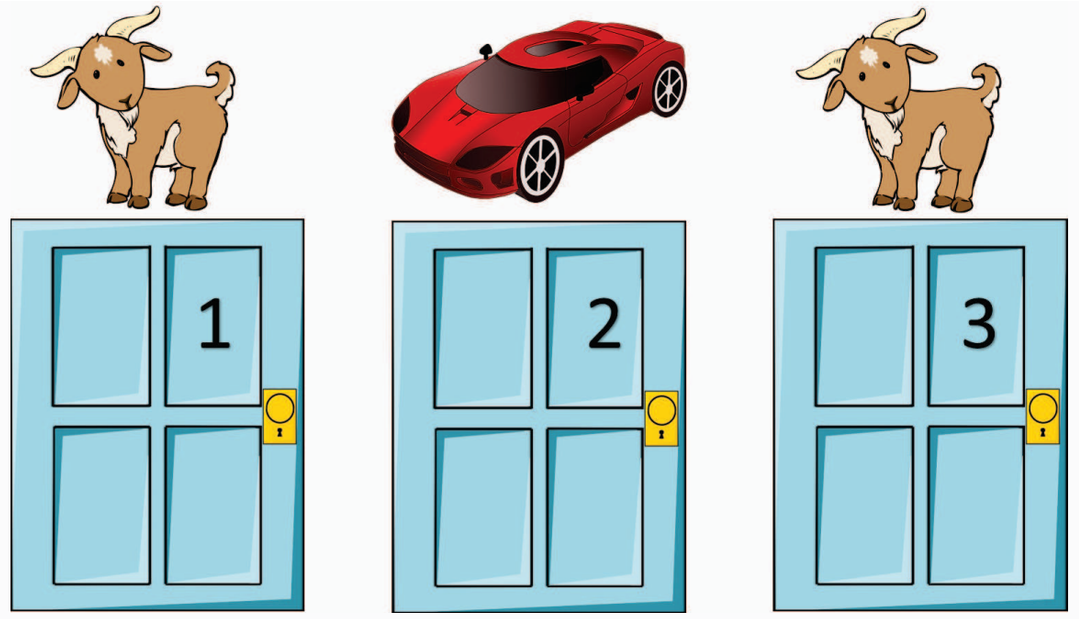
Probabilistic solutions to problems – Monte Carlo & Discrete-event simulations

The Monty Hall Problem

The Monty Hall Problem

The American television game show that started in 1963

The Monty Hall Problem cont'd



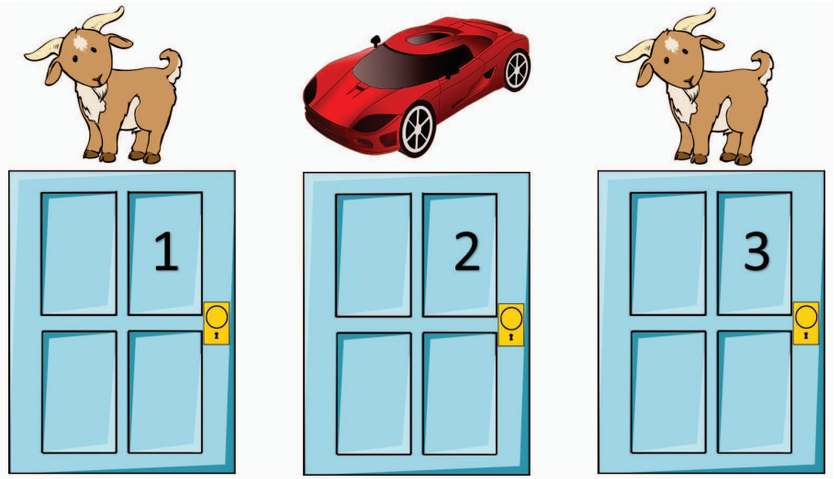
- The game show has 3 doors. A prize car is behind one door, and "the Zonk" (a goat) is behind each of the other two.
- The contestant chooses (randomly) one door.

The Monty Hall Problem cont'd

- The smiling host Monty Hall opens one of the other doors, always choosing one that shows a goat, and offers the contestant a chance to switch to the remaining unopened door

- The constant either chooses to switch, or opt to stick to the first choice

- Monty calls for the remaining two doors to open, and the contestant wins whatever is behind the chosen door



Question: What is the best strategy for the contestant? Does switching increase the chance of winning the car, decrease it, or make no difference?

The best strategy is to switch!

The Monty Hall Problem cont'd

A quick estimate:

Let's label the door chosen by the contestant as Door 1, and the remaining two as Door 2 and Door 3.

Door 1	Door 2	Door 3	Staying	Switching
Car	Goat	Goat	Car	Goat
Goat	Car	Goat	Goat	Car
Goat	Goat	Car	Goat	Car

The Monty Hall Problem cont'd

Note that switching does not always get you the car!

It is just the best strategy in the sense that your chance of winning the car is better. This simply means that if you are the contestant for, say, 10,000 times, and you adopt this strategy, you are a winner more often than you are a loser, on average...

Monte Carlo Method

Introducing the Monte Carlo method

Monte Carlo methods: a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results

Repeated random sampling = picking randomly from a pool of possibilities, many times over

Using randomness to solve problems that might be deterministic in principle .... either because we are unable to obtain a solution to the problem in the deterministic formulation, or more detailed information on the solution is needed

Essential elements in Monte Carlo methods

- Define a domain of possible inputs
- Generate inputs randomly from a probability distribution over the domain
- Perform a deterministic computation on the inputs
- Aggregate the results of the individual computations into the final result

The Monty Hall game as an example

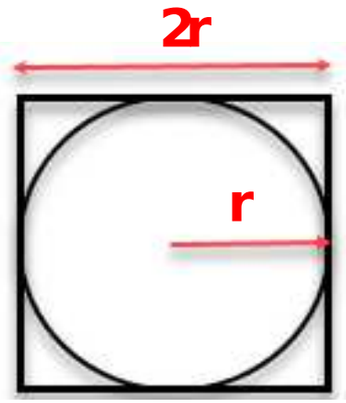
For example, we can use the Monte Carlo method to simulate the 10,000 times you are the contestant in the Monty Hall game

- First the car can be behind any of the 3 doors
- Secondly, you have the choice of 3 doors
- Then you have the choice of switching or not
- Each choice is equally likely – uniform distribution
- “Deterministic computation = opening all doors”, and deciding whether you win the car.
- Do 10,000 of such simulations and take the average to determine whether the overall chance of winning is better with switching, or not.

