Yaning Liu, Giray Ökten

Probability and Simulation: A Python Companion

– Monograph –

September 4, 2020

Springer Nature

To my son and daughter Gavin and Evangeline Liu - YL

Preface

In *Probability and Simulation* the Julia programming language was used throughout the book. In this companion book, we provide translations of the Julia codes used in *Probability and Simulation* to Python.

Each chapter in this book presents the Python codes corresponding to the Julia codes in the corresponding chapter of *Probability and Simulation*. If a project involved programming, we included the entire project statement followed by its solution. If a section of *Probability and Simulation* involved programming, we used the same section title and provided the Python translations.

We hope the companion book will be useful to instructors and students who prefer Python programming language.

Denver, Colorado
July 2020 *Yaning Liu*

Tallahassee, Florida
July 2020 *Giray Ökten*

Contents

Chapter 1
Probability

In Chapter 1 of *Probability and Simulation*, Julia programming was used in Project 1 and Project 2. Below are the complete project statements, followed by their solutions, using Python programming language.

1.1 Project 1: Verifying identities using Python

1. Write a Python function [1] called **verifyid** such that:
   - The function takes three inputs: the polynomials F, G, and the degree of F (and G), call it d.
   - The output is one of the statements: "the polynomials are equivalent" or "the polynomials are not equivalent".

   Here are more details and hints for the code:
   - Initialize a Boolean variable **flag** as flag = True, and initialize $n = 1$.
   - Write a **while** statement that does the following while flag = True and $n \leq 10$:
     - generate a random integer r between 1 and $100d$; this can be done by $r = numpy.random.randint(1, 100 * d + 1)$
     - set **flag** to the truth value of $F(r) == G(r)$
     - increment $n = n + 1$
   - After the **while** statement is executed, print "the polynomials are equivalent" if flag = True. Otherwise print "the polynomials are not equivalent".
2. What is the probability that the function **verifyid** returns "the polynomials are equivalent" when in fact they are not equivalent?
3. Take turns to test your team member's **verifyid** code as follows:
   a. Make up a product, such as $F(x) = (x - 5)(x - 10)(x + 3)(x - 2)(x + 25)$.
   b. Type this in WolframAlpha to find an equivalent expression in standard form. For the above, it is $G(x) = x^5 + 11x^4 - 321x^3 + 865x^2 + 3200x - 7500$.
   c. You have two options: either report F and G as they are to your teammate, or make a little change in G and report it.
   d. Your teammate will check if F and G are equivalent or not.

Solution 1.1

The probability that **verifyid** returns the "the polynomials are equivalent" when in fact they are not equivalent is $(\frac{1}{100})^{10}$.

```
In [1]: def verifyid(F, G, d):
            flag = True
```

[1] A tutorial on Python can be found in Chapter 1 of *First Semester in Numerical Analysis with Python*, http://digital.auraria.edu/l/IR00000195/00001, an open access textbook written by the authors.

```
            n = 1
            while flag is True and n<=10:
                r = np.random.randint(1, 100*d+1)
                flag = (F(r) == G(r))
                n = n+1
            print('n is ', n)
            if flag == True:
                print('The two polynomials are equivalent')
            else:
                print('The two polynomials are not equivalent')
```

```
In [2]: F = lambda x : (x-5)*(x-10)*(x+3)*(x-2)*(x+25)
        G = lambda x : x**5+11*x**4-321*x**3+865*x**2+3200*x-7500
```

```
In [3]: verifyid(F, G, 5)
```

```
n is  11
The two polynomials are equivalent
```

Now let's pick a wrong polynomial:

```
In [4]: H = lambda x : x**5+11*x**4-321*x**3+865*x**2+3200*x-7501
```

```
In [5]: verifyid(F,H,5)
```

```
n is  2
The two polynomials are not equivalent
```

1.2 Project 2: Analysis of Freivalds' algorithm

1. By repeating Freivalds' algorithm many times, we can make the probability of failure much smaller. Suppose we apply the algorithm 100 times, by generating random vectors $r^{(1)}, r^{(2)}, \ldots, r^{(100)}$. Discuss when the algorithm will give the wrong answer in this case, and its probability.
2. How would the conclusion of Theorem 1.1 change if the components of the random vector r were chosen uniformly from $\{0, 1, 2\}$? How about from $\{0, 1, \ldots, d\}$?
3. In Theorem 1.1, we proved that $P(Dr = 0) \leq 1/2$ when $D \neq 0$. In Example 1.1, we observed that this bound is tight: the upper bound $1/2$ was attained for a square matrix of size 2. We want to investigate how the maximum value of $P(Dr = 0)$ changes as the dimension of the matrix D increases. In order to do this, we will write a code that estimates $P(Dr = 0)$ where r is a vector matching the dimension of D, with components generated at random uniformly from $\{0, 1, \ldots, d\}$. Here is an outline for a Python function called **bound** with some hints:

