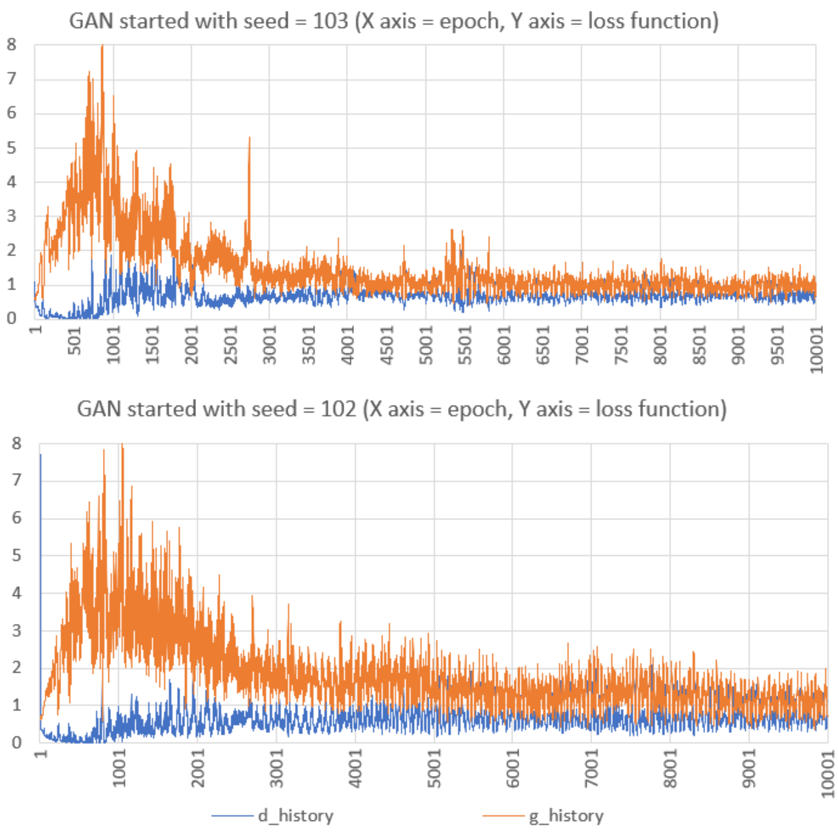
Practical AI & Machine Learning Projects and Datasets



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 3.0, December 2023

Preface

This book is intended to participants in the AI and machine learning certification program organized by my AI/ML research lab MLtechniques.com. It is also an invaluable resource to instructors and professors teaching related material, and to their students. If you want to add enterprise-grade projects to your curriculum, with deep technical dive on modern topics, you are welcome to re-use my projects in your classroom. I provide my own solution to each of them.

This book is also useful to prepare for hiring interviews. And for hiring managers, there is plenty of original questions, encouraging candidates to think outside the box, with applications on real data. The amount of Python code accompanying the solutions is considerable, using a vast array of libraries as well as home-made implementations showing the inner workings and improving existing black-box algorithms. By itself, this book constitutes a solid introduction to Python and scientific programming. The code is also on my GitHub repository.

The topics cover generative AI, synthetic data, machine learning optimization, scientific computing with Python, experimental math, synthetic data and functions, data visualizations and animations, time series and spatial processes, NLP and large language models, as well as graph applications and more. It also includes significant advances on some of the most challenging mathematical conjectures, obtained thanks to modern computations. In particular, intriguing new results regarding the Generalized Riemann Hypothesis, and a conjecture regarding record run lengths in the binary digits of $\sqrt{2}$. Most projects are based on real life data, offered with solutions and Python code. Your own solutions would be a great addition to your GitHub portfolio, bringing your career to the next level. Hiring managers, professors, and instructors can use the projects, each one broken down in a number of steps, to differentiate themselves from competitors. Most offer off-the-beaten path material. They may be used as novel exercises, job interview or exam questions, and even research topics for master or PhD theses.

To see how the certification program works, check out our FAQ posted here, or click on the "certification" tab on our website MLtechniques.com. Certifications can easily be displayed on your LinkedIn profile page in the credentials section. Unlike many other programs, there is no exam or meaningless quizzes. Emphasis is on projects with real-life data, enterprise-grade code, efficient methods, and modern applications to build a strong portfolio and grow your career in little time. The guidance to succeed is provided by the founder of the company, one of the top experts in the field, Dr. Vincent Granville. Jargon and unnecessary math are avoided, and simplicity is favored whenever possible. Nevertheless, the material is described as advanced by everyone who looked at it.

The related teaching and technical material (textbooks) can be purchased at MLtechniques.com/shop/. MLtechniques.com, the company offering the certifications, is a private, self-funded AI/ML research lab developing state-of-the-art open source technologies related to synthetic data, generative AI, cybersecurity, geospatial modeling, stochastic processes, chaos modeling, and AI-related statistical optimization.

About the author

Vincent Granville is a pioneering data scientist and machine learning expert, co-founder of Data Science Central (acquired by TechTarget), founder of MLTechniques.com, former VC-funded executive, author and patent owner.

Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET. Vincent is also a former post-doc at Cambridge University, and the National Institute of Statistical Sciences (NISS). 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 also the author of multiple books, available here. He lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, experimental math and probabilistic number theory.

Contents

1 Getting Started **5**
1.1 Python, Jupyter Notebook, and Google Colab . . . 5
1.1.1 Online Resources and Discussion Forums . . . 6
1.1.2 Beyond Python . . . 7
1.2 Automated data cleaning and exploratory analysis . . . 7
1.3 Tips to quickly solve new problems . . . 8
1.3.1 Original solution to visualization problem . . . 8
1.3.2 New solution, after doing some research . . . 10

2 Machine Learning Optimization **12**
2.1 Fast, high-quality NoGAN synthesizer for tabular data . . . 12
2.1.1 Project description . . . 12
2.1.2 Solution . . . 14
2.1.3 Python implementation . . . 15
2.2 Cybersecurity: balancing data with automated SQL queries . . . 21
2.2.1 Project description . . . 22
2.2.2 Solution . . . 22
2.2.3 Python code with SQL queries . . . 23
2.3 Good GenAI evaluation, fast LLM search, and real randomness . . . 24
2.3.1 Project description . . . 25
2.3.2 Solution . . . 27
2.3.3 Python implementation . . . 29

3 Time Series and Spatial Processes **36**
3.1 Time series interpolation: ocean tides . . . 36
3.1.1 Project description . . . 37
3.1.2 Note on time series comparison . . . 38
3.1.3 Solution . . . 39
3.2 Temperature data: geospatial smoothness and interpolation . . . 43
3.2.1 Project description . . . 44
3.2.2 Solution . . . 45

