
CS/MATH 3414 — Numerical Methods

Lecture 9 Pseudorandom Numbers and Monte Carlo Simulation

Virginia Tech Department of Computer Science, Fall 2026

Instructor: Sumeet Khatri

Date: October 15, 2026

1	Introduction	2
1.1	Why do we need random numbers in a computation?	3
1.2	What does it mean for a number to be random?	5
1.3	If the procedure is deterministic, what makes it random?	6
2	Random numbers, pseudorandom numbers, and NumPy	6
2.1	Creating a source of zeros and ones	6
2.2	States, seeds, and cycles	12
2.3	From an integer recurrence to simulated observations	15
2.4	Create one generator and draw samples	18
2.5	Reproducibility and common sampling mistakes	19
3	Monte Carlo methods and simulation	20
3.1	Estimation by sample averages	20
3.2	Monte Carlo integration	21
3.3	Estimating areas and π	22
3.4	Monte Carlo simulation of waiting times	25
3.5	Monte Carlo and Las Vegas algorithms	26

Learning Objectives

- Understand the explain the difference between random and pseudorandom numbers.
- Provide example of pseudorandom number generators.
- Generate reproducible samples and analyze explicit pseudorandom recurrences.
- Construct Monte Carlo estimates with standard errors and distinguish Las Vegas algorithms.

Primary References

- [1] W. Cheney and D. Kincaid, *Numerical mathematics and computing*, 7th ed., Chapter 10: Monte Carlo Methods and Simulation (Brooks/Cole, Cengage Learning, Boston, 2013).
- [2] SimPy Developers, *Bank renege*, Version 4.1.1: a bank queue with random arrivals, service times, and customer patience, SimPy Documentation, https://simpy.readthedocs.io/en/4.1.1/examples/bank_renege.html (visited on 10/14/2026).
- [3] M. Pharr, W. Jakob, and G. Humphreys, *Path tracing*, Fourth edition, Section 13.2, Physically Based Rendering: From Theory to Implementation, (2023) https://pbr-book.org/4ed/Light_Transport_I_Surface_Reflection/Path_Tracing (visited on 10/14/2026).
- [4] CERN, *Geant4: a powerful tool for medical applications*, Newsletter of the EP Department, <https://ep-news.web.cern.ch/geant4-powerful-tool-medical-applications/> (visited on 10/14/2026).

Primary References (continued)

- [5] J. Goodman, *Class notes: Monte Carlo methods*, Part 1: Introduction (January 20, 2017); Week 2: Simulation and Variance Reduction (January 30, 2017), (2017) <https://math.nyu.edu/~goodman/teaching/MonteCarlo17/resources.html> (visited on 10/14/2026).
- [6] E. Barker, A. Roginsky, and R. Davis, *Recommendation for cryptographic key generation*, SP 800-133, Revision 2 (National Institute of Standards and Technology, June 2020).
- [7] E. Barker and J. Kelsey, *Recommendation for random number generation using deterministic random bit generators*, SP 800-90A, Revision 1 (National Institute of Standards and Technology, June 2015).
- [8] V. Charisopoulos, *Week 4: randomized algorithms*, ORIE 5270/6125 lecture notes: sorting, selection, and data streams, Cornell University, <https://people.orie.cornell.edu/vc333/orie5270/lectures/04/> (visited on 10/14/2026).
- [9] J. Tsitsiklis and P. Jaillet, *Introduction to probability: part i, the fundamentals*, Lectures 5 and 8: discrete and continuous random variables, MIT OpenCourseWare, (2018) <https://ocw.mit.edu/courses/res-6-012-introduction-to-probability-spring-2018/pages/part-i-the-fundamentals/> (visited on 10/14/2026).
- [10] M. S. Turan, E. Barker, J. Kelsey, et al., *Recommendation for the entropy sources used for random bit generation*, SP 800-90B (National Institute of Standards and Technology, 2018).
- [11] S. K. Park and K. W. Miller, “Random number generators: good ones are hard to find”, *Communications of the ACM* **31**, 1192 (1988).
- [12] M. E. O’Neill, *Predictability*, PCG: A Better Random Number Generator, <https://www.pcg-random.org/predictability.html> (visited on 10/14/2026).
- [13] M. E. O’Neill, *PCG: a family of better random number generators*, <https://www.pcg-random.org/> (visited on 10/14/2026).
- [14] G. Marsaglia, “Xorshift RNGs”, *Journal of Statistical Software* **8**, 1 (2003).
- [15] B. Widynski, *Middle-square weyl sequence RNG*, Version 7; introduction describes the original middle-square method, (Mar. 2022) [arXiv:1704.00358 \[cs.CR\]](https://arxiv.org/abs/1704.00358), <https://arxiv.org/pdf/1704.00358v7> (visited on 10/14/2026).
- [16] M. E. O’Neill, *PCG: a family of simple fast space-efficient statistically good algorithms for random number generation*, HMC-CS-2014-0905 (Harvey Mudd College, Sept. 5, 2014).
- [17] G. Valiant and M. Wootters, *CS265/CME309: randomized algorithms and probabilistic analysis. lecture 1: computational models, and the Schwartz–Zippel randomized polynomial identity test*, Notes by Gregory Valiant, updated by Mary Wootters, (Jan. 3, 2022) <https://web.stanford.edu/class/archive/cs/cs265/cs265.1224/Lectures/Lecture1/11.pdf> (visited on 10/14/2026).
- [18] Carnegie Mellon University, *Randomized algorithms*, Spring 2023; Monte Carlo and Las Vegas definitions and timeout conversion, CS 251, (2023) https://s23.cs251.com/Text/17_Randomized_Algorithms/contents.html (visited on 10/14/2026).
- [19] Massachusetts Institute of Technology, *Lecture 6: randomized algorithms*, 6.046J/18.410J Design and Analysis of Algorithms, Spring 2015, MIT OpenCourseWare, (2015) https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/cb55cb123a557eed0738a1187a452c24_MIT6_046JS15_lec06.pdf (visited on 10/14/2026).
- [20] R. Freivalds, “Fast probabilistic algorithms”, in *Mathematical foundations of computer science 1979*, Vol. 74, Lecture Notes in Computer Science (1979), pp. 57–69.

1 Introduction

The problem we consider in this lecture is how to use randomness to solve computational problems that arise in practice. Examples of problems include designing a road network, producing a realistic image, planning a

radiation treatment, or training a machine-learning model on millions of examples. These tasks can involve far more possible events, paths, or data contributions than we can afford to process individually. Randomness gives us ways to select a manageable part of that computation and use it to obtain a useful answer. To implement such methods, a computer needs a procedure for producing random numbers. Randomness also helps us protect information by creating encryption keys that are difficult for an attacker to predict.

1.1 Why do we need random numbers in a computation?

Consider a highway engineer designing a freeway interchange. The engineer wants to determine how much space is needed for waiting vehicles, which traffic flows cause bottlenecks, and how changes in speed limits affect traffic [1, pp. 481–482]. Rather than build every proposed design, the engineer can construct a computer model and simulate traffic passing through it. Random numbers can specify vehicle arrival times and other varying inputs according to an assumed traffic model. One run produces one possible traffic history. Repeating the simulation produces many histories, from which we can estimate average delays, typical queue lengths, or the probability that a queue exceeds the available space. These estimates help compare designs; their relevance to actual traffic also depends on the quality of the model.

Randomness has many practical uses. In each example below, notice the problem we want to solve, what is chosen randomly, and how those choices help solve the problem.

- **Deciding how many bank tellers to schedule.** A bank might want to compare staffing three tellers with staffing four. Sample customer arrival times and service durations, then simulate the queue over many working days. Average the waiting times and estimate the fraction of customers who wait more than ten minutes or leave before being served. This lets the bank compare staffing costs with service quality. SimPy’s bank example models random arrivals, service times, and customer patience [2]. Sampling is useful because following one possible day’s queue is much simpler than listing every possible pattern of arrivals and service times.
- **Rendering a realistic image.** To determine a pixel’s brightness, rendering software must account for light reaching the camera after reflections from surfaces in the scene. There are far too many possible light paths to examine individually. *Path tracing* samples paths randomly and averages their weighted contributions to estimate the pixel’s brightness. More samples generally reduce the grainy noise in the image [3]. Here randomness makes a difficult light-transport calculation manageable by accepting an approximation. Even for a fixed scene and camera, the sampling procedure can be random.
- **Estimating radiation doses for treatment planning.** Given a proposed radiation beam and a model of a patient’s anatomy, simulate many particles moving through tissue. Randomly sample their interactions according to a physical model and record where they deposit energy. Combining these histories estimates the dose delivered to a tumour and surrounding tissue. Geant4 is used for this type of medical modelling [4]. Sampling particle histories provides a way to handle complicated tissue geometry and interactions without enumerating every possible path.
- **Training a machine-learning model on a large dataset.** Suppose we are adjusting an image classifier to reduce its average training loss, a numerical measure of prediction error, on millions of examples. Computing a full gradient at every update requires processing the entire dataset. Instead, randomly select a small batch uniformly and use it to estimate the gradient, giving a cheaper but noisier update. Uniform selection gives each example an equal chance to contribute. This is the idea behind mini-batch stochastic gradient descent [5]. Randomness reduces the work per update; it does not guarantee fewer

updates or a better final model. The dataset and objective can be fixed: randomness is introduced by the solution method.

- **Creating secret keys for encryption.** Suppose a backup service encrypts a private document before storing it. In symmetric encryption, a secret key controls both encryption and decryption: someone who obtains the key can recover the document. The encryption algorithm can be public; security requires that an attacker cannot feasibly discover the key. Random numbers give us a way to choose a key that is difficult to predict [6]. For example, an ideal key consisting of 128 independent fair bits is uniform over 2^{128} possibilities. An attacker with no information about this key succeeds with probability 2^{-128} in one guess. Suppose they tried to guess by choosing one by one. Even choosing at a rate of femtoseconds (10^{-15} seconds), it would take roughly 10^{16} years to go through them all — that's roughly *10.8 million billion years*! A long key generated from a predictable procedure can be much easier to guess. In particular, someone who knows a generator's rule and fixed seed can reproduce its outputs and hence the key.

For this application, an approximately uniform histogram is insufficient: the generated bits must also resist prediction. Practical systems use a *cryptographically secure pseudorandom number generator*, initialized with sufficient entropy unavailable to the attacker and with its internal state protected. Such a generator may use deterministic computation to produce its outputs [7]. The elementary generator in Section 2.1 is not suitable for generating cryptographic keys. Here randomness supplies unpredictable secrets needed to protect information.

