Computational Physics – Lecture 20

(29 April 2020)

Today's plan

In today's class, we want to start a discussion on random numbers.

Introduction

Discussion on what is a random number can quickly become very philosophical.

Knuth famously asked "Is 2 a random number?"

For us, a random number is a member of a sequency of independent numbers with a specified probability distribution.

So (a) each random number is obtained by chance, (b) its value has nothing to do with the value of other random numbers, and (c) there is a specified probability for the value of the number to fall in a range of values.

Introduction

Random numbers are obviously important for computational physics.

They can be used to simulate fundamentally random physical processes, such as quantum systems.

They can also be used to study effectively random physical processes, such as noise, stochastic processes such as Brownian motion.

Random numbers are also fundamentally useful in modern data analysis for statistics, inference and phenomenology.

Random number generation

While trying to use random numbers in computation, the first question is how to get random numbers.

In the early 20th century, random numbers were obtained from mechanical, physical processes that were known to be (or thought to be) random.

For example, Tippett produced a list of 40,000 random digits from the British census in 1927.

Various mechanical random-number machines were manufactured before 1960s.

Random number generation

For example, Kendall and Babington-Smith created a machine and obtained 1,00,000 random numbers in 1939.

The Ferranti Mark 1 computer, manufactured by the British company Ferranti Ltd. in 1951, had an inbuilt random number generator that used resistor noise.

The American international-policy organisation, RAND Corporation, used a special machine to publish a million random numbers.

Most lotteries still use mechanical random number generation. For example, Maharashtra State Lottery publishes random numbers on its web site every week by using “an electric machine or by selecting numbers from drums”.

Random number generation

But such machines and published tables are not very convenient, so in the 1960s people started to come up with ways of generating random numbers using mathematical operations.

For example, John von Neumann suggested that we should choose some large integer, square it, and then take the middle part of the result as a random number.

For example we could take 5772156649, square it to get 33317792380594909201, and then take 7923805949 as our random number.

Random number generation

Note that "random" numbers obtained via such arithmetic operations are not really random.

They are called *pseudorandom numbers*.

During the time of von Neumann, people did not really understand how to quantify randomness or how to test for it. So they kept coming up with methods arbitrarily and haphazardly.

So it was not surprising that most of these methods turned to produce really poor-quality pseudorandom numbers. For example, we now know that von Neumann's method is very bad.

Random number generation

A popular (psuedo-)random number generation scheme used today is called the *linear congruential random number generator*.

This method was invented by Derrick Lehmer (UC Berkeley) in 1949.

This generator is not available in Python. But it is available in C as the `rand()` function. It is also available in GSL.

Linear Congruential Generator

How does this method work?

You begin by choosing four "magic" numbers: m, a, c and X_0.

These numbers are called the "modulus", "multiplier", "increment", and "seed", respectively.

Then the sequency of random numbers is obtained by

$$X_{i+1} = (aX_i + c) \bmod m \tag{1}$$

for $i \geq 0$.

Linear Congruential Generator

This method is so simple that we can code it up it Python:

```
a = 1664525
c = 1013904223
m = 4294967296
x = 1

n = 100
results = []
for i in range(n):
    x = (a*x+c)%m
    results.append(x)
```