   - The function **bound** should take D, d as inputs. The output will be an approximation for $P(Dr = 0)$. Initialize count = 0, and find the dimension of the matrix (Numpy 2D array) by $m = D.shape[1]$.
   - Generate $r^{(1)}, \ldots, r^{(10000)}$ at random, where the components of r are randomly generated numbers from $\{0, 1, \ldots, d\}$. Generating one such vector can be done by the for loop below (notice the code first initializes r as a vector of m 1's, and then overwrites its components):

     ```
     In [ ]: r = numpy.ones(m)
             for j in range(m):
                 r[j] = numpy.random.randint(0, d+1)
     ```

   - Count how many vectors $r^{(i)}$ gives $Dr^{(i)} = 0$, where 0 is the zero vector. This can be done by (below **numpy.zeros(m)** is a vector of m 0's):

     ```
     In [ ]: if numpy.array_equal(numpy.dot(D, r), numpy.zeros(m)):
                 count += 1
     ```

- Return count/10000.
  Use your code to estimate $P(Dr = 0)$ when D is a randomly generated matrix. Here is how to generate a random matrix of dimension 3, where the matrix entries are randomly generated from $\{0, 1, \ldots, 9\}$:

```
In [1]: D = numpy.random.randint(0, 10, (3,3))
        D
Out[1]: array([[3, 6, 5],
               [0, 3, 3],
               [1, 5, 9]])
```

Experiment with matrices with different dimensions. For each dimension, generate several random matrices to find the maximum value of the various estimates you get for $P(Dr = 0)$, and check how this maximum changes as the dimension increases.

Solution 1.2

1. • Generate $r^{(1)}, r^{(2)}, \ldots, r^{(100)}$ at random from $\{0, 1\}^n$
   • Compute $A(Br^{(k)}) - Cr^{(k)}$ for each $k = 1, \ldots, 100$.
   • If $A(Br^{(k)}) - Cr^{(k)} = 0$ for all k, declare $AB = C$. Otherwise declare $AB \neq C$
   Let E_k be the event that the algorithm gives the wrong answer at step k, i.e., $A(Br^{(k)}) = Cr^{(k)}$ but $AB \neq C$. We know that $P(E_k) \leq 1/2$ from Theorem 1.1.
   Let $E = \bigcap_{k=1}^{100} E_k$ - this is the event where the algorithm gives $A(Br^{(k)}) = Cr^{(k)}$ for all k but $AB \neq C$. Since events E_k are independent, we have
   $$P(E) = P(\bigcap_{k=1}^{100} E_k) = \prod_{k=1}^{100} P(E_k) \leq \left(\frac{1}{2}\right)^{100}.$$
2. The upper bound for the probability in the theorem will be 1/3 instead of 1/2. For the general case it will be $1/(d+1)$.
3.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: def bound(D, d):
            count = 0
            m = D.shape[1]
            n = 10000
            zero_mat = np.zeros(m)
            for k in range(n):
                u = np.ones(m)
                for j in range(m):
                    u[j] = np.random.randint(0, d+1)
                if np.array_equal(np.dot(D, u), zero_mat):
                    count += 1
            return count/n
```

```
In [3]: D = np.random.randint(0, 10, (2,2))
        print(D)
```

```
[[2 4]
 [4 7]]
```

```
In [4]: bound(D, 1)
```

```
Out[4]: 0.2491
```

```
In [5]: bound(D, 2)
```

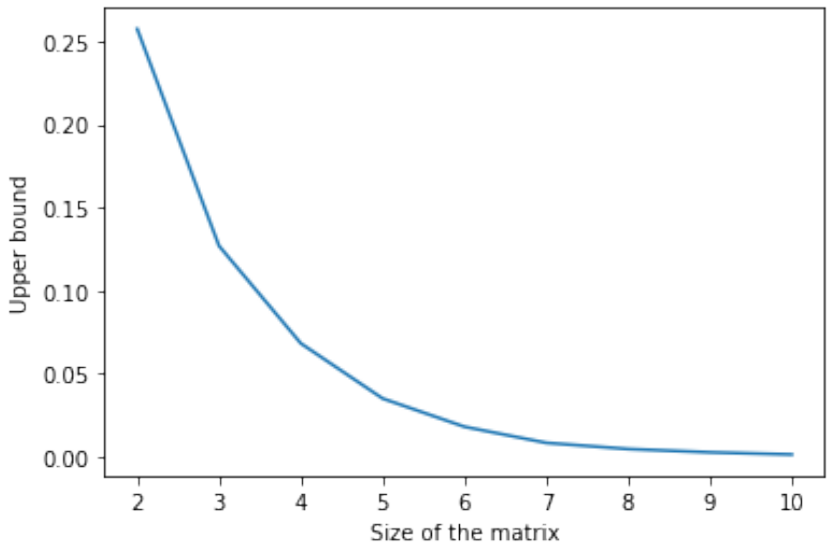
```
Out[5]: 0.1145
```

