old = results[param]
       print("\n-----------------------------------")
       rnd_bits = python_rnd_bits(M, x, x_old, skip_flag)
       print("Tests on last xbin_old + xbin for param =", param)
       prng_test_vg.test_frequencies(Lmax, rnd_bits)
       prng_test_vg.test_autocorrel(maxLag, rnd_bits)
       prng_test_vg.test_collisions(L, rnd_bits)
       prng_test_vg.test_spectral(rnd_bits)
       prng_test_vg.test_compress(rnd_bits)
       prng_test_vg.test_runs('0', rnd_bits)
       prng_test_vg.test_next_bit(rnd_bits)
    return()


import prng_test_vg
test_battery(M, Lmax, maxLag, L, skip_flag, results)
```

8.2.2 PRNG tests of randomness

The functions in this library accept `rnd_bits` as input, an array consisting of bit strings $B_0, B_1, \ldots$ produced by the main program. Typically, B_0 comes from NPG while the other strings from PCG64. They all have the same size. The goal is to check if all these bit strings have the same random behavior, and report the results in Table 8.1 and Figures 8.1 and 8.2, for a fixed seed and a, b values. In addition, each function has its own set of parameters. The code is also on GitHub, here.

```
# prng_test_vg.py

import numpy as np
import lzma


def test_frequencies(Lmax, rnd_bits):

    M = len(rnd_bits)
    print("\nString", end = " ")
    for test in range(M):
       print(" Test %1d" % test, end = " ")
    print()
    for L in range(1,Lmax+1):
       for k in range(2**L):
          arr_stats = []
          kbin = bin(k)[2:]
          kbin = '0' * (L-len(kbin)) + kbin
          for m in range(M):
             bits = rnd_bits[m]
             indices = [i for i in range(len(bits)) if bits.startswith(kbin, i)]
             arr_stats.append(len(indices))
          print("%6s" % (kbin), end = " ")
          for test in range(len(arr_stats)):
             print("%8d" % (arr_stats[test]), end=" ")
          print()
    return()


def test_autocorrel(maxLag, rnd_bits):
```

```
31
32      M = len(rnd_bits)
33      length = min(len(rnd_bits[0])-maxLag, 5000000)
34      arr_max_correl = []
35
36      for m in range(M):
37         bits = rnd_bits[m]
38         auto_correl = []
39         for lag in range(1, maxLag):
40            arr1 = np.array([int(b) for b in bits[0:length]])
41            arr2 = np.array([int(b) for b in bits[lag:lag+length]])
42            correl = np.corrcoef(arr1, arr2)[0, 1]
43            auto_correl.append(correl)
44         auto_correl = np.array(auto_correl)
45         max_ac = np.max(np.abs(auto_correl))
46         arr_max_correl.append(max_ac)
47
48      print("\nMax abs autocorrels up to lag %2d:" %(maxLag-1))
49      for test in range(len(arr_max_correl)):
50         print("Test %2d: %8.6f" % (test, arr_max_correl[test]))
51      return()
52
53
54   def update_hash(hash, key, count):
55      if key in hash:
56         hash[key] += count
57      else:
58         hash[key] = count
59      return(hash)
60
61
62   def test_collisions(L, rnd_bits):
63
64      M = len(rnd_bits)
65      length = len(rnd_bits[0]) - L
66      arr_collisions = []
67
68      for m in range(M):
69         bits = rnd_bits[m]
70         hash = {}
71         for idx in range(0, length, 1):
72            update_hash(hash, bits[idx:idx+L], 1)
73         collisions = 0
74         max_hits = 0
75         for string in hash:
76            cnt = hash[string]
77            if cnt > 1:
78               collisions += 1
79            if cnt > max_hits:
80               max_hits = cnt
81         arr_collisions.append([len(hash), collisions, max_hits])
82
83      print("\n%2d-bits collisions max_hits" %(L))
84      for test in range(len(arr_collisions)):
85         collisions = arr_collisions[test]
86         print("Test %2d: %8d %5d" % (test, collisions[1], collisions[2]))
87      return()
88
89
90   def test_spectral(rnd_bits):
91
92      from math import erfc
93      arr_spectral = []
94      M = len(rnd_bits)
95
96      for m in range(M):
97
98         bits = rnd_bits[m]
99         x = np.array([1 if b == '1' else -1 for b in bits], dtype=float)
100        n = len(x)
101        s = np.fft.fft(x)
102        m = np.abs(s[:n // 2])
103        tau = np.sqrt(np.log(1.0 / 0.05) * n) # threshold
104        n0 = 0.95 * n / 2.0 # expected peaks below threshold
105        n1 = np.sum(m < tau) # peaks below threshold
106        d = (n1 - n0) / np.sqrt(n * 0.95 * 0.05 / 4.0) # test statistic
```

```
        p_value = erfc(abs(d) / np.sqrt(2.0))
        result = 'FAIL'
        if p_value >= 0.01:
            result = 'PASS'
        arr_spectral.append([p_value, result])

    print("\nspectral p_value")
    for test in range(M):
        spectral = arr_spectral[test]
        print("Test %2d: %8.5f %s" % (test, spectral[0], spectral[1]))
    return()


def test_compress(rnd_bits):

    M = len(rnd_bits)
    arr_compress = []

    for m in range(M):
        bit_string = rnd_bits[m]

        padded_bit_string = bit_string.ljust((len(bit_string) + 7) // 8 * 8, '0')
        byte_data = int(padded_bit_string, 2).to_bytes(len(padded_bit_string) // 8, byteorder='big')
        compressed_data = lzma.compress(byte_data, preset=9)
        arr_compress.append(len(compressed_data))

    print("\nLZMA  size  ratio")
    for test in range(len(arr_compress)):
        size = arr_compress[test]
        ratio = size / (len(bit_string)/8)
        print("Test %2d: %6d %8.5f" % (test, size, ratio))
    return()


def test_runs(bit, rnd_bits):

    import matplotlib.pyplot as plt
    import matplotlib as mpl
    M = len(rnd_bits)

    mpl.rcParams['axes.linewidth'] = 0.5
    plt.rcParams['xtick.labelsize'] = 8
    plt.rcParams['ytick.labelsize'] = 8
    plt.rcParams['legend.fontsize'] = 'x-small'

    length = len(rnd_bits[0])
    qualitative_colors = plt.get_cmap('tab10').colors
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
    ax1.set_facecolor('black')
    ax2.set_facecolor('black')

    for m in range(M):

        if m == 0:
            color = 'blue'
        else:
            color = qualitative_colors[m+1]

        arr_run = []
        arr_run_freq = []
        arr_run_first = []

        bit_string = rnd_bits[m]
        hash_runs = {}
        hash_records = {}
        run = 1
        for t in range(1, length):
            Flag = True
            if bit_string[t] == bit_string[t-1] == str(bit):
                run += 1
            else:
                Flag = False
                if run not in hash_runs:
                    hash_records[run] = t
                update_hash(hash_runs, run, 1)
                run = 1
```