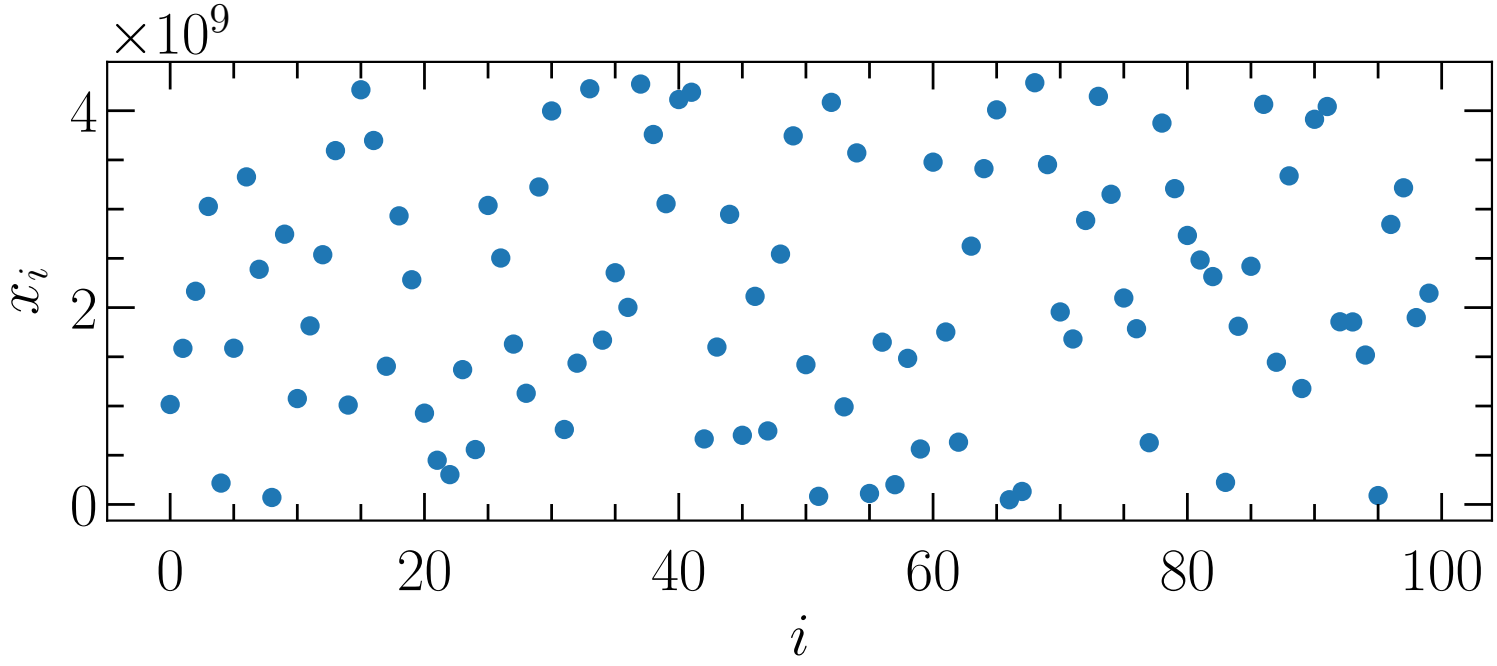
Linear Congruential Generator

This gives us this sequence of numbers:

```
1015568748
1586005467
2165703038
3027450565
 217083232
1587069247
3327581586
2388811721
  70837908
2745540835
1075679462
1814098701
2536995080
3594602695
1009643386
       ...
```

Linear Congruential Generator

These numbers do *look* random:



Linear Congruential Generator

Notice that the Linear Congruential Generator is not actually random.

The programme is deterministic. If we know the four magic numbers, we can reproduce every "random" produced by it.

It is just that the pseudorandom numbers produced by the Linear Congruential Generator are sufficiently random.