```
In [6]: for i in range(10):
            D = np.random.randint(0, 10, (2,2))
            print(bound(D, 1))
```

```
0.2551
0.2493
0.2532
0.2521
0.2505
0.2465
0.2549
0.2522
0.254
0.252
```

```
In [7]: mx = np.empty(0)
        for s in range(2, 11):
            vals = np.empty(0)
            for i in range(10):
                D = np.random.randint(0, 10, (s,s))
                vals = np.append(vals, bound(D,1))
            mx = np.append(mx, vals.max())
```

```
In [8]: plt.plot(np.arange(2, 11), mx)
        plt.xlabel('Size of the matrix')
        plt.ylabel('Upper bound')
```



Chapter 2
Discrete Random Variables

In Example 2.3 of Section 2.1 of *Probability and Simulation*, random harmonic series was discussed using Julia. Below we present the same discussion using Python programming.

Example 2.1 (Random Harmonic Series)

In Calculus, you learned that the harmonic series $\sum_{n=1}^{\infty} \frac{1}{n}$ is divergent. On the other hand, the alternating series $\sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n}$ is convergent:

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots = \infty$$
$$1 - \frac{1}{2} + \frac{1}{3} - \ldots = \log 2$$

Consider the following generalization: $\sum_{n=1}^{\infty} \frac{a_n}{n}$ where a_n is equal to 1 or -1 with probability 1/2 (in other words, a_n is a discrete uniform random variable on $-1, 1$.) What can be said about the convergence of this **random harmonic series**?

Let's investigate this question numerically. Using Python, we will generate 50000 random a_n's to compute $S = \sum_{n=1}^{50000} \frac{a_n}{n}$. We will then repeat this 1000 times to obtain $S^{(1)}, S^{(2)}, \ldots, S^{(1000)}$, and plot a histogram for these values.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We first write a function, **rnd**, which generates the numbers -1 and 1 at random with equal probability:

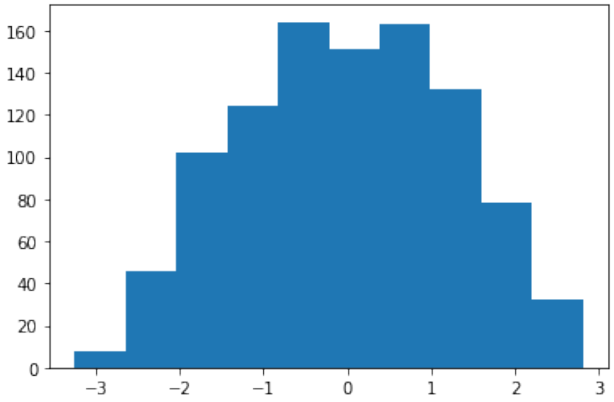
```
In [2]: def rnd():
            x = np.random.rand()
            if x < 0.5:
                return -1.0
            else:
                return 1.0
```

The next function **psum** computes the partial sum $S = \sum_{k=1}^{50000} \frac{a_k}{k}$:

```
In [3]: def psum(n):
            sums = np.zeros(n)
            i = 1
            while i<=n:
                s = 0.0
                for k in range(50000):
                    s += rnd()/(k+1)
                sums[i-1] = s
                i += 1
            plt.hist(sums, 10)
```

Now we compute 1000 partial sums and plot their histogram.