```
183
184         for run in hash_runs:
185             if run > 1:
186                 arr_run.append(run)
187                 arr_run_freq.append(np.log(hash_runs[run]))
188                 arr_run_first.append(np.log(hash_records[run]))
189
190         ax1.plot(arr_run, arr_run_freq, color=color, linewidth = 0.8, label=f"Test {m}")
191         ax2.plot(arr_run, arr_run_first, color=color, linewidth = 0.8, label=f"Test {m}")
192
193     ax1.legend(loc='upper right')
194     ax2.legend(loc='upper left')
195     plt.show()
196     return()
197
198
199 def test_next_bit(rnd_bits):
200
201     from sklearn.neural_network import MLPClassifier
202     from scipy.stats import binom
203
204     M = len(rnd_bits)
205     n = 1000          # samples to extract from string
206     window = 10       # predict next bit based on previous bits in window
207     split = int(0.80 * n) # 80% for training, 20% for testing
208     nobs = n - split
209     arr_accuracy = []
210     length = len(rnd_bits[0])
211     np.random.seed(56) # PCG64 has issues if this seed is 56 or 58
212     offsets = np.random.randint(0, length-window, size=n)
213
214     for m in range(M):
215
216         bits = rnd_bits[m]
217         ibits = np.array([1 if b == '1' else 0 for b in bits], dtype=int)
218         # predict bit at position start+windown based on past window bits
219         X, y = [], []
220         for k in range(n):
221             start = offsets[k]
222             X.append(ibits[start : start + window])
223             y.append(ibits[start + window])
224         X_train, X_test = X[:split], X[split:]
225         y_train, y_test = y[:split], y[split:]
226
227         clf = MLPClassifier(hidden_layer_sizes=(32, 16), max_iter=2000, random_state=42)
228         clf.fit(X_train, y_train)
229         y_pred = clf.predict(X_test)
230         arr_errors = y_pred - y_test # 0 if match; are errors randomly spread or not?
231         accuracy = clf.score(X_test, y_test)
232         arr_accuracy.append([accuracy, arr_errors])
233
234     print("\nnext-bit accuracy p_value (0.50 accuracy means cannot predict next bit)")
235     for test in range(len(arr_accuracy)):
236         arr_output = arr_accuracy[test]
237         accuracy = arr_output[0]
238         lower = min(accuracy, 1-accuracy) * nobs
239         upper = max(accuracy, 1-accuracy) * nobs
240         p_valueL = binom.cdf(lower, nobs, 0.50)
241         p_valueR = 1- binom.cdf(upper, nobs, 0.50)
242         p_value = p_valueL + p_valueR
243         print("Test %2d %8.5f %8.5f" % (test, accuracy, p_value))
244     return()
```

Appendix A

The Pi Day xLLM agent

Pi Day is celebrated worldwide each year on March 14 (the date is sometimes denoted as 3.14). My Pi Day agent, one of several available from our large language model xLLM [16, 20, 21, 22] features a video where the digits of π appears randomly on the screen, one by one, until all of them (among the first 3,000 or so) are displayed. The Python code is also on github here. Watch the video on YouTube, here.

Figure A.1: One of the 300 video frames generated by the Pi Day agent

It works as follows: Each video frame is automatically built as a webpage (HTM code generated with Python, each digit has its own color), then the corresponding webpage is turned into an image (screenshot) also with Python code. All the images are then bundled together into a video. Figure A.1 shows one of the video frames.

```
pi = (
'3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651'
'3282306647093844609550582231725359408128481117450284102701938521105559644622948954930381964428810975665933446128'
'4756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628'
'2540917153643678925903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051185'
'4807446237996274956735188575272489122793818301194912983367336644065643086021394946395224737190702179860943702770'
'5392171762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465'
'4958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973'
'1732816096318595024459455346908302642522308253344685035261931188171010003137838752886587533208381420617177669'
'1473035982534904287554687311595628638823537875937519577818577805321712268066130019278766111959092164201989380'
'9525720106548586327886593615338182796823030195203530185296899577362259941389124972177528347913155748572424541'
'5069595082953311686172785588907509838175463746493931925506040092770167113900984824012858361603567076601047101'
'8194295559619894676783744944825537977472684710404753464620804668425906949129331367702898915210475216205696602'
'4058038150193511253382430035587640247496473263914192722042699227967823547816360093417216412199245863150302861'
'8297455706749838505494588586926995690927210797509302955321165344987202755960236480666549911988183479775366369'
'8074265427527862551818417574672890977727938000816470600161452491921732172147723501414419735685481613611573525'
'5213475741849468438523323907394143334547762416862518983569485562099219222184272550254256887671790494601653466'
'8049886272327917860857843838279679766814541009538837863609506800642251252051173929848960841284862694360424196'
'5250221066116630674427862203919494504712371378696095636437191728746776465757396241389086583264599581339047802'
'7590099465764078951269468398352595709825822620522489407726719478268482601476990902640136394437455305068203496'
'2524517493939651431429809190659250937221696461515708585387410597885959772975489301617539284681382686838689427'
'7415599185592524595395943104997252468084598727364469584865383673622262606991246080512438843904512441365497627'
'8079771569143599770012961608944164868555848406353422072225828488648158450285060168427391522674677888952521385'
'2254995466727823986456596116354862305774564980355936345681743241125150760694794510965960940252202'
```

```
'99272604269922796782354781636009341721641219924586315030286182974555706749838505494588586926995690 9'
'27210797509302955321165344987202755960236480665499119881834797753566369807426542527862551818417574 6'
'72890977772793800081647060016145249192173217214772350141441973568548161361157352552133475741849468 4'
'38523323907394143334547762416862518983569485562099219222184272550254256887671790494601653466804988 6'
'27232791786085784383827967976681454100953883786360950680064225125205117392984896084128488626945604 2'
'41965285022210661186306744278622039194945047123713786960956364371917287467764657573962413890865832 6'
'45995813390478027590099465764078951269468398352595709825822620522489407726719478268482601476990902 6'
'40136394437455305068203496252451749399651431429809190659250937221696461515709858387410597885959772 9'
'75498930161753928468138268683868942774155991855925245953959431049972524680845987273644695848653836 7'
'36222626099124608051243884390451244136549762780797715691435997700129616089441694868555848406353422 0'
'72225828488648158456028506016842739452267467678895252138522549954666727823986456596116354886230577 4'
'56498035593634568174324112515076069479451096596094025228879710893145669136867228748940560101503308 6'
'17928680920874760917824938589009714909675985261365549781893129784821682998948722658804857564014270 4'
'77555132379641451523746234364542858444795265867821051141354735739523113427166102135969536231442952 4'
'84937187110145765403590279934403742007310578539062198387447808478489683321445713868751943506430218 4'
'53191048481005370614680674919278191197939952061419663428754440643745123718192179998391015919561814 6'
'75142691239748940907186494231961567945208095146550225231603881930142093762137855956638937787083039 0'
'69792077346722182562599661501421503068038447734549202605414665925201497442850732518666002132434088 1'
'90710486331734649651453905796268561005508106658796998163574736384052571459102897064140110971206280 4'
'39039759515677157700420337869936007230558763176359421873'
)

import numpy as np
from html2image import Html2Image
from PIL import Image
import moviepy.video.io.ImageSequenceClip
np.random.seed(55)

width = 110
size = len(pi)
lines = 26
print(len(pi), lines)
permut = np.random.permutation(width*lines)