Linear Congruential Generator

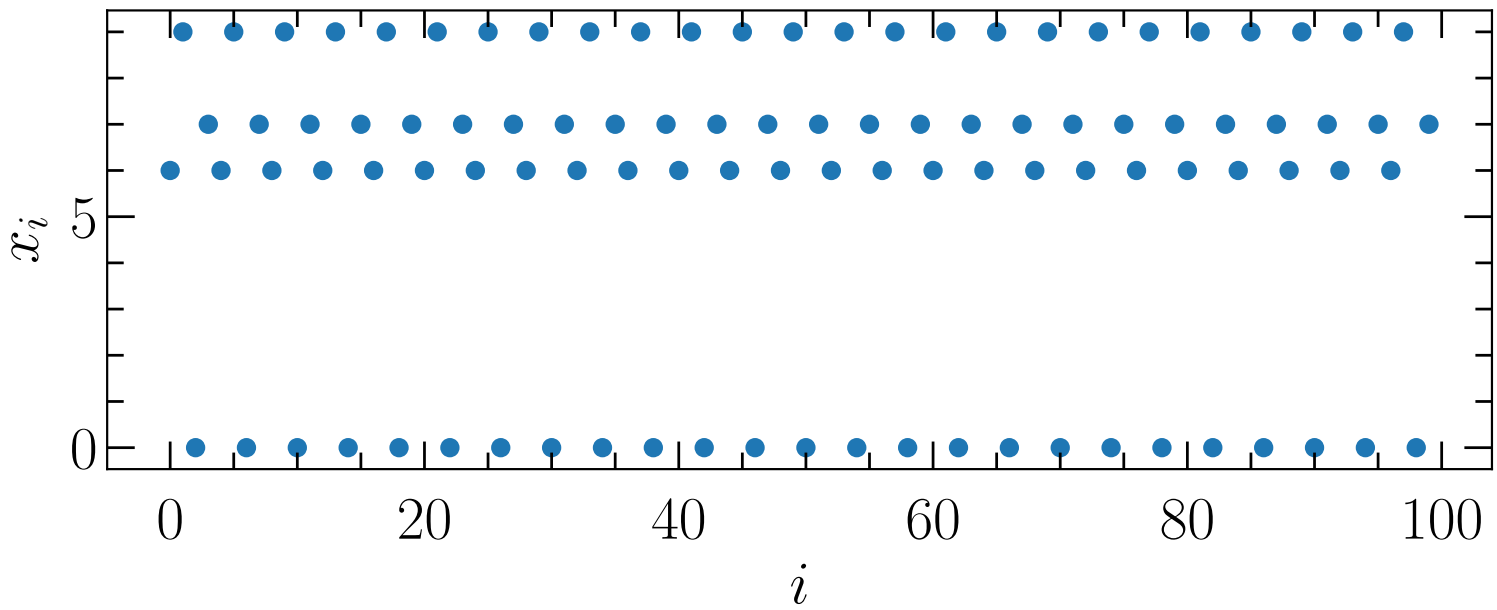
The numbers produced by the Linear Congruential Generator are between 0 and $m - 1$. To obtain numbers in some other range we can scale them as we want.

The Linear Congruential Generator completely breaks down if the numbers m, a, and c are not chosen carefully.

For example, if I take $m = 10$, $a = 7$, $c = 7$, then with $X_0 = 7$, I get the repeating sequence $7, 6, 9, 0, 7, 6, 9, 0, \ldots$.

Linear Congruential Generator

These numbers do *not* look random:



Linear Congruential Generator

You have to set up a seed value to get random number sequence.

For example, in C, if you use the `rand()` function always uses a Linear Congruential Generator with $m = 2^{31}$, $a = 1103515245$, and 12345, with the seed $X_0 = 1$.

To use the same Linear Congruential Generator with another seed, C provides another function called `srand()`, in which the values of m, a and c are the same as above, but the value of X_0 is whatever you specify.

Your random numbers can be bad for some seed values!

Other Generators

Finally, note that the Linear Congruential Generator always produces repeated patterns of numbers with period m.

It turns out that the Linear Congruential Generator is fast but not the best pseudorandom number generator.