4 Scientific Computing **54**
4.1 The music of the Riemann Hypothesis: sound generation . . . 54
4.1.1 Solution . . . 55
4.2 Cross-correlations in binary digits of irrational numbers . . . 57
4.2.1 Project and solution . . . 57
4.3 Longest runs of zeros in binary digits of $\sqrt{2}$ . . . 58
4.3.1 Surprising result about the longest runs . . . 59
4.3.2 Project and solution . . . 60
4.4 Quantum derivatives, GenAI, and the Riemann Hypothesis . . . 62
4.4.1 Cornerstone result to bypass the roadblocks . . . 63
4.4.2 Quantum derivative of functions nowhere differentiable . . . 64
4.4.3 Project and solution . . . 65
4.4.4 Python code . . . 69

5 Generative AI **74**
5.1 Holdout method to evaluate synthetizations . . . 74
5.1.1 Project description . . . 75
5.1.2 Solution . . . 76

5.2 Enhanced synthetizations with GANs and copulas . . . 79
5.2.1 Project description . . . 79
5.2.2 Solution . . . 81
5.2.3 Python code . . . 82
5.3 Difference between synthetization and simulation . . . 92
5.3.1 Frequently asked questions . . . 92
5.3.2 Project: synthetizations with categorical features . . . 93
5.3.3 Solution . . . 94
5.4 Music, synthetic graphs, LLM, and agent-based models . . . 95

6 Data Visualizations and Animations **96**
6.1 Synthesizing data outside the observation range . . . 96
6.1.1 Animated histograms for extrapolated quantiles . . . 96
6.1.2 Python code: video, thumbnails . . . 97
6.2 Curve fitting in bulk . . . 100
6.2.1 Regression without dependent variable . . . 101
6.2.2 Python code . . . 101
6.3 Gradient descent, grids, and maps . . . 106
6.4 Supervised classification with image bitmaps . . . 106
6.5 Miscellaneous topics . . . 106
6.5.1 Agent-based modeling: collision graphs . . . 106
6.5.2 Terrain generation: image and palette morphing . . . 106
6.5.3 Mathematical art . . . 106

7 NLP and Large Language Models **107**
7.1 Synthesizing DNA sequences with LLM techniques . . . 107
7.1.1 Project and solution . . . 107
7.1.2 Python code . . . 110
7.2 Creating high quality LLM embeddings . . . 114
7.2.1 Smart, efficient, and scalable crawling . . . 114
7.2.2 Creating the weighted keyword association graph . . . 118
7.2.3 Category-specific embeddings and other components . . . 118
7.2.4 RAG: retrieval-augmented generation using book catalogs . . . 118
7.3 Scoring content with hidden decision trees . . . 118

A Glossary: GAN and Tabular Data Synthetization **119**

Bibliography **123**

Index **125**

```
print(data_A)

data_C = data_B.drop(['src_port','size'], axis=1)
data_C = data_C.groupby(data_C.columns.tolist(), as_index=False).size()
data_C1 = data_C[(data_C['bidirectional_mean_ps'] == 60) |
             (data_C['bidirectional_mean_ps'] == 1078) |
             (data_C['size'] > 1)]
data_C2 = data_C[(data_C['bidirectional_mean_ps'] != 60) &
             (data_C['bidirectional_mean_ps'] != 1078) &
             (data_C['size'] == 1)]
print(data_C)
data_C1.to_csv('iot_C1.csv')
data_C2.to_csv('iot_C2.csv')

data_B_full = data_B.join(data.set_index(features), on=features, how='inner')
features.remove('src_port')
data_C1_full = data_C1.merge(data_B_full, how='left', on=features)
data_C2_full = data_C2.merge(data_B_full, how='left', on=features)
data_C1_full.to_csv('iot_C1_full.csv')
data_C2_full.to_csv('iot_C2_full.csv')

map_C1 = data_C1_full.groupby('src_port')['src_port'].count()
map_C2 = data_C2_full.groupby('src_port')['src_port'].count()
map_C1.to_csv('iot_C1_map.csv')
map_C2.to_csv('iot_C2_map.csv')

data_C1 = data_C1_full.drop(['src_port','size_x', 'size_y'], axis=1)
data_C1 = data_C1.groupby(data_C1.columns.tolist(), as_index=False).size()
data_C2 = data_C2_full.drop(['src_port','size_x', 'size_y'], axis=1)
data_C2 = data_C2.groupby(data_C2.columns.tolist(), as_index=False).size()
data_C1.to_csv('iot_C1.csv')
data_C2.to_csv('iot_C2.csv')
```

2.3 Good GenAI evaluation, fast LLM search, and real randomness

In this section, I cover several topics in detail. First, I introduce one of the bests random number generators (PRNG) with infinite period. Then, I show how to evaluate the synthesized numbers using the best metrics. Finally, I illustrate how it applies to other contexts, such as large language models (LLM). In particular, the system is based on words with letters from an arbitrary alphabet, and it can be adapted to any prespecified multivariate distribution, not just uniform: the joint ECDF (empirical distribution) attached to a training set in GenAI systems, for instance. At each step, the focus is both on quality and speed, revisiting old methods or inventing new ones, to get solutions performing significantly better and requiring much less computing time. The three components of this system are:

New powerful random number system

In its simplest form, the random numbers are defined as the binary digits $d_n = x_n \bmod 2$, from the sequence $x_{n+1} = 3 \cdot (x_n//2)$, where the double slash is the integer division [Wiki]. It is an improvement over binary digits of quadratic irrationals used previously (see section 4.4 in [12]) in the sense that x_n grows only by a factor $3/2$ at each iteration, rather than 2. All sequences (x_n) that do not grow indefinitely necessarily result in periodic numbers. This is the case for all PRNGs on the market.

In addition, despite having very long periods, these random generators with finite periods exhibit subtle patterns in rather low dimensions: in short, lack of randomness. They can be quite sensitive to the seed and may require many warm-up iterations before reaching higher randomness. See here how you can crack the Mersenne twister used in the Numpy random function. The question is this: how slowly can x_n grow while preserving perfect randomness, fast implementation, and an infinite period? Read on to see how I managed to reduce the aforementioned exponential growth down to linear, while keeping an infinite period.

Ultrafast, robust evaluation metrics

The first step is to define what a strongly random sequence is, when it consists of deterministic digits. Details are again in chapter 4 in [12]. The takeaway: you need a metric that captures just that, when testing your system. This is true for all GenAI systems. Indeed, here I am re-using the full multivariate Kolmogorov-Smirnov distance (KS) specifically implemented in the context of synthetic data generation: see section 6.4.2 in [7] for details. There, I showed how poorly implemented metrics used by vendors fail to capture subtle departures from the target distribution.

In this section, I present a very fast implementation of KS. I also include a few other tests. Very large test batteries exist, for instance Diehard [Wiki]. However, most rely on old statistical practice, offering a large number of disparate, weak tests, rather than a centralized approach to dealing with the problem. You can do a lot better with much fewer tests. This is one of the goals of this project, also with a focus on hard-to-detect patterns.