These examples illustrate several roles for randomness. Sometimes it represents uncertain inputs, such as customer arrival times. Sometimes it is a way to approximate a large calculation, such as the light arriving at a pixel or a gradient averaged over a dataset. In both cases, the quantity we seek can be a fixed number: a model's expected waiting time, for example, has a definite value even though the simulated waiting times vary. Cryptography adds another role: supplying secrets that an attacker cannot feasibly predict.

Using random samples to estimate waiting times, pixel brightness, or deposited radiation energy is an application of *Monte Carlo methods*. The common strategy is to generate many individual outcomes and combine them to estimate a quantity needed for a decision or a calculation. Section 3 develops this strategy and explains how to assess its sampling error. The law of large numbers from the previous lecture explains why averaging can work: under suitable assumptions, averages of many independent observations approach the corresponding expectation.

When can randomness make a problem easier? It is especially useful when following one possible outcome is feasible, but accounting for all possible outcomes directly is too expensive. In rendering, we can trace an individual light path even though we cannot enumerate every path contributing to an image. In staffing, we can simulate a working day even though listing every possible sequence of customer arrivals is impractical. Random sampling lets us estimate the overall behaviour from a manageable collection of such calculations. In machine learning, the saving is more immediate: an update based on a small random batch processes fewer examples than a full-data update. In each case, the benefit depends on whether the resulting approximation is accurate enough for the task and how much work obtaining it requires.

Randomness can also help us choose how to organize a computation. Suppose we want to sort transaction records. Quicksort splits the records around a chosen *pivot* and repeats on the smaller groups. Always using a fixed rule such as “choose the first record” can produce very uneven splits on an unfortunate input ordering. Choosing pivots uniformly at random gives good expected performance for every fixed input of distinct keys, although an unlucky run can still be slow. The completed ordering is exact. Randomness helps this

algorithm avoid dependence on a particular input arrangement; it does not establish an advantage over every deterministic sorting method. Mergesort, for example, is efficient without random choices [8]. We will revisit this example when distinguishing Monte Carlo and Las Vegas algorithms in Section 3.

When is randomness less helpful? Randomness is less helpful when we already have an efficient direct method, when obtaining each sample is itself prohibitively expensive, or when an approximate answer does not meet the task’s requirements. It can also struggle with rare but important outcomes. A staffing plan might work on most simulated days yet fail during an unusual surge in demand; a system can fail in ways that ordinary sampling almost never encounters. Seeing no such events in a simulation does not establish that they are impossible. Specialized sampling methods may be needed to study them efficiently [5]. Greater precision also costs time: for ordinary independent Monte Carlo averages with finite variance, halving the standard error requires roughly four times as many samples. An image with much less sampling noise, for example, can require substantially more rendering work.

Thus randomness can make some problems easier to approximate, make individual steps cheaper, give an algorithm more reliable expected performance, or provide unpredictable secrets for encryption. It is not a universal shortcut, and it does not by itself provide an exact answer or a guarantee of fast computation. More samples reduce sampling uncertainty, but they do not correct an inappropriate model or a defective sampling procedure. We therefore need both an ideal mathematical description of random inputs and a practical way to supply them to an algorithm.

1.2 What does it mean for a number to be random?

In probability theory, a *random number* is an observed value of a *random variable*: a numerical outcome of an experiment governed by a specified probability law [9]. The law tells us how likely different values, or ranges of values, are. Randomness is not a property that we can identify by looking at a numeral alone. The value 4 could be the outcome of a fair die roll or the output of a program that always prints 4; these procedures have very different probability laws. For the ideal fair die,

$$\Pr[X = j] = \frac{1}{6}, \quad j = 1, \dots, 6. \quad (1.1)$$

“Random” does not mean that all outcomes must be equally likely. A biased coin and a normally distributed measurement are also described by random variables. We must specify the distribution appropriate to the problem.

Our starting point is a sequence of numbers uniformly distributed in the unit interval. We use the convention $[0, 1)$, which matches the numerical generators used later; including or excluding the endpoints makes no difference to the ideal continuous uniform law. Writing $U \sim \text{Uniform}[0, 1)$ means

$$\Pr[a \leq U < b] = b - a, \quad 0 \leq a < b \leq 1. \quad (1.2)$$

For instance, an interval of length 0.1 receives probability 0.1 wherever it lies in $[0, 1)$. In many independent draws, approximately one tenth of the values should fall in each of the ten equal subintervals. Uniform samples will be our basic building blocks: Section 2 shows how to transform them into samples from other distributions.

We usually also require the draws $U_1, U_2, \dots$ to be *independent*: knowing earlier draws should not change the probability law of a later draw. Together, uniformity and independence give

$$\Pr[a_i \leq U_i < b_i \text{ for all } i = 1, \dots, n] = \prod_{i=1}^n (b_i - a_i), \quad 0 \leq a_i < b_i \leq 1. \quad (1.3)$$

Such a sequence is called *independent and identically distributed* (iid). Uniformity alone is insufficient: if U_1 is uniform and we set $U_2 = U_1$, then each variable is uniform, but the second draw is completely determined by the first. Treating repeated copies of one observation as independent samples would give misleading conclusions about the amount of information in a simulation.

1.3 If the procedure is deterministic, what makes it random?

A deterministic program started with the same input and internal state produces the same output. A *pseudorandom number generator* (PRNG) uses such a program to produce a sequence of numbers from an initial *seed*. Once the seed and the generator are fixed, the entire sequence is fixed. There is no fresh chance in each step of the calculation.

The word *pseudorandom* describes what the sequence is designed to imitate. Its frequencies, patterns of successive values, and other statistical properties should resemble those of ideal random samples closely enough for the intended computation. For uniform samples, we expect approximately the right fraction in each subinterval, and successive pairs should be spread through the unit square without systematic patterns that distort our calculations. Deterministic sequences can be useful substitutes for random samples because they satisfy suitable tests of random behaviour [1, pp. 482–484].

For example, consider the deterministic sequence

$$u_k = \frac{k \bmod 100}{100}, \quad k = 0, 1, 2, \dots \quad (1.4)$$

Each block of 100 values visits every point of the grid $0, 0.01, \dots, 0.99$ once, so its one-dimensional frequencies look uniform. Yet each value is followed by the next grid point, with a jump back to zero after 0.99. Consecutive pairs lie on a simple pattern, and the sequence repeats after 100 steps. A uniform-looking frequency table therefore does not make this a suitable replacement for independent uniform draws. Section 2 examines more useful generators and tests both distribution and dependence.

Computers can also obtain physical randomness, for example from measured electronic noise, and use it to initialize a generator [10]. Thus the limitation is on deterministic computation from a fixed seed, not on all possible ways of obtaining randomness.

2 Random numbers, pseudorandom numbers, and NumPy

The probability model assumes that we can obtain independent samples from a specified distribution. A computer must implement a concrete procedure that approximates this ideal.

2.1 Creating a source of zeros and ones

How can we construct a source that returns 0 or 1 on each call and approximates the Bernoulli(0.3) distribution? For example, in our traffic simulation, 1 could mean that an arriving vehicle turns left and 0 that it does not. The ideal probability law for each output is

$$\Pr[X = 1] = 0.3, \quad \Pr[X = 0] = 0.7, \quad (2.1)$$

and successive outputs should imitate independent repetitions. We will build the source from an integer seed and arithmetic alone.

Step 1: generate an integer sequence. Store the initial state $z_0 = 1$, called the *seed*. On each call, multiply the current state by 16807 and retain the remainder after division by $m = 2147483647 = 2^{31} - 1$:

$$z_0 = 1, \quad z_{k+1} = (16807z_k) \bmod m. \quad (2.2)$$

This is a multiplicative congruential generator [1, Section 10.1]. For example, the first three updates give

$$\begin{aligned} z_1 &= (16807 \cdot 1) \bmod m = 16807, \\ z_2 &= (16807 \cdot 16807) \bmod m = 282475249, \\ z_3 &= (16807 \cdot 282475249) \bmod m = 1622650073. \end{aligned} \quad (2.3)$$

Nothing is chosen randomly during these updates. The seed and the rule determine every subsequent state. A nonzero seed is essential: starting at zero would leave the state at zero forever.

Why choose $m = 2^{31} - 1$ and the multiplier 16807? We want the recurrence to visit many different integers, with no integer favored over another. These constants were selected to achieve that goal, along with practical considerations about computing the updates.

First, $m = 2147483647$ is *prime*. If both the multiplier and the current state lie between 1 and $m - 1$, their product cannot be divisible by m . Thus a nonzero state never collapses to zero. This particular prime also gives a large state space while fitting in a signed 32-bit integer, which helped motivate its historical use [11]. Primality alone does not ensure a long cycle: a multiplier of 1 would simply repeat the seed forever.

Second, $16807 = 7^5$ has a special property for this modulus: it is a *primitive root modulo m* . This means that its successive powers, reduced modulo m , visit every nonzero remainder before returning to 1. Starting from our seed $z_0 = 1$, repeated substitution in the recurrence gives

$$z_k = 16807^k \bmod m. \quad (2.4)$$

The first return to 1 occurs after exactly $m - 1 = 2147483646$ updates. Consequently, every integer $1, \dots, m - 1$ appears exactly once in the cycle. Any other nonzero seed starts at a different position in this same cycle. The primitive-root property can be verified without generating billions of terms: factor $m - 1 = 2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$ and check that $16807^{(m-1)/q} \bmod m \neq 1$ for each distinct prime factor q in that list. All seven checks hold.

The multiplier 16807 is one established choice, not the only possible one. Park and Miller discuss its selection using three criteria: a full cycle, statistical behavior, and efficient implementation [11]. Its usefulness does not follow merely from being a large number or a power of 7. Also, although the states fit in 32 bits, the product $16807z_k$ can require more bits; Python's ordinary integers handle this automatically.

A small version makes the role of the multiplier visible. Keep the seed 1, but use the prime modulus 11. Multiplication by 2 gives the cycle

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 5 \rightarrow 10 \rightarrow 9 \rightarrow 7 \rightarrow 3 \rightarrow 6 \rightarrow 1. \quad (2.5)$$

All ten nonzero states occur. With the same modulus and seed, multiplication by 3 instead gives

$$1 \rightarrow 3 \rightarrow 9 \rightarrow 5 \rightarrow 4 \rightarrow 1. \quad (2.6)$$

Half of the possible states are missed. Choosing a prime modulus is therefore only part of the construction; the multiplier determines whether a single cycle covers all the nonzero states.

