

Università di Pisa—Dipartimento di Fisica "E. Fermi"

Monte Carlo Methods for Experimental Physics

Lecture notes

Luca Baldini, Elena Graverini

<https://codeberg.org/lbaldini/mcmep>

Table of contents

Preface	4
License	5
You are free to:	5
Under the following terms:	5
Notices:	6
1 Probability and statistics	7
2 Generating random numbers	8
2.1 Prelude: the wheel of fortune	8
2.2 Random (and pseudo-random) sequences	11
2.3 Tables	12
2.4 Digression: modular arithmetic	12
2.5 Pseudo-random sequences via recurrence	12
2.6 Middle square	14
2.7 Linear congruential generators	17
2.8 Tests for randomness	19
2.8.1 The worst LCG ever	21
2.9 Modern generators	21
2.9.1 Period: is that really important?	22
2.10 What am I using under the hood?	22
2.11 Exercises	23
3 Monte Carlo integration	24
3.1 The hit-or-miss method	24
3.2 The sample-mean method	30
3.3 Convergence	32
3.4 The curse of dimensionality	35
3.5 Low-discrepancy sequences	35
3.6 Executive summary	41

Appendices	43
A Version history	43
0.3.0 (2026-10-07)	43
0.2.0 (2026-10-04)	43
0.1.0 (2026-10-02)	43
0.0.1 (2026-09-19)	43

Preface

Version 0.3.0 ([history](#))

License

All the material in this work is distributed under the *Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)* license.



© 2026 Luca Baldini, Elena Graverini

You are free to:

Share — copy and redistribute the material in any medium or format for any purpose, even commercially.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

1 Probability and statistics

2 Generating random numbers

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” (John von Neumann, 1949)

I know, this is abused—you will find this very sentence quoted in any sensible resource about random numbers you might come across. But, on a second thought, it *really* encapsulates well the subject we are covering in this section: generating random sequences by means of arithmetical methods. Quite an oxymoron, eh?

2.1 Prelude: the wheel of fortune

If you were to ask an ordinary person on the street how they would go about drawing a random sequence, chances are that some sort of variant of the wheel of fortune would come up pretty high in the list of the most popular answers—provided that the person would care answering, of course. (I am sure you are familiar with at least one of the many popular TV shows that are based on the very concept of a wheel of fortune, but the [Wikipedia entry](#) on the subject is worth at least a quick read, because the idea seems very deeply rooted, and amusingly so, in western philosophy.)

Now, let us stop for a second and ask ourselves a very fundamental question: what is the magic in the wheel of fortune that makes its output so unpredictable? And I am going to stop for a second and let you think about it.

You got it! It is the fact that *the wheel wraps around at every turn*. Let’s try and make this more quantitative with a toy model. For one thing we shall assume that the wheel has 10 segments, labeled with the integers from 0 to 9, that is, we are generating one decimal digit. If we say that ω_0 is the initial angular velocity that we spin the thing with, and α the (modulo of) the angular deceleration, which for simplicity we shall assume constant in time, it is easy to show that the wheel stops when the angle θ has changed by

$$\theta_0 = \frac{\omega_0^2}{2\alpha}.$$

If you have an uncertainty σ_ω on ω_0 (which you always do), you will have an uncertainty of the final angular coordinate of

$$\frac{\sigma_\theta}{\theta_0} = 2 \frac{\sigma_\omega}{\omega_0} \quad \text{or} \quad \sigma_\theta = \frac{2\theta_0}{\omega_0} \sigma_\omega = \frac{\omega_0}{\alpha} \sigma_\omega$$

WHEEL OF FORTUNE

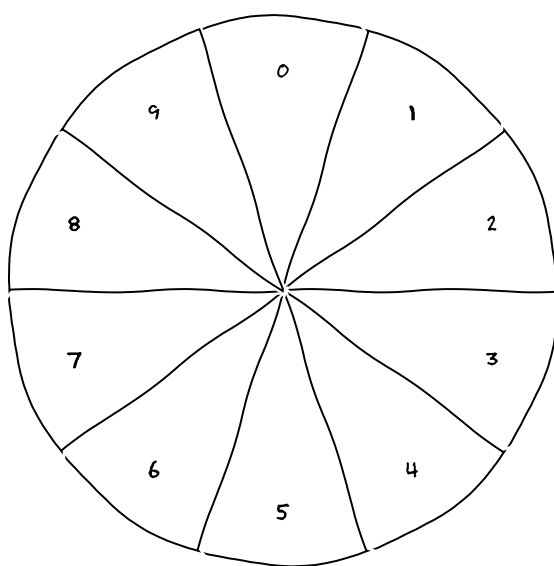


Figure 2.1: A wheel with 10 sectors.

That is: if you have a 1% uncertainty on ω_0 , you end up with a 2% uncertainty on the final value of θ . And this is where things get interesting. If θ was measuring a distance in a given direction and we were talking about a bocce-ball court (or a curling court, for what it's worth) our model would describe a very poor device for generating random sequences: a good player can train to throw a ball (or a rock) very consistently (i.e., with a small error on the initial velocity) which means we can reliably predict where the ball itself will end up. Isn't that the whole point of bocce, by the way?

But if we now go back to our wheel, where θ wraps up at each turn, *and we assume that ω_0 is large enough that the things does many turns*, the situation is completely different. You might have noticed that the absolute error on θ_0 is proportional to ω_0 and, for any value of α and σ_ω there will be a value of ω_0 that is large enough for the error on θ to be

$$\sigma_\theta \geq 2\pi.$$

When that happens, we have lost any capability to predict where the wheel will land. We might be able to predict roughly *how many turns* the wheel will do, but not the actual outcome. (In practice, for this to work, you might have to fine-tune α based on some reasonable assumptions for ω_0 and σ_ω , but this is a technical detail.)

```
import numpy as np

def spin_wheel(omega0, alpha = 0.1, num_segments = 10):
    """Spin our toy wheel of fortune!
    """
    # Calculate the overall angular advance until the wheel stops...
    theta = omega0**2. / 2. / alpha
    # ... wrap around at every turn, i.e., take the modulo 2 pi...
    theta = theta % (2. * np.pi)
    # ... and, finally, calculate the outcome by dividing by the angular with
    # of a sector and taking the integer part.
    return int(theta / (2. * np.pi / num_segments))

# Spin the wheel three times, changing the initial angular velocity by a small
# amount (about 1%), and see that the output is basically random :-)
print(spin_wheel(10.1))
print(spin_wheel(10.2))
print(spin_wheel(10.3))
```

1
7
4

If you got this far, you might be wondering: yeah, this is fun, but how about all this hype on the wheel of fortune? Weren't we supposed to talk about random numbers? Well, as it turns out the fortune wheel is a surprisingly good mechanical analogy to how a typical pseudo-random (ops: did I just say "pseudo"?) number generator works. The mathematical equivalent of the concept of "wrapping around at every turn" is the *modulo* operation in ordinary integer arithmetic and it is interesting to note that, as we shall see in a moment, the modulo is a fundamental ingredient in all the generators that are relevant for everyday use. We shall briefly come back to modular arithmetic in section Section 2.4 but, before we do that, there's a few thing we should sort out. In the meantime, L'Ecuyer (2017) provides an engaging and accessible account of the fascinating history of random number generators.

2.2 Random (and pseudo-random) sequences

While *random numbers* is an expression that you often find in the literature, you might have noticed that we prefer using *random sequence*. The reason is quite simple: there is no such a thing as a random number. "Here is a 5—is that random?" Clearly such a question makes little or no sense. What makes sense, instead, is the concept of a random sequence. That one we like.

So let's say we have a sequence X_n of integers with $0 \leq X_n < m$. (In our toy wheel-of-fortune setup, e.g., we had $m = 10$, but we anticipate that, for all practical purposes, we shall be interested in much larger values for m .) To be concrete, let's say that we spin our wheel of fortune and get

$$X_n = 2, 5, 1, 3, 9, \dots$$

Can we confidently say that this is a random sequence? As it turns out, this is a *very* hard question that has no good answer. We shall briefly come back to the issue in Section 2.8, but we anticipate that, while there are obvious properties that we would like from a truly random sequence (such as the fact that all the possible elements appear with the same frequency, within the expected statistical fluctuations), there is no easy direct *proof* that a given sequence is random. Usually people apply an entire battery of standard tests and declare that a given generator is random if the sequences it produces pass all the tests.

It is also worth mentioning that, while the sequence

$$X_n = 0, 1, 2, 3, 4, 5, \dots$$

definitely does not appear to be random (although all the digits are equi-distributed) in a truly random setting it is just as likely to appear as any other—including the first one we have shown. You get it: assessing randomness is difficult.

