

Lecture 9

Pseudorandom Numbers and Monte Carlo Methods

Virginia Tech Department of Computer Science
Fall 2026 • Sumeet Khatri

Why use randomness in a computation?

A traffic model, realistic image, or radiation treatment can involve far more possibilities than we can examine individually.

Random sampling selects manageable computations whose results can be combined into a useful estimate.

A computer therefore needs a practical source of random inputs. We will build one from a seed and arithmetic.

Choose secret keys for encryption

An ideal key consists of 128 independent fair bits:

$$2^{128} \text{ possible keys,} \quad \Pr[\text{one uninformed guess succeeds}] = 2^{-128}$$

Even one guess per femtosecond would require about 1.08×10^{16} years to try every key. That's **10.8 million billion years!**

Cryptography needs a secure generator, sufficient unpredictable initialization, and protected internal state.

What is a random number?

A random number is an observed value of a random variable governed by a specified probability law.

The numeral 4 could come from a fair die or a program that always prints 4. The numeral alone does not distinguish the procedures.

$$\text{Fair die:} \quad \Pr[X = j] = \frac{1}{6}, \quad j = 1, \dots, 6$$

Random outcomes need not be equally likely. A biased coin is also described by a random variable.

Uniform distribution

$$U \sim \text{Uniform}[0,1) \implies \Pr[a \leq U < b] = b - a$$

For $0 \leq a < b \leq 1$, probability equals interval length. Independence is an additional requirement:

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

The draws are **independent and identically distributed, or iid**, when both properties hold.

If $U_2 = U_1$, each variable is uniform, but knowing the first determines the second.

Transform uniform draws into other distributions

$$U \sim \text{Uniform}[0,1), \quad X = a + (b - a)U$$

X is uniform on $[a, b)$: intervals are stretched by the factor $b - a$.

$$D = 1 + \lfloor 6U \rfloor \quad \implies \quad \Pr[D = j] = \frac{1}{6}, \quad j = 1, \dots, 6$$

$$B = \begin{cases} 1, & U < p, \\ 0, & U \geq p, \end{cases} \quad \Pr[B = 1] = p$$

For $U = 0.375$, the die output is $D = 3$. With $p = 0.2$, the Bernoulli output is $B = 0$.

Computational randomness → pseudorandom numbers

A **pseudorandom number** generator starts from an initial seed and applies *deterministic* update rules.

Given the seed and the rules, the entire output sequence is fixed and completely predictable.

Physical sources such as electronic noise and **qubits** can supply *genuine, physical* randomness.

COMPUTING PRACTICES

Edgar H. Sibley
Panel Editor

Practical and theoretical issues are presented concerning the design, implementation, and use of a good, minimal standard random number generator that will port to virtually all systems.

RANDOM NUMBER GENERATORS: GOOD ONES ARE HARD TO FIND

1988

STEPHEN K. PARK AND KEITH W. MILLER

Build a source of zeros and ones

Goal: create a source that returns 0 or 1 and approximates a Bernoulli distribution:

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

Strategy: update an integer state, scale it into the unit interval, then compare it with 0.3.

We start with a seed and arithmetic alone.

Step one: create a sequence of integers

$$m = 2^{31} - 1 = 2147483647, \quad z_0 = 1 \quad \text{seed!}$$

$$z_{k+1} = (16807 z_k) \bmod m \quad \text{multiplier}$$

$$z_1 = (16807 \cdot 1) \bmod m = 16807,$$

$$z_2 = (16807 \cdot 16807) \bmod m = 282475249,$$

$$z_3 = (16807 \cdot 282475249) \bmod m = 1622650073.$$

Keep the updated state for the next call. Starting at 0 would produce 0 forever.

Why choose this modulus?

The modulus $m = 2^{31} - 1$ is prime. A nonzero state cannot collapse to 0 when multiplied by a nonzero remainder.

It also gives a large state space while fitting each state in a signed 32-bit integer.

Primality alone does not ensure a long cycle:

$$z_{k+1} = 1 \cdot z_k \bmod m \implies z_k = z_0 \text{ for every } k$$

The multiplier must be chosen carefully as well.

Why choose this multiplier?

The multiplier 16807 is a primitive root modulo m : its powers visit every nonzero remainder before returning to 1.

$$z_k = 16807^k \bmod m$$

$$\text{Period} = m - 1 = 2147483646$$

$$16807^{m-1} \equiv 1 \pmod{m}.$$

Every state $1, \dots, m - 1$ appears exactly once per cycle. Any nonzero seed starts elsewhere in that same cycle.

To get a sense of the pattern, without knowing anything about the generator, we would need at least $m - 1 = 2^{31} - 2 = 2,147,483,646$ draws! 68 years drawing one per second. Microseconds at a femtosecond rate.