Step 2: scale the integers to the unit interval. For each newly computed state, set

$$u_k = \frac{z_k}{m}, \quad k = 1, 2, \dots \quad (2.7)$$

The first three values are approximately 0.000007826, 0.131537788, and 0.755605322. These numbers come directly from the recurrence; no uniform random-number source has been assumed. They are our *pseudorandom substitutes* for uniform draws. Because of the full-cycle property, the scaled outputs visit every point of the grid

$$\frac{1}{m}, \frac{2}{m}, \dots, \frac{m-1}{m} \quad (2.8)$$

exactly once per cycle, in the order determined by the recurrence. They therefore have *exactly equal frequencies on this finite grid*. An interval occupying, say, 30% of $(0, 1)$ contains approximately 30% of the grid points; the only discrepancy comes from rounding at its endpoints. More generally, for $0 \leq a < b \leq 1$, the full-cycle fraction is

$$\frac{\#\{k \in \{1, \dots, m-1\} : a \leq u_k < b\}}{m-1} \approx b - a. \quad (2.9)$$

Here # counts the indices satisfying the condition. This is precisely the interval-frequency behavior of a uniform distribution. Our grid spacing is $1/m \approx 4.66 \times 10^{-10}$, so the discrete grid is a very fine approximation to the continuous interval. Dividing by m only rescales the sequence; the full-cycle property is what supplies the equal frequencies.

This argument describes a *complete cycle*. It does not guarantee that every shorter segment is approximately uniform, and it does not establish independence. The order in which the grid points appear still matters for simulation, so a long cycle alone is not a sufficient test of a generator.

What do the generated values look like in practice? We run this same recurrence from the seed $z_0 = 1$ and examine its first $N = 100, 1,000, 10,000$, and $100,000$ scaled outputs. These are successively longer portions of the same deterministic sequence. To draw a histogram, divide $[0, 1)$ into ten equal intervals, $[0, 0.1)$, $[0.1, 0.2)$, $\dots$, $[0.9, 1)$, and plot the fraction of outputs in each interval. For an ideal uniform distribution, each interval has probability 0.1, so a flat histogram at height 0.1 is the target.

Figure 1 shows the results, using the same bins and axis scales in all four panels. With 100 outputs, the bin fractions range from 3% to 14%; with 100,000, they range from 9.863% to 10.285%. Thus, for these sample sizes, the generated values fill the interval increasingly evenly. This is numerical evidence for approximately uniform frequencies over these portions of the sequence. A histogram records how often values occur in each interval; it discards their order and therefore cannot establish independence.

Step 3: turn each value into a bit. Return 1 if the scaled value is below 0.3, and 0 otherwise:

$$x_k = \begin{cases} 1, & u_k < 0.3, \\ 0, & u_k \geq 0.3. \end{cases} \quad (2.10)$$

Thus the first three outputs are 1, 1, 0. The comparison can be made using integers alone: $u_k < 0.3$ is equivalent to $10z_k < 3m$. The following code implements the entire source; it calls no random-number library.

LISTING 1: Creating the Bernoulli source from a seed and an arithmetic recurrence.

```
1 m = 2**31 - 1
2 z = 1                                     # seed: initialize once
```

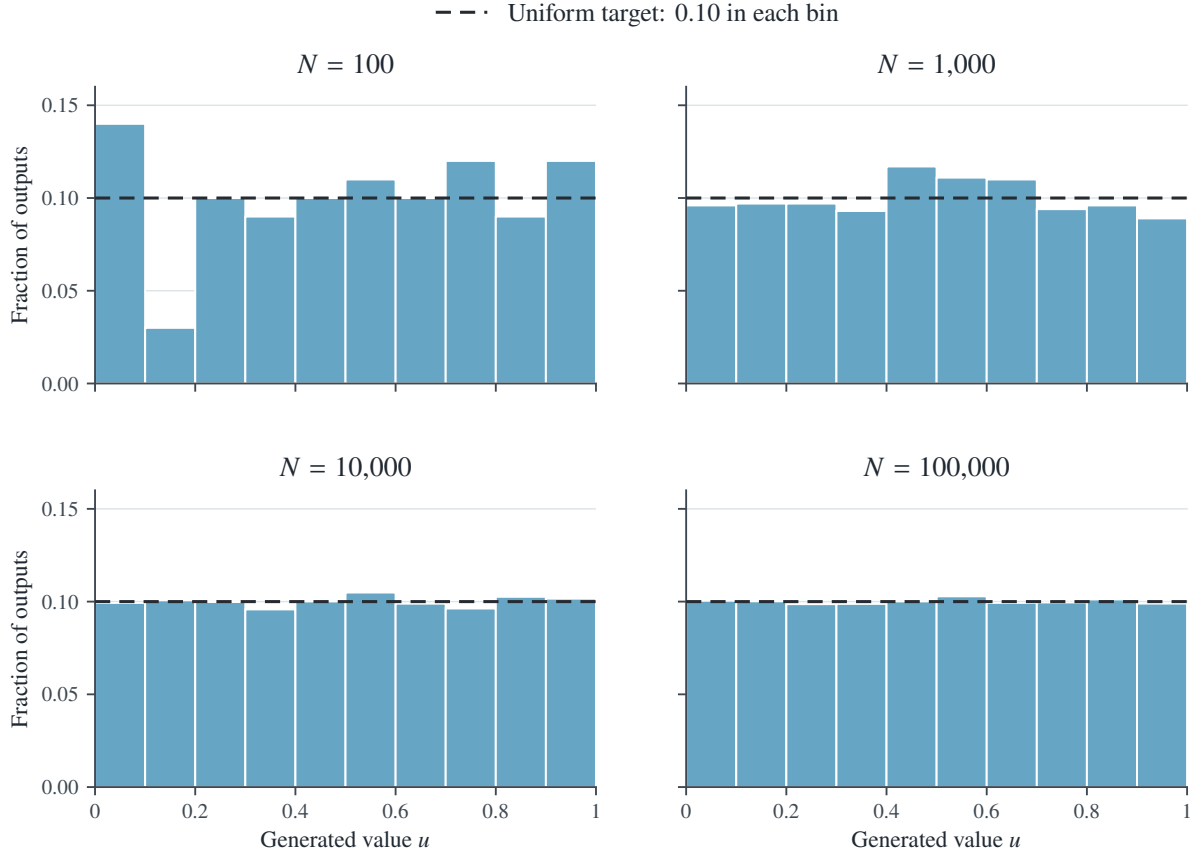


FIGURE 1: Histograms from the source $z_{k+1} = 16807z_k \bmod (2^{31} - 1)$, $u_k = z_k / (2^{31} - 1)$, with seed $z_0 = 1$. Each panel uses the first N outputs of this same sequence. Bar heights are fractions of outputs in ten equal bins of width 0.1; the dashed line marks the uniform target 0.1. All panels have identical axis scales. The histograms become flatter for these increasing sample sizes, illustrating approximate uniformity of individual outputs. They do not test independence.

```

3
4 def next_bit():
5     global z                                # retain state between calls
6     z = (16807 * z) % m                    # compute the next state
7     return int(10 * z < 3 * m)             # equivalent to z / m < 0.3
8
9 samples = [next_bit() for _ in range(10_000)]
10 print(samples[:10])
11 print(sum(samples))
12 print(sum(samples) / len(samples))

```

The first ten outputs are

$$1, 1, 0, 0, 0, 1, 1, 0, 0, 0, \quad (2.11)$$

and the first 10,000 calls produce 2,998 ones: the observed proportion is 0.2998. Keeping z between calls is essential. Resetting it to 1 inside the function would make every call repeat the first update and return 1.

Why does this approximate the desired distribution? An ideal uniform variable falls in $[0, 0.3)$ with probability 0.3, so thresholding independent ideal uniforms would give exactly independent Bernoulli(0.3) variables. Our construction replaces those ideal uniforms with a deterministic sequence whose grid frequencies are nearly uniform. Of the $m - 1 = 2147483646$ nonzero states in a full cycle, exactly 644245094 satisfy $10z < 3m$. Hence the full-cycle fraction of ones is

$$\frac{644245094}{2147483646} \approx 0.300000000093. \quad (2.12)$$

Shorter portions need not have that same fraction: our first ten outputs have four ones, while the first 10,000 have a proportion of 0.2998.

Figure 2 shows the proportions of zeros and ones at the same four sample sizes used for the uniform histograms. We apply the threshold $u_k < 0.3$ to exactly those same outputs, so each histogram describes the binary source we have just constructed. With 100 outputs, the proportions of zeros and ones are 0.73 and 0.27; with 100,000, they are 0.70074 and 0.29926, close to the desired 0.7 and 0.3. The agreement need not improve at every increase in sample size: the proportions 0.7002 and 0.2998 at 10,000 outputs happen to be slightly closer to the targets.

This constructs the requested binary source, but it remains pseudorandom. Its fixed-seed output is completely reproducible and eventually repeats. Matching the proportion of ones does not guarantee that successive bits behave independently: the repeating pattern 1110000000 also has 30% ones. This historical generator makes the construction transparent; assessing a source for a particular simulation also requires examining dependence and other statistical properties. We discuss those tests later in this section.

Restarting with the same seed. Rerunning the entire code listing resets z to the seed 1, so every state z_k , scaled value u_k , and output bit x_k is reproduced in exactly the same order. Thus every run starts with 1, 1, 0, 0, 0, 1, 1, 0, 0, 0 and produces exactly 2,998 ones in its first 10,000 calls. Continuing to call `next_bit()` without resetting z instead continues the existing sequence. Reusing a seed is useful for reproducing and debugging a simulation, but repeated runs with that seed are not independent repetitions. A different seed changes the starting state; it does not make the subsequent updates nondeterministic.