```
In [4]: psum(1000)
```



According to the histogram, most of the partial sums seem to be between −3 and 3. This numerical evidence may be in support of the convergence of the random harmonic series, however, the definitive answer can be given only by theoretical means. In fact, Hans Rademacher[1] proved the convergence of this series in 1922.

Theorem 2.1 (Rademacher, 1922) *If* $\sum_{n=1}^{\infty} c_n^2 < \infty$, *then* $\sum_{n=1}^{\infty} a_n c_n$ *exists with probability 1, where* $a_n = 1$ *or* -1 *with probability 1/2.*

2.1 Project 5: Benford's law

Data is money, it is precious. It runs the engines of many businesses in today's economy. Here is a riddle involving data: if you wrote down the population of all cities, or counties, in the U.S., and examined the first digits of these numbers, would you see each number 1 through 9 appearing as the first digit in roughly equal proportions, or would some numbers appear more often than others? To investigate this question, we download the county population data from the U.S. Census Bureau webpage (`factfinder.census.gov`), and use Python to analyze the distribution of the digits. The data "population_county.csv" and the Python notebook "Benfords Law US population.ipynb" are available on the Springer webpage for the book. The Python notebook can also be found in Appendix A. Here we summarize the results obtained in the Python notebook.

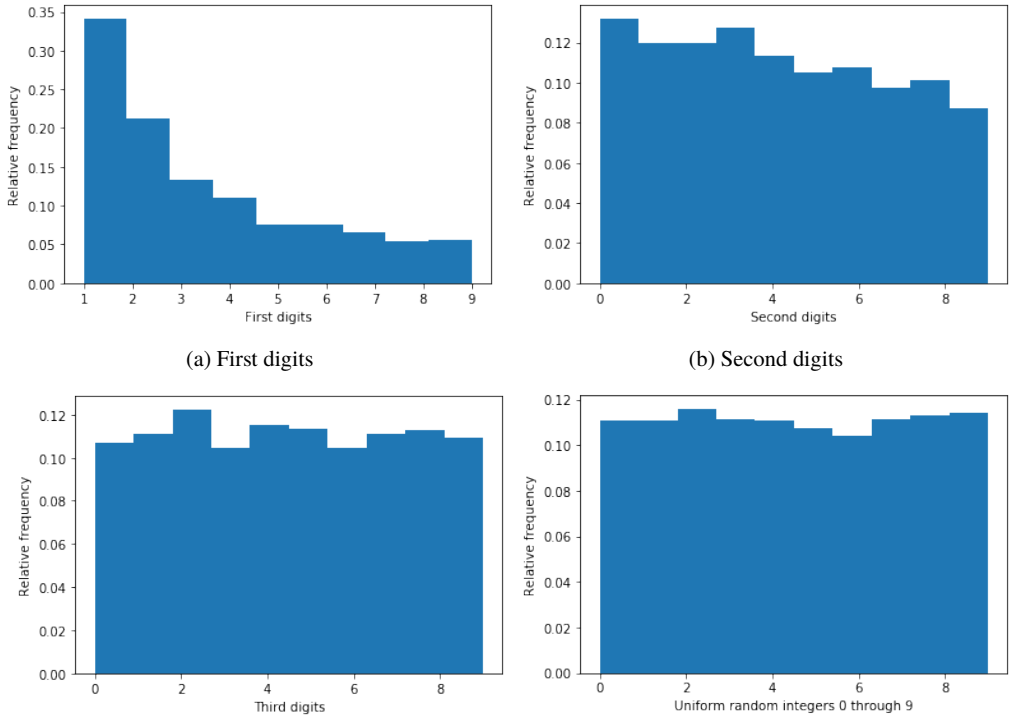
A simple graphical approach to get a sense of the digit distribution is to plot its histogram. There are 3142 numbers in our data. We extract the first digits of these numbers, and plot their relative frequency histogram. We repeat this experiment for the second and third digits as well. (A few populations with two digits were removed from the data.) Figure 2.1 plots these histograms. The top-left histogram is for the first digits, and it clearly shows the distribution is not uniform. Smaller digits appear more often than the larger ones. The distribution of the second digits (top-right) seems to have the same pattern, but less pronounced. The pattern seems to disappear by the third digits: the histogram for the third digits (bottom-left) and the histogram of 3142 random integers 0 through 9 from the uniform distribution (bottom-right) look very alike.

Similar observations were made by Newcomb [4] in 1881, and Benford [1] in 1938, when they examined the digit distributions of some data sets. Newcomb suggested the following probability mass function for the first digit

$$P(\text{first digit } = d) = \log_{10}\left(1 + \frac{1}{d}\right) \tag{2.1}$$

for $d = 1, 2, \ldots, 9$. Benford gave examples of many data sets that follow this probability mass function; this phenomenon is known as the Benford's law, or Benford-Newcomb law.

[1] Rademacher, H., 1922. Einige Sätze über Reihen von allgemeinen Orthogonalfunktionen. Mathematische Annalen, 87(1-2), pp. 112-138.



(a) First digits

(b) Second digits

(c) Third digits

(d) Randomly generated digits

Fig. 2.1: Distributions of digits of county population data and uniform random numbers

Newcomb also investigated the distribution of the second leading digits. He suggested the following probabilities for them

$$P(\text{second digit } = d) = \sum_{k=1}^{9} \log_{10}\left(1 + \frac{1}{10k + d}\right)$$

for $d = 0, 1, \ldots, 9$.

When do we expect data to follow Benford's law? Some intuitive explanations are:

- When data describe the sizes of similar phenomena, with no artificial minimum or maximum constraints, such as weights, distances, temperatures, and dollars, of things. For such data, units are irrelevant, that is data should follow the law independent of the measurement units.
- When data come from different distributions, that is data is a mixture of numbers coming from different populations with different distributions.

Figure 2.2 plots the relative frequency histogram for the first digits of the population data together with the probability mass function (2.1) for the first digits. The fit of the mass function to the data is quite well.

Benford's law can be used to detect fraud. For example, it is widely reported that accounting data follows Benford's law. Then a significant deviation from Benford's law might point to possible fraud. Applications of Benford's law to forensic accounting is the subject of a book by Nigrini [5]. Some researchers claim Benford's law for the first digits can also be used to detect election fraud, although not everyone agrees. Some argue that Benford's law for the second digits is a better choice for election data. Figure 2.3 plots the county by county votes for Democrats in the 2016 U.S. presidential election together with the Benford's probabilities for the first digit, and the fit to the Benford's law looks perfect.

Iran's 2009 presidential election was very controversial. The incumbent Mr. Ahmadinejad won the election with 62% of the vote, while his main challenger Mr. Mousavi received 34% of the vote. Critics claimed widespread election fraud, and street protests started shortly after the election results were announced.

The data "Iran_pres_2009.csv" contains the votes from the ballot boxes, and obtained from http://thomaslotze.com/iran/#Sevens. Download the data and the Python notebook "Benford's Law US population.ipynb", from the

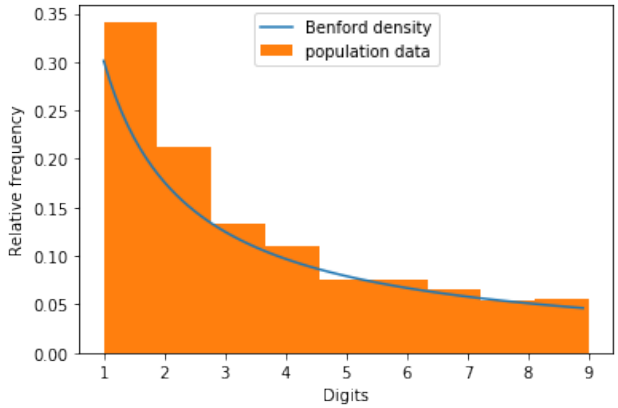


Fig. 2.2: Distribution of first digits of county population data with Benford's probability mass function for the first digits

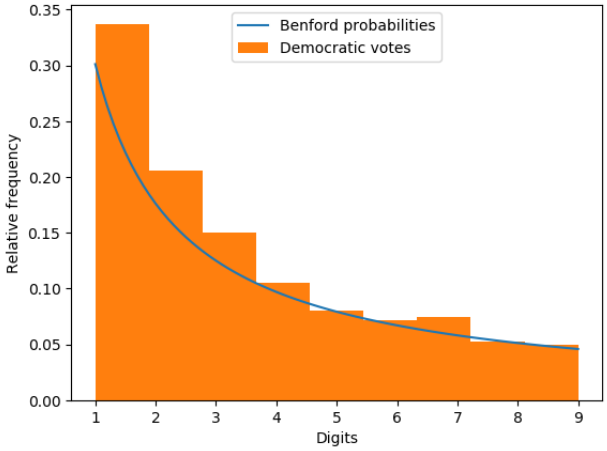


Fig. 2.3: Distribution of first digits of county by county votes for Democrats in 2016 election

Springer webpage for the book, and save them into the same directory. Modify the Python notebook "Benford's Law US population.ipynb" to answer the following questions:

1. The number of votes received by Mr. Ahmadinejad and Mr. Mousavi are given in the columns with headings **Ahmadinejad** and **Mousavi**. Plot relative frequency histograms for the first digits of the votes. Then superimpose the Benford probability mass function for the first digits to the histograms. (In some ballot boxes candidates received no votes. To remove the zeros from the first digit data, use **first_digits_nz = first_digits[first_digits>0]**.)
2. Comment on the fit of the data to Benford's law for Mr. Ahmadinejad and Mr. Mousavi. Do you think the results support possible election fraud?

Solution 2.1

Here is the Python code and the histograms. Load the packages numpy, PyPlot and pandas before running this code.

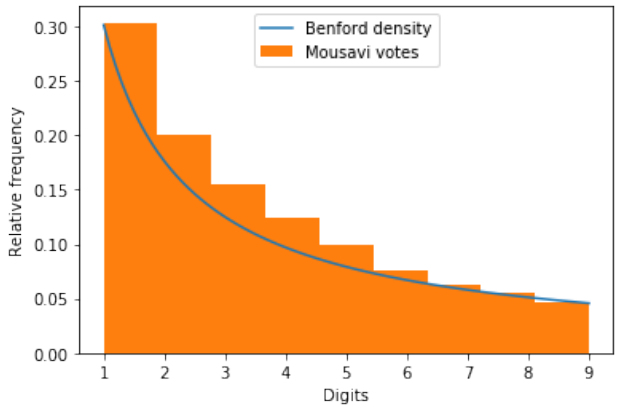
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import pandas as pd

In [2]: p = lambda d : np.log10(1+1/d)
```

```
In [3]: data = pd.read_csv("Iran_pres_2009.csv")
```