Step two: scale into the unit interval

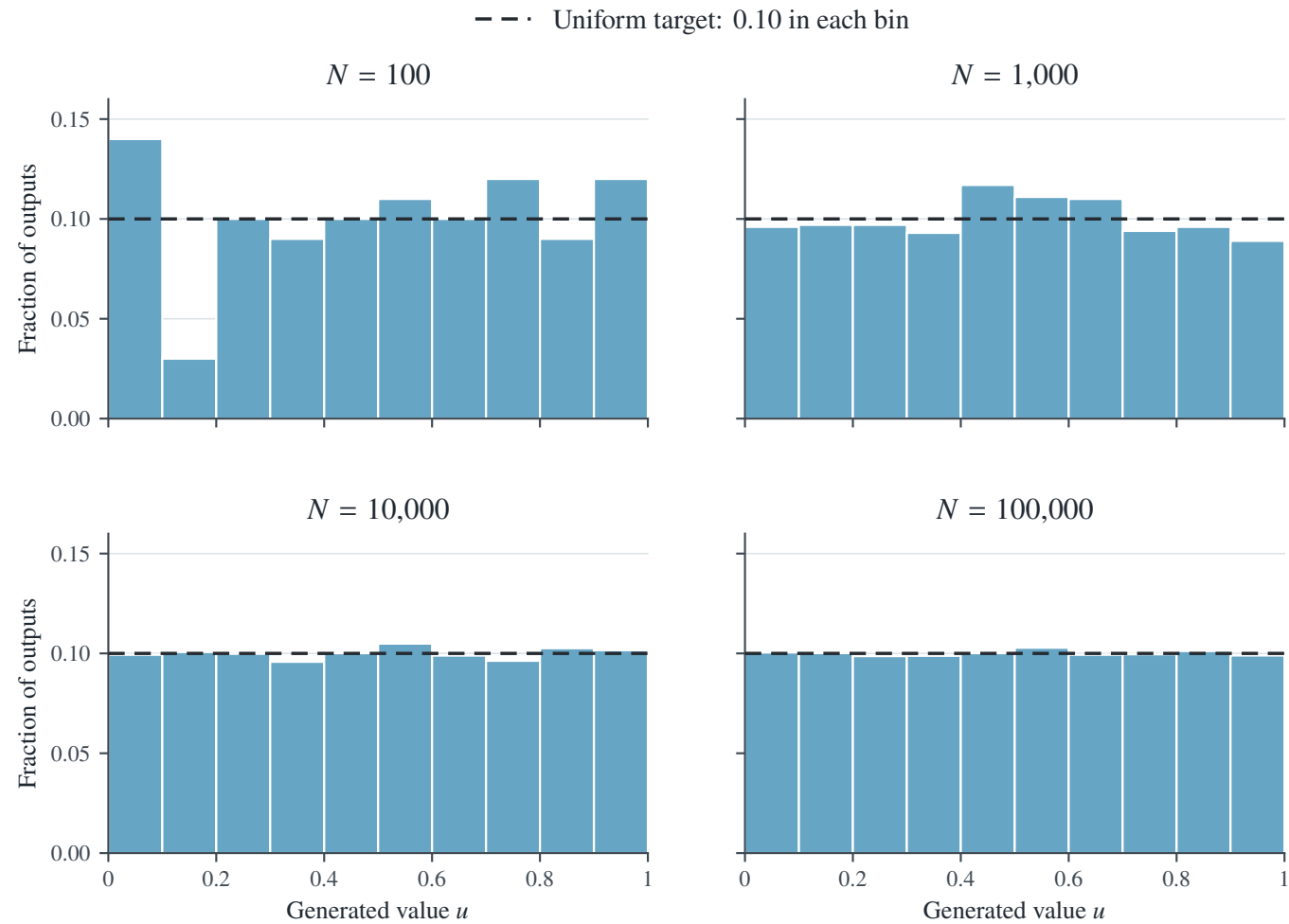
We want to simulate $\text{Uniform}([0,1])$, so need a number between 0 and 1

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

$$u_1 \approx 0.000007826, \quad u_2 \approx 0.131537788, \quad u_3 \approx 0.755605322$$

$$\frac{1}{m}, \frac{2}{m}, \dots, \frac{m-1}{m} \quad \text{are each visited once per cycle}$$

The generated values become evenly distributed



Step three: turn each value into a bit

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

$$u_k < 0.3 \iff 10z_k < 3m$$

The first three bits are 1,1,0. An ideal uniform draw is below 0.3 with probability 0.3.

$$\text{Full-cycle fraction of ones} = \frac{644245094}{2147483646} \approx 0.300000000093$$