Also note that the KS distance relies on the CDF rather than the PDF (probability density function). The latter, used in many tests such as Chi-squared, does not work when you have billions of cross-feature buckets in high dimensions, each with very few observations. As in many GenAI systems, this is what we face. To give you an idea, think about counting occurrences of billions of "words" such as

$$3210232010310223034123103323103003110231 02$$

in a sequence of trillions of digits in base 4 (in this case, the alphabet has 4 letters). Most counts will be zero. Likewise, the base (that is, the size of the alphabet) may be a very large integer. The KS distance handles this problem transparently by looking at closest strings found in the digit sequences, themselves having only one occurrence most of the time. Also, it easily takes care of conditional probabilities when needed.

My previous KS implementation involved thousands of Pandas SQL queries spanning across many features. The new version discussed here is based on the radix numeration system [Wiki], turning long strings in big integers (called blocks), allowing for fast retrieval with simple binary search in a list of big numbers. In this context, a block can have many digits: the k-th feature is the k-th digit, although blocks may have a varying number of digits. I implicitly rely on the Python Bignum library [Wiki] to deal with the computations. Finally, the binary search is further improved and called weighted binary search, accelerating the computations by a factor 3 or 4 in the examples tested. So far, I did not compare with other methods such as vector search based on KNN and ANN (approximate nearest neighbors). But these methods are very relevant in this context.

Connection to LLM

The above paragraphs establish the connection to large language models. The problem is strikingly similar to DNA sequence synthetization discussed in section 7.1, where the alphabet has four letters (A, C, G, T) and the words consist of DNA subsequences. The main difference is that DNA sequences are far from random. Yet, the methodology presented here can easily be adapted to arbitrary target distributions. In particular to empirical distributions like those associated to DNA sequencing, or keyword distributions in ordinary text.

Then, as illustrated in the DNA sequencing problem, predictive analytics for GenAI may rely on conditional probabilities such as $P(B_1|B_2)$, where B_1, B_2 are consecutive blocks. Transitioning from KS and the multivariate CDF to conditional probabilities is straightforward with the formula $P(B_1|B_2) = P(B_1, B2)/P(B_2)$.

2.3.1 Project description

We are dealing with sequences of positive integers defined by a recursion $x_{n+1} = f(x_n) \bmod \tau_n$, where x_0 is the initial condition, referred to as the main seed. The choice of f leads to x_n randomly going up and down as n increases. On average, the trend is up: eventually, x_n gets bigger and bigger on average, albeit rather slowly to allow for fast computations. The **digits** – what the generated random numbers are – are simply defined as $d_n = x_n \bmod b$, where b is called the **base**.

The big contrast with classic random generators is that τ_n depends on n rather than being fixed. Here τ_n strictly increases at each iteration, allowing x_n to grow on average. It leads to more random digits, and an infinite period. In the current implementation, $\tau_{n+1} = s + \tau_n$ where s is a moderately large integer. Thus, we have two additional seeds: the step s, and τ_0. The function f depends on two integer parameters p, q. More specifically,

$$x_{n+1} = p \cdot (x_n//q) \bmod \tau_n, \quad \text{with } p > q > 1,\ x_0, \tau_0 \geq 0, \text{ and } \tau_{n+1} = s + \tau_n. \tag{2.1}$$

Again, the double slash stands for the integer division: $x_n//q = \lfloor x_n/q \rfloor$ where the brackets denote the floor function, or `int` in Python. I use the integer division in Python as it works when x_n becomes extremely large, unlike the standard division combined with `int`.

I implemented three tests of randomness. Before going into the details, let me introduce the concept of **block**. A block consists of a fixed number of digits, used for testing purposes. They are encoded as big integers. For now, let's pretend that a block B has m digits, each with a variable base: the k-th digit is in base b_k, thus with $0 \leq d_k < b_k$. Then B can be uniquely encoded with the one-to-one mapping

$$B = d_1 + d_2 \times b_1 + d_3 \times b_1 b_2 + d_4 \times b_1 b_2 b_3 + \cdots + d_m \times b_1 b_2 \cdots b_{m-1} \tag{2.2}$$

In this case, B is represented by an integer strictly smaller than $b_1 b_2 \cdots b_m$. In the context of real data, each digit represents a feature; the k-th feature can take on b_k different values after approximation or truncation, and a block corresponds to a row in a tabular data set. Of course, the b_k' can be quite large and different, depending on the feature, especially for numerical features. At the other extreme, for a 2-category feature, $b_k = 2$. The numeration system based on (2.2) is called the radix system. It is a generalization of the numeration system in fixed base b. When a new observation is generated, it is first encoded as a block, then the closest match to any row in the training set can be found with a binary search on all long integers B between 0 and $(b_1 \cdots b_m) - 1$, each one representing a potential row in the training set. Note that in general, the existing B's cover a small subset of all potential values: we are dealing with sparsity. We will use the same search algorithm here, with a fixed base b. It makes sense to call it radix search.

Now I can provide a quick overview of the three tests of randomness. The most involved is the block test: it is based on the multivariate KS distance, which in turn relies on radix search. The tests are:

- Run test. The max run test is implemented in section 4.3 for binary digits of quadratic irrationals. The solution presented here is more comprehensive, looking at runs or arbitrary length for all possible digits. These run lengths have a geometric distribution if digits show up randomly. When the base b is very large, there are too many values to fit in a table, so I also compute the following statistics:

$$R(L) = \sum_{d=0}^{b-1} \left(\frac{\rho(L,d) - \rho_0(L,d)}{\sigma(L,d)} \right)^2, \quad L = 1, 2, \dots \tag{2.3}$$

  where $\rho_0(L,d)$ is the number of runs of length L found in the sequence for digit d, and $\rho(L,d)$ is the expected count if the sequence is random; then, it does not depend on d. Finally, $\sigma^2(L,d)$ is the theoretical variance, not depending on d, to normalize $R(L)$ so that it has a Chi-squared distribution with b degrees of freedom. In short, an unexpectedly large $R(L)$ indicates lack of randomness.

- Spectral test. This test looks at autocorrelations of lag 1, 2, and so on in the digit sequence, to check out whether their behavior is compatible or not with randomness. Because the sequences can be extremely long, the computations are done on the flight, updated one new digit at a time using a buffer, without having to store the whole sequence anywhere.

- Block test. Create a sample of blocks B of size m, for instance equally spaced, and independently of the blocks found in the digit sequence. Here m is the number of digits per block, with each digit in base b. Hence, each block should occur in the infinite digit sequence with frequency b^{-m}, assuming randomness. Compute the CDF value $F(B)$ attached to the underlying theoretical (uniform) distribution, for each of these blocks. Then compute the ECDF or empirical CDF value $F_0(B)$ based on block occurrences: mostly nearest neighbors to B, found in the digit sequence. Finally, KS $= \sup |F(B) - F_0(B)|$, and the supremum is over all blocks B in your sample. To magnify any subtle departure from randomness, plot the following: $\delta(B) = F(B) - F_0(B)$ on the Y-axis, and B (in its integer representation) on the X-axis, for all B in your sample.