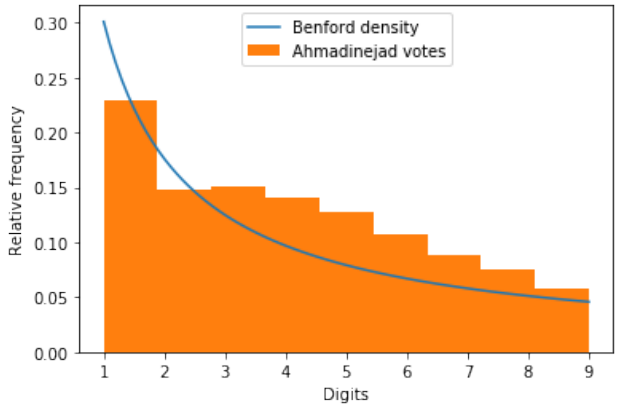
```
In [4]: res = data['Mousavi']
        first_digits = np.array([int(str(n)[0]) for n in res])

        first_digits_nz = first_digits[first_digits>0]
        xaxis = np.linspace(1, 9, 81)
        yvals = p(xaxis)
        plt.plot(xaxis, yvals, label='Benford density')
        plt.xlabel('Digits')
        plt.ylabel('Relative frequency')
        plt.hist(first_digits_nz, 9, density=True, label='Mousavi votes')
        plt.legend(loc='upper center');
```




```
In [5]: res = data['Ahmadinejad']
        first_digits = np.array([int(str(n)[0]) for n in res])

        first_digits_nz = first_digits[first_digits>0]
        xaxis = np.linspace(1, 9, 81)
        yvals = p(xaxis)
        plt.plot(xaxis, yvals, label='Benford density')
        plt.xlabel('Digits')
        plt.ylabel('Relative frequency')
        plt.hist(first_digits_nz, 9, density=True, label='Ahmadinejad votes')
        plt.legend(loc='upper center');
```