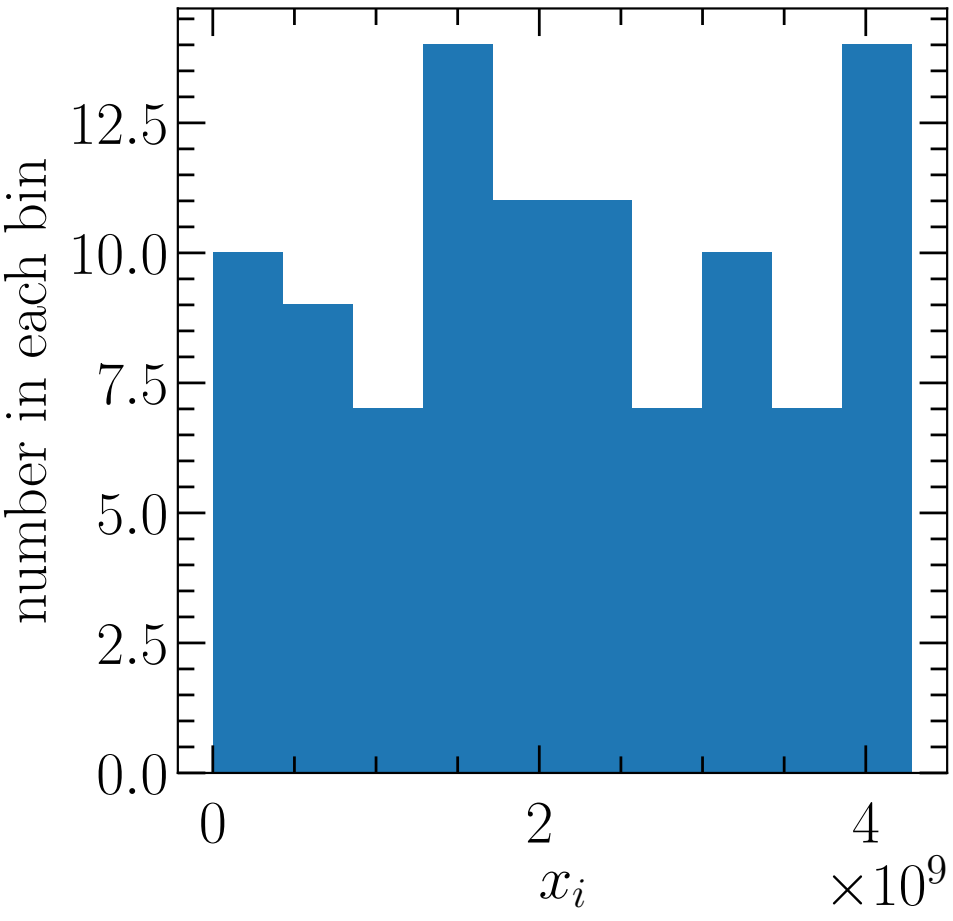
There are other random number generators available. For example, the Python standard library uses the Mersenne Twister (1997) random number generator. Numpy uses the Permuted Congruential Generator (2014).

Uniform deviates

All of the above generators give uniformly distributed random numbers between 0 and m.

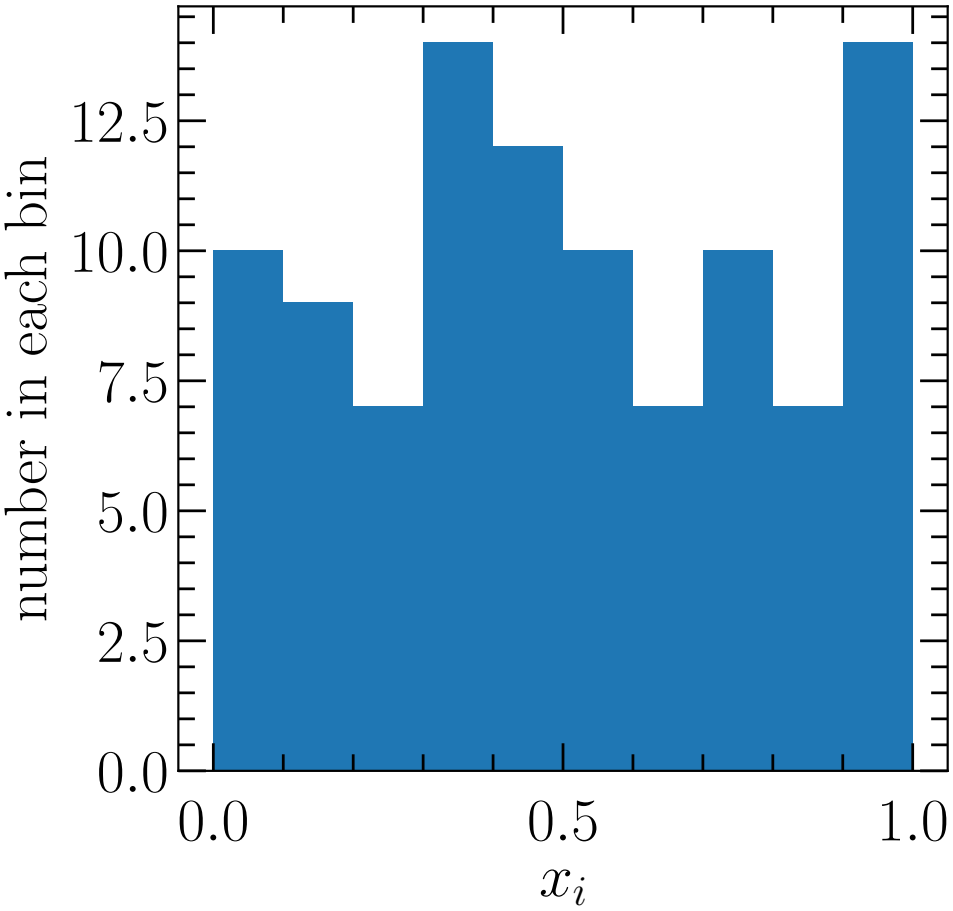
We can check this by plotting a histogram of the 100 random numbers that we produced before.