Let us turn for a second to a slightly different question: how could I conceivably generate a random sequence? Well, arguably I could use a truly random natural process. Although, when you think really hard about this, it is far from trivial to devise a truly random process,

True Random Number Generators (TRNGs) do exist, and serve very precise purposes. Have a look at www.random.org for a good example, and a good source of information on the topic. What we are mainly concerned here, though, is how to programmatically generate deterministic sequences by means of arithmetical methods that behave *as if they were random*. Such generators are called Pseudo-Random Number Generators (PRNGs) and are the main topic of this chapter.

The distinction between TRNG and PRNG is a fundamental one, and its importance can hardly be overstated. The former are by definition aperiodic and non deterministic (i.e., you cannot repeat the same sequence twice) and tend not to be very efficient. Yet, if you want to run a lottery, you should probably have a look at this article about the [the Michael Larsen incident](#) (amusing reading, I promise) before you consider opting away from a TRNG.

PRNGs, on the other hand, are typically *very* efficient and have the advantage that you can regenerate a given sequence as many times as you want. This is definitely your first choice for a Physics simulation—really, it is a no-brainer.

2.3 Tables

There has been a time when tables (and even huge ones) of (true) random digits were actually a thing. The most famous instance of such endeavors is probably the book [A Million Random Digits with 100,000 Normal Deviates](#) published in 1955 (and re-issued in 2001) by the RAND corporation. If you have 70 euros to spare you can buy one of the last paper copies available online. (The reviews of the book are quite funny, too.)

Nuisance to use. Memory was precious in the old days. Both motivated the development of alternative means for producing random numbers.

2.4 Digression: modular arithmetic

2.5 Pseudo-random sequences via recurrence

If you happen to be under impression that pseudo-random number generation is a solved problem, you couldn't be more wrong. For many years linear congruential generators (LCG) have been the de-facto standard. In 1997 Matsumoto and Nishimura (1998) caught the entire World by surprise when they published the Mersenne Twister (MT) algorithm: with a stunning period of $2^{19937} - 1$ and many other desirable properties it immediately underwent widespread adoption, and still is the default engine in many programming languages (including Python) and statistical packages. It took several years for some definite shortcomings and limitations now widely recognized to emerge. In 2014 O'Neill (2014) published the idea of a family of permuted congruential generators (PCG), and this is a notable case of an idea that has never

been published in a formal scientific paper, has been harshly criticized in a number of blogs and other informal forums on the web (see, e.g. [here](#)), and yet had real-world impact, as PCG is the default random generator in numpy (and the [related discussion](#) is a good testament as to how difficult is agreeing on what *the* best PRNG is).

The simplest way to generate a pseudo-random sequence is by a recurrence relation of the type

$$X_{n+1} = f(X_n) \bmod m \tag{2.1}$$

where X_{n+1} and X_n are non-negative integers, m is called *modulo*, and $f(x) : \mathbb{N} \rightarrow \mathbb{N}$ is a suitable function. (More on this later.) The idea is that you start from a *seed* $0 \leq X_0 < m$ and let all the other numbers flow from Equation 2.1.

Although more complex schemes can (and have been) devised, this very basic setting is a good one to discuss some of the general properties that are common to all pseudo-random sequences. The most obvious one is perhaps the fact that all the numbers in the sequence satisfy $0 \leq X_n < m$, which is to say that at most m distinct values can be produced. It follows that, sooner or later, we are bound to stumble across a number that we have already visited, and from that point on we just go on in circle. We can phrase this very thing in a slightly different way by saying that, ultimately, *any pseudo-random sequence is periodic*. In our particular case the period p satisfies $p \leq m$. This is not necessarily an issue if p is large enough, but definitely important to keep in mind.

At the very basic level, some of the good properties that a PRNG should have include

- a long period;
- high speed (i.e., $f(x)$ should be easy to calculate on a computer);
- a small memory footprint;
- good statistical properties (for one thing we will want that any reasonably long subsequence is equi-distributed in the $[0, m]$ interval, but as we shall see in a moment there are many other desirable properties).

We emphasize that, while having a long period is important, it is not by any means the *only* important thing. The sequence

$$X_{n+1} = X_n + 1 \bmod m \quad \text{e.g.} \quad 0, 1, 2 \dots m$$

provides the maximum period m , but it is obviously not random.

Before we move on, you feel be uncomfortable with the fact that we are dealing with integers. Most of our simulations live in the realm of floating-point numbers, and when do we get to that? Well, you just do

$$u_n = X_n / m$$

and here is you sequence of integers turned into a sequence of floating-point numbers between 0 and 1. (It goes without saying, these are by all means plain, rational numbers, but this is as far as floating-point arithmetics goes on a digital computer, and we already knew that.) Appendix

??? provides a cursory summary of floating point arithmetics; in the next chapter we shall discuss how we turn an equi-distributed sequence in the $[0, 1]$ interval into more interesting things.

2.6 Middle square

Devised by John Von Neumann at the turn of the 1940s, the middle-square (MS) method is often regarded as the oldest practical method to produce pseudo-random sequences, as well as an archetypal examples of all the idiosyncrasies that a good generator should stay away from. This [blog post](#) is a *beautiful* reading on the matter.

The underlying algorithm that the name *middle square* is meant to convey is the following: given a seed X_0 with n digits, take its square X_0^2 (this will have at most $2n$ digits, and you can actually make it a $2n$ -digits number by padding with zeros on the left side) and select the n digits in the middle of its numeral representation; this is the next number in the sequence. That is, assuming that we are working with four decimal digits, if we start with 3333, we square it to get 11108889 and the next number in the sequence is 1088; then comes 1837. And so on and so forth.

Quick digression. If you paid attention you will notice that what we are doing is mixing numbers and numerals: calculating the square has a meaning that is independent from the base we are using, but *taking the n mid digits* is something that operates on the string representation of the number, not the number itself. That is, using decimal or binary digits are two different things in this context. Going back to our previous example, decimal 3333 translates to 0b110100000101 (12 binary digits), 11108889 to 0b101010011000001000011001 (24 binary digits), and if we get rid of six digits on both sides we get 0b011000001000, i.e., decimal 1544. The two things do not roundtrip. Not a big deal, but definitely something to be aware of. You will also notice that for the very concept of *selecting the n digits in the middle* not to be ill-defined, n must be even to start with.

Back on track. How do I translate *selecting the n digits in the middle* in a way that can be expressed in a programming language? Well, you *could* convert the number into a string and operate on that, but at a second glance it is easy to see that you can actually do it with a combination of the modulo operator and the integer division. In Python the thing might read vaguely like:

```
def next_middle_square(x):
    """Calculate the next element in a middle square sequence (4 decimal digits).
    """
    return (x ** 2 // 100) % 10000

value = 3333
# The number following 3333 should be 1088.
```

```

value = next_middle_square(value)
print(value)
# The number following 1088 should be 1837.
value = next_middle_square(value)
print(value)

```

```

1088
1837

```

All right. What can we say on the properties of the sequence? Well, for one thing, it fits in our definition given by Equation 2.1 as we can write the recurrence relation as

$$X_{n+1} = \left\lfloor \frac{X_n}{b^{\frac{e}{2}}} \right\rfloor \bmod b^e$$

where the brackets indicate the integer division ignoring the remainder, b is the base (e.g., 10, or 2) in which we operate, and e is the number of digits we are targeting. What can we say about the period we are expecting? Does it depend on the seed? And how about the other statistical properties? This is where things get a little bit difficult, as this is a hard problem to attack mathematically. (And, admittedly, not a very interesting one from any practical standpoint.) But since we have computers, it is very easy to attack it with brute force, and Figure 2.2 shows all the possible sequences that we might ever get with two decimal digits.

A few directions to read the graph. The numbers surrounded by a circle (e.g., 94) are all of those that you only visit if you use them as a seed to start with—they have no other input leading into them. There are exactly four numbers (with two decimal digits) that are transformed into themselves by the middle-square algorithm, and they are: 0, 10, 50 and 60 (make sure you can verify that with pencil and paper); for obvious reasons, they are called *fixed points*, and they are surrounded with a square in the graph. As soon as one stumbles across a fixed point, the sequence is stuck into an infinite cycle of length one. Not good. 50 is doubly peculiar, in that it is both a fixed point, and one that has no input. You only visit it if you use it as a seed, and if you do that you never see anything different; a nice example of a cute little sequence of length 1. 57 and 24 are also peculiar in that they are transformed into each other, and make it for the only non-trivial cycle (of length 2, in this case) that one encounters with two decimal digits. Although the theoretical maximum period in this case is 100, the two longest sequences across all possible seeds are the ones going from either 42 or 69 to 0, and they both have exactly 15 elements. All other sequences are shorter than that, and most of them *much* shorter. Disappointing.