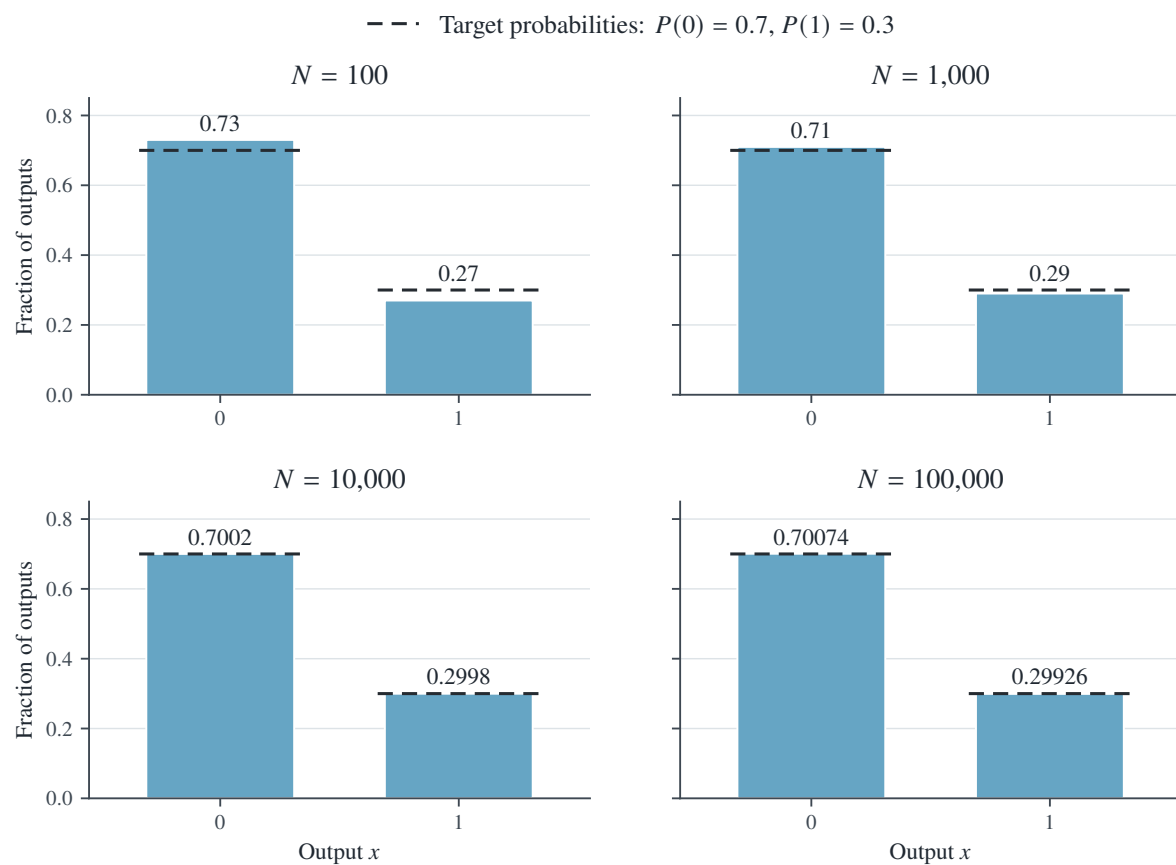


FIGURE 2: Relative-frequency histograms of the Bernoulli outputs obtained by thresholding the same pseudorandom uniform sequence at 0.3, with seed $z_0 = 1$. Each panel uses the first N outputs. The two bars show the observed proportions of zeros and ones, also printed above the bars. Dashed markers show the target probabilities 0.7 for zero and 0.3 for one. All panels use the same axis scales. Larger samples give proportions close to the targets here, although the error need not decrease at every increase in sample size.

2.2 States, seeds, and cycles

The construction in Section 2.1 illustrates the deterministic mechanism underlying a PRNG. We now describe that mechanism in general terms. Its state contains the information needed to determine the next output; the seed initializes that state.

Definition 1 (Pseudorandom number generator). A *pseudorandom number generator* (PRNG) is a deterministic procedure that, starting from a seed, produces a sequence of outputs intended to imitate independent samples from a specified probability distribution. The generator retains an *internal state*: the information it needs to continue producing outputs on later calls.

Formally, let S be a finite set of possible states and A the set of possible output values. An initialization rule converts the seed into $s_0 \in S$. A state-update function $F : S \rightarrow S$ and an output function $G : S \rightarrow A$ specify what happens on each call. Using the convention that we update the state before returning an output, for $k = 0, 1, 2, \dots$,

$$\begin{aligned} s_{k+1} &= F(s_k) && \text{(update and retain the state),} \\ u_{k+1} &= G(s_{k+1}) && \text{(return this value to the calling program).} \end{aligned} \tag{2.13}$$

Thus F specifies how the generator’s memory changes, while G specifies what the caller receives. The state recurrence can be run without computing G : the returned output is not needed to advance it. The purpose of G is to specify the requested kind of output. If the state itself is what we want to return, we simply take $G(s) = s$; no separate conversion is needed.

Determinism makes an experiment reproducible: we can rerun the same sequence to debug a calculation or compare implementations. It also means that a fixed-seed output sequence is not literally an infinite sequence of independent random variables. Theorems about iid samples describe the idealized model used to analyze a simulation. Generator quality determines whether that model is a useful computational approximation.

Statistical quality for simulation. A good PRNG for simulation should imitate the relevant statistical behavior of independent samples from the intended distribution, over the lengths of sequences being used. This concerns both individual values and groups of successive values: matching a histogram is not enough if dependence between outputs distorts the calculation. Such a generator can be useful even if someone who analyzes its algorithm can predict its outputs. Statistical quality and resistance to prediction are distinct requirements [12].

Unpredictability for cryptography. For cryptographic applications, such as generating secret keys, we need a stronger property. Suppose an observer knows the algorithm and has seen previous *outputs*, but does not know the seed or internal state. Informally, a cryptographically secure generator of unbiased bits should prevent any computationally feasible method from predicting the next bit with a meaningful advantage over success probability $1/2$. Its output sequences should also be infeasible to distinguish from independent unbiased bits using computationally feasible tests. This requires initialization with sufficient unpredictable information and protection of the seed and state [7]. If the seed or current state is known, the deterministic rules in Definition 1 let us calculate the subsequent outputs exactly. Thus “hard to predict from previous outputs” is a stronger requirement than the statistical quality normally sought for numerical simulation.

Example 1 (A small linear congruential generator). A *linear congruential generator* (LCG) updates an integer state by

$$z_{k+1} = (az_k + c) \bmod m, \quad u_k = \frac{z_k}{m}, \quad (2.14)$$

where $m \geq 2$, the multiplier a , the increment c , and the seed z_0 are integers, with $0 \leq z_0 < m$. For $m = 16$, $a = 5$, $c = 1$, and $z_0 = 0$, the states are

$$\begin{aligned} &0, 1, 6, 15, 12, 13, 2, 11, \\ &8, 9, 14, 7, 4, 5, 10, 3, 0, \dots \end{aligned} \quad (2.15)$$

Every state appears once before the sequence repeats. Nevertheless, the entire sequence repeats after only 16 steps, and consecutive outputs have visible structure. Equal marginal frequencies alone do not establish independence. This small generator is an illustration, not a recommended simulation tool.

Why congruential generators are common. The word *congruential* refers to equality of remainders: $x \equiv y \pmod{m}$ means that $x - y$ is a multiple of m . Thus (2.14) chooses the remainder of $az_k + c$ as the next state. This keeps the state in a fixed range, and a basic LCG needs only one changing integer and a few arithmetic operations per output. For a modulus 2^w , retaining the lowest w bits implements the reduction, which fits computer arithmetic efficiently. Number theory also gives conditions on the parameters that guarantee long cycles. These properties make congruential generators convenient to implement, analyze, and teach [1, Section 10.1].

Our emphasis on them does not mean that a plain LCG is the best generator for every task. Their simple structure can produce unwanted dependencies, even with a long period. Modern designs such as PCG retain the efficient state update and add an output transformation to improve statistical behavior [13]. Figure 3 illustrates why the behavior of successive outputs matters.

How to read Figure 3. Each dot records two successive outputs of one generator: its horizontal coordinate is the current output U_i , and its vertical coordinate is the next output U_{i+1} . Thus a sequence $U_0, U_1, U_2, \dots$ gives the points $(U_0, U_1), (U_1, U_2), (U_2, U_3), \dots$. For the small generator, the first three points are

$$(0, 1/16), \quad (1/16, 6/16), \quad (6/16, 15/16). \quad (2.16)$$

For example, the point $(0.375, 0.9375)$ says that the output 0.375 is followed by 0.9375. The left panel includes all 16 transitions in a complete cycle, including the transition back to the starting state. Generating another cycle would put dots at exactly the same locations.

Why the line pattern matters. Dividing the small generator's recurrence by 16 gives

$$U_{i+1} = 5U_i + \frac{1}{16} - \left\lfloor 5U_i + \frac{1}{16} \right\rfloor. \quad (2.17)$$

The floor is one of 0, 1, 2, 3, 4 for the states in this cycle, so every point lies on one of five lines of slope 5. Knowing the horizontal coordinate determines the vertical coordinate exactly. Although every allowed individual output occurs equally often, most possible pairs of allowed outputs never occur. This is dependence that a histogram of individual outputs would miss.

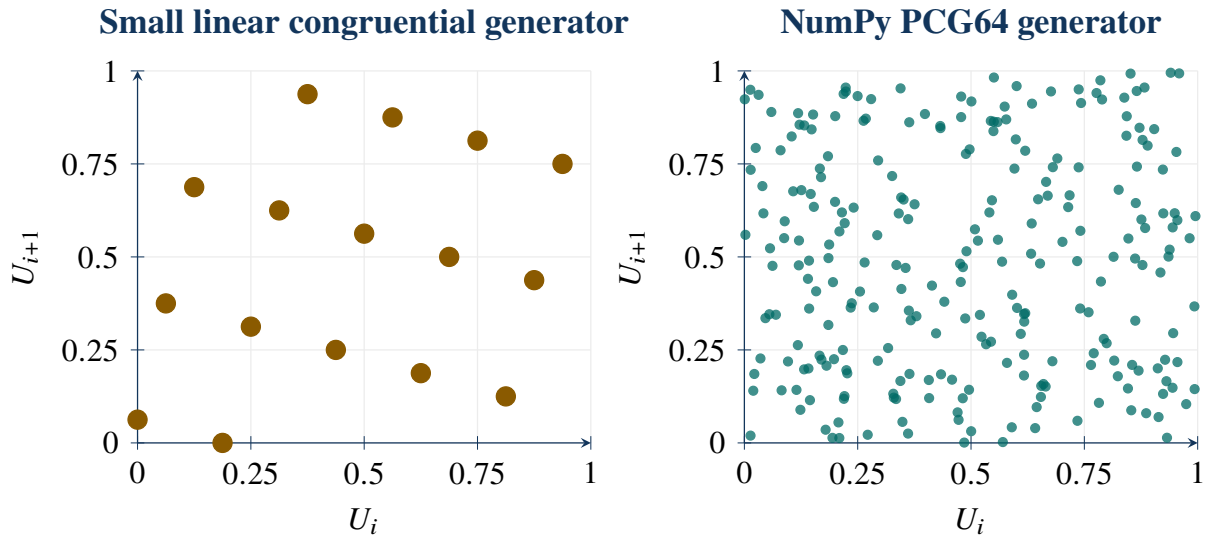


FIGURE 3: A plot of the next output against the current output: each dot is (U_i, U_{i+1}) . Left: all 16 transitions of $z_{i+1} = (5z_i + 1) \bmod 16$, starting from $z_0 = 0$, with $U_i = z_i/16$. The dots lie on five parallel line segments imposed by the recurrence. Right: 256 overlapping pairs formed from 257 outputs of NumPy's PCG64 generator, using seed 3414. The points are spread through the square without the conspicuous line pattern on the left. The panels contain different numbers of points; the comparison concerns their arrangement, not their density. Neither a uniform histogram nor an unstructured-looking scatterplot proves independence.