Uniform deviates



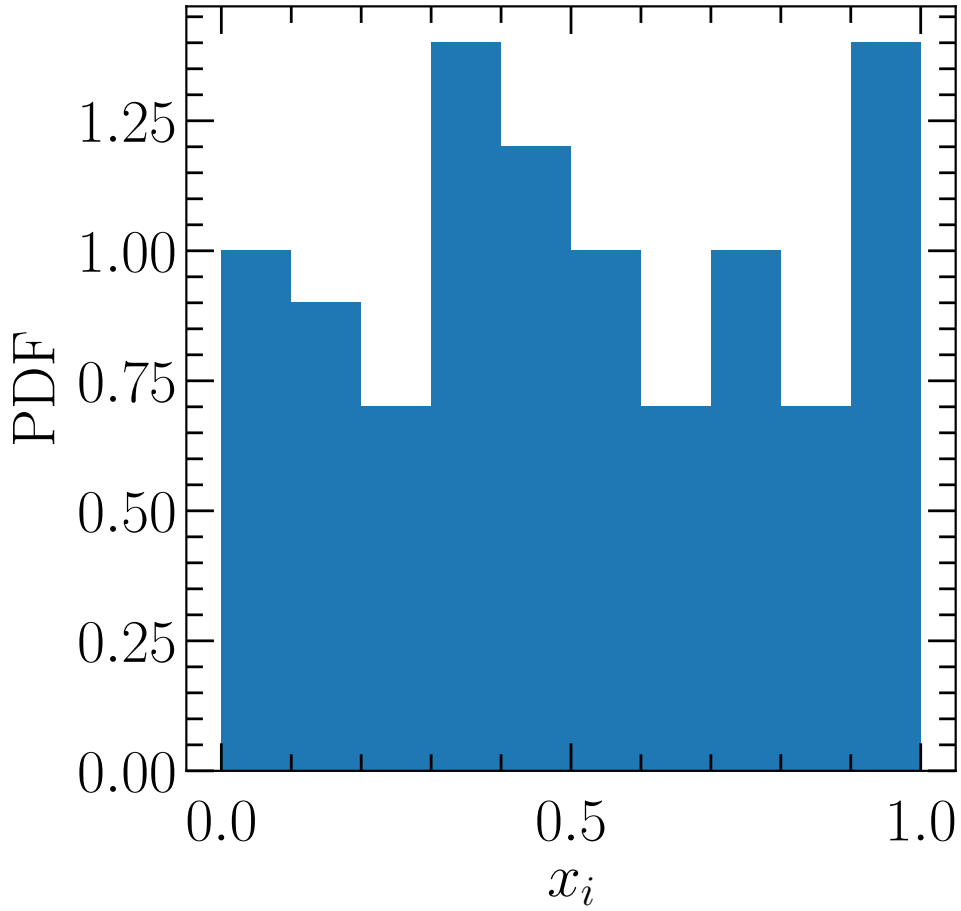
(Histogram of 100 random numbers generated using our linear congruential generator.)