Does this mean that all is lost, or is it rather than two digits are too few, and the thing is well behaved with longer number? How about middle-square sequences with 12 decimal digits? Are those any good? I am afraid the answer is no, unfortunately. No matter how many digits you use, the middle square method tends to be trapped in fixed points or relatively short cycles.

The median sequence length scales as the square root of the modulo. With 12 digits you have a theoretical maximum period of 10^{12} and yet the typical sequence has a period that is a million times shorter. Not exactly an example of efficiency.

2.7 Linear congruential generators

Linear congruential generators (LCG) are among the most studied and well understood PRNG available, and have been widely used since the early 1950s. First discussed in a [paper](#) by D. H. Lehmer, they take the general form

$$X_{n+1} = (aX_n + c) \bmod m \quad (2.2)$$

where a is called the *multiplier*, and c the increment.

A cursory look at the following LCG for two decimal digits clearly show that, unless we devise a good strategy for choosing the underlying constants, a linear congruential generator is not necessarily better than the middle-square—which we said it was terrible.

```
m = 100
a = 40
c = 8
seed = 33
sequence = []

value = seed
while value not in sequence:
    sequence.append(value)
    value = (a * value + c) % m

print(f"Sequence length: {len(sequence)} {sequence}")
```

Sequence length: 2 [33, 28]

(What is happening, here, is that the second value, 28, is transformed into itself, and we are stuck with a sequence of length 2. Disappointing.)

The good news is that we have a good amount of well understood theory that goes with this class of generators. More specifically, the Hull–Dobell theorem states that all we have to require to guaranteed the maximum period m is that

- m and c are coprime (i.e., their greatest common divisor is 1);
- $a - 1$ is divisible by all prime factors of m ;

chosen the multiplier to be a power of the base we are working with, but this is becoming too technical.) The point is: if you are careful, things do get better with larger number, but by all means *having a long period* is not the end of the story. Which brings us to the next question: now that we start dealing with candidate sequences that are not obviously flawed by just looking at them with the naked eye, how do we tell if any particular one is really good?

2.8 Tests for randomness

When discussing the tests for randomness, the specifics depend on what we are testing, exactly, e.g., X_n as integers vs. u_n as floating-point numbers in the $[0, 1]$ interval.

If we concatenate all the subsequent X_n and treat our sequence as an un-interrupted stream of decimal digits, a few obvious desirable things immediately come to mind. For one thing we would like each digit to appear with the same probability $1/10$ in the sequence. (We note that this is hardly enough for the thing to appear as random, as a simple counter $X_n = 0, 1, 2, 3 \dots$ trivially satisfy this.) We may ask that *pairs* of subsequent identical digits, e.g. 2, 2, or 7, 7, appear with the expected probability $1/100$, and this can be easily generalized to triplets, quadruplets or, really, arbitrary high multiplicity.

In reality, there is a nearly unlimited number of properties that one can list and, in absence of a single test that can qualify a given sequence as good enough in terms of randomness, the best that one can do is feed their generator to a full battery of tests, and hope that it passes them all.

There is one, special test, though, that is worth at least a cursory discussion as it offers interesting insight into the very basic process of generating random numbers by arithmetical means. It is called the *spectral test*, and is based on a nice little paper by Marsaglia (1968) titled *Random numbers fall mainly in the planes*. Let's go back for a second to the simple LCG we have discussed in the previous section—more specifically to the maximum-period sequence we have obtained in the last snippet by carefully choosing the multiplier and the increment. Figure 2.3 shows how pairs of successive point u_n, u_{n+1} are distributed in the cartesian plane. (Make sure you grasp the meaning of the plot. This is in u_n space, i.e., we divided the X_n by the modulus, and the small text label indicate the index in the sequence at which each number occurs: the first point is 0.33, 0.04, after which we jump to the upper-left corner 0.04, 0.95, then to the opposite corner 0.95, 0.06 and so on and so forth.)

Well, it is fair to say that this is looking anything but random. At a second though, maybe this is not really unexpected. We are pretending to generate random real numbers, but under the hood all we are doing is truncating stuff in the $[0, 1]$ interval to an accuracy of $1/m$ and therefore we are bound to be unable to properly fill the plane in this representation. With a ridiculously small modulo such as 100 things look understandingly horrible, and one has hope that with a much larger m things are better—and indeed they are—but nonetheless this is a feature that cannot be overcome and we need to keep it in mind. Marsaglia (1968) makes this

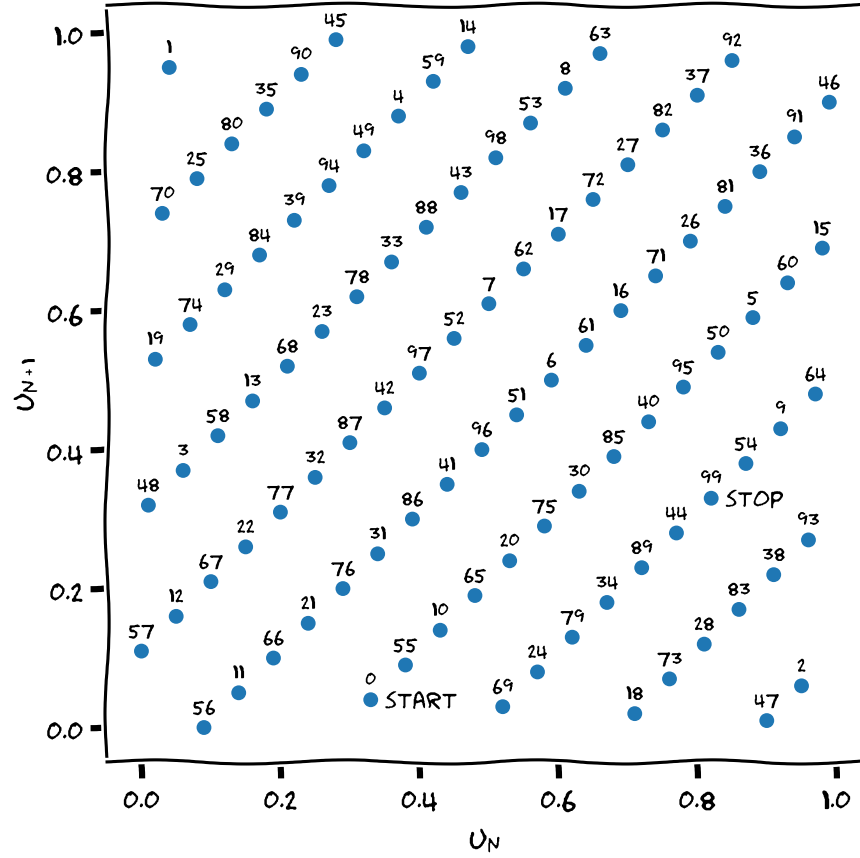


Figure 2.3: Representation of the u_n, u_{n+1} ordered pairs for the simple linear congruential generator with maximum period discussed in the previous section. The decimal integers have been converted into floating-point numbers in the $[0, 1]$ interval dividing by the modulus. The small numbers indicate the order of appearance in the sequence.

all quantitative for LCG, showing that the output of a LCG with modulo m lie on at most 14 straight lines. In Figure 2.3 we do see 10 lines oriented at 45° , so our generator is pretty much as good as it gets with two decimal digits.

The thing can in fact be generalized from 2 to n dimensions (e.g., for $n = 3$ we are dealing with triplets u_n, u_{n+1}, u_{n+2} in the volume, for $n = 4$ with quadruplets, and so on and so forth), and the basic content of Marsaglia's theorem is that the output n -tuples from a LCG will lie on at most

$$(n!m)^{\frac{1}{n}}$$

hyperplanes. For $n = 2$ and $m = 2^{128}$, for instance, this gives about 2×10^{19} lines—something we hardly have to worry about. (And yet, if we are looking at $n = 10$ we are talking about *only* 32296 hyperplanes.)

2.8.1 The worst LCG ever

The spectral test brings us to another piece of random number folklore: RANDU, now universally dubbed as one of the most ill-conceived PRNGs to have ever seen widespread use. Originally introduced by IBM at the end of the 1950s, RANDU is defined as a (purely) linear congruential generator with modulo $m = 2^{31}$, multiplier $a = 65539$ and increment $c = 0$

$$X_{n+1} = 65539 X_n \bmod 2^{31}$$

This was done purely on efficiency consideration, with no regard to the statistical quality of the output: since $65539 = 2^{16} + 3$, the recurrence relation can be rewritten as

$$X_{n+1} = ((X \ll 16) + X + (X \ll 1)) \bmod 2^{31}$$

which on a 32-bit binary computer only involves bit shifts, additions and a truncation (the final modulo), all of which could be done very efficiently in hardware.