Chapter 3
Continuous Random Variables

3.1 Project 8: Monte Carlo integration

Estimation of the integral $I = \int_{(0,1)^s} g(x)dx$ is a classical problem in numerical analysis known as the quadrature problem. The Monte Carlo integration refers to algorithms that estimate this integral using random numbers. Here we will discuss two such methods.

Hit-or-miss Monte Carlo

Let g be a function from (0, 1) to (0, 1). We want to estimate $I = \int_0^1 g(x)dx$, which is the area under the graph of g; see Figure 3.1.

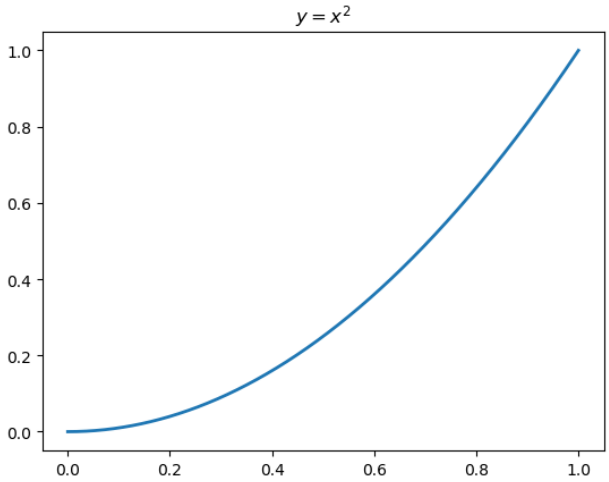


Fig. 3.1: Finding the area under $y = x^2$ with hit-or-miss Monte Carlo

Now suppose we generate N uniform random numbers from the unit square, and count how many fall under the graph of g - say this number is S. Then the area under g is approximately $\frac{S}{N}$, that is