Another example: determination of the value of π

Consider a circle inscribed in a square. Given that

$$\frac{\text{Area of the circle}}{\text{Area of the square}} = \frac{\pi r^2}{(2r)(2r)} = \frac{\pi}{4}$$



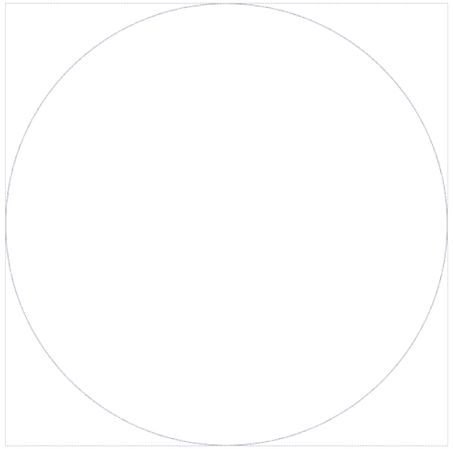
We can approximate the value of π by a Monte Carlo method:

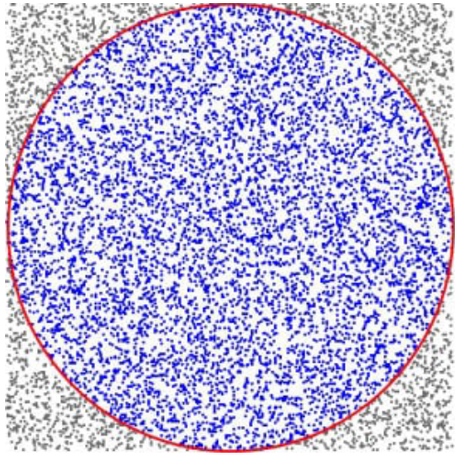
- Draw a square, then inscribe a circle within it
- Generate random points uniformly within the square
- Count the number of points inside the circle
- The ratio of the inside-circle-count to the total-sample-count is an estimate of the ratio of the two areas. Multiply the result by 4 to estimate π

Another example: determination of the value of π cont'd

In the procedure we just described,

- the domain of inputs = all the points within the square that circumscribes the circle
- generate random inputs = generate random points uniformly within the square (aka sampling)
- perform deterministic computation on each input = test whether the point falls within the circle
- aggregate the results = obtain the inside-circle-count and the total-sample-count and take the ratio.





Accuracy of an approximation/estimation

Two important points here:

- The points must be uniformly distributed – otherwise the estimate will be poor. Imagine a biased sampling, where points within the circle is more preferred.
- There should be a large number of inputs (points).

Furthermore, when we complete the procedure once, we get one estimate of π

- As scientists, we need to know what the accuracy of that estimate is
- And as scientists, we repeat the procedure, preferably many times, to see what the variance of the different estimates are.

Computer simulation and generation of random samples

The key idea behind the Monte Carlo method is: the results are computed based on repeated random sampling and statistical analysis.

The repetitive sampling and computation clearly are easier done on a computer.

And how do we generate random inputs (and many many of them ...)?

This takes us to the pseudo random number generators.

Automated (pseudo)random number generators (RNGs)

There are ways to generate deterministic, pseudorandom sequences on a computer, making it easier to test and re-run simulations

- deterministic because the sequences are generated using a well-defined algorithm (or formula)
- pseudorandom because the generated sequence is not truly random, but only “appears” random

An example of a pseudorandom sequence

A typical (pseudo) uniform random number generator uses the multiplicative congruential algorithm with 3 integer parameter, a, c, and m, and an initial value, x_0, the seed.

Uniform random number generator produces pseudo-random numbers in the interval [0, 1] uniformly, i.e. every number in the interval is equally likely to be generated.

A sequence of integers is defined by

$$x_{k+1} = (a x_k + c) \bmod m, \quad k = 0, 1, \cdots$$

- The operation “mod m” means take the remainder after division by m. For example,

$$8 \bmod 2 = 0, \quad \text{and} \quad 123 \bmod 4 = 3$$

An example of a pseudorandom sequence cont'd

$$x_{k+1} = (ax_k + c) \bmod m, \quad k = 0, 1, \cdots$$

Let us take $a = 13$, $c = 0$, $m = 31$, and $x_0 = 1$, then from the formula above, we have

$$ax_0 + c = 13 \times 1 + 0 = 13$$

and 13 is not divisible by 31, so we have $x_1 = 13$

Continuing,

$$ax_1 + c = 13 \times 13 + 0 = 169,$$

and $169 = 5 \times 31 + 14$, giving $x_2 = 14$

The sequence will then look like

$$1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, \ldots$$

An example of a pseudorandom sequence cont'd

- In this example, the numbers x_n generated is bounded: $1 \leq x_n < 31$ (limited by m)
- If a particular number is repeated, then the sequence after it will be the same. (The numbers came from a formula, remember?)
- Choices of a, c, m are important factors that determine the quality (i.e. how close to true randomness) of the (pseudo)random number generator

The infamous IBM RANDU

An example of a bad RNG:

Introducing RANDU or RND, a random number generator included in the Scientific Subroutine Package on IBM mainframe computers in the 1960s

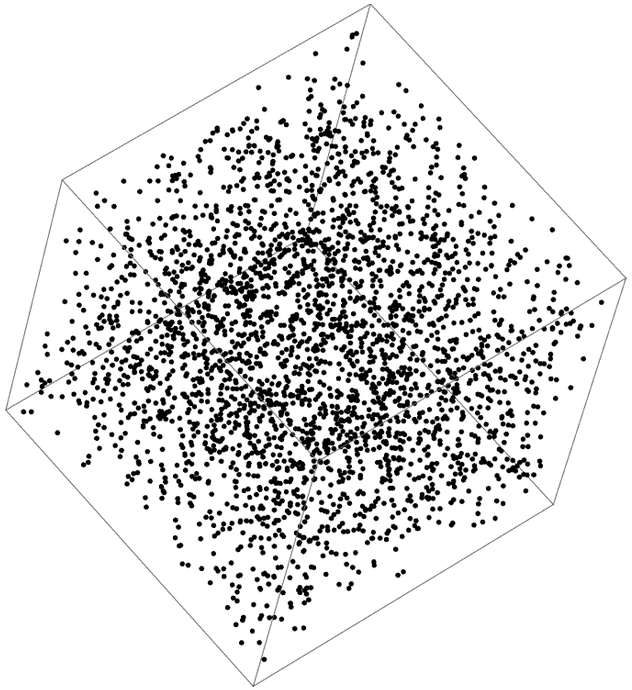
It was a multiplicative congruential with parameters $a = 65539 = 2^{16} + 3$, $c = 0$, and $m = 2^{31}$.