Phrased in a slightly different way, RANDU was blazingly fast on a typical computer of the time. And yet it can be shown that it fails *spectacularly* the spectral test with $n \geq 3$, as the output triplets only sample 15 planes in the unit cube against a theoretical maximum of 2344.

2.9 Modern generators

Cover very quickly F2-linear pseudorandom number generators, and particularly MT and Xoshiro. Also PCG makes it for an interesting story.

Include a few words about crypto generators, and mention the relevant Python module.

2.9.1 Period: is that really important?

While quoting wall-clock time when discussing computer operations is always a little bit silly, nowadays most modern generators run at a speed of a few ns per call on a typical laptop. It would take a time equal to the current age of the universe to generate 10^{27} random numbers, if that was the only thing that we were doing. I'll give you 1000 parallel streams and make that 10^{30} . For comparison, $2^{128} \approx 3.4 \times 10^{38}$ (that is, almost a billion larger), $2^{256} \approx 1.2 \times 10^{77}$ (which is only three orders of magnitude short of the number of protons in the universe), and $2^{19937} \approx 10^{6000}$. (You will have to resort to your high-school knowledge of the logarithms to calculate it, because if you try in a Python prompt you will get an overflow error.)

Now, an obvious question arise: do we *really* need such a ridiculous period from a PRNG?

On the flip side of the story, $35! \approx 10^{40}$, so be aware that, if you are into combinatorial problems, with a period as short as 2^{128} you will not be able to generate all the possible random permutations of a set of 35 elements. Somebody (see, e.g., [this paper]) think that is actually a big deal.

2.10 What am I using under the hood?

It is fair to say that LCG are pretty much dead, nowadays. Many popular programming languages and data-analysis program, while offering many options, default to the MT19937 version of the Mersenne Twister algorithm, although its limitations are widely recognized. The ROOT documentation for the [TRandom](#) class reads verbatim “*this is a very BAD Generator, not to be used*” referring to MT19937, and yet that is used to instantiate the global pointer to the ROOT generator. So goes life. Meanwhile the PCG and Xoshiro families of PRNG are gaining traction. Here is a largely incomplete survey of the situation as of 2026.

Table 2.1: Table of the default random generators in some widely used programming languages and statistical frameworks.

Language/framework	Default generator	Period
Python	MT19937, Matsumoto and Nishimura (1998)	$2^{19937} - 1$
NumPy	PCG XSL RR 128/64, O'Neill (2014)	2^{128}
ROOT	MT19937, Matsumoto and Nishimura (1998)	$2^{19937} - 1$
Julia	Xoshiro256++, Blackman and Vigna (2021)	$2^{256} - 1$
R	MT19937, Matsumoto and Nishimura (1998)	$2^{19937-1}$

Language/framework	Default generator	Period
MATLAB	MT19937, Matsumoto and Nishimura (1998)	$2^{199371} - 1$

2.11 Exercises

Exercise 2.1 (Middle square). Characterize the middle-square method for producing random numbers with n decimal and/or binary digits at least up to $n = 8$, and possibly more. In each case identify all the fixed points and possible loops, and calculate the median period across all possible seeds. Verify that the latter scales as $\sqrt{n}$.

Exercise 2.2 (RANDU). Devise a simple toy problem where the RANDU PRNG clearly fails beyond the expected numerical precision, and verify that a modern generator (e.g., MT or PCG) fixes the problem.

3 Monte Carlo integration

Let us assume that we want to calculate the definite integral

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

Before you ask: there is nothing really special about this function, except the fact that it is smooth, and its primitive has a simple, analytical form—and therefore our integral is readily evaluated:

$$I = \arctan(1) = \frac{\pi}{4}.$$

How can we put a generator of equi-distributed random numbers to use to estimate this integral?

3.1 The hit-or-miss method

One of the features of the integration region we have chosen for our example is that it can be enclosed in the unitary square—so why don't we go ahead and throw a whole bunch of random points in this square and count how many of them fall below the curve? The fraction of the latter should asymptotically tend to I .

```
import numpy as np
from matplotlib import pyplot as plt

rng = np.random.default_rng(313)
n = 100

def f(x):
    return 1. / (1. + x**2.)

# Evaluate the function on a grid.
x = np.linspace(0., 1., 100)
y = f(x)

# Throw some random numbers in the unitary square.
```

```

px = rng.random(size=n)
py = rng.random(size=n)

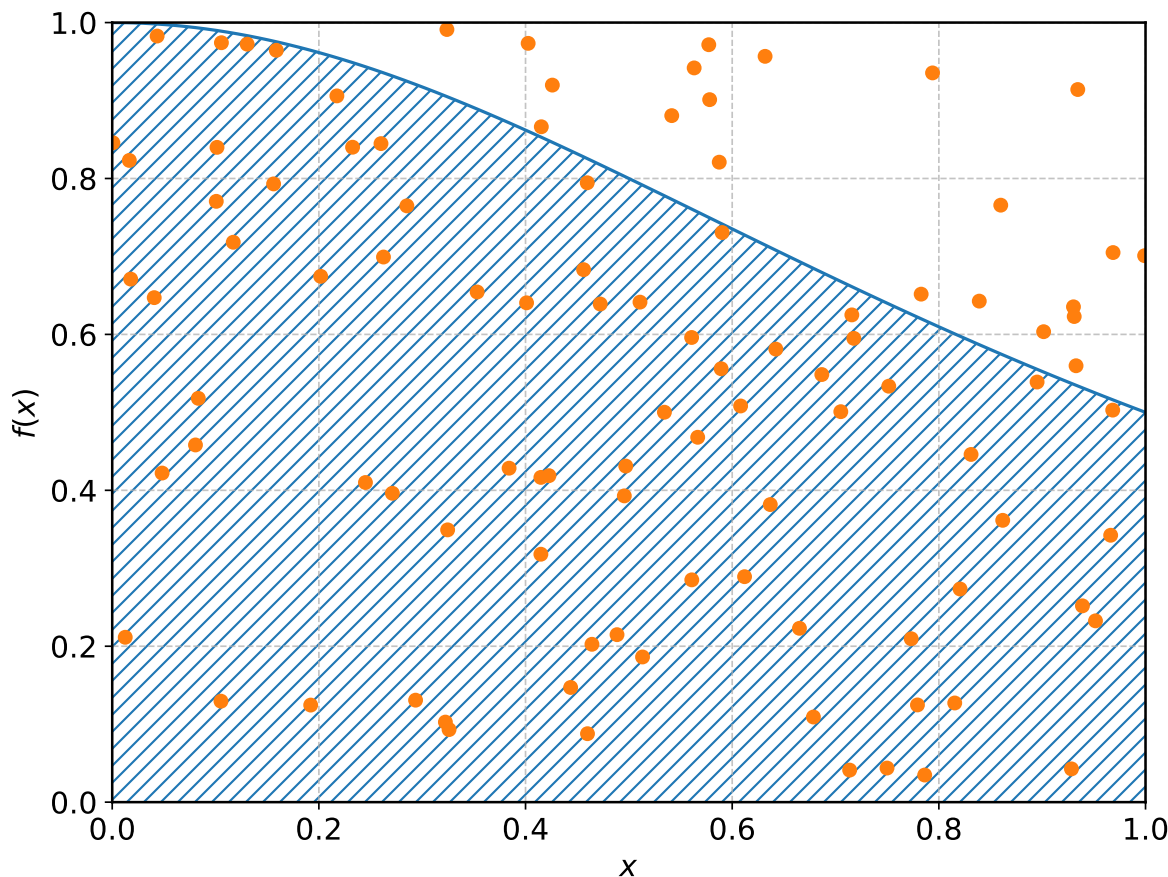
# Count how many of them fall below the curve.
k = (py < f(px)).sum()
I = k / n

plt.plot(x, y)
plt.gca().fill_between(x, 0, y, facecolor="none", edgecolor="tab:blue",
    hatch="///", linewidth=0)
plt.plot(px, py, "o")
plt.axis([0., 1., 0., 1.])
plt.xlabel("$x$")
plt.ylabel("$f(x)$")

print(f"{k}/{n} = {I} (expected {np.pi / 4.})")

```

79/100 = 0.79 (expected 0.7853981633974483)



This works as advertised, and generally goes under the name of *hit-or-miss* Monte Carlo integration. (Spoiler alert: it is unlikely that you will use it extensively, but the technique is germane to the one bearing the same name that is used for *sampling* distribution, so it is instructive to take a look.)