There is a considerable amount of new material to master, if you were to implement the whole system on your own. The goal is not to create your own code from scratch, but rather to understand and use mine. You are welcome to improve it and create Python objects for the various components. One of the easy tasks is to use the code to compare different random generators and assess the impact of parameters. Another goal is to generalize the code to other contexts such as synthesizing DNA sequences, where the target distribution is not uniform, but comes as an ECDF (empirical CDF) computed on the training set. This also involves conditional probabilities: predicting the next block given the previous ones, when blocks are auto-correlated.

The project consists of the following steps:

Step 1: Read and understand the code in section 2.3.3. Identify the different components: the three tests, the base, the digits, the generation of the digits, the blocks and block encoding, the seeds and parameters, and the binary search to locate nearby blocks in the sequence, given a specific block. Run the code (it is on GitHub), see and analyze the results. Understand the computations for the autocorrelation coefficients, as it is done on the fly, in a non-traditional way using a buffer.

Step 2: What are the values of $\rho(L,d)$ and $\sigma(L,d)$ in Formula 2.3? How would you optimize a binary search so that it requires fewer steps? To help answer this question, look at the variable `trials`, which counts the combined number of steps used in all the binary searches. Finally, save the generated digits in a file if the sequence is not too long.

Step 3: To improve block tests, work with blocks of various sizes. Also, compute the autocorrelations in the block sequence, in the same way it is done in the digit sequence. Given a small-size block B consisting of two sub-blocks B_1, B_2, estimate $P(B_2|B_1)$ on the data, and confirm the independence between B_1 and B_2, that is, $P(B_2|B_1) = P(B_2)$.

- **Step 4**: Try different seeds x_0, τ_0 and s to check out how the random generators is sensitive to the seeds. Does it change the quality of the randomness? In particular, try to find the lowest possible seed values that still lead to good random numbers. Then, try different combinations of p, q and b. Some work, some don't. Can you identify necessary conditions on p, q, b to guarantee good random numbers? Finally, can you find a good combination that makes the sequence (x_n) grow as slowly as possible, while still generating great random numbers. Obviously, you need $p > q$, for instance $p = q + 1$, and s as small as possible. A large base b also helps to extract as much as possible from the sequence (x_n), yet some combinations (p, q, b) don't work, and usually $b > q$ causes problems, making $b = q$ ideal.

2.3.2 Solution

Since much of the solution is my code in section 2.3.3, I cover only specific items in this section, with a focus on explaining and interpreting the output tables and visualizations. The main parameters are initialized in section [3] in the code, see lines 278 – 288. In particular, the current values of x_n, τ_n are stored in `x` and `tau` respectively, while s in Formula (2.1) is denoted as `step` in the code.

Figure 2.3 shows the function $\delta(B) = F(B) - F_0(B)$, that is, the difference between a perfectly uniform CDF and the approximate ECDF computed on the generated digits, using 500 equally-spaced test blocks, each with 6 digits. The number of blocks (500 here) is specified by `n_nodes` in the code. The higher the number, the better the approximation. The total number of digits if 640,000. I tried four sets of parameters labeled HM_1 to HM_4. The parameter sets are specific combinations of p, q, b. See lines 294, 316, 337 and 358 for the respective values in the code. I also added Numpy.random for comparison purposes. Note that HM stands for "Home-Made", by contrast to Numpy.

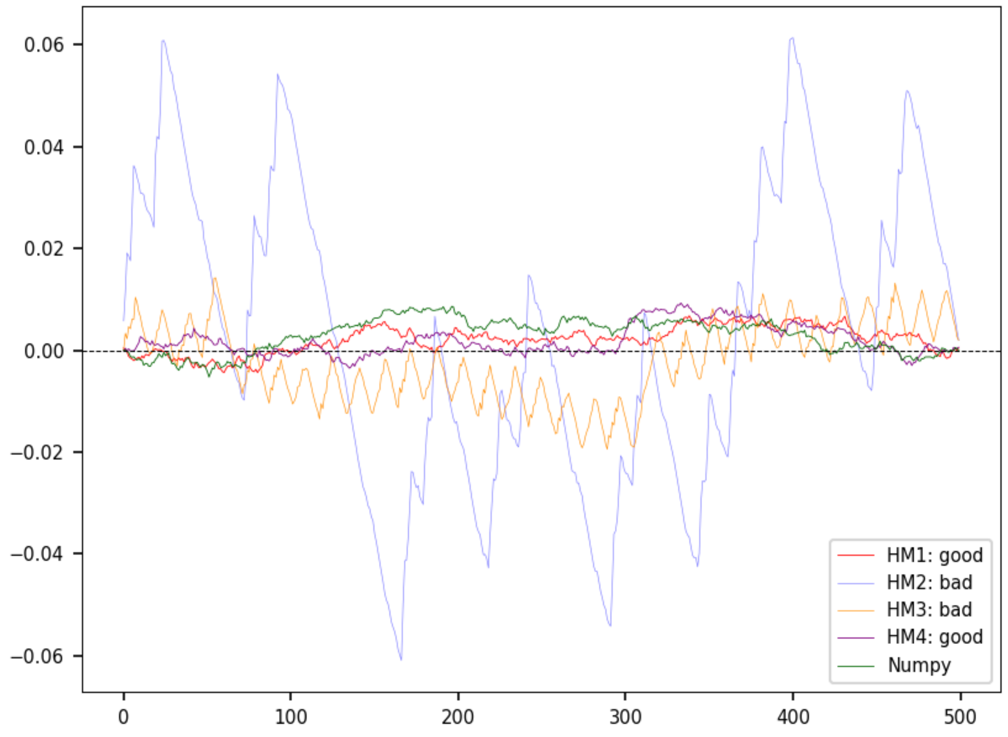


Figure 2.3: Four versions of my generator (HMx) compared to numpy.random

Clearly, HM_2 and HM_3 generate non-random numbers. As for HM_1, HM_4 and Numpy, they pass this test. It does not mean that they generate random digits. More tests are needed for verification, in particular with larger block sizes (the parameter `block_size`) and more nodes (the parameter `n_nodes`).

Table 2.1 focuses on HM_1 only. It shows the number of occurrences for runs of length L, for each of the 4 digits ($b = 4$) and various values of L. The sequence has 640,000 digits. The 4 rightmost columns are summary statistics: `Exp` and `Avg` are respectively the expected and observed counts, while `Norm` indicates whether or not all the counts, for a specific L, are compatible with perfect randomness. If the digits are truly random, then `Norm` approximately follows a standard normal distribution. In particular, `Norm` $= (R(L) - b)/(2b)$ where $R(L)$ is defined by (2.3). Clearly, HM$_1$ passes this test.