cols = ['888888', 'FF8888', '00FF00', 'FFFF00', 'FF00FF',
      'CCCCCC', 'AAFF00', '00FFFF', 'FF0000', 'AA66AA' ,
      '8888FF']

def generate_frame(cols, nchars, lines, width):

   html = "<html>\n<body style='background-color:#000000; font-size: 3'>\n"
   for line in range(lines):
      start = width*line
      end = start + width
      linestring = pi[start:end]
      strcolor = ''
      for k in range(len(linestring)):
         char = linestring[k]
         if char == '.':
            ichar = 0
         else:
            ichar = int(char) + 1
         idx = line * width + k
         if permut[idx] < nchars:
            strcolor = "<font size = '6' color = #" + cols[ichar] + ">" + char + "</font>"
         else:
            strcolor = "<font size = '6' color = #" + '333333' + ">" + char + "</font>"
         html += strcolor
      html += "<br>\n\n"
   html += "</body>\n</html>"
   return(html)

nchars = 0
frame = 0
step = 12 # number of frames = lines * width / step
icontinue = True
flist = []

while icontinue:

   if nchars >= lines * width:
      nchars = lines * width - 1
      icontinue = False
   html = generate_frame(cols, nchars, lines, width)
   hti = Html2Image()

```

```
    # https://stackoverflow.com/questions/60598837/html-to-image-using-python
    hti.screenshot(html_str=html, save_as='pi.png')

    image = "pi.png"
    original = Image.open(image)
    pwidth, pheight = original.size # Get dimensions
    left = 0 # width/4
    top = 0 #height/4
    right = 0.925 * pwidth ## 3 * width/4
    bottom = 0.8 * pheight ## 3 * height/4
    cropped = original.crop((left, top, right, bottom))
    fimage = "pi" + f"{frame}" + ".png"
    cropped.save(fimage)
    flist.append(fimage)
    print("Frame:", frame)
    frame += 1
    nchars += step

clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=15)
clip.write_videofile('pi_day.mp4')
```

Appendix B

Convolution, Approximations, and Signal Processing

In just a few pages, I cover signal processing and convolution quite thoroughly, even advanced concepts. I illustrate the techniques with the Riemann zeta function and the kernel method, along with short, home-made Python code that shows all the detailed steps, rather than based on Blackbox Python libraries. The presentation looks like a cheat sheet. It may be useful for scientists who need a refresher on the topic, or professionals in quantitative fields who don't know much about the topic and want to get a deep overview without spending weeks reading introductory to advanced books. References are provided as needed. The focus is on practical aspects, good practices, discrete convolution, and working with real data rather than explaining the theory and listing theorems.

Finally, I discuss interesting approximation techniques, again, with a focus on the Riemann zeta function. As usual, for practical purposes, I use its sister, the Dirichlet eta function, which is easier to handle especially for people not familiar with analytic continuation.

B.1 Approximations to mathematical function

The theory presented here is applied to the Dirichlet eta function. For more on this topic, see section B.2.

B.1.1 Finding the roots of ζ with fast-converging series

The method described here relies on a combination of formulas (6.1), (6.2) and (6.4). Given an integer $m > 1$, let $\mathscr{D}_m$ be the set consisting of the first primes $p_2, \ldots, p_m$, omitting $p_1 = 2$. We have:

$$\zeta(s) = \frac{1}{1-2^{1-s}} \cdot \left(\prod_{k=2}^{m} \frac{1}{1-p_k^{-s}} \right) \cdot \left(\sum_{k=1}^{\infty} \frac{\delta_m(k)}{k^s} \right), \tag{B.1}$$

where $\delta(k) \in \{-1, 0, +1\}$. More specifically, $\delta_m(k) = 0$ if k is a multiple of any prime $p \in \mathscr{D}_m$. Otherwise, $\delta_m(k) = -1$ if k is even and $+1$ is k is odd. By choosing a large m, you eliminate most of the terms in the sum on the right in (B.1). However, you increase the number of factors in the middle product. Thus, this approach is efficient when you are only interested in finding the roots of ζ. In that case, you can ignore (not compute) the product in the middle and choose a very large m. It also shows that only extremely large primes have any impact on the value of these zeros, since m can be arbitrarily large.

For a fixed $m > 1$, the proportion of $\delta_m(k)$'s not equal to zero is the probability ρ_m that an arbitrary k in (B.1) is not a multiple of any prime p_i with $2 \leq i \leq m$. It is equal to

$$\rho_m = \prod_{i=2}^{m} \left(1 - \frac{1}{p_i}\right) \sim \frac{e^{-\gamma}}{\log m} \text{ as } m \to \infty \tag{B.2}$$

where $\gamma = 0.57721\ldots$ is the Euler–Mascheroni constant [Wiki]. The asymptotic result in (B.2) is a consequence of the third Mertens theorem [Wiki]. As an illustration, see how the sum in (B.1) gets shortened when $m = 4$, that is, with $\mathscr{D}_4 = \{3, 5, 7\}$:

$$\sum_{k=1}^{\infty} \frac{\delta_4(k)}{k^s} = 1 - \frac{1}{2^s} - \frac{1}{4^s} - \frac{1}{8^s} + \frac{1}{11^s} + \frac{1}{13^s} - \frac{1}{16^s} + \frac{1}{17^s} + \frac{1}{19^s} - \frac{1}{22^s} + \frac{1}{23^s} - \frac{1}{26^s} \cdots \tag{B.3}$$

It is tempting to try increasingly large values of m to study the properties of the zeros of the Riemann zeta function $\zeta(s)$, in particular, their distribution on the critical line $\Re(s) = \frac{1}{2}$. However, there is no guarantee that the framework proposed here makes it any easier. In particular, the convergence of a series such as (B.3) has some peculiarities. If you evaluate the sum at a root of ζ, using the first n terms ($k = 1, \ldots, n$) on the left sum in (B.3), how close you get to zero strongly depends on which modulo class n belongs to. Eventually, convergence is guaranteed, but it does not happen smoothly.

Figure 6.2 shows in green how well you approximate $||\zeta(s_3)|| = 0$ using the first n terms ($k = n$) in the series on the left side in (B.3), evaluated at $s = s_3 = \frac{1}{2} + 25.010858\,i$, the third root of ζ. The number of actual terms is less than $n/2$ when you eliminate those with $\delta_4(k) = 0$. For many values of n higher than 1500. You periodically get a much better approximation to zero (the correct value when $n = \infty$) than with the standard η function in red. However, for small n's the standard η does better.

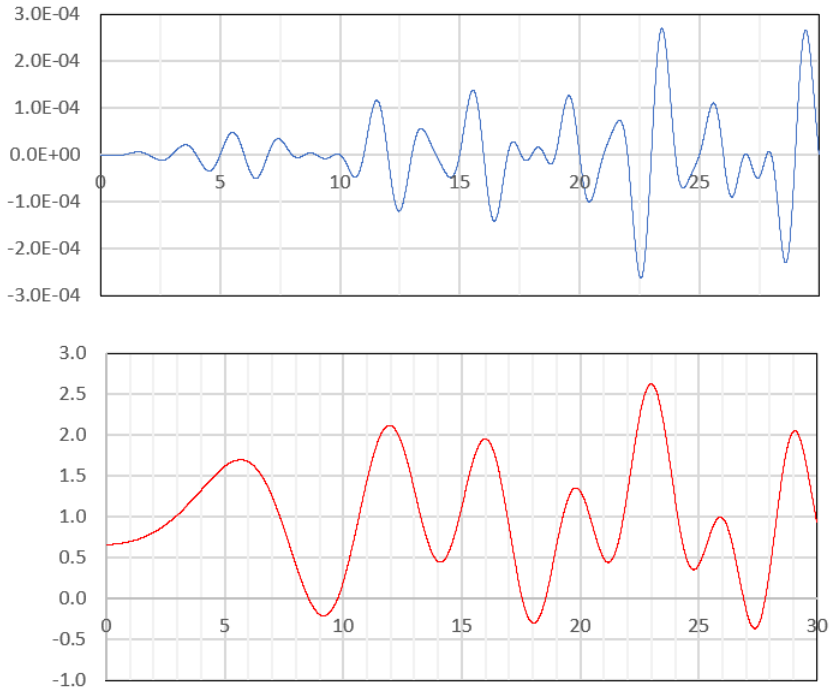


Figure B.1: $\eta_r(\sigma + it)$ with $\sigma = 0.8$, t on X-axis (bottom) and interpolation error (top)

B.1.2 Approximation based on quantization

This may be particularly useful for time series interpolation. Assume that f is a smooth, slow growing even function, and $f(t)$ is known if t is a positive integer. Then, if we know $f(t)$ for $t = 0, 1, 2$ and so on, we can interpolate f exactly with the formula

$$f(t) = \frac{\sin \pi t}{\pi} \cdot \Big[\frac{f(0)}{t} + \phi'(t) \sum_{k=1}^{\infty} (-1)^k \frac{f(k)}{\phi(t) - \phi(k)}\Big]. \tag{B.4}$$

It works if $\phi(t) = t^2$, $\phi'(t) = 2t$ (derivative with respect to t) and the function f can be represented as

$$f(t) = \sum_{k=0}^{\infty} \alpha_k \cos \beta_k t, \text{ with } |\beta_k| < \pi. \tag{B.5}$$

The approximation is exact if you include all the infinitely many terms. It can be used for time series disaggregation [23]. A potential application is to break down hourly temperature predictions into 5 minute increments. A detailed discussion about this interpolation formula and its generalization, can be found here. Now let's see how it works for the real part of the Dirichlet eta function $\eta(\sigma + it)$. The real part of the Dirichlet eta function (closely linked to the Riemann Hypothesis) is

$$\eta_r(\sigma + it) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\cos(t \log k)}{k^{\sigma}}, \quad \sigma > 0. \tag{B.6}$$

The Dirichlet series (B.6) converges very slowly and chaotically, especially if σ is small or t is large. One way to accelerate the convergence is to use Euler's transform [Wiki]. See also Exercise 25 in [13]. Other approximations

exist, for instance using Dirichlet polynomials [12]. Let us assume $\eta_r(\sigma + ik)$ is known and denoted as $\eta_k(\sigma)$ if k is a positive integer. Using (B.4), the new approximation is as follows:

$$\eta_r(\sigma + it) \approx \frac{\sin \pi t}{\pi} \cdot \left[\frac{\eta_0(\sigma)}{t} + 2t \sum_{k=1}^{n} (-1)^k \frac{\eta_k(\sigma)}{t^2 - k^2} \right] \tag{B.7}$$

Figure B.1 shows how accurate the interpolation formula is, here for $\sigma = 0.80$. The full function was reconstructed, based on $f(k)$ computed at $k = 0, 1, \ldots, 249$. The horizontal axis represents t. Note that to estimate $f(t)$ beyond $t = 30$, more than 250 terms are needed in formula (B.4), to keep the error smaller than 3×10^{-4}. Interestingly, the interpolation formula seems to be working even though condition (B.5) is not satisfied. At integer arguments, the error is minimum in absolute value, and smaller than 10^{-6}.

For the imaginary part – an odd function – you can multiply it by $\sin \lambda t$ to turn it into an even function, then apply the same methodology to the transformed function to interpolate it, then divide back by $\sin \lambda t$. Here $\lambda \neq 0$ is an arbitrary constant.

B.2 Non-causal discrete convolution with Gaussian kernel

The discrete Gaussian convolution of the Riemann zeta function, one of the most famous in number theory, leads to some curious and unexpected results. Here I provide a solid introduction to this topic, of interest to anyone dealing with signal processing problems. Starting with the initial conditions $h_1(k) = \frac{1}{3}$ if $-1 \leq k \leq 1$ and $h_1(k) = 0$ otherwise, I iteratively define the kernel h_{n+1} as the convolution $h_{n+1} = h_1 * h_n$, that is:

$$h_{n+1}(k) = \sum_{j=-n}^{n} h_1(j) h_n(k - j). \tag{B.8}$$