$$I = \int_0^1 g(x)dx \approx \frac{S}{N}.$$

1. Plot $y = x^2$ on a board, enclosed within a unit square, and take turns to throw 40 darts at the unit square.[1] Count S, N and compute $\frac{S}{N}$. How close is it to the exact area $\int_0^1 x^2 dx = \frac{1}{3}$?
2. Let's analyze this method theoretically. Let

$$X_i = \begin{cases} 1 \text{ if the } i\text{th dart falls under the graph} \\ 0 \text{ otherwise.} \end{cases}$$

   a. What is the probability $P\{X_i = 1\}$ for any i?
   b. Write S in terms of X_i. What is the distribution of the random variable S?
   c. Find the expectation $E[S/N]$.
   d. Find the variance $Var(S/N)$.

Sample mean Monte Carlo

Let $U_1, U_2, \ldots, U_N, \ldots$ be independent uniform random variables on $(0, 1)$. Then the sample mean

$$\frac{1}{N}\sum_{i=1}^{N} g(U_i) = \frac{g(U_1) + \ldots + g(U_N)}{N}$$

is an approximation to the expectation

$$E[g(U)] = \int_0^1 g(x)dx = I,$$

with the error of the approximation converging to 0 (with probability 1) as $N \to \infty$. This follows from the strong law of large numbers we discussed earlier.

1. Write a Python code to approximate $\int_0^1 e^{x^2} dx$ using sample mean Monte Carlo. The code should take N as the input, and return $(g(U_1) + \ldots + g(U_N))/N$ as the output. The U_i's will be replaced by random numbers in your code. Recall that the function **numpy.random.rand()** generates a random number from the uniform distribution on $(0, 1)$. Run your code with $N = 1000$ and $N = 10,000$. Compare your results with the approximation WolframAlpha gives for the integral.
2. The sample mean Monte Carlo generalizes to higher dimensions in a straightforward way. If g is a function on $(0, 1)^s$, then

$$\frac{1}{N}\sum_{i=1}^{N} g(U_i) \approx E[g(U)] = \int_{(0,1)^s} g(x)dx = I$$

   where U_i is an s-dimensional vector, with components independent uniform random variables on $(0, 1)$. Write a Python code to estimate $\int_0^1 \int_0^1 e^{(x+y)^2} dxdy$.
3. Sample mean Monte Carlo can be generalized to estimate integrals defined on a finite interval (a, b). Prove that

$$\int_a^b g(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} g(a + (b-a)u_i)$$

   where u_i are uniform random numbers from $(0, 1)$. (Hint: Use the substitution $x = \frac{y-a}{b-a}$ to turn the integral to one with domain $(0, 1)$.)
4. Sample mean Monte Carlo is a better method than hit-or-miss Monte Carlo. These methods estimate $\int_0^1 g(x)dx$ by computing
   - S/N in hit-or-miss MC
   - $\frac{1}{N}\sum_{i=1}^{N} g(U_i)$ in sample mean MC.

[1] YSP students at Florida State University shoot a nerf gun aiming at the unit square drawn on the blackboard.

Claiming that sample mean Monte Carlo is better than hit-or-miss Monte Carlo means the variance of its estimates is smaller than the variance of the estimates of hit-or-miss, that is

$$Var\left(\frac{1}{N}\sum_{i=1}^{N} g(U_i)\right) \leq Var(S/N).$$

Prove this inequality.

Solution 3.1

Hit-or-miss Monte Carlo

Solutions to problem 2:

1. $P\{X_i = 1\} = I$.
2. $S = \sum_{i=1}^{N} X_i$. S is a binomial distribution with parameters N and probability of success I.
3. $E[S] = NI$, and thus $E[S/N] = I$.
4. $Var(\frac{S}{N}) = \frac{1}{N^2}Var(S) = \frac{1}{N^2}I(1-I)N = \frac{I(1-I)}{N}$.