L	$d=0$	$d=1$	$d=2$	$d=3$	Exp	Avg	Chi2	Norm
1	8968	8932	9074	8999	9000	8993	1.44	−0.3202
2	2296	2227	2294	2149	2250	2242	6.81	0.3511
3	548	532	620	587	563	572	9.05	0.6315
4	146	132	143	131	141	138	1.44	−0.3204
5	36	39	38	36	35	37	0.69	−0.4136
6	3	11	8	10	9	8	4.61	0.0759
7	2	3	3	2	2	3	0.62	−0.4223
8	0	2	0	0	1	1	3.83	−0.0211

Table 2.1: Runs of length L per digit for HM$_1$, with summary stats

Table 2.2 shows summary statistics for the 4 home-made generators (HM) and Numpy. One of them (HM$_3$) has $b = 256$. The normal approximation to `Norm` is especially good when $b > 10$, and bad when the counts are small, that is, when L is large. All the values not compatible with the randomness assumption are highlighted in red. Thus, HM$_2$ and HM$_3$ do not produce random digits, confirming the findings from Figure 2.3. Here, AC stands for autocorrelations within digit sequences

Metric	Level	HM$_1$	HM$_2$	HM$_3$	HM$_4$	Numpy
Norm	$L=1$	−0.3202	1530.7	136.38	0.0885	−0.0471
	$L=2$	0.3511	249.3	95.80	−0.1944	0.3141
	$L=3$	0.6315	80.1	41.37	0.5242	0.8902
AC	Lag 1	0.0010	−0.0046	−0.0731	−0.0002	−0.0007
	Lag 2	−0.0011	0.0022	0.0048	0.0022	−0.0014
KS		0.00676	0.06136	0.01932	0.00929	0.00859

Table 2.2: High-level comparison of HM and Numpy generators, with red flags

Besides listing the parameters used in the simulation, Table 2.3 features two interesting metrics: last x_n, and `Trials`. The former indicates how fast the sequence x_n grows. Numpy is not based on growing sequences, resulting in digits that keep repeating themselves past the finite period. And for random binary digits based on quadratic irrationals, the last x_n would be of the order $2^N \approx 10^{193{,}000}$, where $N = 640{,}000$ is the number of digits in the sequence. By contrast, for the HM generators explored here, it is around 10^{10}.

Feature	HM$_1$	HM$_2$	HM$_3$	HM$_4$	Numpy
Last x_n	10^{10}	10^9	10^{10}	10^9	n/a
Trials	178	2197	3497	2293	150
Base	4	4	8	256	4
Block size	6	6	6	6	6
p	7	6	7	401	n/a
q	4	4	4	256	n/a

Table 2.3: Top parameters and special metrics

Finally, `Trials` represents the total number of steps needed in the binary search, combined over the 500 test blocks (500 = `n_nodes`). For each test block, the goal is to find the closest neighbor block in the digit sequence (640,000 digits). The `Trials` value depends on the base, the block size, the number of digits in the sequence, and the number of test blocks. A very small value means that most of the test blocks are already present in the digit sequence: for these test blocks, the binary search is not needed. In the table, a value of 2197 means that on average, each test block requires 2197/500 = 4.39 trials before finding the closest neighbor block. The lower `Trials`, the faster the computations. Because I use an optimized binary search, it is already 3–4 times faster than the standard method. The generation of random digits is also just as fast as Numpy.

Regarding the project steps, I now provide answers to selected questions. Let N be the number of digits in base b in the sequence, and π be the probability for a run to be of length $L > 0$, for any digit $d < b$. We are dealing with a binomial distribution of parameter (N, π). Thus,

$$\rho(L,d) = N\pi, \quad \sigma^2(L,d) = N\pi(1-\pi), \quad \text{with } \pi = \Big(\frac{b-1}{b}\Big)^2 \cdot \Big(\frac{1}{b}\Big)^L,$$

Note that are $\rho(L, d)$ and $\sigma^2(L, d)$ are respectively the expectation and variance of the binomial distribution. This answers the first question in Step 2. For the second question about the binary search, see lines 184 – 197 in the code. If you set `A=1` and `B=1` respectively in lines 187 and 188, it becomes a standard binary search, with computational complexity $O(\log_2 n)$ in all cases. With my choice of `A` and `B`, it is similar to interpolation search [Wiki], which is $O(\log_2 \log_2 n)$ for the average case when the underlying distribution is uniform. See also the recent article on interpolated binary search [23].