Note that $h_n(k) = 0$ if $|k| > n$. For $-n \leq k \leq n$, the integers $3^n h_n(k)$ are known as trinomial coefficients [Wiki], a generalization of binomial coefficients. As $n \to \infty$, their distribution approximates a Gaussian one with zero mean and variance $\sigma_n^2 = 1/(2\pi h_n^2(0))$. This is illustrated in Figure B.3, where you can not distinguish h_n from its Gaussian approximation. Thus h_n is called a Gaussian kernel. Also, since negative values of k are allowed, it is non-causal. Given a discrete signal X_t, I define its convolution by h_n as

$$Y_t = \sum_{k=-n}^{n} h_n(k) X_{t-k}. \tag{B.9}$$

Here, the input signal is a mixture of cosines with incommensurate periods, thus non periodic, and defined as

$$X_t = \sum_{k=r}^{m} (-1)^{k+1} \cdot \frac{1}{k^\sigma} \cos\Big(\theta(t - \tau) \log k\Big). \tag{B.10}$$

When $r = 1, m = \infty, \theta = 1$ and $\tau = 0$, X_t is the real part of the Dirichlet eta function, a close sister of the Riemann zeta function, also used as a universal function in deep neural networks in chapter 4 in [22]. Here $\sigma = 0.75$. Since the the convolution of a cosine signal with a Gaussian kernel is also a cosine of the same period, the resulting Y_t has the following form:

$$Y_t = \sum_{k=r}^{m} (-1)^{k+1} \cdot \frac{\lambda_k}{k^\sigma} \cos\Big(\varphi_k + \theta(t - \tau) \log k\Big) \tag{B.11}$$

where λ_k, φ_k can be computed exactly and indicate a change respectively in amplitude and phase (but not in period) in each cosine term after the convolution. Figure B.2 shows scaled X_t in green, here with $r = 1$, $\theta = 0.5$, $\tau = 4.16$ and $\sigma = 0.75$. The kernel h_n has $n = 200$. The resulting output Y_t looks like a perfect cosine of period $L = 2\pi/(\theta \log 2)$. That is, all the terms in (B.11) vanish except for $k = 2$. In other words, $\lambda_k = 0$ unless $k = 2$. The red curve features Y_t when $r = 5$, with all other parameters unchanged. This corresponds to a truncated X_t that no longer contains the term $k = 2$, and this time the output Y_t is no longer a pure cosine.

B.2.1 Problem

Since the convolution transforms X_t into a simple cosine in Figure B.2, at least almost perfectly if not perfectly, it is not invertible for all practical purposes. And if it was, inverting would be a numerically unstable operation. Yet in Figure B.4, Y_t seems more complex at first glance. Now the problem consists of addressing the following questions:

- How sensitive is the output Y_t to the parameters $m, r, \sigma, \tau, \theta$ and the kernel h_n?
- Estimate the phase and period of the cosine output in the case shown in Figure B.2, without using curve fitting techniques. These two values correspond to φ_2 and λ_2 in (B.11), as the other λ_k for $k \neq 2$ are almost or exactly zero.
- Is Y_t periodic Figure B.4?
- Discuss the inversion operation (deconvolution) and its feasibility.
- Even if deconvolution is not possible due to multiple X_t signals leading to the same Y_t, is the output Y_t still useful to discover interesting features of X_t? Discuss the case in Figure B.2.

You are free to ask questions to AI to help you answer the above questions. This may help you understand the inner mechanics even if you know very little about signal processing. As a side note, the problem discussed in chapter 1 is also based on convolutions and strings auto-convolutions in particular, though my presentation of the topic has been simplified in this book to facilitate the reading. See also numerical approximation via quantization in section B.1.2. Finally, zoom in on any figure to get a much higher resolution.

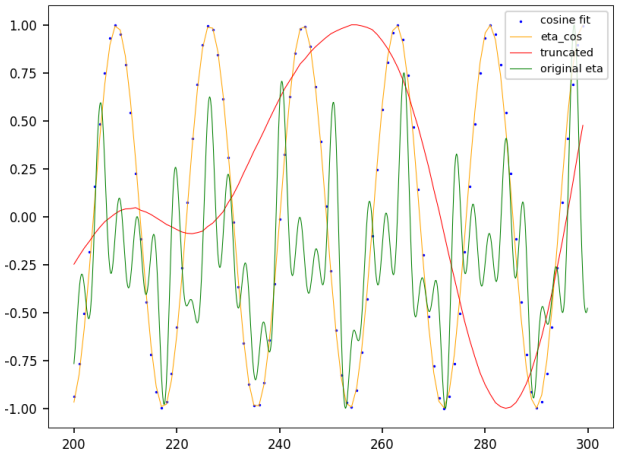


Figure B.2: X_t in green, Y_t in orange with blue dots for cosine fit, with t on X-axis. See text for red curve

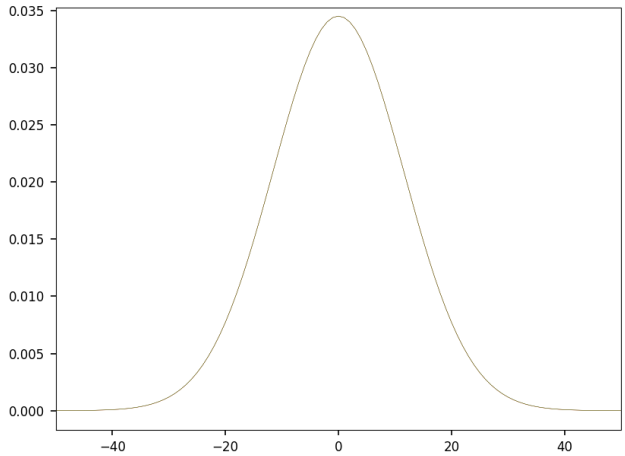


Figure B.3: Coefficients $h_n(k)$ with k on the X-axis, and their Gaussian approximation ($n = 200$)

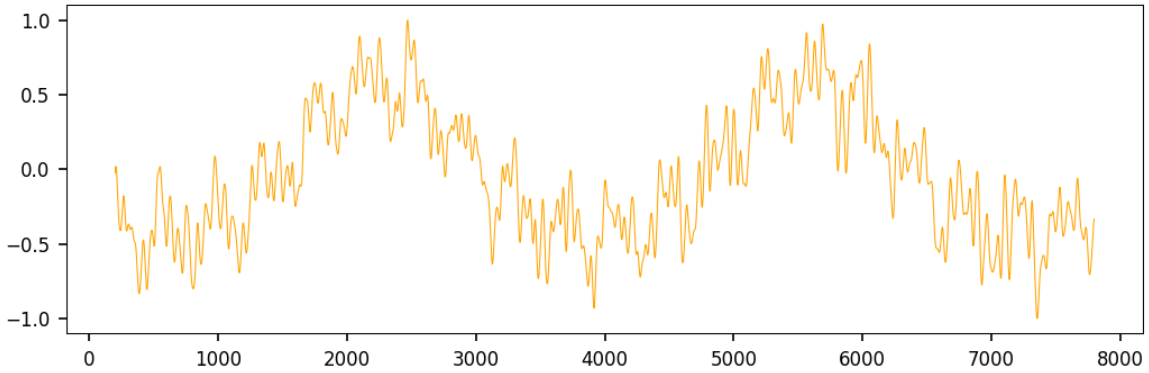


Figure B.4: Y_t based on different X_t with $\theta = 4.9$ instead of $\theta = 0.5$ in Figure B.2 (t on X-axis)

B.2.2 Solution

If the convolution is invertible, retrieving the original signal X_t from the output Y_t is done with the following formula, involving an infinite number of weights $h_n^*(k)$ even though the kernel h_n has a finite number of non-zero values $h_n(k)$:

$$X_t = \sum_{k=-\infty}^{\infty} h_n^*(k) Y_{t-k}. \tag{B.12}$$

Here h^* is the inverse of h. A search for "inverse transform to $y(t) = h(-1)x(t-1) + h(0)x(t) + h(1)x(t+1)$" on (say) Perplexity.ai will tell you the mechanics and how to proceed, see here. Additional questions such as "what

are the required conditions to make discrete convolution invertible" leads to full details, see here. You may also ask for book references on the topic, asking for the PDF versions of these books, see here. One that stands out is chapter 2 in the book "Discrete–Time Signal Processing" [35]. Answers from AI are typically short compared to courses on the topic, yet detailed enough to help you dig further while starting with a high level summary. In short, to compute $h_n^*(k)$, you need to write $1/H(z)$ as a Laurent series [Wiki] (a Taylor series with positive and negative powers), and the coefficients in that series are the $h_n^*(k)$ in question:

$$H(z) = \sum_{k=-n}^{n} h_n(k) z^{-k}, \quad \frac{1}{H(z)} = \sum_{k=\infty}^{\infty} h_n^*(k) z^k \tag{B.13}$$

Formula (B.13) involves Z-transforms and the transfer function H. For the convolution to be invertible, one needs the polynomial $z^n H(z)$ to have no zero on the unit disc in the complex plane. For the inverse to be stable, all roots must be inside that disc.

I am now about to address the other questions. Before starting, note that we work with integer values of t. In some cases, Figures B.2 and B.4 can be misleading as X_t is not an exact value but an interpolation if t is not an integer. In particular, if Y_t is a simple cosine with a non-integer period, its interpolated values in the picture may make it appear as a more complicated function than it really is. Below is a summary of my answers:

- The convolution is not sensitive to σ, τ, m or the kernel, but it is to r and θ. To see this, run the Python code in section B.2.3 with different parameter sets. The sharp contrast between the two orange curves Y_t in Figures B.2 and B.3 shows the sensitivity to θ, while $r > 2$ cannot produce the pure cosine seen with $r = 1, 2$ as it relies solely on the term $k = 2$ in (B.11). The length of the signal X_t also has some impact. In the code, it is denoted as `M`.
- In Figure B.3, the pattern observed between $t = 4000$ and $t = 7500$ seems to repeat itself when you look at a much larger time window, suggesting periodicity. But it is not exactly periodic and may fade away when t becomes very large.
- You can fit a cosine function to Y_t in Figure B.2 as follows. First scale Y_t so that the values are in $[-1, 1]$. Then detect the peaks. The average distance between two successive peaks is the period. Then look at the abscissa modulo the period, corresponding to the peaks. The average value is the phase. In short, you can fit Y_t to $\alpha \cos(\beta t + \gamma)$ without standard curve fitting technique to determine the optimum α, β, γ. See the functions `rescale` and `detect_period` in the code. I also fitted a Gaussian distribution to the kernel without curve fitting, see `fit_kernel_to_Gaussian` in the code.
- Despite the convoluted Y_t acting as a blurring filter, losing some information about the original signal (not being invertible in particular), it still carries important indicators about X_t. For instance, see Figure B.2, where minima in X_t almost match those in Y_t.

B.2.3 Python code

I use `N` to represent n, and `offset` to represent τ. Also `M` is the time window, that is the number of integer values used for X_t starting at $t = 0$. Note that to get a decent precision on the Dirichlet eta function, you need to use $m \geq 5000$ with even a much larger m when t is large. Also, you can use the function `fit_kernel_to_Gaussian` to get a good approximation of $\sqrt{2\pi}$. The program named `zeta2.py` is on my Google drive, here.