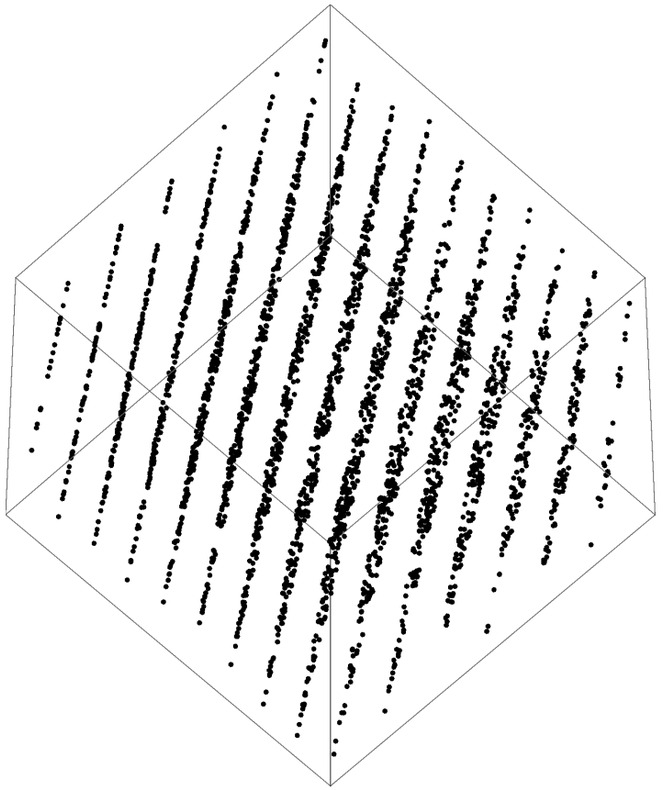
Unfortunately, the numbers generated are correlated with each other, and hence are not all random in the true sense of the word! We shall show this in the next slide

The infamous IBM RANDU cont'd

A plot of 32,000 triples of the form $[x_0, x_1, x_2]$, $[x_3, x_4, x_5]$, ... taken from 96,000 consecutive RANDU values demonstrate that they "clump" along a mere 15 parallel planes!

This means that the numbers generated are not all random in the true sense of the word!





The infamous IBM RANDU cont'd

With $x_{n+2} = (2^{16}+3)x_{n+1} \mod 2^{31}$, and $x_{n+1} = (2^{16}+3)x_n \mod 2^{31}$, we have

$$x_{n+2} = (2^{16}+3)x_{n+1} \bmod 2^{31} = (2^{16}+3)^2 x_n \bmod 2^{31}$$

Expanding the quadratic factor,

$$x_{n+2} = (2^{32} + 6 \times 2^{16} + 9)x_n \bmod 2^{31}$$

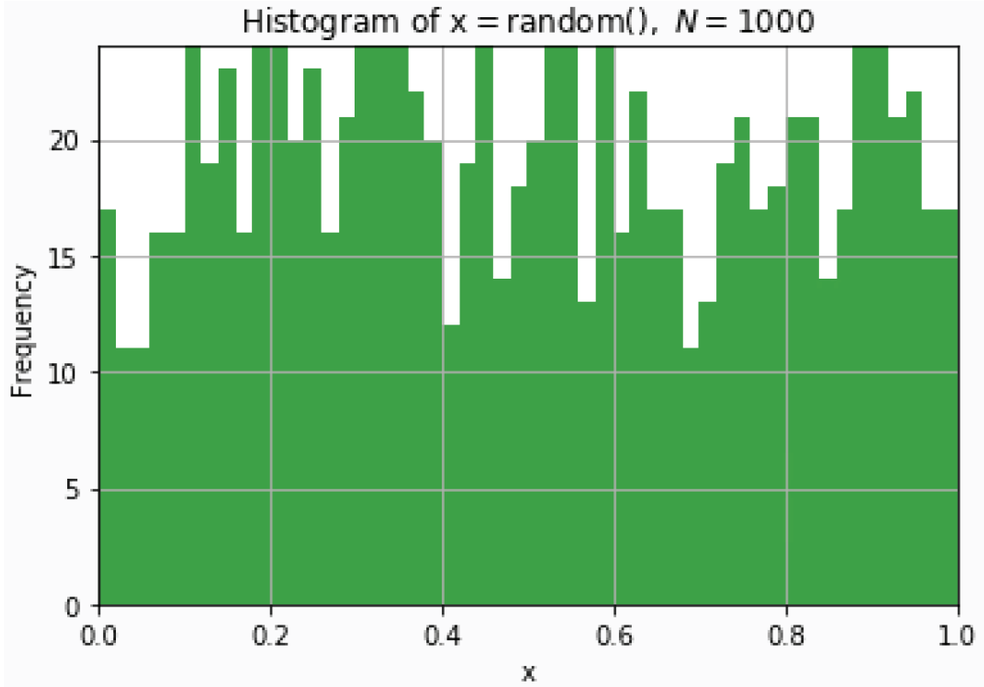
Now since $2^{32} = 2 \times 2^{31}$ and so when this is divided by 2^{31}, you get exactly 2 with no remainder. Hence $2^{32} \bmod 2^{31}$ is 0. This gives

$$\begin{aligned} x_{n+2} &= (6 \times 2^{16} + 9)x_n \mod 2^{31} = (6 \times 2^{16} + 18 - 9)x_n \mod 2^{31} \\ &= [6(2^{16}+3)x_n - 9x_n] \mod 2^{31} = (6x_{n+1} - 9x_n) \mod 2^{31} \end{aligned}$$