Let us look at this technique in some more details. Let D be our domain of integration (with area I) and R (typically a rectangle, with area A_R) the region we are throwing our random points in: assuming that we throw n , uniformly distributed random points in R , and k of them end up inside D the quantity

$$\hat{I} = \frac{k}{n} A_R \quad (3.1)$$

is an unbiased estimator for A_D , in the sense that

$$E[k] = n \frac{I}{A_R} \quad \text{and therefore} \quad E[\hat{I}] = I.$$

How do we estimate the uncertainty associated to our estimator? Well, k is a binomial random

variable with $p = \frac{I}{A_R}$ and therefore

$$\text{Var}(\hat{I}) = \frac{A_R^2}{n^2} \text{Var}(k) = \frac{A_R^2}{n^2} np(1-p) = \frac{I(A_R - I)}{n} \quad \text{and} \quad \sigma_{\hat{I}} = \sqrt{\frac{I(A_R - I)}{n}}$$

Let's look at this expression for a second. First thing first, we notice that this estimates includes the *unknown* integral I —this is sometimes called an *a priori* error analysis. This will suffice for our scope: we can always plug in our estimate of I and get an approximate expression that only depends on measured quantities:

$$\sigma_{\hat{I}} \approx \sqrt{\frac{\hat{I}(A_R - \hat{I})}{n}}. \quad (3.2)$$

This allows to derive a number of interesting facts. The uncertainty on the integral asymptotically tends to zero for large n with a convergence rate of

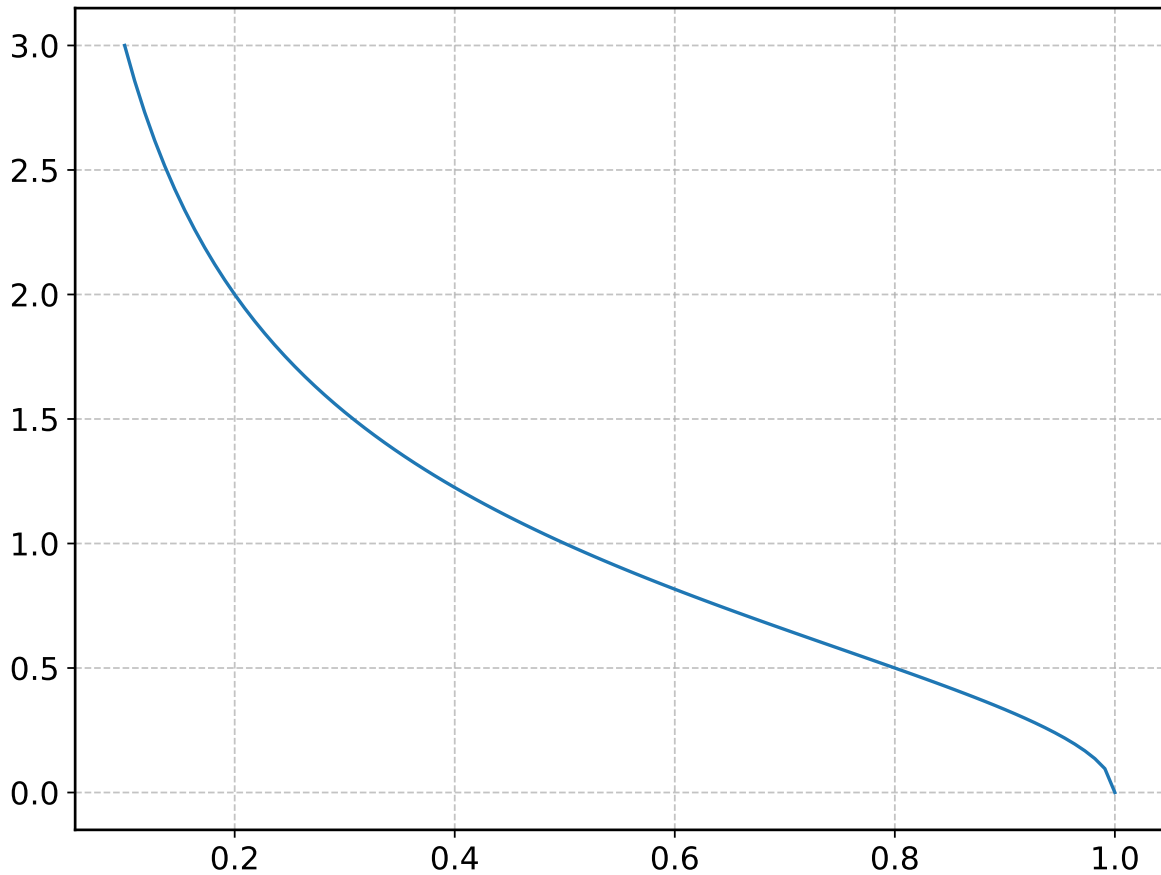
$$\sigma_{\hat{I}} \propto n^{-\frac{1}{2}},$$

which, as we shall see, is a basic property of Monte Carlo integration, independently of the number of dimensions of our problem. The multiplicative constant for the convergence depends on the actual values of I and A_R , and it is more conveniently expressed in relative terms:

$$\frac{\sigma_{\hat{I}}}{\hat{I}} \approx \sqrt{\frac{(1 - \varepsilon)}{\varepsilon}} n^{-\frac{1}{2}} \quad \text{where} \quad \varepsilon = \frac{\hat{I}}{A_R}.$$

Interesting: the constant only depends on the ratio of the original integral to the area of the bounding region. How? Easily shown!

```
epsilon = np.linspace(0.1, 1., 100)
plt.plot(epsilon, np.sqrt((1. - epsilon) / epsilon))
```



That is: for best performance you want the bonding region to be as close as possible to your integration domain—when the two are the same the uncertainty is *exactly* zero, as you know A_R and, therefore, you know the answer. (Keep in mind this is not always easy to achieve, as it's easy to throw random points uniformly distributed in a rectangle, not so much in an arbitrarily complex region.) For $\varepsilon = 1/2$, which you can take as a typical example of an integration setup that is not ill-posed, the constant is equal to unity. Easy to remember.

And here is a fully fledged, vectorized Python function that will calculate the definite integral (and associated uncertainty) of a generic one dimensional function by means of a hit-or-miss Monte Carlo technique.

```
import numpy as np

rng = np.random.default_rng(313)

def hit_or_miss_integral(func, a, b, ymin, ymax, n):
```

```
"""Calculate the definite integral of a function by means of the hit-or-miss
Monte Carlo method.
```

```
Note this requires  $y_{\min} \leq f(x) \leq y_{\max}$  on the  $[a, b]$  interval.
```

```
Arguments
```

```
-----
```

```
func : callable
```

```
    The integrand  $f(x)$ . It must accept a NumPy array as the only argument
    and return an array of function values with the same shape.
```

```
a : float
```

```
    Lower bound of the integration interval.
```

```
b : float
```

```
    Upper bound of the integration interval (must be  $> a$ ).
```

```
ymin : float
```

```
    Lower vertical bound of the integration interval.
```

```
ymax : float
```

```
    Upper vertical bound of the integration interval.
```

```
n : int
```

```
    Number of points for the Monte Carlo sampling.
```

```
Returns
```

```
-----
```

```
I : float
```

```
    Estimate of the integral.
```

```
sigma_I : float
```

```
    Estimated one-standard-deviation statistical uncertainty of
    the integral estimate.
```

```
"""
```

```
A_R = (b - a) * (ymax - ymin)
```

```
x = rng.uniform(a, b, size=n)
```

```
y = rng.uniform(ymin, ymax, size=n)
```

```
I = np.mean(y < func(x))
```

```
sigma_I = np.sqrt(I * (A_R - I) / n)
```

```
return I + (b - a) * ymin, sigma_I
```

```
def f(x):
    return 1. / (1. + x**2.)

I, sigma_I = hit_or_miss_integral(f, 0., 1., 0., 1., 100000)
target = np.pi / 4.
rel_error = abs(I - target) / target
print(f"Integral = {I:.4f} +- {sigma_I:.4f} (relative error: {rel_error:.1e})")
```

Integral = 0.7870 +- 0.0013 (relative error: 2.0e-03)

3.2 The sample-mean method

We can recast the problem of calculating a definite integral

$$I = \int_a^b g(x)dx$$

in a completely different fashion. If we take x to be a continuous random variable uniformly distributed between a and b , i.e.,

$$p(x) = \frac{1}{(b-a)} \quad \text{for } a \leq x \leq b$$

we might take advantage of the fact that the probability density function is a simple constant and rewrite our integral as an *expectation value* for g :

$$I = (b-a) \int_a^b g(x)p(x)dx = (b-a)E[g(x)].$$

At a second look, this is hardly surprising: the integral of a function over an interval is equal to the width of the interval, times the mean value of the function—who would have guessed?