For comparison, if successive outputs were independent uniform random variables, the probability of falling in any rectangle within the unit square would equal its area:

$$\Pr[a \leq U_i < b, c \leq U_{i+1} < d] = (b - a)(d - c), \quad 0 \leq a < b \leq 1, \quad 0 \leq c < d \leq 1. \quad (2.18)$$

We would therefore expect points spread throughout the square, with ordinary gaps and clusters from finite sampling. The right panel is consistent with that visual expectation. It does not show the obvious defect of the small generator, but it also does not establish that PCG64 outputs are independent or genuinely random. Even under the ideal model, neighboring plotted pairs share one output, so the dots themselves are not independent observations. This plot is a useful diagnostic of successive outputs; other tests examine other kinds of structure.

What Figure 3 says about predictability. For the small generator on the left, the displayed recurrence predicts the next output exactly from the current one, and the line pattern also reveals a failure to imitate independent uniform samples. On the right, the absence of a comparable visible pattern does not establish cryptographic unpredictability: an algorithm could exploit structure that this plot does not show. Indeed, the stated seed 3414 lets anyone reproduce the PCG64 sequence using the same implementation. That reproducibility is useful in simulation. NumPy’s generators, including PCG64, are designed for statistical modeling and simulation, not for generating cryptographic secrets.

Key Idea

Every deterministic generator with a finite state space eventually enters a cycle. Useful generators require a sufficiently long period, good behavior in several dimensions, efficient implementation, and well-designed state initialization.

2.3 From an integer recurrence to simulated observations

For the LCG in Example 1, “modulo m ” means the remainder after division by m . For instance, starting from $z_0 = 0$,

$$z_1 = 1, \quad z_2 = 6, \quad z_3 = 31 \bmod 16 = 15, \quad z_4 = 76 \bmod 16 = 12. \quad (2.19)$$

The corresponding scaled values are $1/16$, $6/16$, $15/16$, and $12/16$. The following implementation outputs the *updated* state each time:

LISTING 2: A deliberately small generator for inspecting the mechanism.

```
1 z = 0
2 states = []
3 for _ in range(16):
4     z = (5 * z + 1) % 16
5     states.append(z)
6 u = [z / 16 for z in states]
7 print(states)
8 print(u[:4]) # [0.0625, 0.375, 0.9375, 0.75]
```

Parameter choices matter. The recurrence $z_{k+1} = 6z_k \bmod 16$ with $z_0 = 1$ gives $1, 6, 4, 8, 0, 0, \dots$ and becomes trapped at zero. Even the full-period generator above is inadequate: since $z_{k+1} \equiv z_k + 1 \pmod{2}$, its lowest bit alternates deterministically. Having every state occur once per cycle does not imply independence.

Another example is the historical multiplicative generator

$$z_{k+1} = 16807z_k \bmod (2^{31} - 1), \quad u_{k+1} = \frac{z_{k+1}}{2^{31} - 1}, \quad (2.20)$$

with a nonzero seed [1, Section 10.1]. For $z_0 = 1$, its first states are 16807, 282475249, and 1622650073. It has a much longer period than the toy example.

Other ways to construct a PRNG. A congruential update is a design choice, rather than a requirement in Definition 1. Other choices include extracting digits from a squared state, as in the example below; combining bit strings using shifts and exclusive-or, as in *xorshift* generators [14]; and updating secret state with cryptographic hash functions or block ciphers [7]. Some of these operations can themselves be expressed using modular arithmetic: exclusive-or is addition modulo 2 separately in each bit position. What changes is the construction of the state and output rules. Each construction needs its own analysis of period and statistical quality.

Example 2 (Middle-square: a source that quickly fails). In the original *middle-square method*, square the state and retain its middle digits [15]. Here the state z_k has four decimal digits, including leading zeros when needed. Write its square with exactly eight digits, retain the middle four as z_{k+1} , and return $u_{k+1} = z_{k+1}/10,000$ to approximate a uniform sample. For the seed $z_0 = 6752$, the first two steps are

$$\begin{aligned} 6752^2 = 45\mathbf{5895}04 &\implies z_1 = 5895, & u_1 = 0.5895, \\ 5895^2 = 34\mathbf{7510}25 &\implies z_2 = 7510, & u_2 = 0.7510. \end{aligned} \quad (2.21)$$

Continuing the same rule gives

$$(z_0, z_1, \dots, z_9) = (6752, 5895, 7510, 4001, 0080, 0064, 0040, 0016, 0002, 0000). \quad (2.22)$$

For instance, $4001^2 = 16008001$ gives the state 0080; $80^2 = 00006400$ gives 0064. After nine updates the state is zero, and squaring zero keeps it there forever.

The first 100 returned values therefore contain 92 zeros, and 97 of those 100 values lie in $[0, 0.1)$. A uniform distribution assigns only probability 0.1 to this interval. As we generate more values, the proportion of zeros approaches 1, so this source fails to approximate a uniform distribution. Thresholding these values at 0.3 to make Bernoulli(0.3) outputs would eventually return 1 on every call.

Zero is not the only problematic state: $2500^2 = 06250000$ returns 2500 again, so that seed produces 0.25 forever. The failure here is not merely eventual repetition, which occurs for every deterministic finite-state generator. These valid seeds enter cycles of length one almost immediately and give severely biased output frequencies.

NumPy’s PCG64. PCG stands for *permuted congruential generator*. It combines an LCG state update with a separate transformation of the state’s bits before returning an output [13]. The number 64 refers to the width of a raw output integer. The changing internal state has 128 bits. In the small LCG, we exposed the state directly by dividing it by the modulus. PCG64 instead transforms that state, with the aim of avoiding simple output patterns such as those in Figure 3. This transformation is deterministic; it does not supply new randomness.

Definition 2 (PCG64: the PCG XSL RR 128/64 generator). Let

$$M = 2^{128}, \quad a = 47026247687942121848144207491837523525. \quad (2.23)$$

Given an initialized state $s_0 \in \{0, \dots, M-1\}$ and a fixed odd increment $c \in \{1, 3, \dots, M-1\}$, update, for $k \geq 0$,

$$s_{k+1} = (as_k + c) \bmod M. \quad (2.24)$$

Split the updated state into its upper and lower 64 bits, and extract its top six bits:

$$h_{k+1} = \left\lfloor \frac{s_{k+1}}{2^{64}} \right\rfloor, \quad \ell_{k+1} = s_{k+1} \bmod 2^{64}, \quad r_{k+1} = \left\lfloor \frac{s_{k+1}}{2^{122}} \right\rfloor. \quad (2.25)$$

Here $h_{k+1}, \ell_{k+1} \in \{0, \dots, 2^{64} - 1\}$ and $r_{k+1} \in \{0, \dots, 63\}$. Write $x \oplus y$ for the *bitwise exclusive-or* of two 64-bit integers: each result bit is 1 exactly when the corresponding input bits differ. Define a cyclic right rotation by r bits, for integers x and r , as

$$\text{rotr}_{64}(x, r) = \left\lfloor \frac{x}{2^r} \right\rfloor + 2^{64-r} (x \bmod 2^r), \quad 0 \leq x < 2^{64}, \quad 0 \leq r < 64. \quad (2.26)$$

This moves the last r bits to the front; for $r = 0$ it leaves x unchanged. The raw output is

$$w_{k+1} = \text{rotr}_{64}(h_{k+1} \oplus \ell_{k+1}, r_{k+1}) \in \{0, \dots, 2^{64} - 1\}. \quad (2.27)$$

These are the state and output rules implemented by NumPy's PCG64.

Thus one step first advances the large integer state, then combines its two halves by exclusive-or, then rotates the result by an amount determined by the state. For a smaller illustration of the bit operations, the exclusive-or of the four-bit strings 1010 and 1100 is 0110; rotating 1001 right by one position gives 1100.

Because $a \equiv 1 \pmod{4}$ and c is odd, the state recurrence has period 2^{128} : for a fixed increment it visits every 128-bit state before repeating. To appreciate how large this period is, suppose we could generate one sample every femtosecond (10^{-15} seconds). Even at this hypothetical rate, generating 2^{128} samples would take approximately 1.08×10^{16} years—about 10.8 million billion years. For comparison, the universe is approximately 13.8 billion years old, so this would take about 780,000 times the current age of the universe.

Why PCG64 transforms the state before returning it. Even PCG64's long state cycle has simple patterns: because a and c are odd, $s_{k+1} \equiv s_k + 1 \pmod{2}$, so the lowest state bit alternates between zero and one. Returning the state directly would expose that pattern. In (2.27), exclusive-or combines corresponding bits from the two halves of the state. The rotation then moves those combined bits to positions selected by the state itself. Thus the lowest returned bit is not simply the alternating lowest state bit. The state-dependent rotation is an essential part of the statistical improvement; the exclusive-or alone is not sufficient [16, Section 6.3.3].

This separates two design tasks: F supplies an efficient update with a known long cycle, while G improves the statistical behavior of the returned sequence. It adds no new randomness and does not lengthen the state cycle. Neither the transformation nor a long period establishes independence or cryptographic security. Figure 3 illustrates the different output patterns, but it does not isolate the effect of G : the generators also differ in state size, parameters, and number of plotted points. The right-hand sequence remains deterministic; its structure is less apparent in this plot of successive outputs.