```
import numpy as np
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['legend.fontsize'] = 'x-small'

# The coefficients h_n(k) are stored in the array h[].
# X(t) and Y(t) are stored respectively in the arrays x[] and y[].


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


def build_kernel(N):
```

```
23
24      h = {}
25      h[0] = 1
26      h[1] = 1
27      h[2] = 1
28      sum = h[0] + h[1] + h[2]
29
30      b = {}
31      for n in range(1, N):
32         for k in range(0, 2*n+3):
33            if k in h:
34               update_hash(b, k, h[k])
35               update_hash(b, k+1, h[k])
36               update_hash(b, k+2, h[k])
37         sum = 0
38         for k in range(0, 2*n+3):
39            h[k] = b[k]
40            b[k] = 0
41            sum += h[k]
42
43      for k in range(0, 2*N+1):
44         h[k] = h[k]/sum
45      return(h)
46
47
48   def eta(type, t, params):
49
50      # different offsets lead to same period in cosine fit, but different phases
51      theta = params[0]
52      offset = params[1]
53      sigma = params[2]
54      start = params[3]
55      nterms = params[4]
56      if 'eta' in type:
57         start = 1
58      fval = 0
59      mod = 1
60      for k in range(start, nterms):
61         if type in ('eta_cos','truncated'):
62            wave = np.cos(theta * (t + offset) * np.log(k))
63         elif type == 'eta_sin':
64            wave = np.sin(theta * (t + offset) * np.log(k))
65         fval += mod * wave / k**sigma
66         mod = -mod
67      return(fval)
68
69
70   def f(type, M, params):
71
72      x = {}
73      for t in range(M):
74         if type in ('truncated','eta_cos','eta_sin'):
75            x[t] = eta(type, t, params)
76         elif type == 'basic':
77            x[t] = np.cos(0.1 * t)
78      return(x)
79
80
81   def detect_period(yval, N):
82
83      # cosine fit to yval[t] for t in [N, M-N[: find period and phase
84      max_arg = []
85      npeaks = 0
86      for t in range(2,len(yval)-1):
87         if yval[t] > max(yval[t-1] , yval[t+1]):
88            max_arg.append(N+t)
89            npeaks += 1
90      if npeaks > 1:
91         period = (max_arg[npeaks-1]-max_arg[0])/(npeaks-1)
92         moduli = np.mod(np.array(max_arg), period)
93         phase = np.mean(moduli)
94      else:
95         period = -1
96      return(period, phase)
97
98
```