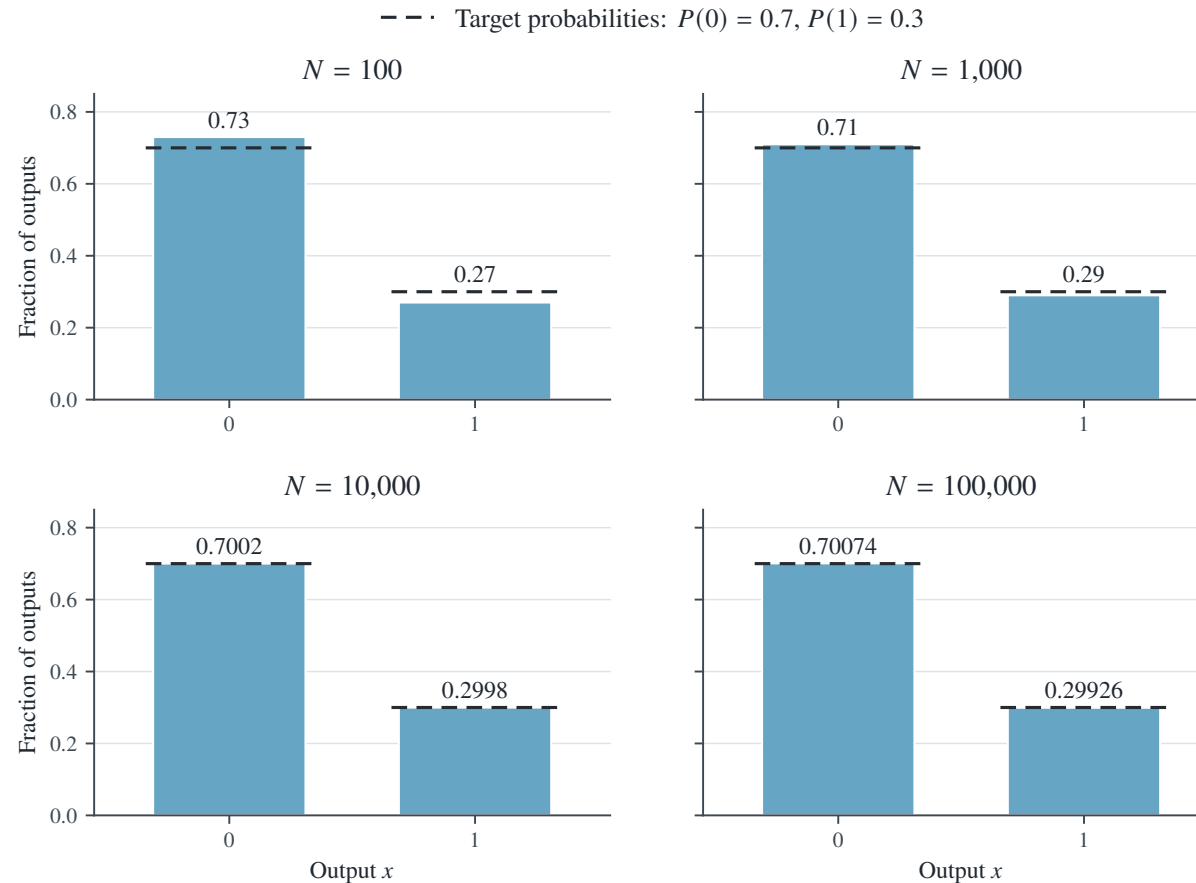
The complete source uses arithmetic alone

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

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

First ten: 1,1,0,0,0,1,1,0,0,0. First 10000 calls: 2998 ones, proportion 0.2998.

The binary proportions approach their targets



Targets: 0.7 zeros and 0.3 ones. The error need not decrease every time the sample grows.

Pseudorandom number generator (PRNG)

A PRNG starts from a seed and deterministically produces outputs intended to imitate independent samples from a particular distribution.

The seed initializes $s_0 \in S$. Each call changes the stored state and returns an output in A :

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

F controls the memory; G specifies what the program receives.

Simulation quality and cryptographic security

For simulation: imitate individual values and groups of successive values from the intended distribution.

For cryptography: with the seed and state hidden, past outputs should not let a feasible computation predict future outputs.

For unbiased bits: prediction success should not meaningfully exceed $\frac{1}{2}$.

The initialization must contain enough *unpredictable information*, and the seed and state must remain protected.

A known seed reproduces the sequence exactly. Reproducibility is useful for simulation.

The “middle-square” construction

Square a four-digit state. Pad to eight digits, keep the middle four, and divide the new state by 10000.

$$6752^2 = 45\mathbf{5895}04 \quad \Longrightarrow \quad z_1 = 5895, \quad u_1 = 0.5895$$

$$5895^2 = 34\mathbf{7510}25 \quad \Longrightarrow \quad z_2 = 7510, \quad u_2 = 0.7510$$

$$z_k : \quad 6752, 5895, 7510, 4001, 0080, \\ 0064, 0040, 0016, 0002, 0000, \dots$$

The “middle-square” construction

Starting from 6752, the first 100 returned values contain 92 zeros; 97 lie in $[0,0.1)$.
This is a bad approximation to a uniform distribution!

$$\text{Uniform target:} \quad \Pr[0 \leq U < 0.1] = 0.1$$

Once the state reaches zero, it stays there. Thresholding at 0.3 eventually returns 1 on every call.

Starting from 2500 is worse:

$$2500^2 = 06\mathbf{2500}00 \implies u_k = 0.25 \text{ forever}$$

These seeds enter one-state cycles almost immediately, with severely biased frequencies.