Uniform deviates



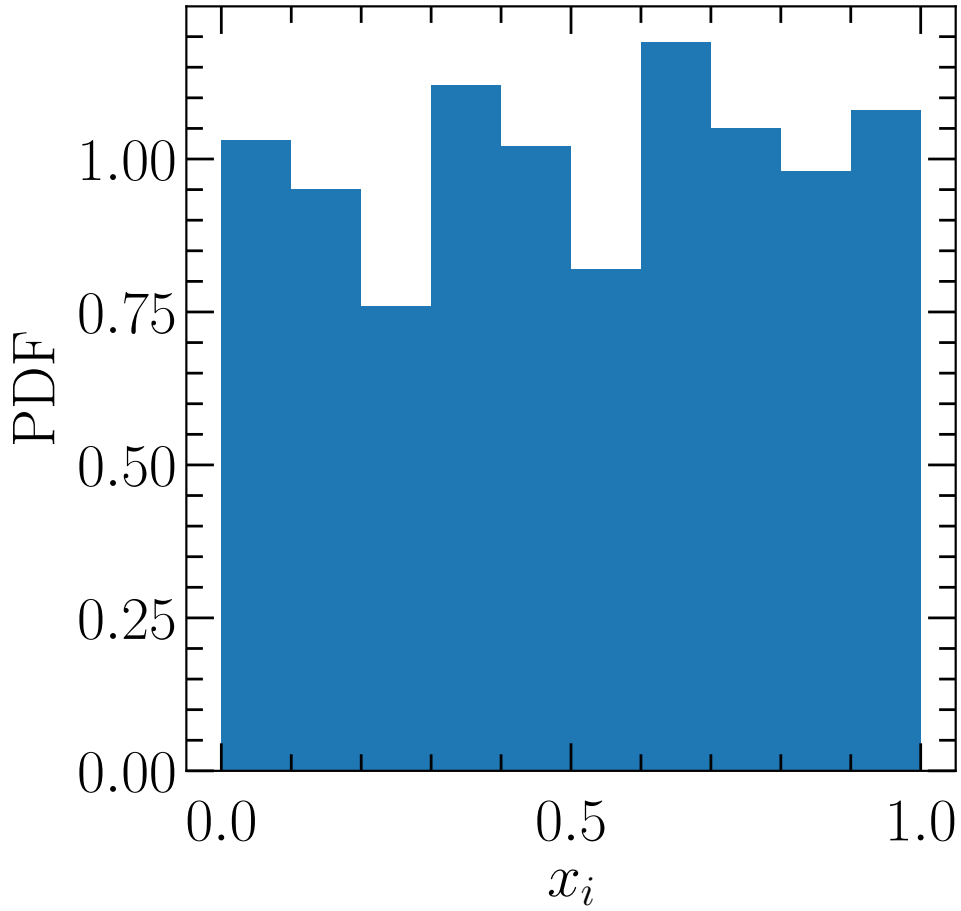
(Histogram of 100 uniform deviates between 0 and 1.)

Uniform deviates



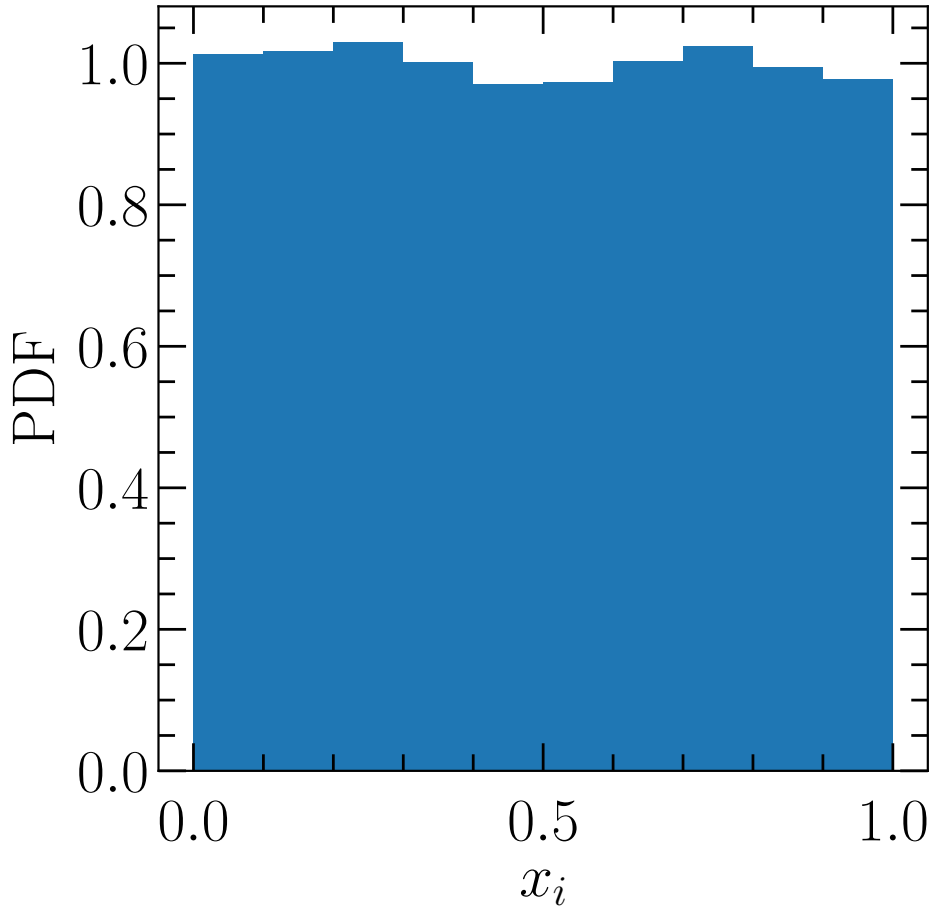
(A PDF of our 100 uniform deviates between 0 and 1.)