From a seed to a uniform sample. The call `np.random.default_rng(3414)` initializes PCG64 using NumPy’s SeedSequence, which processes the seed to determine the initial state and increment. It does not simply set $s_0 = 3414$. Restarting with the same seed reproduces the same initialized values and hence the same raw output sequence. The Generator object supplies the distribution-sampling methods on top of this bit generator. For an ordinary double-precision `rng.random()` call, the 64-bit output w is converted by

$$u = \frac{\lfloor w/2^{11} \rfloor}{2^{53}}. \quad (2.28)$$

This retains the leading 53 bits and scales the resulting integer into $[0, 1)$. The possible values are $0, 2^{-53}, 2 \cdot 2^{-53}, \dots, 1 - 2^{-53}$, a finite grid approximating the continuous uniform model. If the raw word were exactly uniform, each of those grid values would have equal probability, since each has 2^{11} raw words mapping to it. In a fixed-seed computation, these are still deterministic samples. Other methods, such as `rng.integers` and `rng.normal`, use the bit generator with transformations appropriate to their requested distributions.

Example 3 (Turning uniform draws into other distributions). If $U \sim \text{Uniform}[0, 1)$, then

$$X = a + (b - a)U, \quad D = 1 + \lfloor 6U \rfloor, \quad (2.29)$$

are respectively uniform on $[a, b)$ and on $\{1, \dots, 6\}$ in the ideal model. Also, $B = 1$ when $U < p$ and $B = 0$ otherwise is a Bernoulli variable with success probability p . For example, $U = 0.375$ gives $D = 3$ and, when $p = 0.2$, gives $B = 0$.

For an exponential waiting time with rate $\lambda > 0$, set

$$T = -\frac{\log(1 - U)}{\lambda}. \quad (2.30)$$

For $t \geq 0$, the event $T \leq t$ is equivalent to $U \leq 1 - e^{-\lambda t}$, giving the required distribution function. In numerical code, `-np.log1p(-u) / rate` evaluates this expression accurately for small u . These constructions illustrate *inverse transform sampling*: convert cumulative probabilities into values. Library routines may use other, more efficient methods.

2.4 Create one generator and draw samples

NumPy recommends constructing a Generator with `np.random.default_rng`. The current documented default bit generator is PCG64, defined in Definition 2; the Generator transforms its output into samples from the requested distributions. Create a generator once and then reuse it:

LISTING 3: A generator with a fixed seed.

```
1 import numpy as np
2
3 rng = np.random.default_rng(3414)
4 u = rng.random(5)           # Five floats in [0, 1)
5 dice = rng.integers(1, 7, size=5) # 1, ..., 6
6 print(u)
7 print(dice)
```

The seed 3414 is an arbitrary nonnegative integer. It identifies a reproducible experiment; choosing a particular seed does not guarantee a more accurate estimate. Omitting the seed, as in `default_rng()`, initializes the generator using entropy supplied by the operating system. Each call above advances the generator's state.

Call, after creating <code>rng</code>	Meaning
<code>rng.random(size=n)</code>	n approximately uniform samples in $[0, 1)$.
<code>rng.integers(1, 7, size=n)</code>	n die rolls; the upper endpoint is excluded.
<code>rng.choice(values, size=n, p=p)</code>	Draw from a finite distribution, with replacement by default.
<code>rng.binomial(1, p, size=n)</code>	n Bernoulli outcomes with success probability p .
<code>rng.binomial(m, p, size=n)</code>	n counts, each from m Bernoulli trials.
<code>rng.geometric(p, size=n)</code>	n waiting times in trials, counting the first success; values start at 1.
<code>rng.poisson(lam=3.0, size=n)</code>	n Poisson counts with mean 3.
<code>rng.normal(0.0, 1.0, size=n)</code>	n standard normal samples; the second parameter is standard deviation.
<code>rng.exponential(scale=2.0, size=n)</code>	n exponential samples with mean 2; <code>scale</code> is not the rate.
<code>rng.permutation(n)</code>	A shuffled ordering of $0, \dots, n - 1$.

For example, `rng.normal(65.0, 10.0, size=n)` samples the model $\mathcal{N}(65, 10^2)$: pass the standard deviation 10, not the variance 100. The realized sample mean and standard deviation fluctuate; they are not forced to equal 65 and 10. The exponential call illustrates another continuous family. Floating-point outputs occupy a finite set of representable values, so they approximate a continuous distribution. For large simulations, use array-valued draws rather than generating one number at a time in a Python loop.

2.5 Reproducibility and common sampling mistakes

Reinitializing a generator with the same seed restarts the same sequence. The following experiment verifies this behavior in the same environment:

```
1 g1 = np.random.default_rng(3414)
2 g2 = np.random.default_rng(3414)
3 assert np.array_equal(g1.integers(0, 100, 20),
4                       g2.integers(0, 100, 20))
```

Those two arrays are identical, so they are not two independent replications. In particular, recreating `default_rng(3414)` inside every loop iteration repeatedly produces the same first draw.

Key Idea

Initialize once, then draw repeatedly. For a reproducible experiment, record the seed, the generator, the software version, and the sequence of sampling operations.

Sampling without replacement also changes the probability model. Drawing five distinct student IDs using `rng.choice(100, size=5, replace=False)` does not produce iid IDs: after one ID is selected, it cannot appear again. For repeated draws from the fixed three-point distribution above, use replacement. To create streams for separate simulations, NumPy provides child seeds through `SeedSequence.spawn`, designed to avoid overlapping streams with very high probability:

```

1 seeds = np.random.SeedSequence(3414).spawn(4)
2 generators = [np.random.default_rng(s) for s in seeds]
3 replicates = [g.binomial(1, 0.3, size=1000)
4               for g in generators]

```

For simple sequential replications, continuing to use one generator is also appropriate. A plot or histogram can reveal obvious errors, but agreement with a histogram alone does not establish independence or generator quality.

3 Monte Carlo methods and simulation

Random numbers are useful because a difficult calculation can sometimes be expressed as an average over randomly chosen inputs. We then approximate the average by sampling. The quantity we want may be completely deterministic: an integral has a fixed value even when our method for estimating it uses random numbers. This is the central idea behind Monte Carlo methods [1].

Definition 3 (Monte Carlo method). A *Monte Carlo method* uses random sampling to estimate a numerical quantity, such as an expectation, probability, integral, or volume. A *simulation* generates outcomes from a mathematical model; repeated simulations can supply the samples used in a Monte Carlo estimate.

For example, simulating one customer's arrival and service produces one possible experience. Repeating the simulation and averaging the waiting times estimates an expected waiting time. Both the probability model and the sampling procedure matter: many samples from an inappropriate model do not produce an accurate description of the real system. Numerical Monte Carlo can also introduce randomness into a problem that originally contains none [5].

3.1 Estimation by sample averages

Suppose the desired quantity is $\mu = \mathbb{E}[Y]$. Generate independent observations $Y_1, \dots, Y_n$ with the same distribution as Y , and form

$$\hat{\mu}_n = \bar{Y}_n = \frac{1}{n} \sum_{i=1}^n Y_i. \quad (3.1)$$

If $\mathbb{E}[|Y|] < \infty$, the law of large numbers says that this average converges to μ with probability one as n increases. If, in addition, $\text{Var}[Y] = \sigma^2 < \infty$, independence gives

$$\mathbb{E}[\hat{\mu}_n] = \mu, \quad \text{Var}[\hat{\mu}_n] = \frac{\sigma^2}{n}, \quad \text{SD}(\hat{\mu}_n) = \frac{\sigma}{\sqrt{n}}. \quad (3.2)$$

Thus, the estimate is *unbiased*: its average over repeated runs equals the target. This does not mean that the estimate from any one run equals the target. The standard deviation of an estimator is called its *standard error*; it describes the typical variation between independent repetitions of the whole calculation [5].

Usually σ is unknown. For $n \geq 2$, estimate it from the observations:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (Y_i - \bar{Y}_n)^2, \quad \widehat{\text{SE}}(\hat{\mu}_n) = \frac{s}{\sqrt{n}}. \quad (3.3)$$

For a sufficiently large sample, the central limit theorem motivates the approximate 95% confidence interval

$$\hat{\mu}_n \pm 1.96 \frac{s}{\sqrt{n}}. \quad (3.4)$$

The interpretation concerns repeated experiments: approximately 95% of intervals produced this way contain the fixed value μ . It is not a deterministic error bound. The approximation can be poor with small samples, strongly skewed data, or rare events. The stated formulas assume independent samples and finite variance; correlated samples require a different variance calculation. In computation, these ideal assumptions are approximated using a suitable pseudorandom generator.

3.2 Monte Carlo integration

Let X be uniform on $[a, b]$, where $a < b$. Its density is $1/(b - a)$, so

$$\begin{aligned} I &= \int_a^b f(x) \, dx \\ &= (b - a) \int_a^b f(x) \frac{1}{b - a} \, dx \\ &= (b - a) \mathbb{E}[f(X)]. \end{aligned} \quad (3.5)$$

This identity turns integration into expectation estimation. Draw independent U_i uniformly from $[0, 1]$, put $X_i = a + (b - a)U_i$, and compute

$$\hat{I}_n = \frac{b - a}{n} \sum_{i=1}^n f(X_i). \quad (3.6)$$

The formula is the interval length multiplied by an average sampled height. For the error formulas above, use $Y_i = (b - a)f(X_i)$.

Example 4 (Integrating x^2). Consider $I = \int_0^1 x^2 \, dx = 1/3$. Suppose four sampled points are 0.1, 0.4, 0.6, and 0.9. Then

$$\hat{I}_4 = \frac{0.1^2 + 0.4^2 + 0.6^2 + 0.9^2}{4} = \frac{1.34}{4} = 0.335. \quad (3.7)$$

This illustrative sample happens to give a close answer, but four samples do not justify expecting this accuracy. For a uniform U on $[0, 1]$,

$$\text{Var}[U^2] = \mathbb{E}[U^4] - (\mathbb{E}[U^2])^2 = \frac{1}{5} - \frac{1}{9} = \frac{4}{45}. \quad (3.8)$$

Consequently, the standard error is $2/\sqrt{45n}$, approximately 0.149 when $n = 4$. The actual error for these particular four points is much smaller, illustrating the distinction between an observed error and the uncertainty of the sampling procedure.

Figure 4 shows what this average means geometrically. The orange segments have heights $f(U_i)$. Averaging their heights produces a rectangle whose width is the interval length. Its area is the Monte Carlo estimate in (3.6).