A small linear congruential generator

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

$$m = 16, \quad a = 5, \quad c = 1, \quad z_0 = 0$$

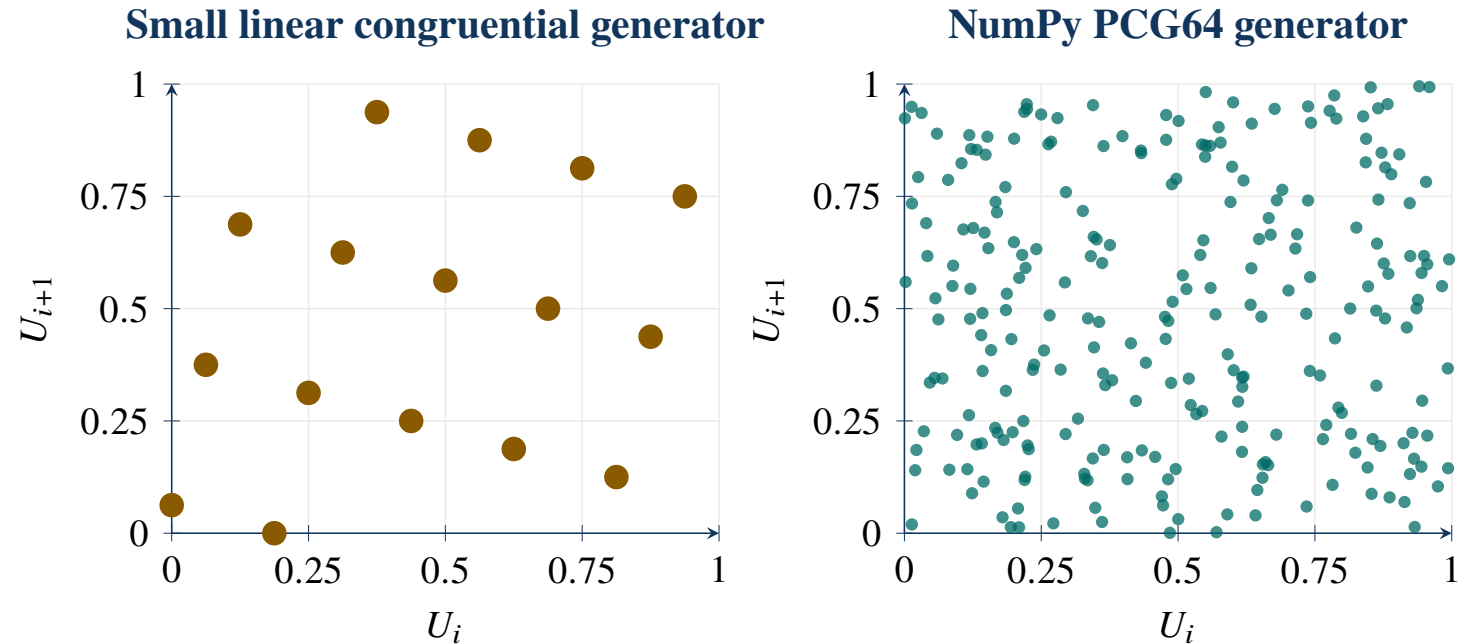
$$z_1 = 1, \quad z_2 = 6, \quad z_3 = 31 \bmod 16 = 15, \quad z_4 = 76 \bmod 16 = 12$$

$$z_k : \quad 0, 1, 6, 15, 12, 13, 2, 11, \\ 8, 9, 14, 7, 4, 5, 10, 3, 0, \dots$$

Every state occurs once per cycle, but the entire output sequence repeats after only 16 steps.

General pattern:
$$U_{i+1} = 5U_i + \frac{1}{16} - \left\lfloor 5U_i + \frac{1}{16} \right\rfloor$$

Read the next-output plot



$$(U_0, U_1) = (0, 1/16), \quad (U_1, U_2) = (1/16, 6/16), \quad (U_2, U_3) = (6/16, 15/16)$$

Left: all 16 transitions. **Right:** 256 PCG64 pairs.

Why this construction is common

$$x \equiv y \pmod{m} \iff m \text{ divides } x - y$$

An LCG keeps one changing integer in a fixed range and uses a few arithmetic operations per output.

For modulus 2^w , reduction means keeping the lowest w bits. Number theory gives conditions for long cycles.

A congruential recurrence is a useful design choice.

NumPy's PCG64

(Permuted congruential generator with 64 bits)

An LCG update followed by a transformation of the state's bits

$$M = 2^{128}, \quad a = 47026247687942121848144207491837523525$$

$$s_0 \in \{0, \dots, M - 1\}, \quad c \in \{1, 3, \dots, M - 1\} \text{ fixed}$$

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

The state has 128 bits. The raw returned integer has 64 bits; it is formed from the updated state.

