*5.3 SIMULATING PROBABILITY DISTRIBUTIONS

Pseudorandom number generators can be used to approximate various *probability distributions* that can be used to simulate natural phenomena. A probability distribution assigns probabilities, which are values in [0,1], to a set of random *elementary events*. The sum of the probabilities of all of the elementary events must be equal to 1. For example, a weather forecast that calls for a 70% chance of rain is a (predicted) probability distribution: the probability of rain is 0.7 and the probability of no rain is 0.3. Mathematicians usually think of probability distributions as assigning probabilities to numerical values or intervals of values rather than natural events, although we can always define an equivalence between a natural event and a numerical value.

Python provides support for several probability distributions in the `random` module. Here we will just look at two especially common ones: the uniform and normal distributions. Each of these is based on Python's `random` function.

The uniform distribution

The simplest general probability distribution is the *uniform distribution*, which assigns equal probability to every value in a range, or to every equal length interval in a range. A PRNG like the `random` function returns values in [0,1) according to a uniform distribution. The more general Python function `random.uniform(a, b)` returns a uniformly random value in the interval [a, b]. In other words, it is equally likely to return any number in [a, b]. For example,

```
>>> random.uniform(0, 100)
77.10524701669804
```

The normal distribution

Measurements of most everyday and natural processes tend to exhibit a "bell curve" distribution, formally known as a *normal* or *Gaussian* distribution. A normal distribution describes a process that tends to generate values that are centered around a mean. For example, measurements of people's heights and weights tend to fall into a normal distribution, as do the velocities of molecules in an ideal gas.

A normal distribution has two parameters: a mean value and a standard deviation. The standard deviation is, intuitively, the average distance a value is from the mean; a low standard deviation will give values that tend to be tightly clustered around the mean while a high standard deviation will give values that tend to be spread out from the mean.

The normal distribution assigns probabilities to intervals of real numbers in such a way that numbers centered around the mean have higher probability than those that are further from the mean. Therefore, if we are generating random values according to a normal distribution, we are more likely to get values close to the mean than values farther from the mean.

The Python function `random.gauss` returns a value according to a normal distribution with a given mean and standard deviation. For example, the following returns a value according to the normal distribution with mean 0 and standard deviation 0.25.

```
>>> random.gauss(0, 0.25)
-0.3371607214433552
```

Copyright Taylor and Francis, 2021

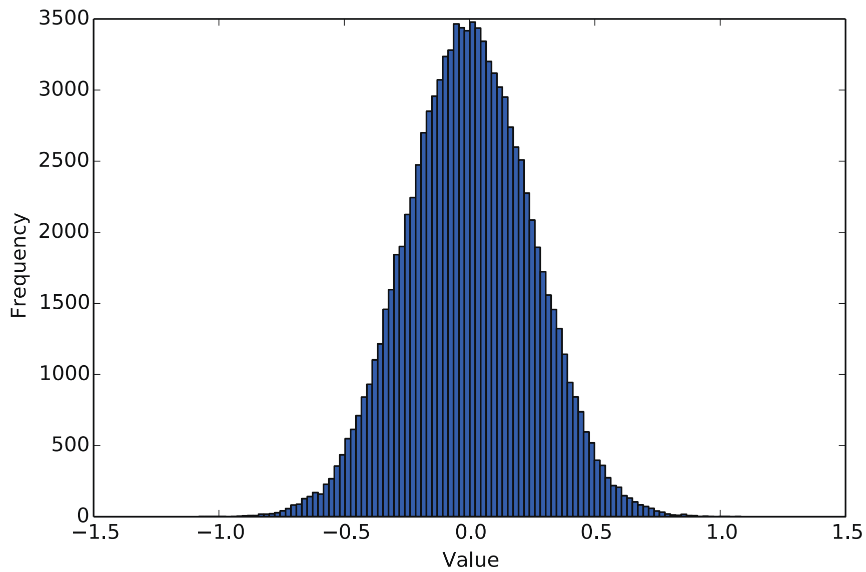


Figure 1 A histogram of 100,000 samples from `random.gauss(0, 0.25)`.

Figure 1 shows a ***histogram*** of 100,000 values returned by `random.gauss(0, 0.25)`. Notice that the “bell curve” of frequencies is centered at the mean of 0.