This clearly tells us how the 3 consecutive numbers in the generated sequence are related. eg 1, 65539, 393225, 1769499, 7077969, 26542323, 95552217, $\cdots$

Python: Uniform random number generator

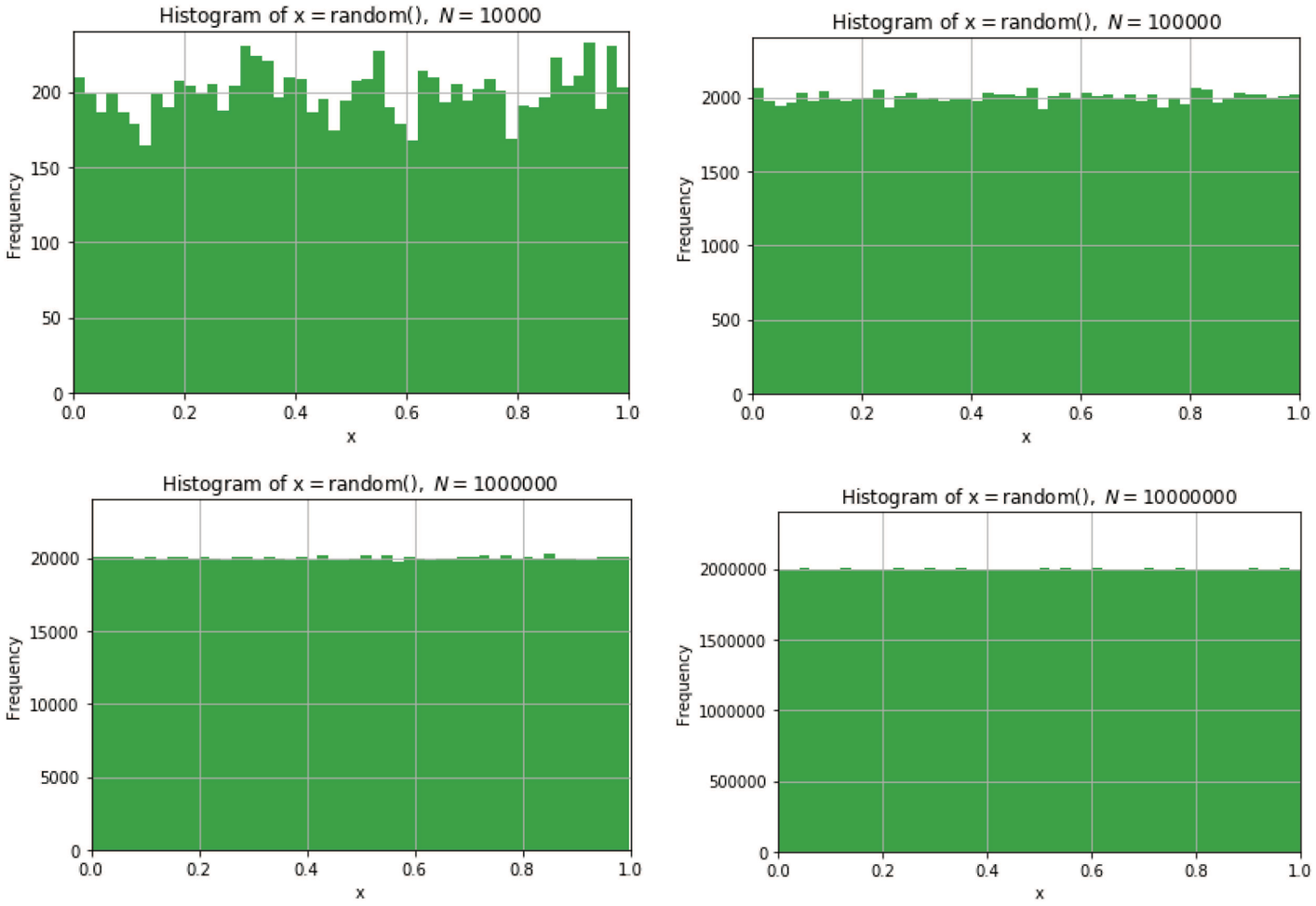
If we generate 1000 pseudorandom numbers using the Python random() generator, the histogram of the distribution is



Doesn’t look very “uniform”. But it should be clear that if you generate only a few numbers, the equally likely is not going to be obvious.

Python: Uniform random number generator cont'd

Let's generate more to see what happens to the distributions:



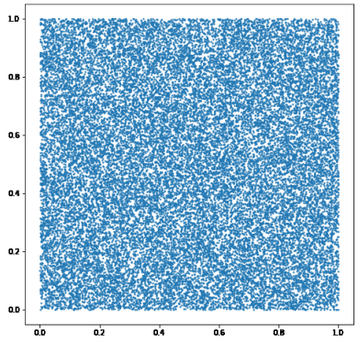
“Uniform” distribution

- From these generated histograms of the random numbers generated, we see that the uniform distribution is only apparent when a very large number of such random numbers are generated.
- When a computer simulation uses only a finite number of random numbers, even if they are truly uniformly distributed (in the large samples limit), there will be statistical errors/uncertainties.
- Uniformity in distribution is only the lowest level check for a good random number generator.

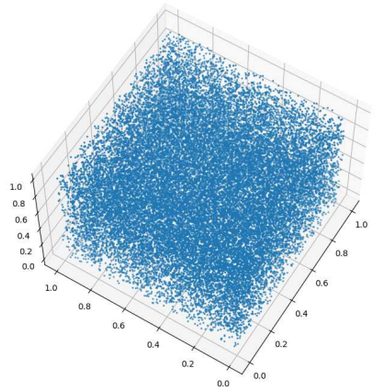
The doubles & triples

Just like before, we construct the sequence of triples: $\{[x_0, x_1, x_2], [x_3, x_4, x_5], \ldots\}$ and doubles $\{[x_0, x_1], [x_2, x_3], \ldots\}$

We can treat each of these doubles as the coordinates of a point on a two-dimensional plane and plot them. The doubles from a good pseudo-random number generator should be uniformly distributed on the plane



These triples should also be evenly (uniformly) distributed in the cube and not fall in planes!



