

1D random walk

CONTENTS

[Summary](#)[Learning objectives](#)[Background: random walk in 1D](#)[Exercise 1: random walk in 1D](#)[Exercise 2: split-screen output](#)[Acknowledgements](#)

Summary

In this challenge, we'll play around with random processes by looking at one-dimensional random walks. A simple coin toss will be our main example of randomness. You'll get to simulate random walks on a grid where the edges "wrap around," making it feel like the space goes on forever. The activities include keeping track of coin toss results—like how the fraction of heads changes over time—and seeing how they line up with the probabilities we expect. Along the way, while doing some coding, you'll also get an intuitive sense of how things can converge (or not) to their statistical predictions.

Learning objectives

- Understand Random Walks in One Dimension:
 - Gain a conceptual and practical understanding of random walks in a one-dimensional space, using coin tosses to simulate random movement on a lattice.
- Implement Periodic Boundary Conditions:
 - Learn how to implement periodic boundary conditions in a simulation, allowing an object to "wrap around" when it reaches the edges of a defined space.
- Simulate and Analyse Probabilistic Events:

- Develop skills to simulate a series of probabilistic events (coin tosses) and analyze their outcomes, including the convergence of relative frequencies to expected probabilities.
- Visual Representation of Data:
 - Practice visualizing the results of a random walk in a simple graphical format, enhancing understanding of the random drift over time.
- Modular Code Development Using Functions:
 - Improve coding skills by implementing modular code using functions and a main block to perform specific tasks, like tossing the coin and drawing the lattice.
- Statistical Convergence and Divergence:
 - Observe the phenomena of statistical convergence (e.g., fraction of heads) and divergence (e.g., random walk distance) in practical simulations.

These outcomes help you grasp key concepts in probability and stochastic processes, while reinforcing programming skills.

Background: random walk in 1D

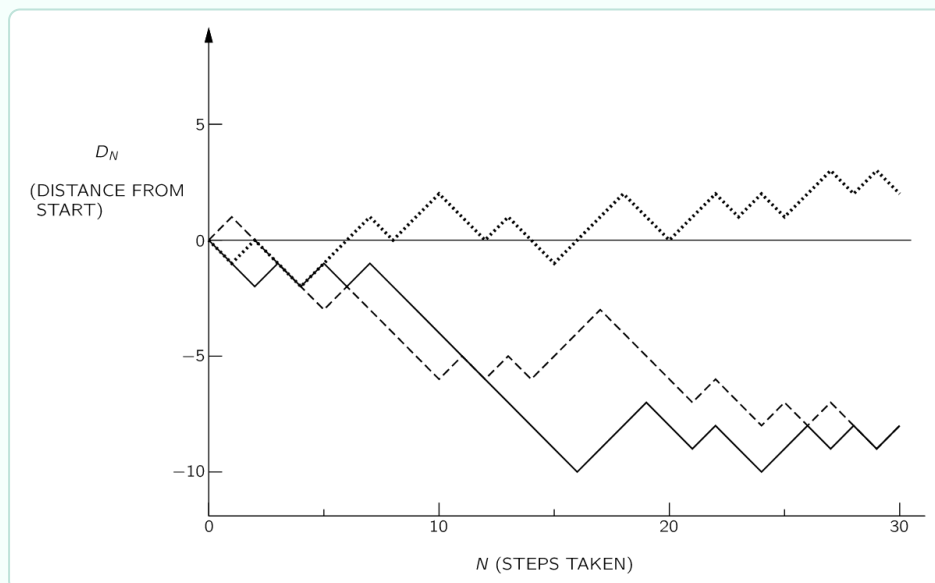


Figure 1: Random walk.

Look at Figure 1, which is taken from the *Feynman Lectures on Physics* (see the acknowledgements section). The figure shows the accumulated result of a sequence of coin tosses. At each step, N , a coin is tossed: heads means one step in one direction and tails means one step in the other. The sequence runs for a total of 30 steps in this instance.