Uniform deviates



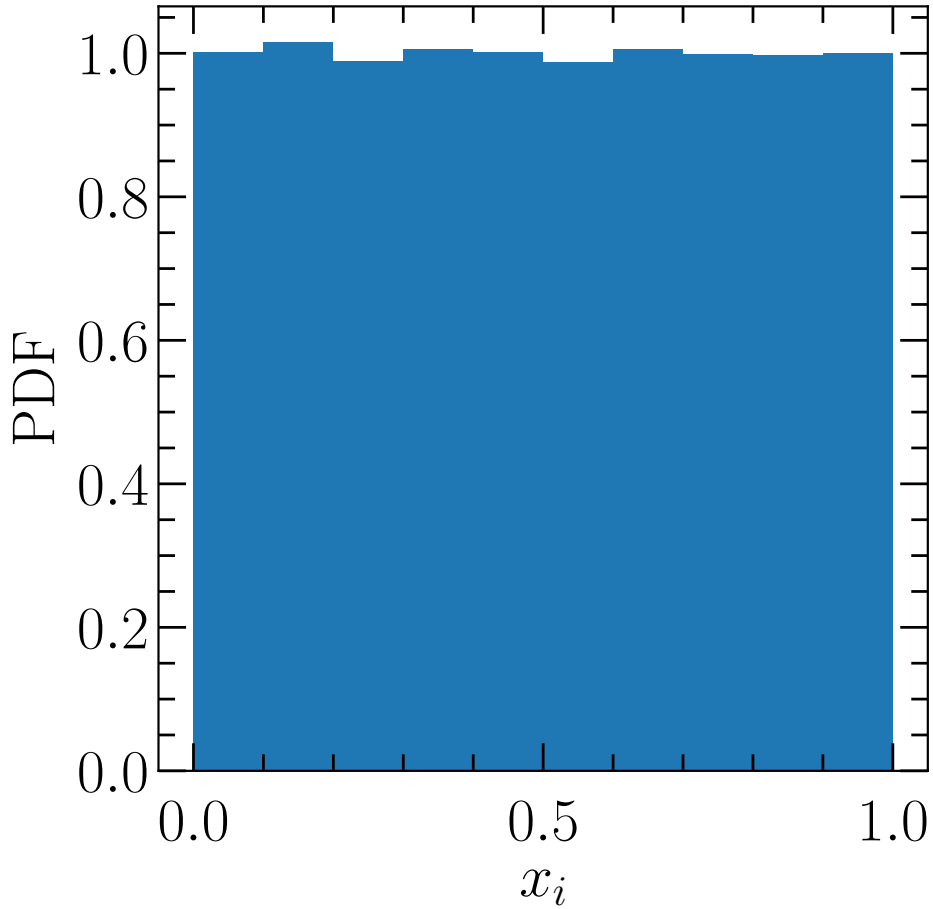
(A PDF of 1,000 uniform deviates between 0 and 1.)