This, in turn, is opening up an entirely new avenue for us, as we can approximate the *true* average with the *sample* average over a finite number of random variates x_i equi-distributed between 0 and 1 (i.e., exactly the thing that we just became good at simulating)

$$\hat{I} \approx \frac{(b-a)}{n} \sum_{i=1}^n g(x_i). \quad (3.3)$$

It is easy to see that this, too, is an unbiased estimator of the original integral, in the same way the sample mean is an unbiased estimator of the mean of a distribution.

(Note we are not *sampling* the original function $g(x)$ —we are *applying* $g(x)$ to a sample or random deviates equi-distributed between 0 and 1.)

How do we estimate the uncertainty, this time around? Well, once again we have to calculate the variance of our estimator

$$\text{Var}(\hat{I}) = \frac{(b-a)^2}{n^2} \text{Var} \left(\sum_{i=1}^n g(x_i) \right) = \frac{(b-a)^2}{n^2} \sum_{i=1}^n \text{Var}(g(x_i)) = \frac{(b-a)^2}{n} \text{Var}(g(x))$$

Now, written in this form this is not particular nice looking, as the variance of our function contains both the original integral and the integral of $g^2(x)$, which is most likely even more difficult—you will see the expression made explicit as

$$\text{Var}(\hat{I}) = \frac{1}{n} \left[(b-a) \int_a^b g^2(x) dx - I^2 \right] = \frac{(b-a)}{n} \int_a^b \left(g(x) - \frac{I}{(b-a)} \right)^2 dx.$$

Sampling comes to rescue, again, as we can approximate the variance of the function as

$$\text{Var}(g(x)) \approx s^2 = \frac{1}{n-1} \sum_{i=1}^n (g(x_i) - m)^2 \quad \text{where} \quad m = \frac{1}{n} \sum_{i=1}^n g(x_i) = \frac{\hat{I}}{(b-a)}.$$

That is:

$$\sigma_I \approx (b-a) s n^{-\frac{1}{2}} \quad \text{or} \quad \frac{\sigma_I}{\hat{I}} \approx \frac{s}{m} n^{-\frac{1}{2}} \quad (3.4)$$

Here is a complete Python implementation.

```
import numpy as np

rng = np.random.default_rng(313)

def sample_mean_integral(func, a, b, n):
    """Calculate the definite integral of a function by means of the sample mean
    Monte Carlo method.

    Arguments
    -----
    func : callable
        The integrand f(x). It must accept a NumPy array as the only argument
        and return an array of function values with the same shape.

    a : float
        Lower bound of the integration interval.

    b : float
```

```

    Upper bound of the integration interval (must be > a).

n : int
    Number of points for the Monte Carlo sampling.

Returns
-----
I : float
    Estimate of the integral.

sigma_I : float
    Estimated one-standard-deviation statistical uncertainty of
    the integral estimate.
"""
x = rng.uniform(a, b, size=n)
I = np.mean(func(x))
sigma_I = (b - a) * np.std(func(x)) / np.sqrt(n)
return I, sigma_I

def f(x):
    return 1. / (1. + x**2.)

I, sigma_I = sample_mean_integral(f, 0., 1., 100000)
target = np.pi / 4.
rel_error = abs(I - target) / target
print(f"Integral = {I:.4f} +- {sigma_I:.4f} (relative error: {rel_error:.1e})")

```

```

Integral = 0.7851 +- 0.0005 (relative error: 3.7e-04)

```

3.3 Convergence

You might have noticed that, in our particular case, the standard error for the sample mean is smaller than that for the hit-or-miss integration, and so is the relative deviation of the central value from the known target.

The example function we have picked to illustrate the problem is easy enough that the standard error can be analytically calculated for both techniques

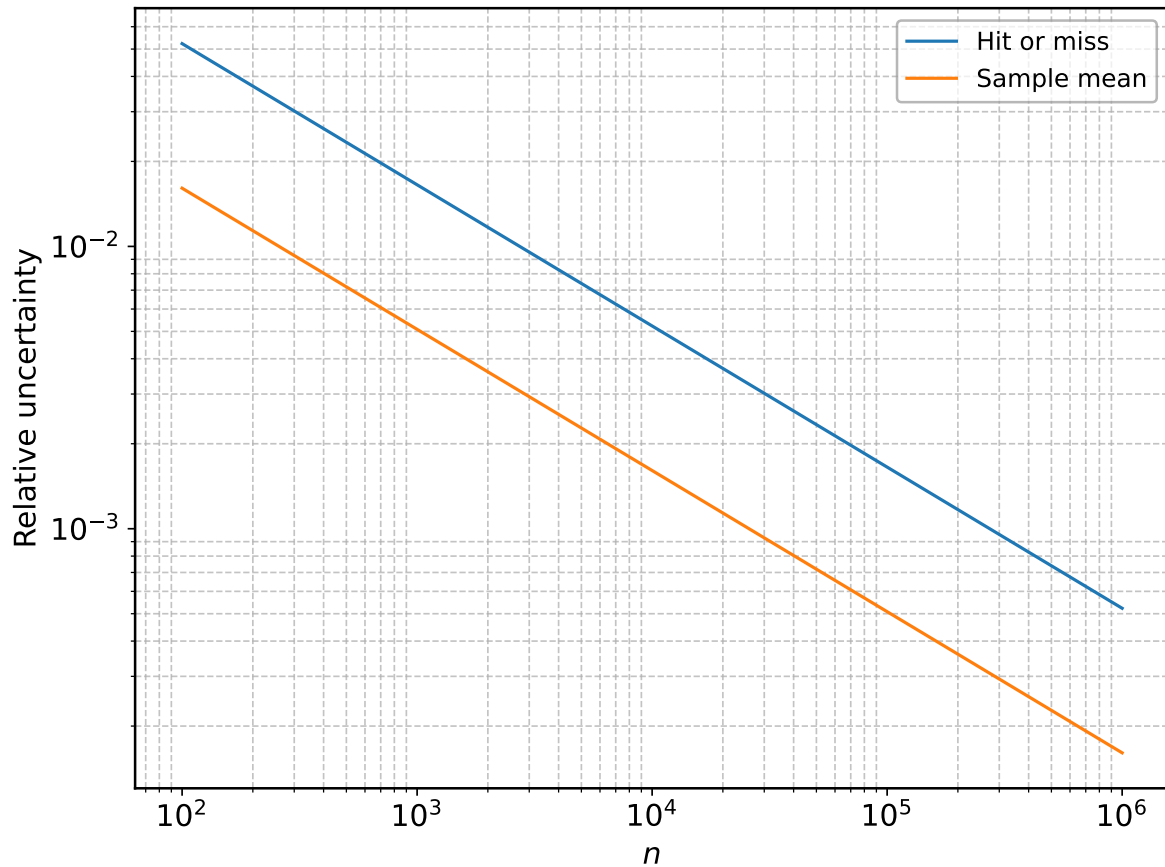
$$\frac{\sigma_I^{\text{HM}}}{\hat{I}} = \sqrt{\frac{4}{\pi} - 1} n^{-\frac{1}{2}} \quad \text{and} \quad \frac{\sigma_I^{\text{SM}}}{\hat{I}} = \frac{\sqrt{4 + 2\pi - \pi^2}}{4} n^{-\frac{1}{2}}.$$

Though the numerical factors are only valid for our particular toy problem, it is instructive to see how they compare to each other: the scaling is the same, but the prefactors are different.

```
import numpy as np
from matplotlib import pyplot as plt

n = np.geomspace(100, 1000000)
rel_err_hm = np.sqrt(4. / np.pi - 1.) * (n**-0.5)
rel_err_sm = np.sqrt(4. + 2. * np.pi - np.pi**2.) / 4 * (n**-0.5)

plt.loglog(n, rel_err_hm, label="Hit or miss")
plt.loglog(n, rel_err_sm, label="Sample mean")
plt.grid(which="both")
plt.xlabel("$n$")
plt.ylabel("Relative uncertainty")
plt.legend()
```



This is a general feature of Monte Carlo integration—the sample-mean method is more efficient than the hit-or-miss one: with half of the random numbers it provides a lower variance. Let us assume that

$$0 \leq g(x) \leq M.$$

We can write the difference between the two variances as

$$\begin{aligned} \text{Var}(\hat{I}^{\text{HM}}) - \text{Var}(\hat{I}^{\text{SM}}) &= \frac{1}{n}(b-a)MI - I^2 - \frac{1}{n} \left[(b-a) \int_a^b g^2(x)dx - I^2 \right] = \\ &= \frac{(b-a)}{n} \left[MI - \int_a^b g^2(x)dx \right] = \frac{(b-a)}{n} \int_a^b g(x)[M - g(x)]dx \geq 0, \end{aligned}$$

which is positive by construction. (The two are the same only if $g(x)$ is constant.)