2.3.3 Python implementation

The code is also on GitHub, here.

```
1  import numpy as np
2  from collections import OrderedDict
3  
4  #--- [1] Functions to generate random digits
5  
6  def get_next_digit(x, p, q, tau, step, base, option = "Home-Made"):
7  
8     if option == "Numpy":
9        digit = np.random.randint(0, base)
10    elif option == "Home-Made":
11       x = ((p * x) // q) % tau # integer division for big int
12       tau += step
13       digit = x % base
14    return(digit, x, tau)
15 
16 
17 def update_runs(digit, old_digit, run, max_run, hash_runs):
18 
19    if digit == old_digit:
20       run += 1
21    else:
22       if (old_digit, run) in hash_runs:
23          hash_runs[(old_digit, run)] += 1
24       else:
25          hash_runs[(old_digit, run)] = 1
26       if run > max_run:
27          max_run = run
28       run = 1
29    return(run, max_run, hash_runs)
30 
31 
32 def update_blocks(digit, m, block, base, block_size, hash_blocks):
33 
34    if m < block_size:
35       block = base * block + digit
36       m += 1
37    else:
38       if block in hash_blocks:
39          hash_blocks[block] += 1
40       else:
41          hash_blocks[block] = 1
42       block = 0
43       m = 0
44    return(m, block, hash_blocks)
45 
```

```python

def update_cp(digit, k, buffer, max_lag, N, cp_data):

   # processing k-th digit starting at k=2
   # buffer stores the last max_lag digits
   # cp stands for the cross-product part (in autocorrelation)

   mu = cp_data[0]
   cnt = cp_data[1]
   cp_vals = cp_data[2]
   cp_cnt = cp_data[3]

   buffer[(k-2) % max_lag] = digit
   mu += digit
   cnt += 1

   for lag in range(max_lag):
      if k-2 >= lag:
         cp_vals[lag] += digit * buffer[(k-2-lag) % max_lag]
         cp_cnt[lag] += 1

   cp_data = (mu, cnt, cp_vals, cp_cnt)
   return(cp_data, buffer)


def generate_digits(N, x0, p, q, tau, step, base, block_size, max_lag, option =
    "Home-Made"):

   # Main function. Also produces output to test randomnes:
   #    - hash_runs and max_run: input for the run_test function
   #    - hash_blocks: input for the block_test function
   #    - cp_data input for the correl_test function

   # for run and block test
   hash_runs = {}
   hash_blocks = {}
   block = 0
   m = 0
   run = 0
   max_run = 0
   digit = -1

   # for correl_test
   mu = 0
   cnt = 0
   buffer = np.zeros(max_lag)
   cp_vals = np.zeros(max_lag) # cross-products for autocorrel
   cp_cnt = np.zeros(max_lag)
   cp_data = (mu, cnt, cp_vals, cp_cnt)

   x = x0

   for k in range(2, N):

      old_digit = digit
      (digit, x, tau) = get_next_digit(x, p, q, tau, step, base, option)
      (run, max_run, hash_runs) = update_runs(digit, old_digit, run, max_run, hash_runs)
      (m, block, hash_blocks) = update_blocks(digit, m, block, base, block_size,
          hash_blocks)
      (cp_data, buffer) = update_cp(digit, k, buffer, max_lag, N, cp_data)

   print("----------------")
   print("PRNG = ", option)
   print("block_size (digits per block), digit base: %d, %d", block_size, base)
   if option == "Home-Made":
      print("p, q: %d, %d" %(p, q))
      print(len(str(x)), "decimal digits in last x")
   return(hash_runs, hash_blocks, max_run, cp_data)


#--- [2] Functions to perform tests of randomness

def run_test(base, max_run, hash_runs):

   # For each run, chi2 has approx. chi2 distrib. with base degrees of freedom
   # This is true assuming the digits are random

   print()
   print("Digit ", end = " ")
   if base <= 8:
      for digit in range(base):
         print("%8d" %(digit), end =" ")
   print("  Exp", end = " ")
   print("  Avg", end = " ")  # count average over all digits
   print(" Chi2", end = " ")  # degrees of freedom = base
   print(" norm", end = " ")
   print("\n")

   for run in range(1, max_run+1):

      print("Run %3d" % (run), end = " ")
      prob = ((base-1)/base)**2 * (1/base)**run
      exp = N*prob
      var = N*prob*(1-prob)
      avg = 0
      chi2 = 0

      for digit in range(0, base):
         key = (digit, run)
         count = 0
         if key in hash_runs:
            count = hash_runs[key]
            avg += count
            chi2 += (count - exp)**2 / var
         if base <= 8:
            print("%8d" %(count), end =" ")

      avg /= base
      norm = (chi2 - base) / (2*base)
      print("%8d" %(int(0.5 + exp)), end =" ")
      print("%8d" %(int(0.5 + avg)), end =" ")
      print("%8.2f" %chi2, end =" ")
      print("%8.4f" %norm, end =" ")
      print()

   return()


def get_closest_blocks(block, list, trials):

   # Used in block_test
   # Return (left_block, right_block) with left_block <= block <= right_block,
   #  - If block is in list, left_block = block = right_block
   #  - Otherwise, left_block = list(left), right_block = list(right);
   #             they are the closest neighbors to block, found in list
   #  - left_block, right_block are found in list with weighted binary search
   #  - list must be ordered
   # trials: to compare spped of binary search with weighted binary search

   found = False
   left = 0
   right = len(list) - 1
   delta = 1
   old_delta = 0

   if block in list:
      left_block = block
      right_block = block

   else:
      while delta != old_delta:
         trials += 1
         old_delta = delta
         A = max(list[right] - block, 0) # in standard binary search: A = 1
         B = max(block - list[left], 0) # in standard binary search: B = 1
         middle = (A*left + B*right) // (A + B)
         if list[middle] > block:
            right = middle
         elif list[middle] < block:
            left = middle
         delta = right - left

      left_block = list[middle]
      right_block = list[min(middle+1, len(list)-1)]

   return(left_block, right_block, trials)


def true_cdf(block, max_block):
   # Used in block_test
   # blocks uniformly distributed on {0, 1, ..., max_block}
   return((block + 1)/(max_block + 1))


def block_test(hash_blocks, n_nodes, base, block_size):

   # Approximated KS (Kolmogorov-Smirnov distance) between true random and PRNG
   # Computed only for blocks with block_size digits (each digit in base system)
   # More nodes means better approximation to KS

   hash_cdf = {}
   hash_blocks = OrderedDict(sorted(hash_blocks.items()))
   n_blocks = sum(hash_blocks.values())
   count = 0
   trials = 0 # total number of iterations in binary search


   for block in hash_blocks:
      hash_cdf[block] = count + hash_blocks[block]/n_blocks
      count = hash_cdf[block]

   cdf_list = list(hash_cdf.keys())
   max_block = base**block_size - 1
   KS = 0
   trials = 0  # total number of iterations in binary search
   arr_cdf = [] # theoretical cdf, values
   arr_ecdf = [] # empirical cdf (PRMG), values
   arr_arg = [] # arguments (block number) associated to cdf or ecdf

   for k in range(0, n_nodes):

      block = int(0.5 + k * max_block / n_nodes)
      (left_block, right_block, trials) = get_closest_blocks(block, cdf_list, trials)
      cdf_val = true_cdf(block, max_block)
      ecdf_lval = hash_cdf[left_block] # empirical cdf
      ecdf_rval = hash_cdf[right_block] # empirical cdf
      ecdf_val = (ecdf_lval + ecdf_rval) / 2 # empirical cdf
      arr_cdf.append(cdf_val)
      arr_ecdf.append(ecdf_val)
      arr_arg.append(block)
      dist = abs(cdf_val - ecdf_val)
      if dist > KS:
         KS = dist

   return(KS, arr_cdf, arr_ecdf, arr_arg, trials)


def autocorrel_test(cp_data, max_lag, base):

   mu = cp_data[0]
   cnt = cp_data[1]
   cp_vals = cp_data[2]
   cp_cnt = cp_data[3]

   mu /= cnt
   t_mu = (base-1) / 2
   var = cp_vals[0]/cp_cnt[0] - mu*mu
   t_var = (base*base -1) / 12
   print()
   print("Digit mean: %6.2f (expected: %6.2f)" % (mu, t_mu))
   print("Digit var : %6.2f (expected: %6.2f)" % (var, t_var))
   print()
   print("Digit autocorrelations: ")
   for k in range(max_lag):
      autocorrel = (cp_vals[k]/cp_cnt[k] - mu*mu) / var
      print("Lag %4d: %7.4f" %(k, autocorrel))

   return()


#--- [3] Main section

# I tested (p, q) in {(3, 2), (7, 4), (13, 8), (401, 256)}

N = 64000        # number of digits to generate
p = 7            # p/q must > 1, preferably >= 1.5
q = 4            # I tried q = 2, 4, 8, 16 and so on only
base = q         # digit base, base <= q (base = 2 and base = q work)
x0 = 50001       # seed to start the bigint sequence in PRNG
tau = 41197      # co-seed of home-made PRNG
step = 37643     # co-seed of home-made PRNG
digit = -1       # fictitious digit before creating real ones
block_size = 6   # digits per block for the block test; must be integer > 0
n_nodes = 500    # number of nodes for the block test
max_lag = 3      # for autocorrel test
seed = 104       # needed only with option = "Numpy"
np.random.seed(seed)

#- [3.1] Home-made PRNG with parameters that work

p, q, base, block_size = 7, 4, 4, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                             base, block_size, max_lag,
                                             option="Home-Made")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf1, arr_arg1, trials) = block_test(hash_blocks, n_nodes,
                                         base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)
```

```python

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))


#- [3.2] Home-made, with parameters that don’t work

p, q, base, block_size = 6, 4, 4, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                              base, block_size, max_lag,
                                              option="Home-Made")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf2, arr_arg2, trials) = block_test(hash_blocks, n_nodes,
                                          base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))

#- [3.3] Home-made, another example of failure

p, q, base, block_size = 7, 4, 8, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                              base, block_size, max_lag,
                                              option="Home-Made")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf3, arr_arg3, trials) = block_test(hash_blocks, n_nodes,
                                          base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))

#- [3.4] Home-made, using a large base

p, q, base, block_size = 401, 256, 256, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                              base, block_size, max_lag,
                                              option="Home-Made")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf4, arr_arg4, trials) = block_test(hash_blocks, n_nodes,
                                          base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))


#- [3.5] Numpy PRNG (Mersenne twister)

# here p, q are irrelevant
base, block_size = 4, 6

# generate random digits, home-made PRNG
(hash_runs, hash_blocks, max_run, cp_data) = generate_digits(N, x0, p, q, tau, step,
                                             base, block_size, max_lag,
                                             option="Numpy")
# run_test
run_test(base,max_run,hash_runs)

# block test
(KS, arr_cdf, arr_ecdf5, arr_arg5, trials) = block_test(hash_blocks, n_nodes,
                                         base ,block_size)
# autocorrel_test
autocorrel_test(cp_data, max_lag, base)

print()
print("Trials = ", trials)
print("KS = %8.5f\n\n" %(KS))


#--- [4] Scatterplot cdf (true random) versus ecdf (based on the two PRNGs)

import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 7
plt.rcParams['ytick.labelsize'] = 7

arr_cdf = np.array(arr_cdf)
delta_ecdf1 = (np.array(arr_ecdf1) - arr_cdf)
delta_ecdf2 = (np.array(arr_ecdf2) - arr_cdf)
delta_ecdf3 = (np.array(arr_ecdf3) - arr_cdf)
delta_ecdf4 = (np.array(arr_ecdf4) - arr_cdf)
delta_ecdf5 = (np.array(arr_ecdf5) - arr_cdf)

# print()
# print("blocks (arguments) used to compute ecdf1:\n")
# print(arr_arg1)

plt.plot(delta_ecdf1, linewidth = 0.4, color = 'red', alpha = 1)
plt.plot(delta_ecdf2, linewidth = 0.3, color = 'blue', alpha = 0.5)
plt.plot(delta_ecdf3, linewidth = 0.3, color = 'darkorange', alpha = 1)
plt.plot(delta_ecdf4, linewidth = 0.4, color = 'purple', alpha = 1)
plt.plot(delta_ecdf5, linewidth = 0.4, color = 'darkgreen', alpha = 1)
plt.axhline(y = 0.0, linewidth = 0.5, color = 'black', linestyle = 'dashed')
plt.legend(['HM1: good', 'HM2: bad', 'HM3: bad', 'HM4: good', 'Numpy'],
         loc='lower right', prop={'size': 7}, )
plt.show()
```