Split the updated state into bit fields

$$h_{k+1} = \left\lfloor \frac{s_{k+1}}{2^{64}} \right\rfloor$$

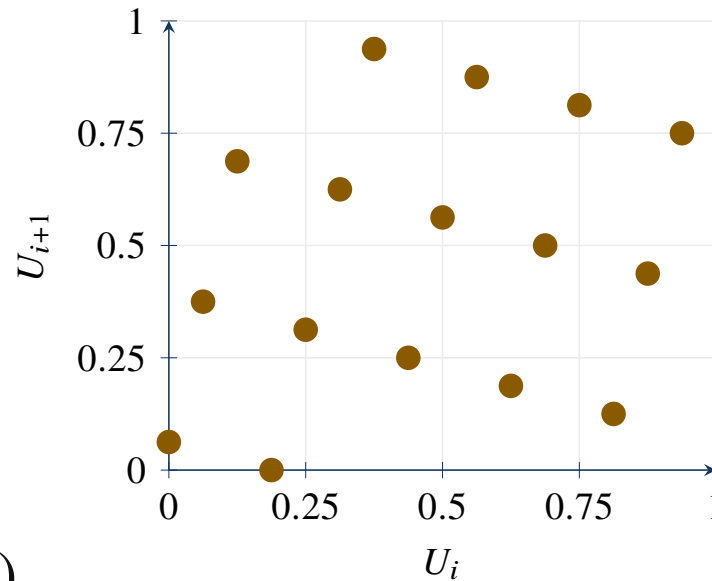
$$\ell_{k+1} = s_{k+1} \bmod 2^{64}$$

$$r_{k+1} = \left\lfloor \frac{s_{k+1}}{2^{122}} \right\rfloor$$

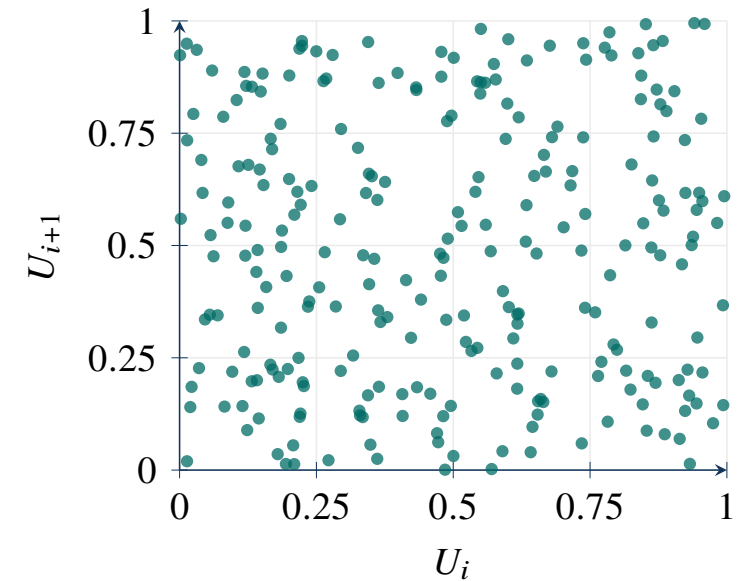
$$w_{k+1} = \text{rotr}_{64}(h_{k+1} \oplus \ell_{k+1}, r_{k+1})$$

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

Small linear congruential generator



NumPy PCG64 generator



How large is the state period?

$$a \equiv 1 \pmod{4}, \quad c \text{ odd} \implies \text{state period} = 2^{128}$$

For a fixed increment, every 128-bit state is visited before the state sequence repeats.

$$2^{128} \times 10^{-15} \text{ seconds} \approx 1.08 \times 10^{16} \text{ years}$$

Even at one sample per femtosecond: about 10.8 million billion years, or 780000 times the universe's age.

This describes the state cycle. Individual 64-bit output values can recur much sooner.

Create one generator and 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)
```

Every call advances the state. Omitting the seed uses entropy from the operating system for initialization.

Draw discrete samples

```
dice = rng.integers(1, 7, size=n)
x = rng.choice(values, size=n, p=probabilities)
bits = rng.binomial(1, p, size=n)
counts = rng.binomial(m, p, size=n)
trials = rng.geometric(p, size=n)
arrivals = rng.poisson(lam=3.0, size=n)
```

Continuous samples and shuffled orderings

```
u = rng.random(size=n)
z = rng.normal(0.0, 1.0, size=n)
scores = rng.normal(65.0, 10.0, size=n)
waits = rng.exponential(scale=2.0, size=n)
order = rng.permutation(n)
```

The same seed restarts the same sequence