We have a winner. At a second thought, though, the scaling of the relative errors is revealing for another, unrelated reason. How about using an old, plain, boring quadrature?

```
import time
from scipy.integrate import quad

def f(x):
    return 1. / (1. + x**2.)

t0 = time.time()
I, sigma_I = quad(f, 0, 1)
dt = time.time() - t0
print(f"Quadrature calculated in {dt:.2e} s...")

target = np.pi / 4.
rel_error = abs(I - target) / target
print(f"Integral = {I:.16f} +- {sigma_I:.16f} (relative error: {rel_error:.1e})")
```

```
Quadrature calculated in 6.41e-05 s...
Integral = 0.7853981633974484 +- 0.00000000000000087 (relative error: 1.4e-16)
```

Really? We got 15 significant digits in less than 100 μs ? In order to achieve that with the scaling of the sample-mean Monte Carlo integration we would need something like 10^{30} random numbers. Even under the generous assumption that you can throw pseudo-random numbers at 1 ns per call, this translates into 10^{21} s, or 10^{14} years. This is much larger than the age of the universe.

(And, in reality, there are other fundamental reasons why it would be hard to achieve this kind of precision even if we had the time.)

3.4 The curse of dimensionality

At this point you might have mixed feelings about our entire discussion on Monte Carlo integration. On one hand, we have a well defined techniques that works as advertised with basically no assumption. On the other hand, a classic quadrature is so spectacularly better for our toy problem, that we cannot help asking if Monte Carlo techniques are of any practical interest.

The point is: *Monte Carlo methods are extremely inefficient for one-dimensional integrals, and especially smooth ones.* Just to be clear: quadrature will not provide a relative accuracy at the level of the intrinsic capabilities of floating-point arithmetics for all the functions at hand. And you can find a counter-example where quadrature methods fail miserably and a Monte Carlo integration is comparatively better behaved (all you need is a wildly oscillating function), but you have to build that on purpose. One-dimensional integrals are definitely not where Monte Carlo methods shine.

More precisely: compared to the $n^{-\frac{1}{2}}$ of the typical Monte Carlo integration, any reasonable quadrature method will perform much better. Midpoint rectangle or trapezoidal integrals will converge as n^{-2} , where n is the number of intervals, provided that the second derivative of the function exists and is bounded; the Simpson rule provides a convergence rate of n^{-4} if the fourth derivative of the function is good enough. In general the convergence rate for a typical quadrature methods can be written as

$$\frac{\sigma_I}{\hat{I}} \propto n^{-p} \quad \text{where } p \geq 2. \quad (\text{In dimension 1.})$$

So, what are we talking about? Well, fortunately not all the problems are one-dimensional. And this is where Monte Carlo methods get right back into business. When you move from dimension 1 to dimension d in the context of a typical quadrature methods, that means that, in order to maintain the same level of accuracy, you have to use a grid with n points on all the d dimensions—that means an overall factor of n^d , which in turn implies that the convergence rate is now

$$\frac{\sigma_I}{\hat{I}} \propto n^{-\frac{p}{d}} \quad (\text{In dimension } d.)$$

For a Monte Carlo integration, nothing changes—the *convergence rate is independent from the dimensionality of the problem*. That means that when $d \geq 4$, Monte Carlo methods become to be competitive. For even higher dimensionality, and especially when the integrand is not smooth, they are the choice.

3.5 Low-discrepancy sequences

There is one simple thing about Equation 3.3 that is worth pointing out explicitly: if we take an ensemble of x_i evenly spaced on a grid, that is rigorously equivalent to the rectangle rule.

(Make sure you do picture this in your mind before you move on.)

Think about it for a second: did we really go into the trouble of all the first three chapters of these notes just to find out what we are really doing is the dumbest possible way to calculate an integral? And it's even worse than that: in our particular setting rectangle integration is generally *superior* to the Monte Carlo sample mean. (And vastly so, too.)

```
import numpy as np

rng = np.random.default_rng(313)

def f(x):
    return 1. / (1. + x**2.)

n = 1024

I1 = np.mean(f(rng.uniform(0., 1., size=n)))
I2 = np.mean(f(np.linspace(0., 1., n)))

target = np.pi / 4.
print(f"Monte Carlo: relative error = {abs(I1 - target) / target:.1e}")
print(f"Regular grid: relative error = {abs(I2 - target) / target:.1e}")
```

```
Monte Carlo: relative error = 8.3e-03
Regular grid: relative error = 4.4e-05
```

What the heck? At a second thought, though, this is not really surprising. When we use a regular grid, mileage may vary depending on how the grid points line up with the *features* (if any) of our function, but if we fix the number of points, we always get the same answer. If we start throwing random numbers, instead, we are adding additional variance, as the result is slightly different every time. A pure Monte Carlo method is bound to be worse in the sense of the variance.

i Note

Note that, by definition, the relative error of the Monte Carlo estimate with respect to the true value is only representative of this particular realization, and not an estimate of the uncertainty of the underlying method—change the seed and you will get a different answer. The regular grid, on the other hand, will give you the same estimate every time.

Monte Carlo methods have a distinct advantage on regular grids, though: you can continue enlarging the sample and refining your results without throwing away what you did so far. On

the other hand, if you want to make your regular grid one point larger, you have to recalculate the entire thing and start over.

Now you might ask yourself: I wonder if there is a sensible way to get the best of both worlds. Good news—the answer is yes, and it is called a *low-discrepancy sequence*, which lives in the realm of the so-called *Quasi Monte Carlo* (QMC) methods. Sobol sequences (Sobol' 1967; Joe and Kuo 2008) are one of the most widely know alternatives on the market and Scipy offer a fully-fledged implementation in the `qmc` submodule.

The easiest way to get acquainted with low-discrepancy sequence is maybe just look at how the compare with regular pseudo-random numbers.

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import qmc

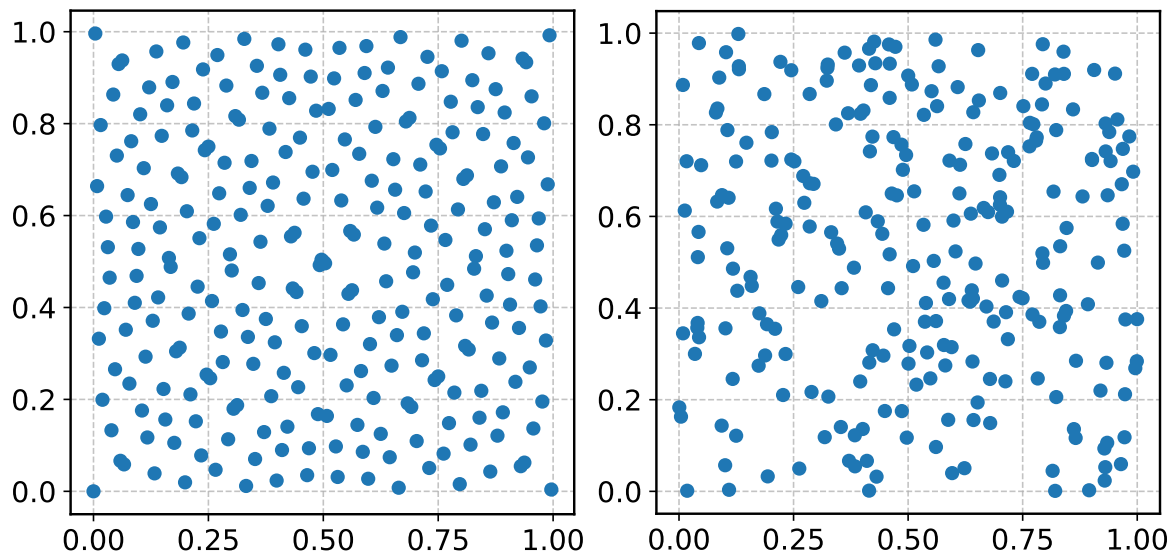
rng = np.random.default_rng(313)

m = 8
n = 2**m

fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True)

# Generate n points using a Sobol sequence in 2 dimensions.
sampler = qmc.Sobol(d=2, scramble=False)
x, y = sampler.random_base2(m).T
ax1.plot(x, y, "o")
ax1.set_aspect("equal")

# Generate plain Monte Carlo numbers in the unit square.
x = rng.uniform(size=n)
y = rng.uniform(size=n)
ax2.plot(x, y, "o")
ax2.set_aspect("equal")
```



Can you guess who is who? (Yes: left is a Sobol sequence, right is the usual PRNG.) A quick look at the two plots should make you jump on your chair and scream: *ah ah—now I can see where the variance reduction comes from!*

⚠ Warning

As most QMC constructions, Sobol sequences are designed to generate samples of special sizes—in this case a power of 2. Generating 100 numbers might get you a less accurate estimate for an integral than generating 64. Make sure you read the fine prints in the [Scipy documentation](#).

How do we get the machinery to work for our purposes? Easier done than said.