Computer-based Pseudo-random number generators

Summary:

- Pseudo-random numbers are generated by deterministic algorithms that produce sequences that “resemble” random sequences.

PROS

- simple
- fast
- convenient

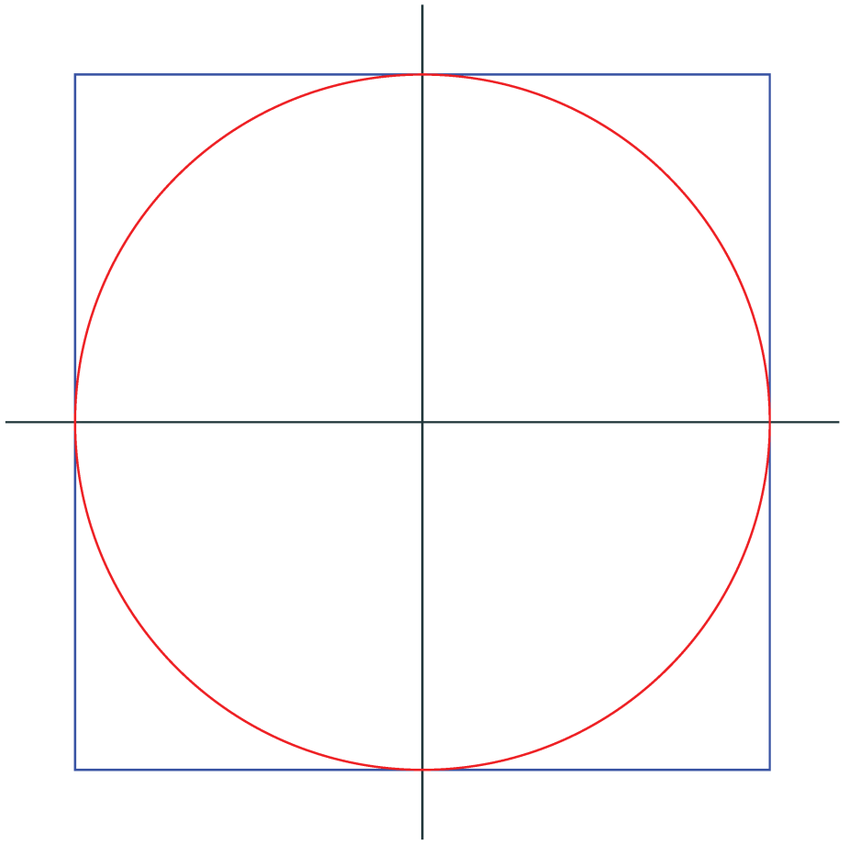
CONS

- periodicity
- correlations

Monte Carlo Simulations

How would a MC program for estimating π look like?

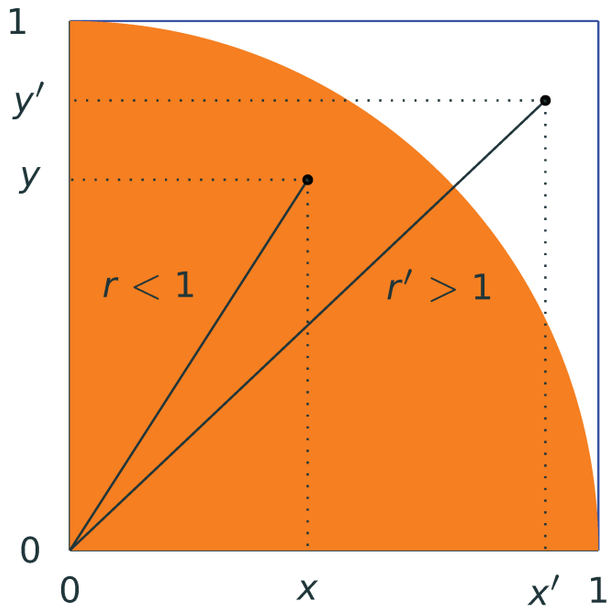
First, we can try to be more efficient, by observing that a circle enclosed by a unit square has 4 symmetrical (and effectively identical) pieces.



How would a MC program for estimating π look like?

So we can just focus on one quadrant:

For a circle of unit radius, the area is simply π, so the orange region will have area equal to $\pi/4$



This means if we sample points within the unit square uniformly randomly, then

$$\pi \approx 4 \times \frac{\text{No. of points in the Orange region}}{\text{Total no. of points within the square}}$$

The algorithm

A Monte Carlo algorithm would involve the following steps:

1. Fix total number N of samples
2. Set up counter N_{in} and initialize to 0
3. Call the uniform RNG to generate two random numbers in the interval [0, 1] – these will be taken as the (x, y) coordinates of a point in the square
4. Decide whether the point falls within the circular quadrant: if $x^2 + y^2 < 1$ $(\text{or } \leq 1)$, the counter N_{in} is then incremented by 1
5. Repeat Steps 3 & 4 for another $N - 1$ times
6. Compute $4N_{\text{in}}/N$. This is the MC estimate of π

How to use the Random module in Python

- import numpy as np
- np.random.random() – returns a random number x in the interval [0, 1), i.e., $0 \leq x < 1$.

  e.g. myran = np.random.random()
- np.random.randint(a,b) – (a, b must be integers) returns a random integer N such that $a \leq N < b$ (i.e. inclusive of a but not b)

  e.g. myranInt = np.random.randint(1, 5)

  myranInt will be an integer from (1, 2, 3, 4)

Python code for the Monte Carlo estimation of π


```
import numpy as np

# Number of points that land inside the circle.
inside = 0

# Total number points to sample.
N = 1000

# Iterate for the number of points.
for i in range(0, N):

    # Generate random x, y in [0, 1].
    x2 = np.random.random()**2
    y2 = np.random.random()**2

    # Increment if inside unit circle.
    if np.sqrt(x2 + y2) < 1.0:
        inside += 1

# inside / total = pi / 4
pi = (float(inside) / N) * 4
print(pi)
```

Other examples: Random walks

A random walk is the process by which randomly-moving objects wander away from where they started.