```
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))
```

These arrays are identical, so they are not independent replications. Continuing to use `g1` advances its state.

Do not recreate the same seeded generator inside every loop iteration.

Record the seed, generator, software version, and sequence of sampling operations.

Monte Carlo simulation

A simulation generates outcomes from a mathematical model. A Monte Carlo method uses random samples to estimate a numerical quantity.

$$\mu = \mathbb{E}[Y], \quad \hat{\mu}_n = \bar{Y}_n = \frac{1}{n} \sum_{i=1}^n Y_i$$

For independent samples with the same distribution and $\mathbb{E}[|Y|] < \infty$, the **law of large numbers** gives $\hat{\mu}_n \rightarrow \mu$.

$$\mathbb{E}[\hat{\mu}_n] = \frac{1}{n} \sum_{i=1}^n \mathbb{E}[Y_i] = \mu$$

An integral is an interval length times an expectation

Let X be uniform on $[a, b]$, with density $1/(b - a)$. Then

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

$$X_i = a + (b - a)U_i, \quad U_i \sim \text{Uniform}([0, 1]), \quad \hat{I}_n = \frac{b - a}{n} \sum_{i=1}^n f(X_i)$$

The estimate is the interval length multiplied by an average sampled height.

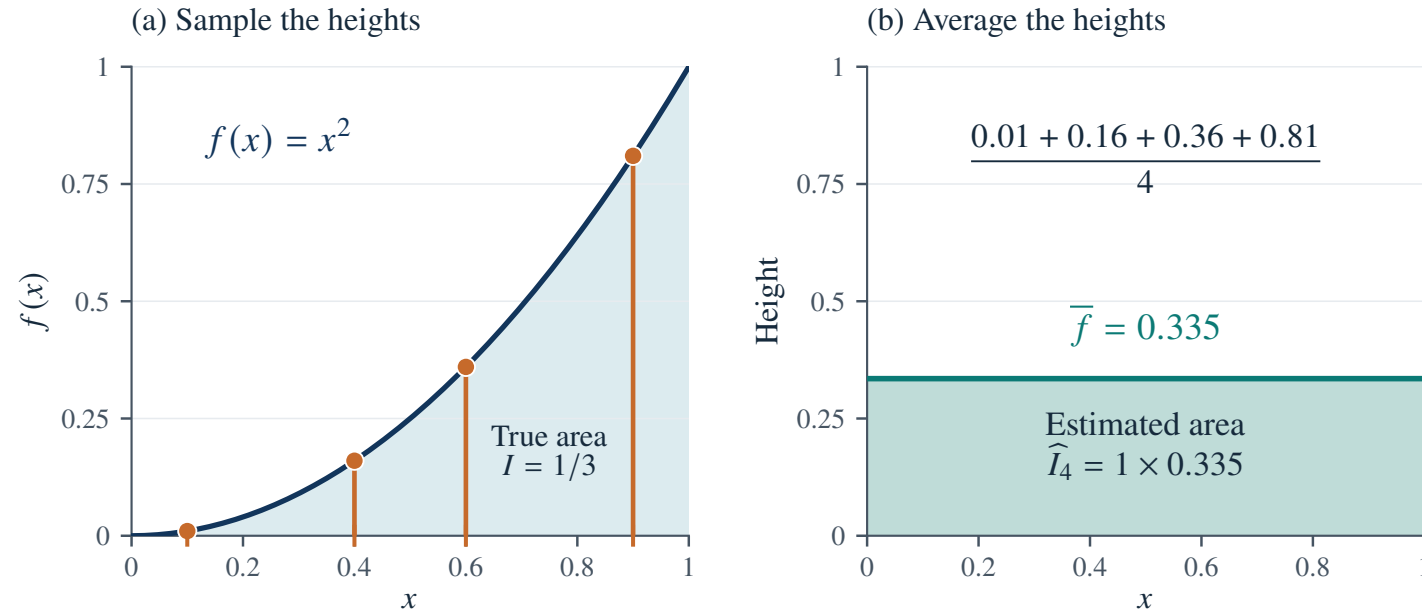
Integration using Monte Carlo

$$I = \int_0^1 x^2 dx = \frac{1}{3}$$

U_i	0.1	0.4	0.6	0.9
U_i^2	0.01	0.16	0.36	0.81

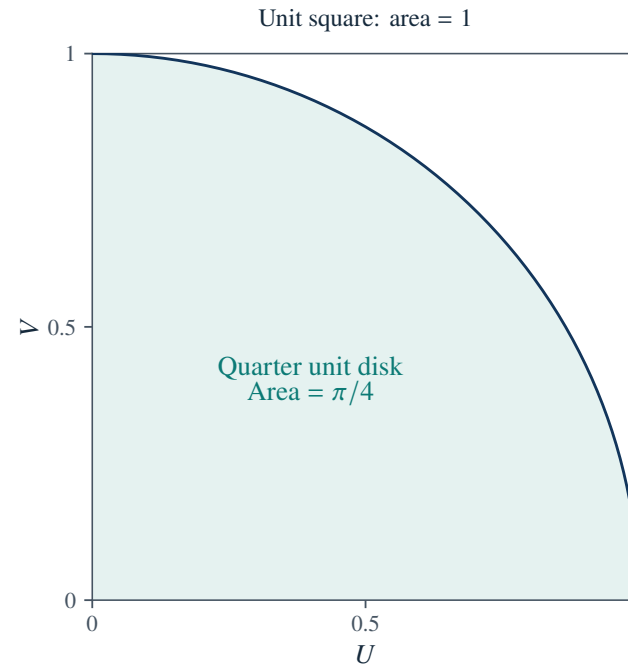
$$\hat{I}_4 = \frac{0.01 + 0.16 + 0.36 + 0.81}{4} = \frac{1.34}{4} = 0.335$$