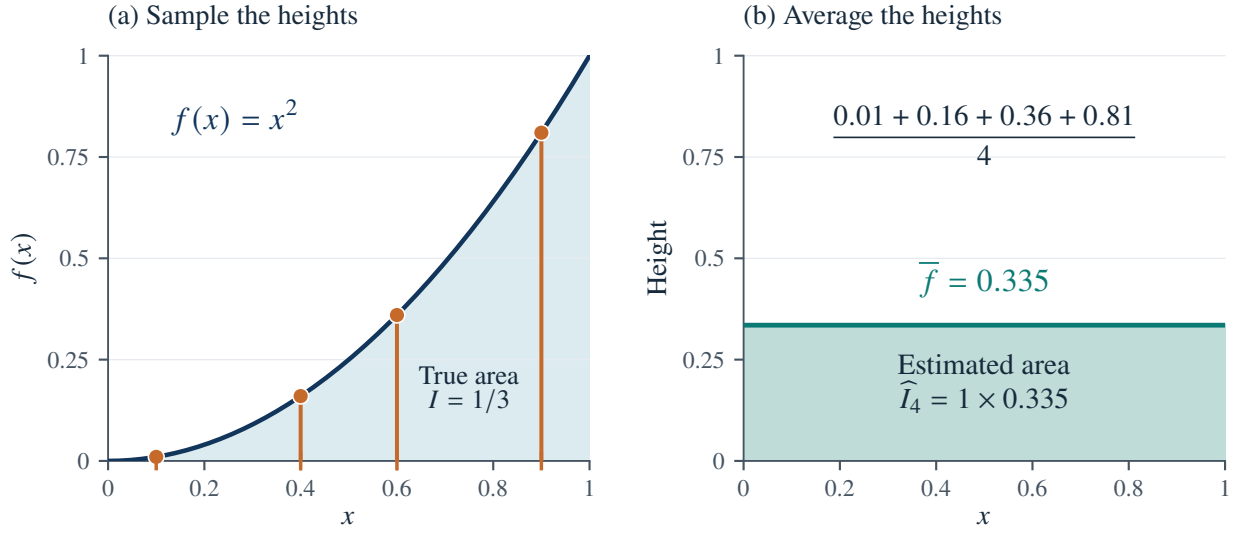


FIGURE 4: Monte Carlo integration by averaging sampled heights. Left: the four illustrative locations 0.1, 0.4, 0.6, 0.9 from the example give heights 0.01, 0.16, 0.36, 0.81. The shaded area under x^2 is the target $I = 1/3$. Right: the average height is 0.335, so the unit-width rectangle has estimated area $\hat{I}_4 = 0.335$. These four points illustrate the calculation; their unusually close estimate does not guarantee accuracy from a small random sample.

Why use randomness when quadrature rules already exist? For a smooth function of one variable, the trapezoidal rule, Simpson's rule, or Gaussian quadrature will often achieve much greater accuracy for the same work. Monte Carlo becomes especially useful when there are many input variables or when evaluating the integrand already involves a complicated simulation. A tensor grid with m points in each of d dimensions has m^d points. Independent Monte Carlo sampling still has the $n^{-1/2}$ standard-error factor, provided the integrand has finite variance. However, its variance and the cost of evaluating each sample can depend strongly on d ; high-dimensional problems are not automatically easy.

3.3 Estimating areas and π

Sample a point (U, V) uniformly in the unit square. The probability that it lies in the quarter unit disk equals the disk's area divided by the square's area, namely $\pi/4$; see Figure 5. Define the indicator variable

$$Y_i = \begin{cases} 1, & U_i^2 + V_i^2 \leq 1, \\ 0, & U_i^2 + V_i^2 > 1. \end{cases} \quad H_n = \sum_{i=1}^n Y_i, \quad \hat{\pi}_n = 4 \frac{H_n}{n}. \quad (3.9)$$

Here the pairs (U_i, V_i) are independent and uniform on the square. Since Y_i is Bernoulli with success probability $p = \pi/4$,

$$\mathbb{E}[\hat{\pi}_n] = \pi, \quad \text{SD}(\hat{\pi}_n) = 4 \sqrt{\frac{p(1-p)}{n}} \leq \frac{2}{\sqrt{n}}. \quad (3.10)$$

Proof of (3.10). Since Y_i takes values 0 and 1, we have $Y_i^2 = Y_i$. Consequently, $\mathbb{E}[Y_i] = p$ and $\text{Var}[Y_i] = \mathbb{E}[Y_i^2] - (\mathbb{E}[Y_i])^2 = p - p^2 = p(1-p)$. Linearity of expectation gives the first equality below. Independence

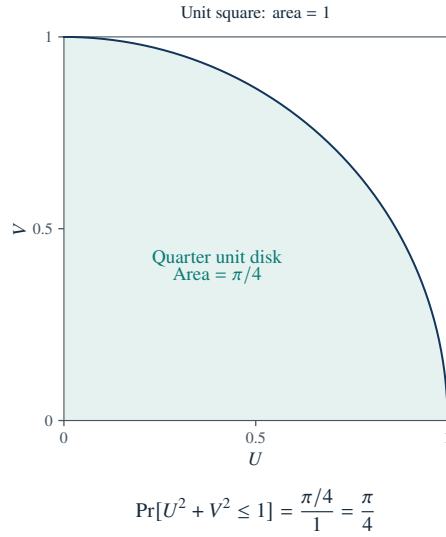


FIGURE 5: By sampling points uniformly on the unit square, the probability that a point lies in the quarter unit disk is equal to the area of the quarter unit disk, which is $\pi/4$.

makes the covariances zero, allowing the variances to add in the second:

$$\begin{aligned}\mathbb{E}[\hat{\pi}_n] &= \frac{4}{n} \sum_{i=1}^n \mathbb{E}[Y_i] = 4p = \pi, \\ \text{Var}[\hat{\pi}_n] &= \frac{16}{n^2} \sum_{i=1}^n \text{Var}[Y_i] = \frac{16p(1-p)}{n}.\end{aligned}\tag{3.11}$$

Taking the square root gives $\text{SD}(\hat{\pi}_n) = 4\sqrt{p(1-p)/n}$. Finally, $p(1-p) = 1/4 - (p - 1/2)^2 \leq 1/4$, which gives $\text{SD}(\hat{\pi}_n) \leq 2/\sqrt{n}$. ■

Figure 6 follows one growing sample. Teal points fall inside the quarter disk and orange points fall outside. Multiplying the fraction of teal points by four gives the displayed estimate. At $n = 100, 1000, 10,000, 100,000$, the absolute errors are approximately 0.1416, 0.0384, 0.0144, 0.00323, respectively. The running curve also shows temporary movements away from π : improvement is not monotone. It is the standard error, rather than the error of every individual run, that decreases proportionally to $n^{-1/2}$.

The following NumPy example computes both estimates and their estimated standard errors. The fixed seed makes the experiment reproducible. Increasing the number of samples changes the precision; changing the seed produces another realization of the experiment.

LISTING 4: Monte Carlo integration and an area estimate.

```
1 import numpy as np
2
3 rng = np.random.default_rng(3414)
4 n = 100_000
5
6 y = rng.random(n)**2
```

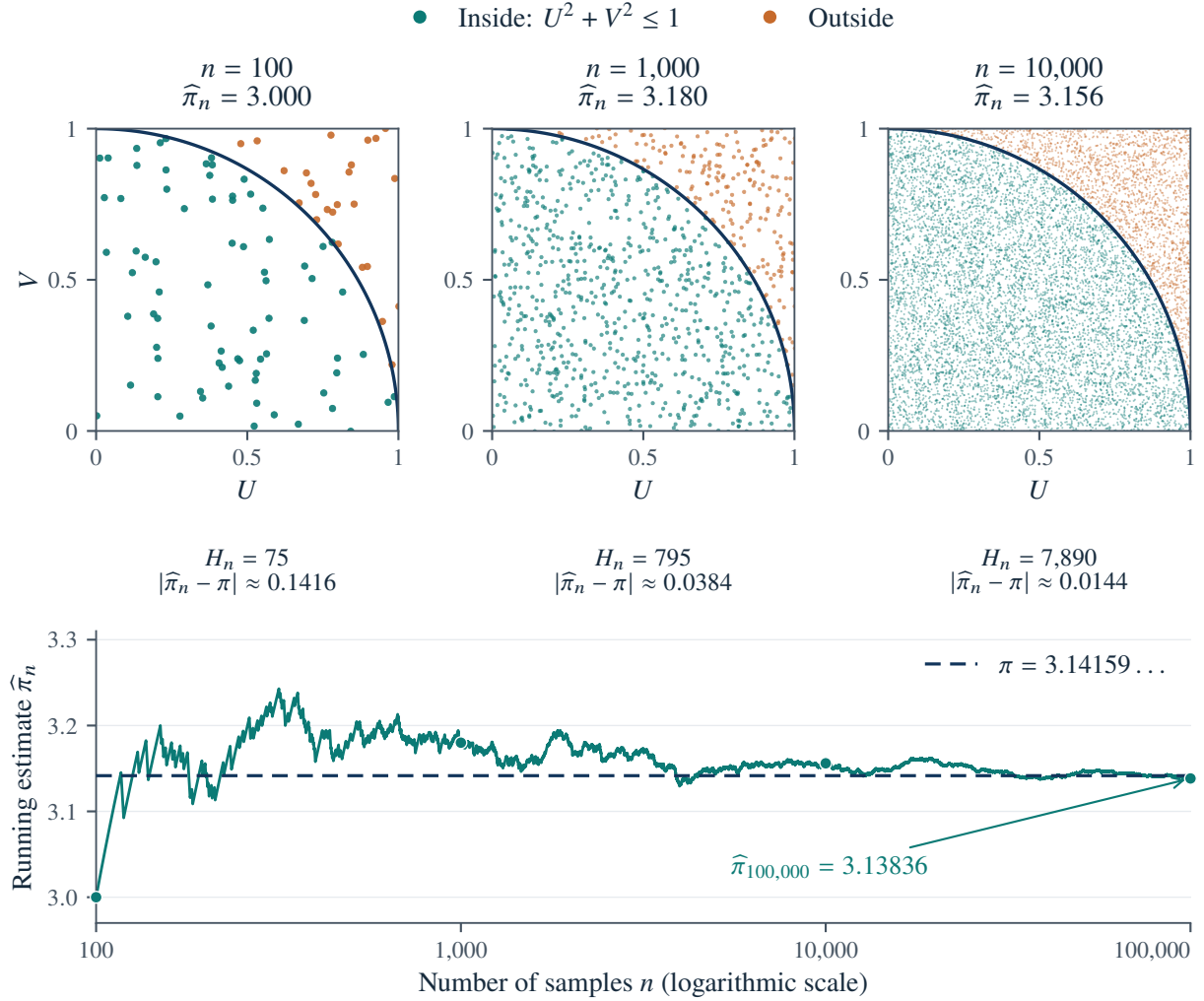


FIGURE 6: Estimating π by counting points in the quarter unit disk. The top panels use the first 100, 1000, and 10,000 points of one sample; each smaller sample is contained in the next. The labels give the number H_n inside, the estimate $4H_n/n$, and its absolute error. The bottom panel follows the running estimate through 100,000 points, with a logarithmic horizontal axis and a dashed line at π . The final count is $H_{100,000} = 78,459$, giving $\hat{\pi}_{100,000} = 3.13836$. The points come from Listing 4 with seed 3414 and $n = 100,000$, after its integration draws; all panels reuse prefixes of that same point array.

```

7 integral = y.mean()
8 integral_se = y.std(ddof=1) / np.sqrt(n)
9
10 points = rng.random((n, 2))
11 inside = (points**2).sum(axis=1) <= 1.0
12 pi_estimate = 4.0 * inside.mean()
13 pi_se = 4.0 * inside.std(ddof=1) / np.sqrt(n)
14
15 print("integral:", integral, "estimated SE:", integral_se)
16 print("pi:", pi_estimate, "estimated SE:", pi_se)