A typical example is the zig-zag way a drunk walks along a street. In his intoxicated state, he is as likely to take a step forward as backward: 50% chance in either of the two directions. How far will he move from the original spot after n steps? How long does it take for him to reach the next junction that is, say, M steps away?

Of course, in realistic situations, the drunk is not likely to walk in a straight line, but could also take steps to the left or right, in addition to forward or backward. Let us focus on the backward-forward motion for now - imagine him walking in a very narrow alley ...

A drunkard's walk ... in one dimension

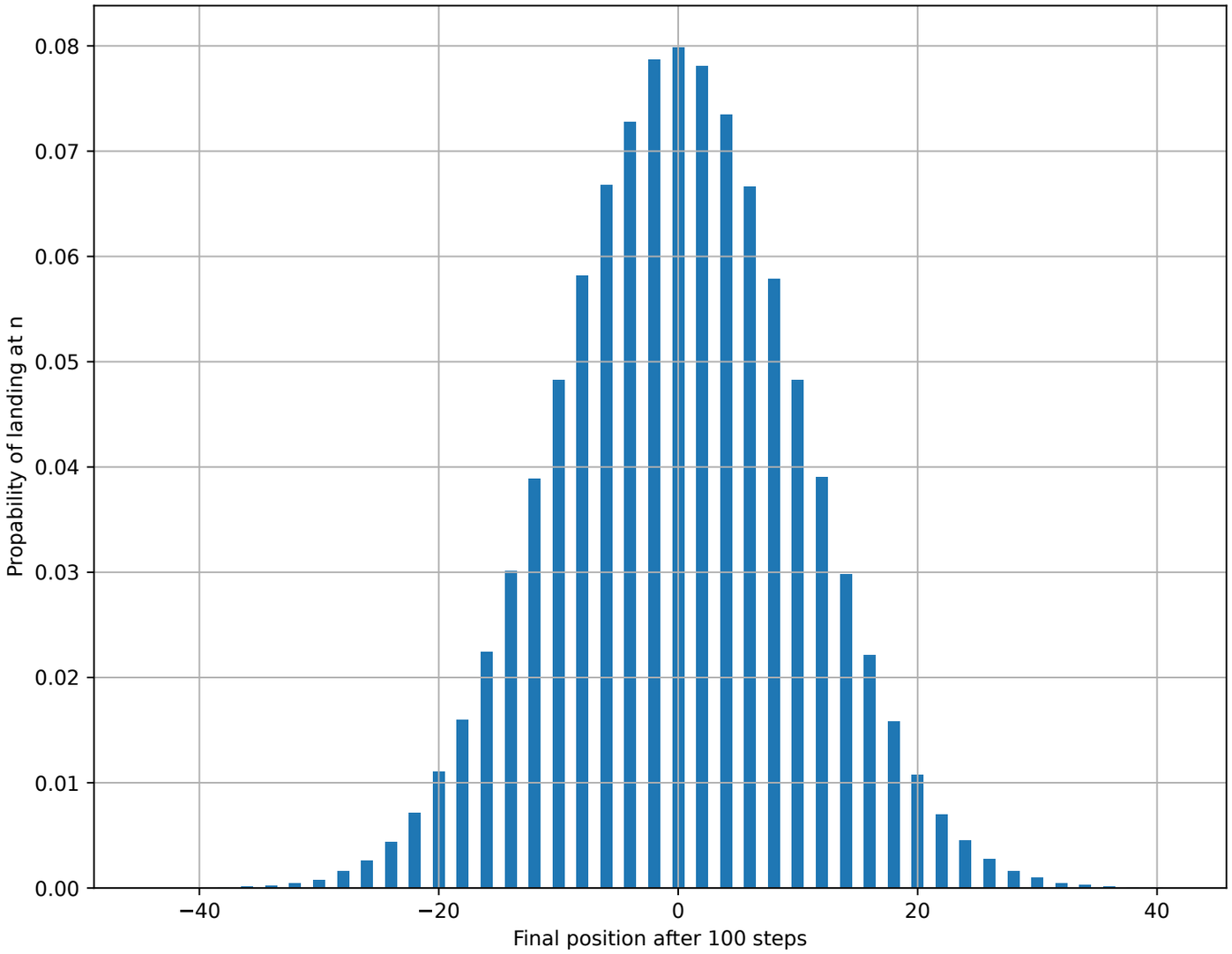
Flip a Coin, Take a Step

The one-dimensional random walk is constructed as follows:

- You walk along a line, each pace being the same length
- Before each step, you flip a coin
- If it's head, you take one step forward. If it's tail, you take one step back.
- The coin is unbiased, so the chances of heads or tails are equally probable.

The problem is to find the probability of landing at a given spot after a given number of steps, and, in particular, to find how far away you are on average from where you started.

How far can I get?



Applications of random walks

When generalized to two or three dimensions, random walks provide means of simulating many physical and natural phenomena:

- Einstein used the random walk to find the size of atoms from the Brownian motion.
- Diffusion of atoms/molecules in a medium
- Movement, dispersal and population redistribution of animals and micro-organisms.

Discrete Event Simulation (DES)

Discrete Event Simulation (DES)

Discrete Event Simulation (DES) is the discrete-time approach to the simulation study of dynamical systems that include some probabilistic/random elements in the time evolution

There are two approaches: fixed-increment time progression and next-event time progression

Here, we shall be concerned only with the fixed-increment time progression. In this approach, time is broken up into small time intervals and the system state is updated according to the set of events/activities happening during the time interval

Between these event times, it can be viewed as changing in a simple, predictable fashion

Traffic jams – Could random driver behavior be a cause?