Uniform deviates



(A PDF of 10,000 uniform deviates between 0 and 1.)

Uniform deviates



(A PDF of 1,00,000 uniform deviates between 0 and 1.)

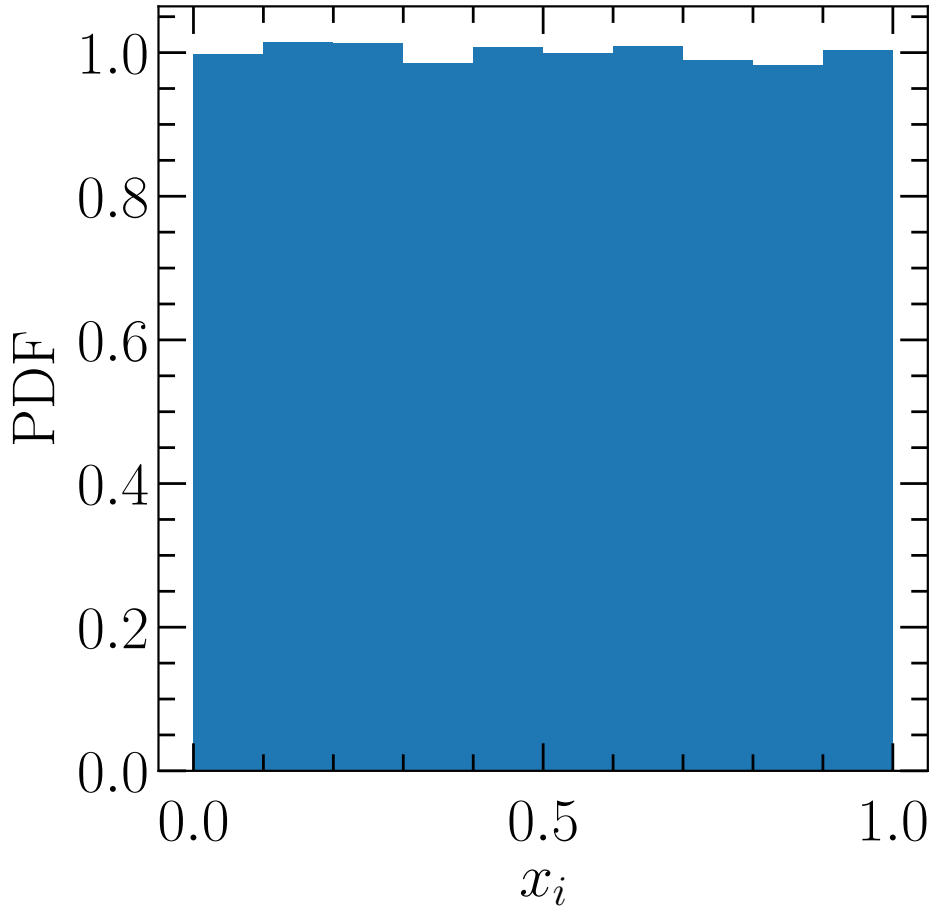
Uniform deviates using Numpy

The `np.random.random` function in Numpy generates uniformly distributed random numbers between 0 and 1.

We can plot 1,00,000 random numbers using this function:

```
x = np.random.random(100000)
plt.hist(x, range=(0.0, 1.0), density=True)
```

Uniform deviates



(A PDF of 1,00,000 uniform deviates obtained using Numpy.)

Uniform deviates using Numpy

The function `np.random.randint` gives uniformly distributed integers.

For example, `np.random.randint(1, 6, size=10)` gives these ten numbers between 1 and 6 on my computer:

```
array([2, 4, 1, 4, 5, 4, 4, 4, 3, 1])
```

Activity

Simulate the rolling of two dice a million times (10 lakh times). Count the number of times you get a double six. Divide that number by million and report the answer.

A warning

"Random" modifications to a randon number generator do not give a better random number generator!

Always use generators the way their inventor wants you to use them.