Average the sampled heights



Left: evaluate $f(U_i)$ at the sampled locations. Right: use their average height to form a rectangle of area $\hat{I}_4 = 0.335$.

From areas to estimating π



$$\Pr[U^2 + V^2 \leq 1] = \frac{\pi/4}{1} = \frac{\pi}{4}$$

Sample (U, V) uniformly in the unit square. The chance of landing in the quarter disk is its area divided by the square's area: $\pi/4$.

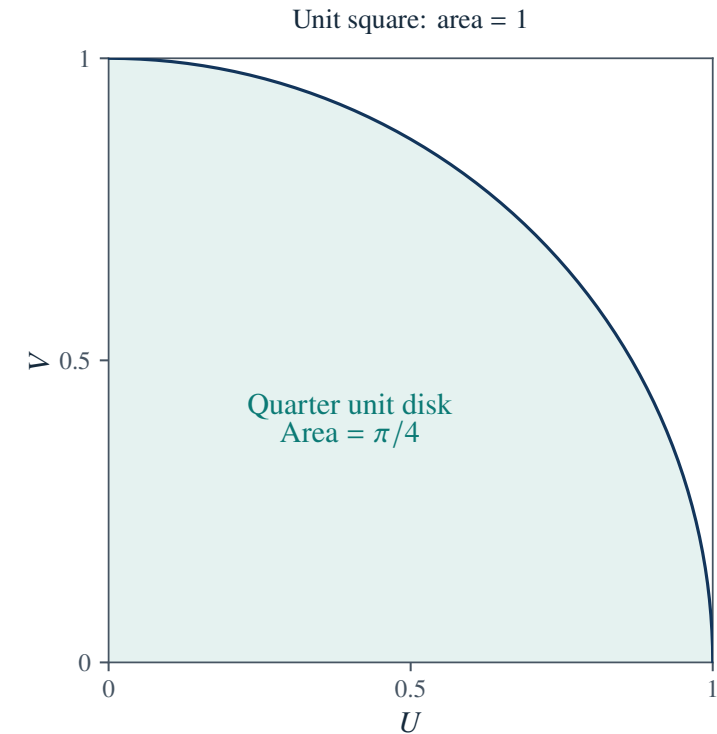
Count points inside the quarter disk

$$Y_i = \begin{cases} 1, & U_i^2 + V_i^2 \leq 1, \\ 0, & U_i^2 + V_i^2 > 1, \end{cases} \quad Y_i \sim \text{Bernoulli}(p), \quad p = \frac{\pi}{4}$$

$$H_n = \sum_{i=1}^n Y_i, \quad \hat{\pi}_n = 4 \frac{H_n}{n}$$

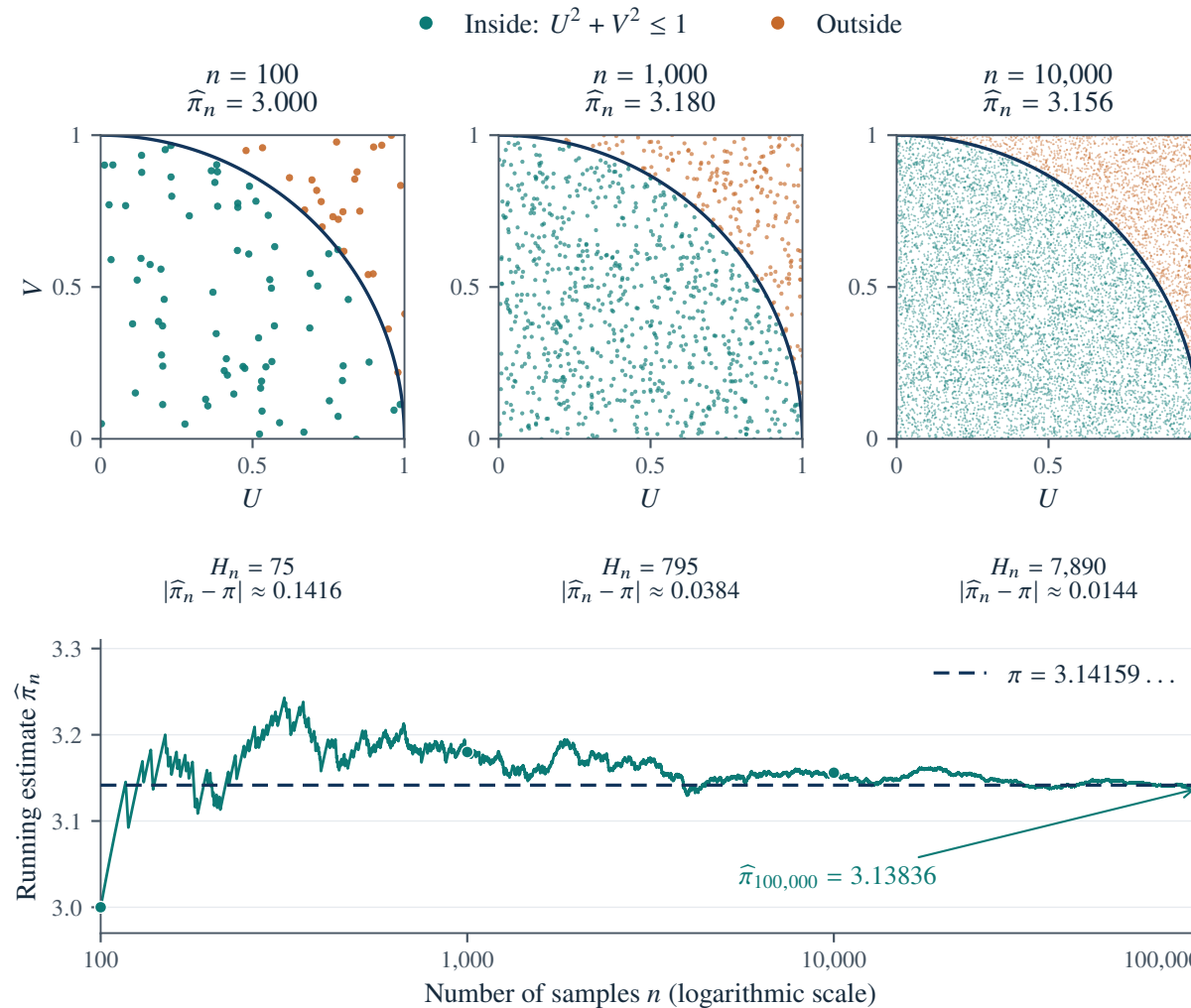
The observed fraction inside estimates the probability of landing inside. Multiply that fraction by four.

Procedure: take n samples, compute their average, multiply by 4.



$$\Pr[U^2 + V^2 \leq 1] = \frac{\pi/4}{1} = \frac{\pi}{4}$$

Count points inside the quarter disk



At $n = 100,000$,
 $H_n = 78,459$ gives
 $\hat{\pi}_n = 3.13836$. The running
 error can increase
 temporarily as more points
 are added.

How long until a sensor transmission succeeds?

Each attempt succeeds independently with probability $p = 0.2$ and takes two seconds. Let G count attempts, including the first success.

$$G \sim \text{Geometric}(0.2), \quad T = 2G \text{ seconds}$$