The central limit theorem

Intuitively, the normal distribution is so common because phenomena that are the sums of many additive random factors tend to be normal. For example, a person’s height is the result of several different genes and the person’s health, so the distribution of heights in a population tends to be normal. To illustrate this phenomenon, let’s create a histogram of the sums of numbers generated by the `random` function. Below, the `sumRandom` function returns the sum of `n` random numbers in the interval [0,1). The `sumRandomHist` function creates a list named `samples` that contains `trials` of these sums, and then plots a histogram of them.

```
import matplotlib.pyplot as pyplot
import random

def sumRandom(n):
    """Returns the sum of n pseudorandom numbers in [0,1).

    Parameter:
        n: the number of pseudorandom numbers to generate

    Return value: the sum of n pseudorandom numbers in [0,1)
    """

    total = 0
    for index in range(n):
        total = total + random.random()
    return total

def sumRandomHist(n, trials):
    """Displays a histogram of sums of n pseudorandom numbers.

    Parameters:
        n:      the number of pseudorandom numbers in each sum
        trials: the number of sums to generate

    Return value: None
    """

    samples = [ ]
    for index in range(trials):
        samples.append(sumRandom(n))
    pyplot.hist(samples, 100)
    pyplot.show()
```

Reflection 1 *Call* `sumRandomHist` *with 100,000 trials and values of* `n` *equal to 1, 2, 3, and 10. What do you notice?*

When we call `sumRandomHist` with `n = 1`, we get the expected uniform distribution of values, shown in the upper left of Figure 2. However, for increasing values of `n`, we notice that the distribution of values very quickly begins to look like a normal distribution, as illustrated in the rest of Figure 2. In addition, the standard deviation of the distribution (intuitively, the width of the "bell") appears to get smaller relative to the mean.

The mathematical name for this phenomenon is the *central limit theorem*. Intuitively, the central limit theorem implies that any large set of independent measurements of a process that is the cumulative result of many random factors will look like a normal distribution. Because a normal distribution is centered around a mean, this also implies that the average of all of the measurements will be close to the true mean.

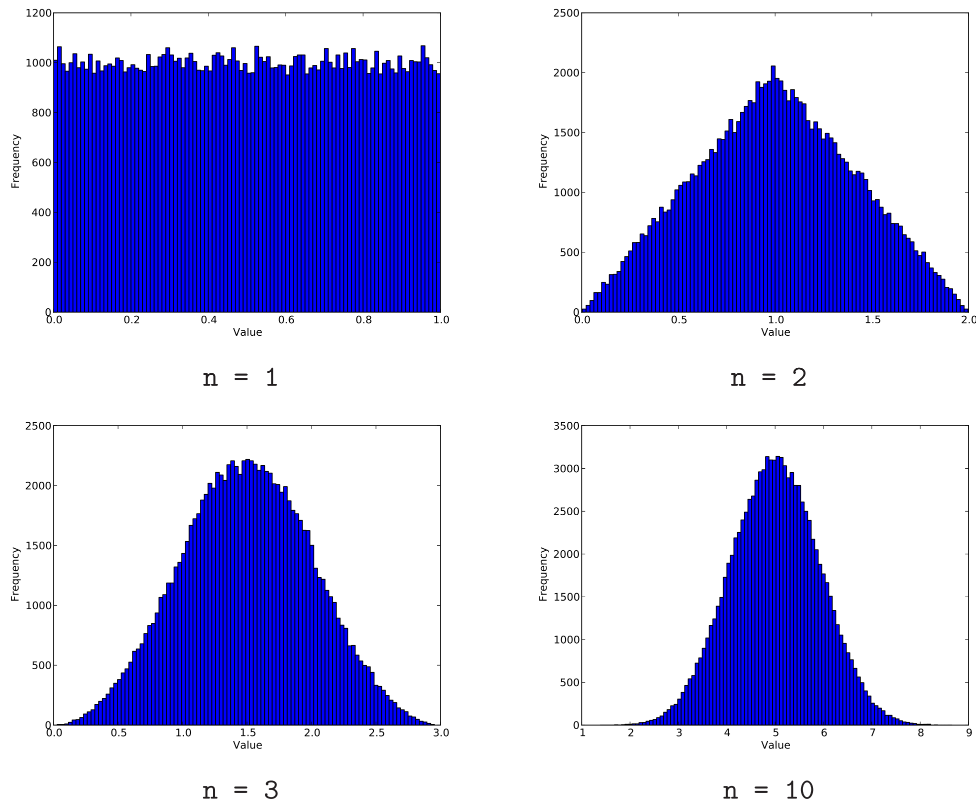


Figure 2 Results of `sumRandomHist(n, 100000)` with `n = 1, 2, 3, 10`.