Bibliography

[1] Adel Alamadhi, Michel Planat, and Patrick Solé. Chebyshev's bias and generalized Riemann hypothesis. *Preprint*, pages 1–9, 2011. arXiv:1112.2398 [Link]. 65

[2] K. Binswanger and P. Embrechts. Longest runs in coin tossing. *Insurance: Mathematics and Economics*, 15:139–149, 1994. [Link]. 58

[3] Ramiro Camino, Christian Hammerschmidt, and Radu State. Generating multi-categorical samples with generative adversarial networks. *Preprint*, pages 1–7, 2018. arXiv:1807.01202 [Link]. 94

[4] Fida Dankar et al. A multi-dimensional evaluation of synthetic data generators. *IEEE Access*, pages 11147–11158, 2022. [Link]. 93

[5] Antónia Földes. The limit distribution of the length of the longest head-run. *Periodica Mathematica Hungarica*, 10:301–310, 1979. [Link]. 59

[6] Louis Gordon, Mark F. Schilling, and Michael S. Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72:279–287, 1986. [Link]. 58

[7] Vincent Granville. *Statistics: New Foundations, Toolbox, and Machine Learning Recipes*. Data Science Central, 2019. 24

[8] Vincent Granville. *Synthetic Data and Generative AI*. MLTechniques.com, 2022. [Link]. 100

[9] Vincent Granville. Feature clustering: A simple solution to many machine learning problems. *Preprint*, pages 1–6, 2023. MLTechniques.com [Link]. 82

[10] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 75

[11] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [Link]. 12, 15, 121

[12] Vincent Granville. *Gentle Introduction To Chaotic Dynamical Systems*. MLTechniques.com, 2023. [Link]. 24, 60, 64

[13] Vincent Granville. How to fix a failing generative adversarial network. *Preprint*, pages 1–10, 2023. MLTechniques.com [Link]. 14

[14] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 13, 82

[15] Vincent Granville. Smart grid search for faster hyperparameter tuning. *Preprint*, pages 1–8, 2023. MLTechniques.com [Link]. 13, 80, 82

[16] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 38, 43, 44, 45, 47, 48, 54, 55, 57, 63, 74, 75, 79, 80, 82, 83, 85, 93, 94

[17] Elisabeth Griesbauer. *Vine Copula Based Synthetic Data Generation for Classification*. 2022. Master Thesis, Technical University of Munich [Link]. 82

[18] Emil Grosswald. Oscillation theorems of arithmetical functions. *Transactions of the American Mathematical Society*, 126:1–28, 1967. [Link]. 65

[19] Adam J. Harper. Moments of random multiplicative functions, II: High moments. *Algebra and Number Theory*, 13(10):2277–2321, 2019. [Link]. 65

[20] 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]. 65

[21] Adam J. Harper. Almost sure large fluctuations of random multiplicative functions. *Preprint*, pages 1–38, 2021. arXiv [Link]. 65

[22] Zsolt Karacsony and Jozsefne Libor. Longest runs in coin tossing. teaching recursive formulae, asymptotic theorems and computer simulations. *Teaching Mathematics and Computer Science*, 9:261–274, 2011. [Link]. 59