```
99  def rescale(yval):
100
101     yval = np.array(yval)
102     ymin = np.min(yval)
103     ymax = np.max(yval)
104     yval = -1 + 2*(yval - ymin)/(ymax - ymin)
105     return(yval)
106
107
108 def fit_kernel_to_Gaussian(a):
109
110     from scipy.stats import norm
111     # a is a hash (dictionary), turn it to array a_array
112     a_array = list(a.values())
113
114     xaxis = range(-N, N+1)
115     plt.plot(xaxis,a_array, linewidth=0.2,color='red')
116     mu = 0
117     # a_array[N] is central value out of 2N+1
118     sigma = 1 / (a_array[N]*np.sqrt(2*np.pi))
119     pdf_values = norm.pdf(xaxis, loc=mu, scale=sigma)
120     plt.plot(xaxis, pdf_values, linewidth=0.2, color='green')
121     plt.show()
122     return()
123
124
125 #--- Main
126
127 N = 200  # length of kernel a
128 M = 800  # length of signal x (starts at 0)
129 if M < 2*N:
130     print("Error: M must be > 2N")
131     exit()
132
133 h = build_kernel(N)
134 fit_kernel_to_Gaussian(h)
135
136 params = [0.5, 4.16, 0.75, 5, 500]
137 ctypes = {'eta_cos':'orange','truncated':'red'}
138
139 for type in ctypes:
140
141     x = f(type, M, params)
142     y = {}
143     xval = []
144     yval = []
145     for t in range(N, M-N):
146         y[t] = 0
147         for k in range(2*N+1):
148             # line below does this: y[t] += h[k]*x[t-N+k]
149             update_hash(y, t, h[k]*x[t-N+k])
150         xval.append(t)
151         yval.append(y[t])
152     xval = np.array(xval)
153     yval = rescale(yval)
154     if type == 'eta_cos':
155         period, phase = detect_period(yval, N)
156         print("Period: %8.5f\nPhase: %8.5f" %(period, phase))
157         cosine_fit = np.cos((xval-phase)*2*np.pi/period)
158         plt.scatter(xval, cosine_fit, s=0.4, c='blue',label='cosine fit')
159     plt.plot(xval, yval, linewidth=0.5, c=ctypes[type], label=type)
160
161 eta_cos = []
162 xval = []
163 for t in np.arange(N, M-N, 0.1):
164     xval.append(t)
165     eta_cos.append(eta('eta_cos',t,params))
166 eta_cos = rescale(eta_cos)
167 plt.plot(xval, eta_cos, label='original eta',c='green',linewidth=0.5)
168 plt.legend(loc='upper right')
169 plt.show()
```

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]. 13, 15, 24, 33, 46

[2] Christoph Aistleitner et al. Normal numbers: Arithmetic, computational and probabilistic aspects. 2016. Workshop [Link]. 13, 24, 33, 46

[3] Adel Alamadhi, Michel Planat, and Patrick Solé. Chebyshev's bias and generalized Riemann hypothesis. *Preprint*, pages 1–9, 2011. arXiv:1112.2398 [Link]. 85

[4] Gökalp Alpan and Maxim Zinchenko. Lower bounds for weighted Chebyshev and orthogonal polynomials. *Preprint*, 2024. arXiv:2408.11496v [Link]. 57

[5] David Bailey, Jonathan Borwein, and Neil Calkin. *Experimental Mathematics in Action*. A K Peters, 2007. 12, 24, 33, 46, 62

[6] Verónica Becher, A. Marchionna, and G. Tenenbaum. Simply normal numbers with digit dependencies. *Mathematika*, 69:988–991, 2023. arXiv:2304.06850 [Link]. 13, 24, 33, 46

[7] Frederik Broucke. On zero-density estimates for Beurling zeta functions. *Preprint*, pages 1–24, 2024. arXiv:2409:1051v1 [Link]. 78

[8] James Dolan. Carrying is a 2-cocycle. *Preprint*, pages 1–9, 2023. [Link]. 13, 24, 33, 46

[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]. 71

[10] Taylor Dupuy and David E. Weirich. Bits of in binary, Wieferich primes and a conjecture of Erdős. *Journal of Number Theory*, 158:268–280, 2016. [Link]. 103

[11] 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]. 13, 24, 33, 46

[12] P. M. Gauthier. Approximating the Riemann zeta-function by polynomials with restricted zeros. *Canadian Mathematical Bulletin*, 62(3):475–478, 2018. [Link]. 132

[13] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLT, 2022. [Link]. 78, 131

[14] Vincent Granville. *Synthetic Data and Generative AI*. MLT, 2022. [Link]. 78

[15] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLT, 2023. [Link]. 8, 11, 12, 13, 15, 16, 19, 24, 33, 45, 46, 58, 62, 70, 71, 78, 79, 84, 85, 98, 101

[16] Vincent Granville. *Building Disruptive AI & LLM Technology from Scratch*. MLT, 2024. [Link]. 16, 24, 33, 46, 127

[17] Vincent Granville. *State of the Art GenAI & LLMs, Creative Projects & Solutions*. MLT, 2024. [Link]. 21, 85

[18] Vincent Granville. *Statistical Optimization for AI and Machine Learning*. MLT, 2024. [Link]. 79

[19] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 83

[20] Vincent Granville. *Blueprint: Next-Gen Enterprise RAG & LLM 2.0 – Nvidia PDFs Use Case*. 2025. MLT [Link]. 127

[21] Vincent Granville. Simple, efficient, secure, accurate enterprise AI xLLM 2.0 architecture & operating system. 2025. BondingAI internal report `bdai-scores.pdf`, July 2025. 127

[22] Vincent Granville. *No-Blackbox, Secure, Efficient AI and xLLM Solutions*. MLT, 2026. [Link]. 96, 101, 127, 132

[23] Vincent Granville and Richard L Smith. Disaggregation of rainfall time series via Gibbs sampling. *NISS Technical Report*, pages 1–21, 1996. [Link]. 131

[24] Emil Grosswald. Oscillation theorems of arithmetical functions. *Transactions of the American Mathematical Society*, 126:1–28, 1967. [Link]. 85

[25] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 86

[26] 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]. 86

[27] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 86

[28] C. P. Hughes and A. Nikeghbali. The zeros of random polynomials cluster uniformly near the unit circle. *Compositio Mathematica*, 144:734–746, 2008. [Link]. 109

[29] Christopher Lutsko, Athanasios Sourmelidis, and Niclas Technau. Pair correlation of the fractional parts of αn^{θ}. *Journal of the European Mathematical Society*, 27:4069–4082, 2025. arXiv:2106.09800 [Link]. 72

[30] 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]. 33, 46

[31] Marcus Michelen and Oren Yakir. Limit law for root separation in random polynomials. *Preprint*, pages 1–77, 2025. arXiv:2505.02723 [Link]. 109

[32] 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]. 13, 24, 33, 46

[33] Mohammadamin Moradi et al. Data-driven model discovery with Kolmogorov-Arnold networks. *Preprint*, pages 1–6, 2024. arXiv:2409.15167 [Link]. 25, 33, 46

[34] 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]. 24, 33, 46

[35] Alan Oppenheim and Ronald W. Schafer. *Discrete–Time Signal Processing*. Prentice Hall, 1999. 134

[36] Michel Planat and Patrick Solé. Efficient prime counting and the Chebyshev primes. *Preprint*, pages 1–15, 2011. arXiv:1109.6489 [Link]. 85

[37] Eric S. Rowland. Regularity versus complexity in the binary representation of 3^n. *Complex Systems*, 18:367–377, 2009. [Link]. 103

[38] Frank Ruskey. Generating functions for the digital sum and other digit counting sequences. *Journal of Integer Sequences*, 12:1–9, 2009. [Link]. 95

[39] Klaus Schiefermayr and Maxim Zinchenko. Norm estimates for Chebyshev polynomials, i. *Journal of Approximation Theory*, 265, 2021. [Link]. 57

[40] Jan-Christoph Schlage-Putcha and Jasson Vindas. The prime number theorem for Beurlings generalized numbers – new cases. pages 1–26, 2011. [Link]. 78

[41] Maxwell Schneider and Robert Schneider. Digit sums and generating functions. *Preprint*, pages 1–10, 2018. arXiv:1807.06710 [Link]. 95

[42] Terence Tao. Limit law for root separation in random polynomials. *Tao's blog*, 2013. [Link]. 109