Exercises

5.3.1* A more realistic random walk has the movement in each step follow a ***normal distribution***. In particular, in each step, we can change both x and y according to a normal distribution with mean 0. Because the values produced by this distribution will be both positive and negative, the particle can potentially move in any direction. To make the step sizes small, we need to use a small standard deviation, say 0.5:

```
x = x + random.gauss(0, 0.5)
y = y + random.gauss(0, 0.5)
```

Modify the `randomWalk` function so that it updates the position of the particle in this way instead. Then use the `rwMonteCarlo` and `plotDistances` functions to run a Monte Carlo simulation with your new `randomWalk` function. As we did earlier, call `plotDistances(1000, 10000)`. How do your results differ from the original version?

5.3.2* Write a function

```
uniform(a, b)
```

that returns a number in the interval [`a`,`b`) using only the `random.random` function. (Do not use the `random.uniform` function.)

5.3.3. Suppose we want a pseudorandom integer between 0 and 7 (inclusive). How can we use the `random.random()` and `int` functions to get this result?

5.3.4. Write a function

```
randomRange(a, b)
```

that returns a pseudorandom integer in the interval [`a..b`] using only the `random.random` function. (Do not use `random.randrange` or `random.randint`.)

5.3.5. Write a function

```
normalHist(mean, stdDev, trials)
```

that produces a histogram of `trials` values returned by the `gauss` function with the given mean and standard deviation. In other words, reproduce Figure 1. (This is very similar to the `sumRandomHist` function.)

5.3.6. Write a function

```
uniformHist(a, b, trials)
```

that produces a histogram of `trials` values in the range [`a`,`b`] returned by the `uniform` function. In other words, reproduce the top left histogram in Figure 2. (This is very similar to the `sumRandomHist` function.)

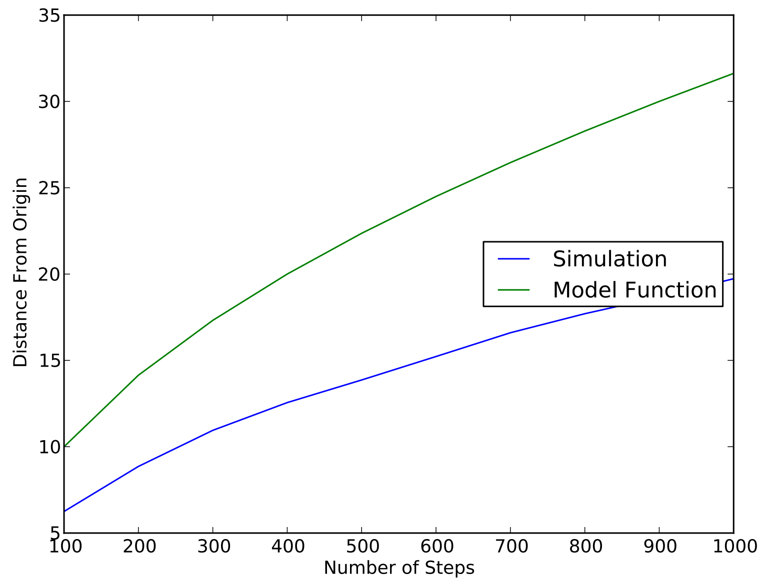
5.3.7. Write a function

```
plotChiSquared(k, trials)
```

that produces a histogram of `trials` values, each of which is the sum of k squares of values given by the `random.gauss` function with mean 0 and standard deviation 1. (This is very similar to the `sumRandomHist` function.) The resulting probability distribution is known as the *chi-squared distribution* (χ^2 distribution) with k degrees of freedom.

Selected Exercise Solutions

5.3.1
```
def randomWalk(steps, tortoise, draw):
    x = 0
    y = 0
    moveLength = 10
    for step in range(steps):
        x = x + random.gauss(0, 0.5)
        y = y + random.gauss(0, 0.5)
        if draw:
            tortoise.goto(x * moveLength, y * moveLength)
    return distance(0, 0, x, y)
```



5.3.2
```
def uniform(a, b):
    return a + random.random() * (b - a)
```