[23] Adnan Saher Mohammed, Şahin Emrah Amrahov, and Fatih V. Çelebi. Interpolated binary search: An efficient hybrid search algorithm on ordered datasets. *Engineering Science and Technology*, 24:1072–1079, 2021. [Link]. 29

[24] Tamas Mori. The a.s. limit distribution of the longest head run. *Canadian Journal of Mathematics*, 45:1245–1262, 1993. [Link]. 59

[25] Michel Planat and Patrick Solé. Efficient prime counting and the Chebyshev primes. *Preprint*, pages 1–15, 2011. arXiv:1109.6489 [Link]. 65

[26] M.S. Schmookler and K.J. Nowka. Bounds on runs of zeros and ones for algebraic functions. *Proceedings 15th IEEE Symposium on Computer Arithmetic*, pages 7–12, 2001. ARITH-15 [Link]. 58

[27] Mark Shilling. The longest run of heads. *The College Mathematics Journal*, 21:196–207, 2018. [Link]. 58

[28] Chang Su, Linglin Wei, and Xianzhong Xie. Churn prediction in telecommunications industry based on conditional Wasserstein GAN. *IEEE International Conference on High Performance Computing, Data, and Analytics*, pages 186–191, 2022. IEEE HiPC 2022 [Link]. 93

[29] Terence Tao. Biases between consecutive primes. *Tao's blog*, 2016. [Link]. 65

[30] Ruonan Yu, Songhua Liu, and Xinchao Wang. Dataset distillation: A comprehensive review. *Preprint*, pages 1–23, 2022. Submitted to IEEE PAMI [Link]. 80

Index

agent-based modeling, 45, 48
Anaconda, 5
analytic continuation, 63
analytic functions, 63
antialiasing, 97
augmented data, 79, 82
autocorrelation, 38

Bard (Google AI), 115
Bernoulli trials, 58
Bignum library, 25
binary search, 25
    interpolated binary search, 29
    interpolation search, 29
    weighted, 25
binomial distribution, 29
block test, 26
bootstrapping, 47
Brownian motion, 64
bucketization, 80, 93

Chebyshev's bias, 64–66
checksum, 7
Chi-squared distribution, 26
Chi-squared test, 25
cloud regression, 100
Colab, 5, 6
command prompt, 5
computational complexity, 29
conditional convergence, 63
confidence intervals
    model-free, 15
connected components, 82
copula, 79
correlation distance, 74, 80
correlation distance matrix, 93
Cramér's V, 93
cross-validation, 23, 44
curse of dimensionality, 38
curve fitting, 66, 100

data distillation, 13, 80
Deming regression, 101
Diehard tests (randomness), 25
diffusion, 15
Dirichlet L-function, 63
Dirichlet character, 63
Dirichlet eta function, 54
Dirichlet series, 63
Dirichlet's theorem, 68
dummy variables, 93

ECDF, 12
EM algorithm, 82
empirical distribution, 24, 96
    multivariate, 12, 23, 24
empirical quantile, 82
ensemble method, 80, 82
epoch (neural networks), 80, 85
Euler product, 62
experimental math, 62
explainable AI, 12, 80
exploratory analysis, 7

faithfulness (synthetic data), 74

GAN (generative adversarial network), 12, 79
Gaussian mixture model, 43, 82
GenAI, 24
generative adversarial network, 14, 79
generative AI, 67
generative AI (GenAI), 24
geospatial data, 44
GitHub, 6
GMM (Gaussian mixture model), 82
goodness-of-fit, 66
GPT, 107, 114
gradient descent, 80, 93, 100
grid search, 13

hash table, 114
Hellinger distance, 43, 109
Hessian, 45
hierarchical clustering, 82
holdout method, 23, 74, 93
Hurst exponent, 47
hyperparameter, 13, 23
hyperrectangles, 13

identifiability (statistics), 38, 92
integer square root, 60
interpolation, 36, 44

Jupyter notebook, 5

Keras (Python library), 80
Kolmogorov-Smirnov distance, 12, 23, 24, 74, 80

Lagrange multiplier, 100
large language models (LLM), 24
LaTeX, 6
law of the iterated logarithm, 60
learning rate, 80

lightGBM, 81
lim sup, 60
Littlewood's oscillation theorem, 65
LLM, 24, 107
logistic regression, 79
loss function, 80

Markdown, 6
Markov chain, 107
Matplotlib, 9
mean squared error, 8
Mersenne twister, 24
metadata, 83, 115
metalog distribution, 92
mode collapse, 93
Monte-Carlo simulations, 38, 92
Moviepy (Python library), 54
MPmath (Python library), 54
multinomial distribution, 13, 22, 94
multiplication algorithm, 57
multiplicative function (random), 65
multivalued function, 101

node (interpolation), 40
normalization, 38

OpenAI, 114, 115
overfitting, 12, 43

Pandas, 6, 22
parallel computing, 80
PCA (principal component analysis), 79
Plotly, 8
pointwise mutual information, 109
prime race, 65
principal component analysis, 79
PRNG, 24, 58
pseudo-random number generator, 57, 58
Python library
    Copula, 82
    Gmpy2, 60
    Keras, 80
    Matplotlib, 9
    Moviepy, 54, 97
    MPmath, 54, 62, 65
    Osmnx (Open Street Map), 43
    Pandas, 6
    Plotly, 8
    PrimePy, 65
    Pykrige (kriging), 43
    Request, 115
    Scipy, 65, 82
    SDV, 81, 83
    Sklearn, 82
    Statsmodels, 43
    TabGAN, 81
    TensorFlow, 6
    Torpy, 115

quantile, 13
    extrapolated quantile, 96
    quantile function, 96
quantum derivative, 64
quantum state, 65

R-squared, 67
Rademacher distribution, 67
Rademacher function (random), 65
radix numeration system, 25, 26
random forest classifier, 82
random numbers (PRNG, 24
random walk, 64
records (statistical distribution), 60
regular expression, 7
regularization, 100
resampling, 47
Riemann Hypothesis, 62
Riemann zeta function, 55, 63
run (statistical theory), 58
run test, 26

scaling factor, 66
Scipy (Python library), 82
SDV (Python library), 81, 83
seed (random number generators), 24, 80
Sklearn (Python library), 82
smoothness, 44
softmax function, 94
spectral test, 26
stationarity, 42
Statsmodels (Python library), 43
synthetic data, 96
    geospatial, 44
synthetic function, 67

TabGAN (Python library), 81
taxonomy creation, 107
TensorFlow (Python library), 6
time complexity, 13
time series
    autocorrelation function, 43
    disaggregation, 36
    interpolation, 36
    stationarity, 42
training set, 23

Ubuntu, 5
unstructured data, 107

validation set, 12, 23, 79, 82
vector search, 25
vectorization, 13
versioning, 6
virtual machine, 5
Voronoi diagram, 47

Wasserstein GAN (WGAN), 93

XGboost, 12, 93