$$\mathbb{E}[G] = \frac{1}{p} = 5, \quad \text{Var}[G] = \frac{1-p}{p^2} = 20$$

$$\mathbb{E}[T] = 10 \text{ seconds}, \quad \text{Var}[T] = 80 \text{ seconds}^2$$

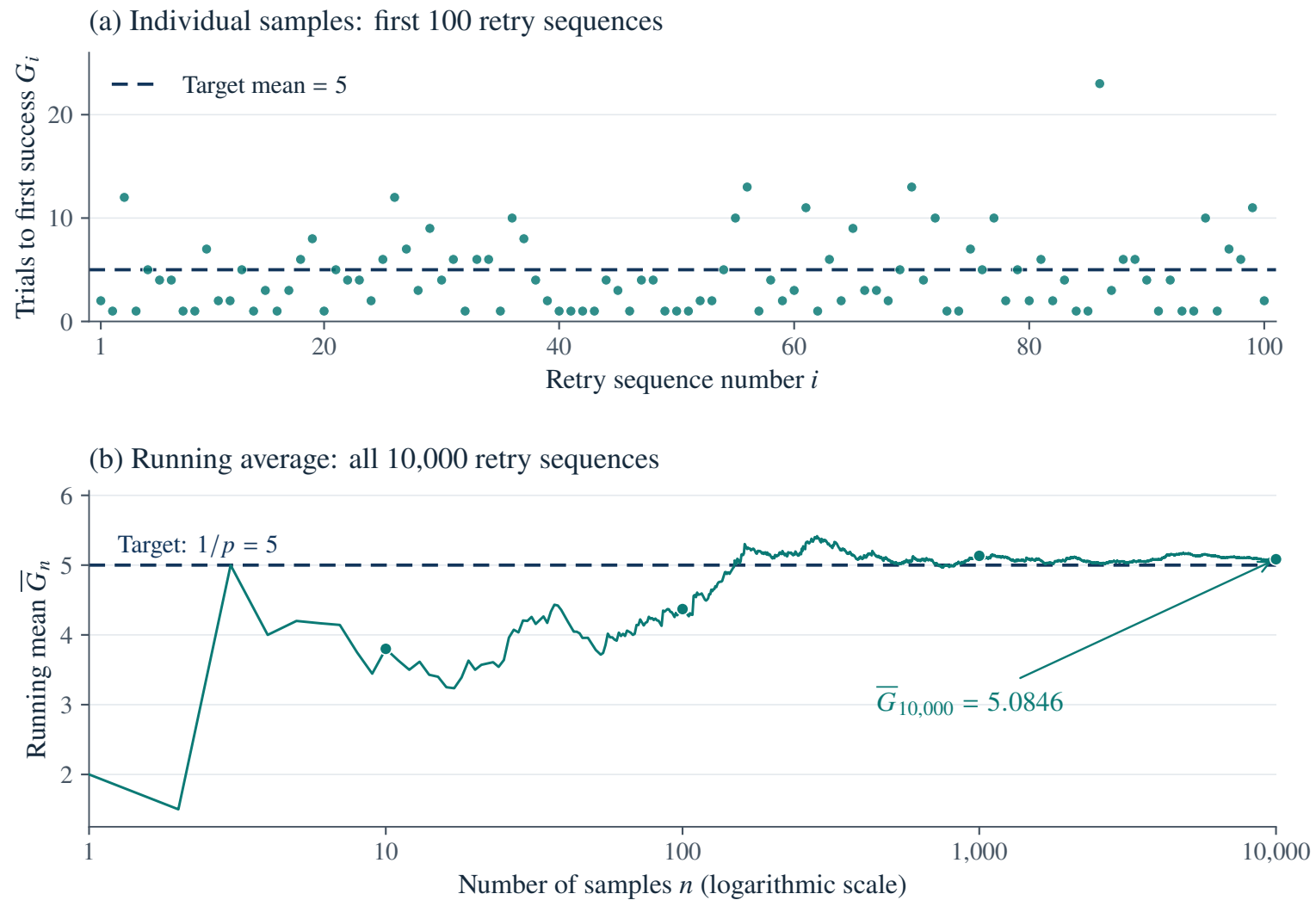
Each new observation is a fresh complete retry history under this model.

How long until a sensor transmission succeeds?

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)
```

How long until a sensor transmission succeeds?



Monte Carlo algorithms: bound work cost and control the error

Fix input x of size n . Let $\mathcal{C}(x)$ be its correct outputs, R the random choices, $A(x, R)$ the answer, and $T(x, R)$ the work.

Monte Carlo guarantee: there is a finite budget $b(n)$ and error bound $0 \leq \delta < 1$ such that, for every valid input x ,

$$T(x, r) \leq b(n) \quad \text{for every choice sequence } r$$

$$\Pr[A(x, R) \notin \mathcal{C}(x)] \leq \delta$$

Every run respects the work budget. An incorrect answer is allowed with a controlled probability.

Las Vegas algorithms: correctness with expected work cost

Fix input x of size n . Let $\mathcal{C}(x)$ be its correct outputs, R the random choices, $A(x, R)$ the answer, and $T(x, R)$ the work.

$$T(x, r) < \infty \implies A(x, r) \in \mathcal{C}(x) \quad \text{for every } r$$

$$\mathbb{E}[T(x, R)] \leq t(n) < \infty$$

Every returned answer is correct. Finite expected work implies termination with probability one, but individual runs may take longer.