[43] Terence Tao. Biases between consecutive primes. *Tao's blog*, 2016. [Link]. 85

[44] Van Vu Terence Tao. Limit law for root separation in random polynomials. *Preprint*, pages 1–56, 2014. arXiv:1307.4357 [Link]. 109

[45] Yury V. Tiumentsev and Mikhail V. Egorchev. *Neural Network Modeling and Identification of Dynamical Systems*. Elsevier, 2019. 24, 33, 46

[46] Chukwudubem Umeano and Oleksandr Kyriienko. Ground state-based quantum feature maps. *Preprint*, pages 1–8, 2024. arXiv:2024.07174 [Link]. 13, 24, 33, 46

[47] Joseph Vandehey. On the binary digits of $\sqrt{2}$. *Preprint*, pages 1–6, 2017. arXiv:1711.01722 [Link]. 12, 24, 33, 46, 62

[48] Troy Vasiga and Jeffrey Shallit. On the iteration of certain quadratic maps over GF(p). *Discrete Mathematics*, 277:219–240, 2004. [Link]. 24, 33, 45

[49] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, second edition, 2012. 13, 24, 33, 46

[50] Nhan D. V. Nguyen Yen Q. Do. Limit law for root separation in random polynomials. *Electronic Journal of Probability*, 30:1–37, 2025. [Link]. 109

[51] 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]. 24, 33, 46

Index

p-adic numbers, 6, 115
p-adic valuation, 75
p-value, 120

agent (artificial intelligence), 25
agent-based modeling, 34
AI–generated art, 111
algebraic number, 33
analytic continuation, 83
analytic functions, 82
asymptotic periodicity, 96
attractor distribution, 12
autoconvolution (see self-convolution), 19
autocorrelation, 70
autocorrelation function, 34, 97, 118
    time lag, 43

basin of attraction, 79
Benford's law, 103
Bernoulli convolution, 15
Bernoulli distribution, 95
Bernoulli process, 45
beta distribution, 12
Beurling prime, 74
bifurcation phase, 20
binomial coefficients, 25
    central coefficient, 15
blancmange curve, 13, 24, 33, 46
BQ (base quadratic map), 42
Brownian motion, 20, 34, 84, 98

canonical form (strings), 6
Cantor set, 46
carry digit function, 13, 24, 33, 46
carry-lookahead addition, 119
Catalan numbers, 15
causal model
    non-causal, 132
central-limit theorem, 20
Champernowne constant, 119
chaotic convergence, 94
chaotic phase, 20, 42
characteristic polynomial, 96
Chebyshev polynomials, 56, 57
Chebyshev's bias, 83, 85, 86
checksum, 111
class (string or number), 7
cocycle, 13, 24, 33, 46
Collatz conjecture, 96
collision test, 118
combinatorial complexity, 117
compression, 118
computational intelligence, 25
congruential class, 21, 31, 43
conjugate maps, 19
convergence
    absolute, 74
    acceleration, 78
    chaotic convergence, 94
    conditional, 83
    fractal convergence, 94
convolution, 7
    auto-convolution, 7
    iterated self-convolution, 29
    self-convolution, 19, 29
convolution product
    deconvolution, 133
    discrete Gaussian, 132
Copeland–Erdős constant, 119
coprime integers, 15
correlation, 71
    cross-correlations, 15, 34
    empirical correlation, 15, 70
Cramér's conjecture, 78
cryptographically secure, 117
cubic equation, 96
curve fitting, 86, 134

deconvolution, 133
deep neural network, 24, 33, 46, 119
deterministic AI, 115
deterministic PRNG, 117
digit count, 13, 19
digit sum function, 13, 15, 19, 24, 33, 42, 46, 94, 105
    adjusted digit sum, 30, 38, 42
Dirichlet L-function, 83
Dirichlet character, 83
Dirichlet eta function, 74, 132
Dirichlet series, 83
Dirichlet's theorem, 87
dyadic map, 11, 12, 15, 19, 29, 56
dyadic rational, 32, 33, 102, 106
dynamical systems, 41, 79
    chaotic, 52, 79, 100
    quadratic map, 58
    state space, 58

empirical distribution function, 101
empirical probability density function, 98
entropy test, 118
equidistribution, 58, 118, 119
ergodic mean, 101

ergodicity, 12, 19, 100
  ergodic mean, 100
Euler product, 74, 82
Euler's transform, 131
Euler-Mascheroni constant, 130
experimental math, 82

filter
  blurring, 134
fixed point, 33, 52
flat polynomial, 107
fractal, 45
fractal convergence, 94
fractional part function, 15
functional equation, 12, 101

generating function, 15, 48, 95, 105
generative AI (GenAI), 87
geometric mean
  complex numbers, 57
goodness-of-fit, 86
gradient descent
  stochastic, 96

Hamming weight, 13, 24, 33, 46, 94
high-performance computing, 57, 116
homeomorphism, 11, 19

invariant measure, 12, 19, 29, 72, 100, 101
inverse transform, 29
irrational number, 15

kernel method
  Gaussian kernel, 132
Kolmogorov complexity, 119

Laurent series, 134
law of the iterated logarithm, 20
leading digits, 103
Littlewood polynomial, 107
Littlewood's oscillation theorem, 85
LLM (large language model), 24, 34
logarithmic capacity, 57
logistic map, 12, 19, 24, 29, 33, 41, 45
loss function
  adaptive, 96
LZMA compression, 118

Mandelbrot set, 41
map (dynamical systems), 12
Mersenne Twister, 117, 119
Mertens' theorem, 130
multi-branch function, 21
multiplicative function (random), 86
multiplicative order, 13, 21
Möbius function, 108

Newman polynomial, 107
next-bit test, 119
normal number, 12, 19, 24, 33, 34, 46, 49, 56, 103, 107, 109, 119
  simply normal, 45
  strongly normal, 58
NPG (non-periodic PRNG), 115
number representation, 6

odometer map, 15
orbit (dynamical systems), 52, 105

partition function, 116
PCG64, 117
  PCG64DXSM, 117
period (rational numbers), 15
periodicity
  asymptotic, 96
Perron-Frobenius operator, 101
pigeonhole principle, 49, 101
Pisot number, 107
prime race, 85
primorial, 30, 32, 35
PRNG (pseudo-random generator), 24, 33, 34, 46, 115
  cryptographically secure (CSPRNG), 117
  deterministic, 117
  Mersenne Twister, 117
  non-periodic, 115
  NPG, 115
  PCG64, 117
  PCG64DXSM, 117
  PRNG state, 116
  strong PRNG, 15, 71, 117
  tests of randomness, 118
Python library
  Gmpy2, 34
  MPmath, 82, 86
  PrimePy, 86
  Scipy, 86

quadratic convergence, 33
quadratic dynamical systems, 24, 29, 33, 45
quadratic irrational, 15
quadratic map, 41, 58
  base quadratic map, 41
quantization, 103
quantum cryptography, 13, 24, 33, 46
quantum derivative, 21, 84
quantum dynamics, 31, 34, 42
quantum function, 21, 30
quantum map, 13, 24, 33, 46
quantum state, 24, 43, 78, 85, 96
quantum system
  quantum convergence, 77
  sub-quantum states, 74

R-squared, 86
Rademacher distribution, 86
Rademacher function (random), 86
random number generation (see PRNG), 58
random polynomial, 108
random walk, 20, 84
randomness test, 71
reciprocal distribution, 12, 19, 29
replicability, 71, 111, 115, 116
residue class, 30, 96
Riemann Hypothesis, 78, 82, 131

Riemann zeta function, 74, 82, 105
root–finding algorithm, 111
run (of same digit), 15
run test, 118

Salem number, 107
scaling factor, 86
seed (dynamical systems), 12
seed (PRNG), 72, 115, 116
seed string, 8, 19, 29, 42
    reverse seed, 30
Shapiro polynomial, 107
signal processing, 132
slow growth function, 48
spectral radius, 58
spectral test, 118
spectral view, 43
square root (of a string), 7
square root operator, 29
state (PRNG), 116
state space, 58
stationary process
    non-stationary, 43
Stern's diatomic series, 96
Stirling numbers, 95
stochastic gradient descent, 96
string class, 6
string convergence, 7
swarm optimization, 96
symbolic mathematics, 96
synthetic data, 34
synthetic function, 87
synthetic numbers, 74, 78

Thue-Morse sequence, 107
time series
    disaggregation, 131
transfer function, 134
trinomial coefficients, 132
truncation, 6, 19

undecidable, 119
universal property, 111

Weyl criterion (normality), 56, 58
Weyl's equidistribution theorem, 102

Z-transform, 134