The diagram shows the outcomes of three independent ‘experiments’, each conducted under the same conditions. At the initial step, $N = 0$, the distance from the origin is 0 in every case.

Let's consider a scenario that we can output to a console, to illustrate these concepts and set the context for the task. Imagine a coin, C , that sits in the middle of a lattice of size 51, at index 25 (counting from 0). This is equivalent to the starting position 0 in Figure 1:

console output

```
#####C#####
```

Over time, after 7 tosses, we would expect to see some random drift from the initial position and the lattice might look like this:

console output

```
#####C#####
#####C#####
#####C#####
#####C#####
#####C#####
#####C#####
#####C#####
#####C#####
```

Figure 2: Intended output example

Exercise 1: random walk in 1D

Create a script called `random_walk_1D.py`. Simulate a number, M , of coin tosses (for example, $M = 100$), assuming a non-biased coin ($P_{\text{HEADS}} = 1 - P_{\text{TAILS}} = 0.5$), on a lattice of size 51. N is the number of tosses so far, so it runs from 1 to M . At $N = 0$, the distance from the origin is 0. Each time the coin is tossed the ‘ C ’ changes position: one cell to the right for heads and one cell to the left for tails. Print the

lattice after every toss, as in Figure 2. Keep track of the *real* distance from the origin, but ensure that the 'C' graphic is wrapped around the lattice using the % operator. As the simulation progresses, store the observed fraction of heads in a variable, say `frac_heads`, which is a growing list that reaches size M at the end of the simulated sequence; each value in the list is given by $(\text{number_of_heads} / N)$. Print out the list at the end of the simulation and think about what is happening.

Periodic boundary: ensure that the lattice on which C is drifting has periodic boundaries at each edge – i.e., a boundary condition that simulates the lattice as if it extends infinitely in all directions. Instead of treating the lattice as a finite space with edges, a periodic boundary condition enforces the idea that the lattice "wraps around" itself. In other words, if you move beyond one edge of the lattice, you re-enter from the opposite edge.

Checking your output: because the tosses are random, your output will be different every time you run the script, so there is no single right answer to compare with. Some things should always be true, though. The 'C' moves exactly one cell on every line. The list `frac_heads` has M values, and the first is either 0.0 or 1.0. The early values jump around a lot and the later ones settle near 0.5. For example, one run with $M = 100$ gave a list that started [1.0, 0.5, 0.667, 0.5, 0.4, 0.5, 0.571, ...] and finished at 0.43 (values rounded to 3 decimal places).

A test with a known answer: temporarily set `P_HEADS = 1`, so that every toss is heads. Now you know exactly what should happen: the 'C' moves one cell to the right on every line, steps off the right-hand edge on toss 26 and re-enters at the left-hand edge. Every value in `frac_heads` is 1.0 and the real distance from the origin finishes at exactly M , even though the 'C' has wrapped around. Set `P_HEADS` back to 0.5 when you are done.

Exercise 2: split-screen output (optional extension)

Think creatively about your output. Maybe you could have ‘split screen’ graphics in your console, where half of the screen represents the coin ‘C’ and the other half is some representation of how far the latest value of `frac_heads` is from `P_HEADS`.

Whereas the drift associated with the real distance in Exercise 1 is a form of *divergence*, you should observe the value of `frac_heads` *converge* to `P_HEADS = 0.5`.

Finally, run your simulation several times and compare the final distance from the origin with the square root of M . What do you notice as you increase M ?

What you should find: any single run can finish close to the origin or a long way from it, but the typical distance is roughly the square root of M : around 10 for $M = 100$ and around 32 for $M = 1000$. So the distance keeps growing as M increases, but more slowly than M itself, and that is exactly why `frac_heads` converges: its error, $|\text{frac_heads} - \text{P_HEADS}|$, is the distance divided by $2M$, which shrinks towards 0.

Acknowledgements

Random walk images taken from The Feynman Lectures on Physics

Figure 1: https://www.feynmanlectures.caltech.edu/I_06.html