```
import numpy as np
from scipy.stats import qmc

def sobol_integral(func, a, b, m):
    """Calculate the definite integral of a function by means of the sample mean
    method using a one-dimensional Sobol sequence.

    Arguments
    -----
    func : callable
        The integrand f(x). It must accept a NumPy array as the only argument
        and return an array of function values with the same shape.
```

```

a : float
    Lower bound of the integration interval.

b : float
    Upper bound of the integration interval (must be > a).

m : int
    The power of 2 defining the number of points in the sequence (n = 2**m).

Returns
-----
I : float
    Estimate of the integral.
"""
sampler = qmc.Sobol(d=1, scramble=False)
x = sampler.random_base2(m)
x = qmc.scale(x, a, b)
return np.mean(func(x))

def f(x):
    return 1. / (1. + x**2.)

I = sobol_integral(f, 0., 1., m=10)

target = np.pi / 4.
print(f"Sobol sequence: relative error = {abs(I - target) / target:.1e}")

```

Sobol sequence: relative error = 3.1e-04

If you paid attention, you will have noticed that we are not returning an estimate for the uncertainty, here. This points out to a very fundamental fact: Sobol sequences are strictly deterministic—if you don't do anything specific they really behave as a regular grid.

```

from scipy.stats import qmc

m = 3

sampler1 = qmc.Sobol(d=1, scramble=False)
sampler2 = qmc.Sobol(d=1, scramble=False)

print(sampler1.random_base2(m).T)
print(sampler2.random_base2(m).T)

```

```
[[0.    0.5   0.75  0.25  0.375 0.875 0.625 0.125]]
[[0.    0.5   0.75  0.25  0.375 0.875 0.625 0.125]]
```

Since the values in the sequence are not statistically independent (nor they are designed to look as if they were), you cannot use your statistics textbook to attach an uncertainty to the estimate starting from the first principles, as we did with the Monte Carlo methods.

And this is where the `scramble` arguments comes into play. If you set it to `True` (which, by the way, is its default value) two things happen:

1. the resulting points are randomized;
2. you get different values every time.

```
from scipy.stats import qmc

m = 3

sampler1 = qmc.Sobol(d=1)
sampler2 = qmc.Sobol(d=1)

print(sampler1.random_base2(m).T)
print(sampler2.random_base2(m).T)
```

```
[[0.09037848 0.71159615 0.80082258 0.42203643 0.26132263 0.89035292
 0.59679725 0.22582372]]
[[0.94870749 0.25122471 0.22366582 0.54608277 0.66192123 0.10933817
 0.38647442 0.81399164]]
```

And yet, all of this happens preserving the low-discrepancy structure of the sequence, so you get all the good properties the thing was defined for, but now you can take the variation across a number of independently scrambled sequences to estimate the uncertainty in you calculation.

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.stats import qmc

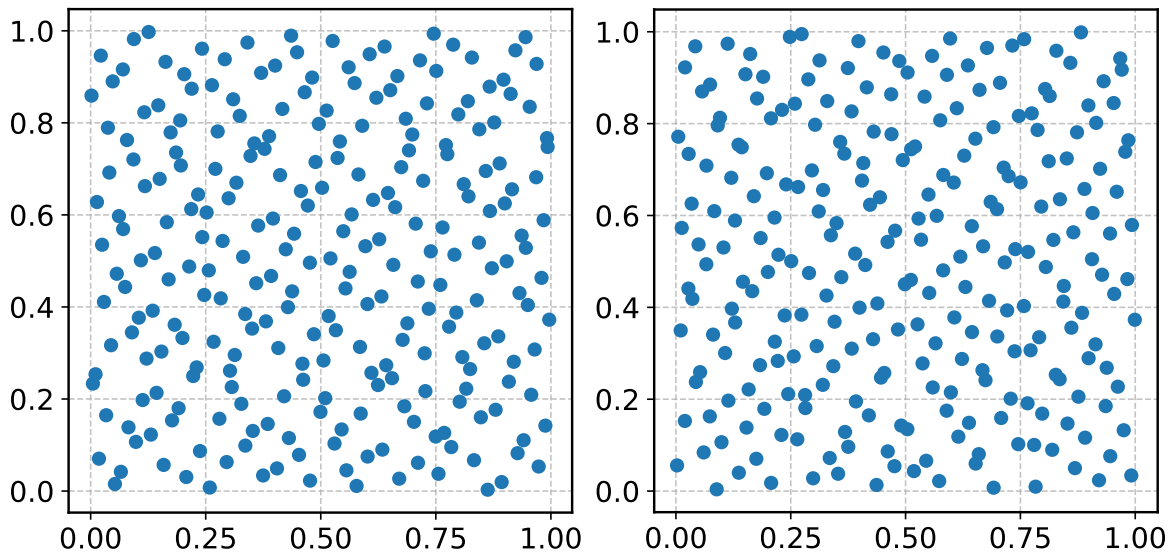
rng = np.random.default_rng(313)

m = 8
n = 2*m

fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True)
```

```
sampler1 = qmc.Sobol(d=2, scramble=True)
x, y = sampler1.random_base2(m).T
ax1.plot(x, y, "o")
ax1.set_aspect("equal")

sampler2 = qmc.Sobol(d=2, scramble=True)
x, y = sampler2.random_base2(m).T
ax2.plot(x, y, "o")
ax2.set_aspect("equal")
```



The typical convergence rate for a well-behaved QMC integral is far better than the PRNG counterpart,

$$\frac{\sigma_I}{\hat{I}} \propto n^{-1}.$$

3.6 Executive summary

When integrating a one-dimensional function, classical quadrature methods such as the ones provided by `scipy.quad()` are hard to beat—yes, you can find counter-examples to this, but you really have to look hard for them.

As the dimensionality of your problem grows, Monte Carlo and QMC techniques become an attractive alternative, as the convergence rate remains constant. For the purpose of integration, Monte Carlo methods are limited to a convergence rate of $n^{-\frac{1}{2}}$, while QMC typically achieves a much better rate of n^{-1} (or even better, in some situations) and is therefore to be preferred.

This all said, there are cases (and many, too) where you *really* want sequences whose elements look statistically independent from each other, and that is precisely what we shall start exploring in the next section.

A Version history

0.3.0 (2026-10-07)

- New chapter on Monte Carlo integration ([PR 10](#))

0.2.0 (2026-10-04)

- Complete refactoring of the code generating the figures ([PR 8](#))

0.1.0 (2026-10-02)

- Added cover page ([PR 7](#))

0.0.1 (2026-09-19)

- Initial draft of a single chapter.

- Blackman, David, and Sebastiano Vigna. 2021. “Scrambled Linear Pseudorandom Number Generators.” *ACM Trans. Math. Softw.* 47 (4). <https://doi.org/10.1145/3460772>.
- Joe, Stephen, and Frances Y. Kuo. 2008. “Constructing Sobol Sequences with Better Two-Dimensional Projections.” *SIAM Journal on Scientific Computing* 30 (5): 2635–54. <https://doi.org/10.1137/070709359>.
- L’Ecuyer, Pierre. 2017. “History of Uniform Random Number Generation.” In *Proceedings of the 2017 Winter Simulation Conference*. WSC ’17. Las Vegas, Nevada: IEEE Press.
- Marsaglia, George. 1968. “Random Numbers Fall Mainly in the Planes.” *Proceedings of the National Academy of Sciences of the United States of America* 61 1: 25–28. <https://api.semanticscholar.org/CorpusID:30098699>.
- Matsumoto, Makoto, and Takuji Nishimura. 1998. “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.” *ACM Trans. Model. Comput. Simul.* 8 (1): 3–30. <https://doi.org/10.1145/272991.272995>.
- O’Neill, Melissa E. 2014. “PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.” HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College.

Sobol', I. M. 1967. "On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals." *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki* 7 (4): 784–802.