```

3.4 Monte Carlo simulation of waiting times

Example 5 (Simulating the time to the first successful transmission). Suppose each transmission succeeds independently with probability 0.2. NumPy's `geometric` method counts trials *including* the first success. Each entry below represents one complete sequence of sensor retries, and a new entry represents a fresh independent repetition under the model. Each attempt takes two seconds.

LISTING 5: Waiting times and error bars for their estimated mean.

```

1 import numpy as np
2
3 rng = np.random.default_rng(20261013)
4 n = 10_000
5 trials = rng.geometric(p=0.2, size=n)
6 delivery_seconds = 2.0 * trials
7 mean_seconds = delivery_seconds.mean()
8 sd_seconds = delivery_seconds.std(ddof=1)
9 se_seconds = sd_seconds / np.sqrt(n)
10 ci95 = (mean_seconds - 1.96 * se_seconds,
11         mean_seconds + 1.96 * se_seconds)
12
13 print("Mean delivery time and target:", mean_seconds, 10.0)
14 print("Spread of delivery times:", sd_seconds)
15 print("Estimated SE of mean:", se_seconds)
16 print("Approximate 95% CI for mean:", ci95)
17 print("Estimated probability of waiting over 20 seconds:",
18       np.mean(delivery_seconds > 20.0))

```

The SD estimates the spread of individual delivery times, whose model value is $\sqrt{80}$ s: if G is the trial count, $\mathbb{E}[G] = 1/p = 5$ and $\text{Var}[G] = (1-p)/p^2 = 20$, so $2G$ has mean 10 and variance 80. The SE estimates the uncertainty in the mean, whose model value here is $\sqrt{80/10,000}$ s ≈ 0.08944 s. Plotting the mean with `sd_seconds`, `se_seconds`, or the two endpoints in `ci95` therefore communicates three different quantities. The confidence interval is a large-sample approximation for the mean, not a prediction interval for the next transmission. Sampling more complete retry sequences improves estimation of the mean; it does not make a particular transmission arrive faster.

3.5 Monte Carlo and Las Vegas algorithms

The distinction is clearest when we separate two questions: *must the answer be correct?* and *must the computation finish within a prescribed budget?* The following definition makes the guarantees precise [17, 18].

Definition 4 (Monte Carlo and Las Vegas algorithms). Fix a computational problem. For an input x of size n , let $C(x)$ be the set of correct outputs. For example, a yes/no question has one correct answer, whereas a search problem may have several valid solutions. Let R denote the algorithm's random choices, $A(x, R)$ its returned answer, and $T(x, R)$ its running time. Set $T(x, R) = \infty$ if the run never terminates. Probabilities and expectations below are over R , with x held fixed.

1. A **Monte Carlo algorithm** with error bound δ , where $0 \leq \delta < 1$, has a finite work bound $b(n)$ such that, for every valid input x ,

$$\begin{aligned} T(x, r) &\leq b(n) \quad \text{for every choice sequence } r, \\ \Pr[A(x, R) \notin C(x)] &\leq \delta. \end{aligned} \tag{3.12}$$

Thus, every run finishes within the budget, but the answer is allowed to be wrong with probability at most δ . In applications, we usually choose δ much smaller than 1.

2. A **Las Vegas algorithm** has a finite expected-work bound $t(n)$ such that, for every valid input x ,

$$\begin{aligned} T(x, r) < \infty &\implies A(x, r) \in C(x) \quad \text{for every } r, \\ \mathbb{E}[T(x, R)] &\leq t(n) < \infty. \end{aligned} \tag{3.13}$$

Thus every returned answer is correct. The expected running time is bounded, and finite expectation implies termination with probability one. An individual run may take much longer than $t(n)$.

Monte Carlo bounds the work on every run and controls the chance of an error; Las Vegas guarantees correctness and bounds the work in expectation. Running time can vary in both cases: a Monte Carlo algorithm may stop early, and a Las Vegas algorithm may also have a finite worst-case bound. **These are overlapping guarantees, not mutually exclusive classes of algorithms.** Since the Monte Carlo definition permits $\delta = 0$, every always-correct algorithm with a finite worst-case work bound satisfies both definitions.

To make a useful comparison, state the *work bound as well as the correctness guarantee*. An $O(b(n))$ -time Monte Carlo guarantee bounds *every run* by that order of work and permits an error probability at most δ . An $O(t(n))$ -time Las Vegas guarantee bounds the *expected work* by that order and allows no incorrect answer.

Example 6 (Checking a matrix product: Freivalds' algorithm). Suppose a computation on another machine returns an $N \times N$ integer matrix C , claiming that $C = AB$. Recomputing the product by the ordinary algorithm costs $O(N^3)$ arithmetic operations. A randomized check instead uses three matrix–vector products, costing $O(N^2)$ operations per round [19].

Choose a column vector $r \in \{0, 1\}^N$ by independent fair coin flips. Compute Br , then $A(Br)$, and compare it with Cr . A mismatch proves that $C \neq AB$. If they agree, the product passes this check, but it may still be wrong. Notice that we never form AB during the check.

For a small worked example, take

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix}, \quad C = \begin{pmatrix} 4 & 4 \\ 10 & 9 \end{pmatrix}. \quad (3.14)$$

The true product has bottom-right entry 8, so C is incorrect. The random vector determines whether a single check notices this:

$$\begin{aligned} r = \begin{pmatrix} 1 \\ 0 \end{pmatrix} : \quad A(Br) &= \begin{pmatrix} 4 \\ 10 \end{pmatrix} = Cr \quad (\text{error missed}), \\ r = \begin{pmatrix} 0 \\ 1 \end{pmatrix} : \quad A(Br) &= \begin{pmatrix} 4 \\ 8 \end{pmatrix} \neq Cr = \begin{pmatrix} 4 \\ 9 \end{pmatrix} \quad (\text{error detected}). \end{aligned} \quad (3.15)$$

Here a check misses the error exactly when the second component of r is zero, so its probability of missing it is $1/2$.

The general error bound follows from the same idea [20]. Put $D = AB - C \neq 0$ and choose a nonzero entry D_{ij} . After fixing every component of r except r_j , the equation $(Dr)_i = 0$ can hold for at most one of the two choices $r_j = 0, 1$. Thus $Dr = 0$ has probability at most $1/2$. With K independent checks, accept C only if all checks pass. For any fixed incorrect C ,

$$\Pr[\text{all } K \text{ checks pass}] \leq 2^{-K}. \quad (3.16)$$

For $K = 20$, this bound is about 9.54×10^{-7} . A correct product always passes; an incorrect one can escape detection. This is a *one-sided* Monte Carlo error. The bound uses exact arithmetic; replacing equality with a floating-point tolerance needs a separate error analysis.

Example 7 (Las Vegas search: why randomness can help). Suppose a vector contains $N = 1,000,000$ entries, and our task is to find *any nonzero entry*. We are promised that at least one exists, but the search procedure is not told where the nonzero entries occur. Assume that accessing and checking any chosen entry has comparable cost.

Why might a sequential search be expensive? Consider an input with exactly $r = 100,000$ nonzero entries, all located in the final 100,000 positions. Scanning from the beginning requires 900,001 checks before finding the first nonzero entry.

Instead, use this randomized procedure:

1. Choose an index i uniformly from $\{1, \dots, N\}$.
2. Inspect a_i . If $a_i \neq 0$, return i and stop. Otherwise, repeat with a fresh, independent random choice.

Indices are sampled with replacement, so an index can be chosen again. The procedure does not need to know r .

A possible run. For the input described above, the random choices might give

Trial	1	2	3	4
Chosen index	142,073	806,512	374,291	953,217
Entry inspected	0	0	0	nonzero
Action	repeat	repeat	repeat	stop

(3.17)

This run returns index 953,217 after four checks. Another run might succeed immediately or require many more checks.

The expected advantage. Each trial succeeds with probability $r/N = 0.1$, regardless of where the r nonzero entries are located. Under independent uniform sampling, the number T of checks therefore satisfies

$$\Pr[T > k] = \left(1 - \frac{r}{N}\right)^k, \quad \mathbb{E}[T] = \frac{N}{r} = 10. \quad (3.18)$$

Thus, on this input, the randomized procedure needs only 10 checks on average, compared with 900,001 for scanning from the beginning. The useful guarantee is that the expected cost N/r holds for *every arrangement* of the r nonzero entries.

Why is this Las Vegas? The procedure returns an index only after verifying that its entry is nonzero. Consequently, every returned answer is correct. Unlucky random choices increase the running time but cannot produce an incorrect answer. There is no fixed upper bound on the number of trials: $\Pr[T > k] > 0$ for every finite k . Nevertheless, this probability tends to zero, so the procedure terminates with probability one and has finite expected running time.

When does randomness help? A large vector alone does not justify random sampling. If its first entry is nonzero, sequential scanning finishes immediately. If there is only one nonzero entry, sampling with replacement takes N checks on average and can waste work revisiting entries. The advantage here comes from having many qualifying entries while avoiding dependence on their locations relative to a fixed scanning order. The term *Las Vegas* describes the correctness and expected-work guarantees; it does not mean that the algorithm is always faster than a deterministic one.