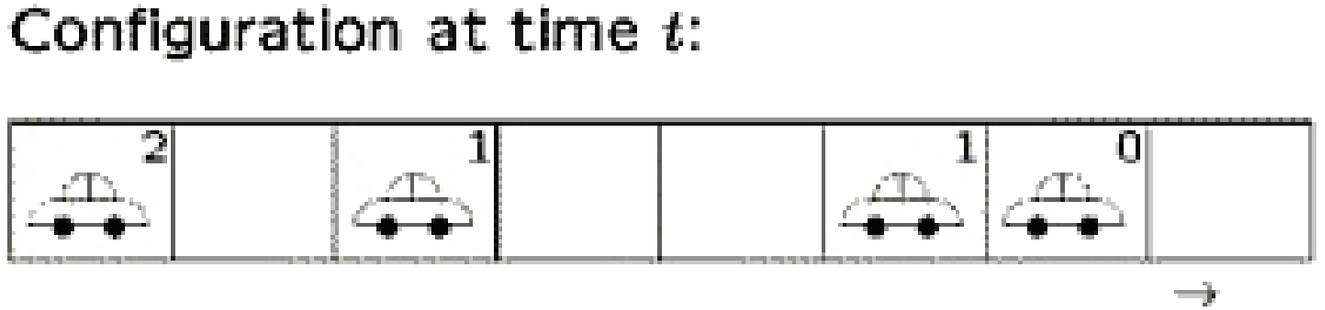
Let's see how we can go about simulating a very simple traffic model, just investigating random slow down by drivers.

Some assumptions:

- drivers generally speed up when it is judged to be safe (i.e. a good distance away from the car in front)
- drivers slow down to avoid collision
- at any one time, certain driver slows down for no apparent reason

How can we construct a simulation to check if this could be one reason for traffic jams?

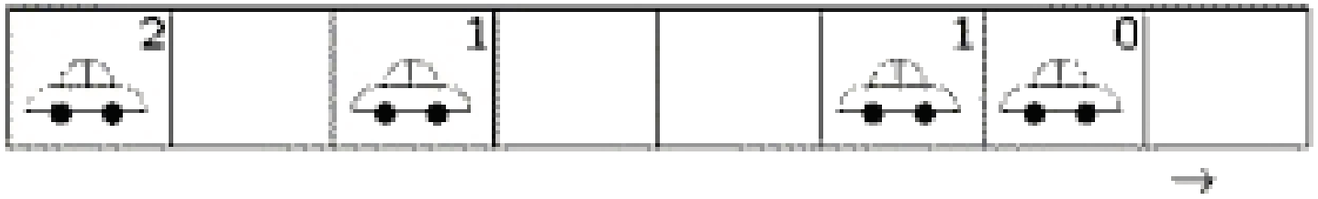
Nagel-Schreckenberg model



- Take the simplest case of a single lane of traffic
- Standardize units of distance and speed: distance to be measured in terms of number of fixed length cells, and speed to be measured in terms of the number of cells per unit time
- Set the maximum speed $v_{\max}$ - input based on local traffic rules

Nagel-Schreckenberg model cont'd

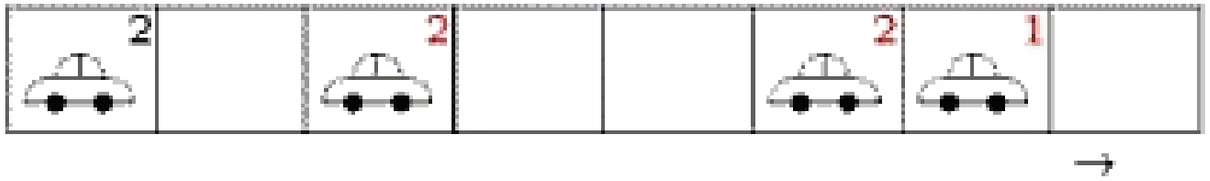
Configuration at time t:



We must incorporate some additional elements into the model:

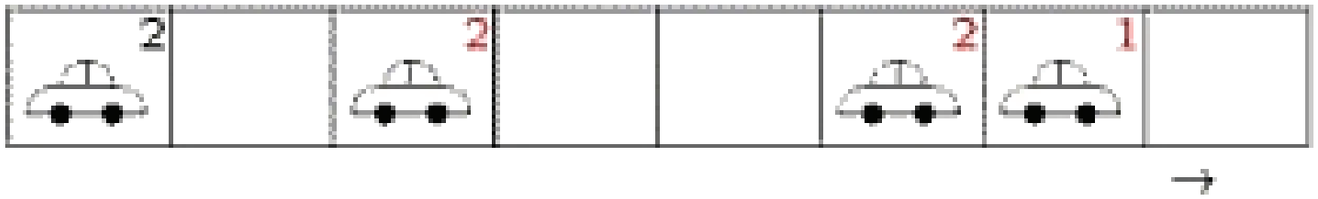
1. acceleration: All cars that have not already reached the maximum speed $v_{\max}$ will accelerate the speed by one unit. ie $v \to v + 1$

a) Acceleration ($v_{max} = 2$):



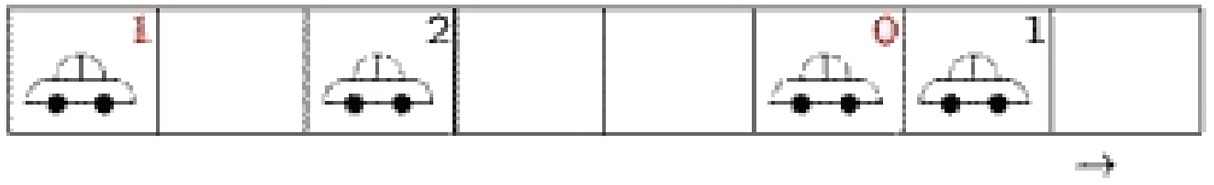
Nagel-Schreckenberg model cont'd

a) Acceleration ($v_{max} = 2$):



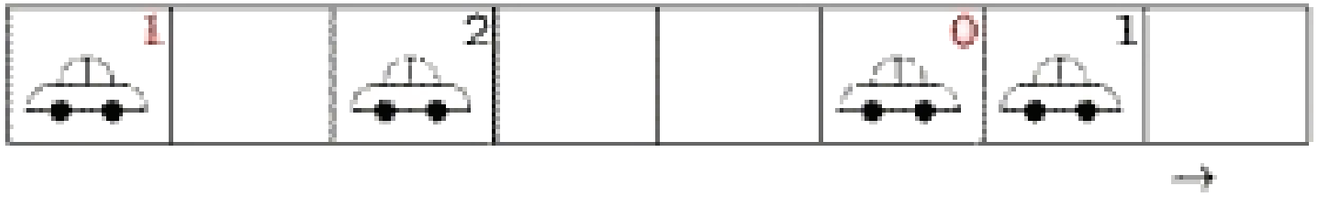
2. braking: If a car has d empty cells in front of it and its speed is larger than d, then it reduces the speed to d: $v \rightarrow \min\{d, v\}$

b) Braking:



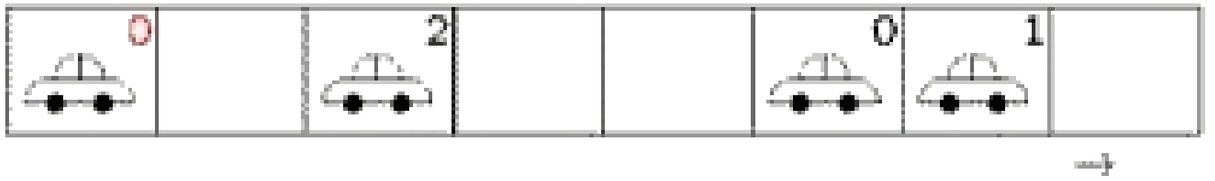
Nagel-Schreckenberg model cont'd

b) Braking:



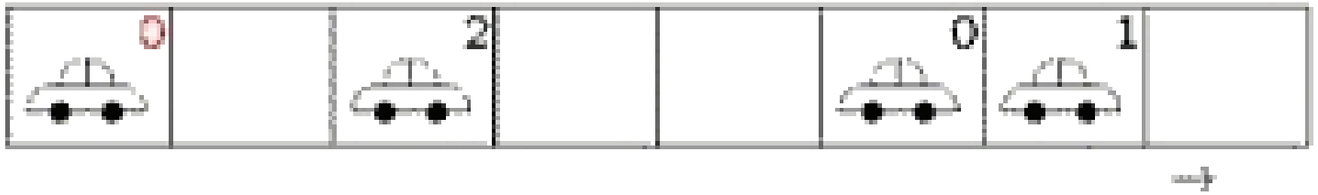
3. random driver behavior (random slowing down): With probability p, the speed of a car is reduced by one unit ie $v \to v - 1$

c) Randomization ($p = 1/3$):



Nagel-Schreckenberg model cont'd

c) Randomization ($p = 1/3$):



4. moving of the cars: With the new speed, each car (labeled by n) moves forward by v_n cells ie. $x_n \to x_n + v_n$

d) Driving (= configuration at time $t + 1$):

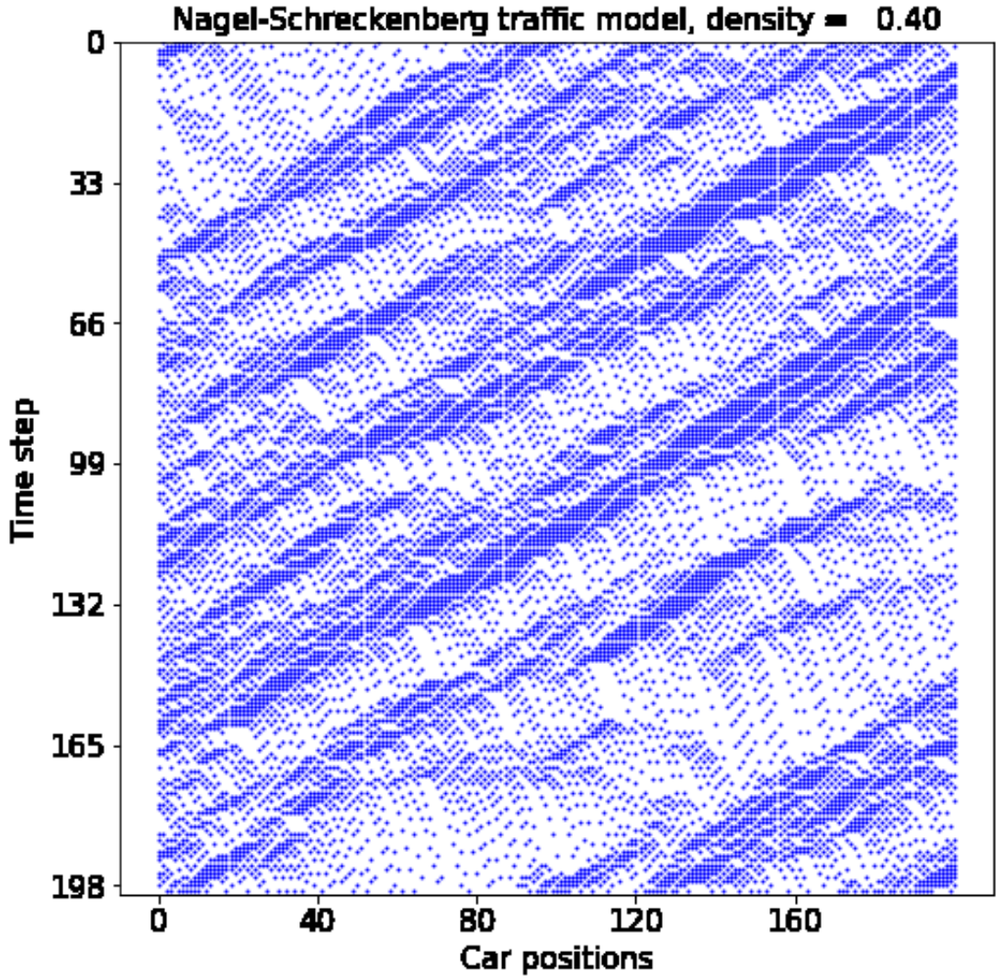
Nagel-Schreckenberg model cont'd

The model that we have arrived at is the Nagel-Schreckenberg model

One can simulate the traffic condition depicted in this model on a computer:

- You will need to input an initial configuration of the N cars: i.e. where (which cells) the cars are, and their speeds
- You will need to provide the parameter $v_{\max}$

Traffic pattern in the Nagel-Schreckenberg model



Traffic pattern in the Nagel-Schreckenberg model