Sample mean Monte Carlo

1. ```
   In [1]: import numpy as np
   ```

   ```
   In [2]: def mc(N):
               sum = 0.
               for i in range(N):
                   sum += np.exp(np.random.rand()**2)
               return sum/N
   ```

   ```
   In [3]: mc(1000)
   ```

   ```
   Out[3]: 1.439953251926987
   ```

   ```
   In [4]: mc(10000)
   ```

   ```
   Out[4]: 1.4615463223140084
   ```

   Let's also code the basic Simpson's quadrature to compare with Monte Carlo.

   ```
   In [5]: def simpson(f, a, b):
               h = (b-a)/2
               return h*(f(a)+4*f(a+h)+f(b))/3
   ```

   ```
   In [6]: simpson(lambda x: np.exp(x**2), 0, 1)
   ```

   ```
   Out[6]: 1.4757305825350018
   ```

2. Next problem is $\int_0^1 \int_0^1 e^{(x+y)^2} dxdy$.

   ```
   In [7]: def mct(N):
               sum = 0.
               for i in range(N):
                   sum += np.exp((np.random.rand()+np.random.rand())**2)
               return sum/N
   ```

   ```
   In [8]: mct(1000)
   ```
   ```

```
Out[8]: 5.014005358322982

In [9]: mct(10000)

Out[9]: 4.901648872047895
```

Wolfram alpha's approximation for the above integral is: 4.89916.

3. This is for the integral of any function defined on (a, b).

```
In [10]: def mc(g, a, b, N):
             sum = 0.
             for i in range(N):
                 sum += g(a+(b-a)*np.random.rand())*(b-a)
             return sum/N
```

4. Note that

$$Var\left(\frac{1}{N}\sum_{i=1}^{N} g(U_i)\right) = \frac{1}{N}\left(\int g^2(x)dx - I^2\right)$$

and from part 1

$$Var(S/N) = \frac{1}{N}I(1-I).$$

Then it suffices to prove $\int g^2(x)dx \leq I = \int g(x)dx$. This statement is true since $0 < g^2(x) \leq g(x)$, which follows from $0 < g(x) < 1$.

3.2 Exponential and normal random variables

In Section 3.3 of *Probability and Simulation*, Julia was used to analyze exponential and normal distributions. We present a Python version of this discussion next.

Exponential distribution in Python

We start with loading the subpackage **stats** from scipy which contains a large collection of probability distributions.

```
In [1]: from scipy import stats

In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Let's define X as an exponential random variable with $\theta = 2$.

```
In [3]: X = stats.expon(scale=2)
```

The pdf and cdf of X can be evaluated by simply using the Python functions:

```
In [4]: X.pdf(0.1)

Out[4]: 0.475614712250357

In [5]: X.cdf(3)

Out[5]: 0.7768698398515702
```

Next we plot the pdf and cdf of X:

```
In [6]: xval = np.linspace(0, 10, 100)
        y = X.pdf(xval)
        z = X.cdf(xval)
        plt.plot(xval, y, label='PDF')
        plt.plot(xval, z, label='CDF')
        plt.legend(loc='upper right');
```

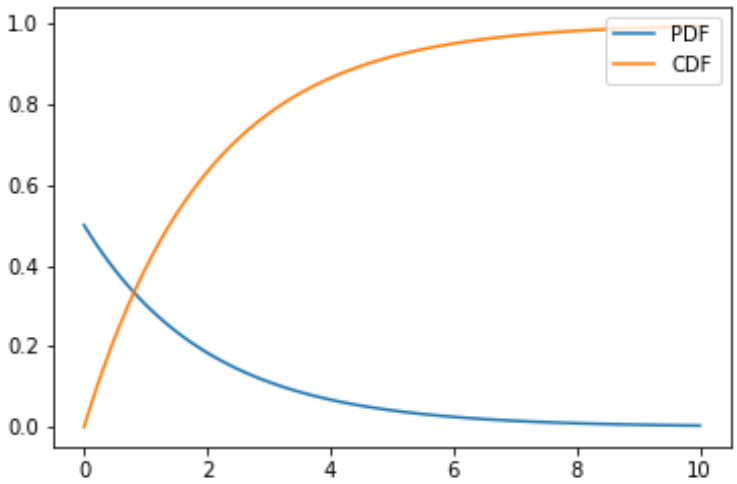


Fig. 3.2: The pdf and cdf of the exponential random variable with $\theta = 2$

Generating random numbers from exponential distribution

The function **X.rvs()** generates a random number from the distribution X.

```
In [7]: X.rvs()
```

```
Out[7]: 0.10264495621736651
```

Multiple random numbers from X can be obtained simply by:

```
In [8]: X.rvs(size=5)
```

```
Out[8]: array([0.12616957, 2.04605907, 0.32213927, 7.72715273, 0.29900646])
```

Below we generate 10000 random numbers and plot their relative frequency histogram in Figure 3.3. Notice how the histogram matches the shape of the exponential probability density function in Figure 3.2.

```
In [9]: plt.hist(X.rvs(size=10000), density=True);
```

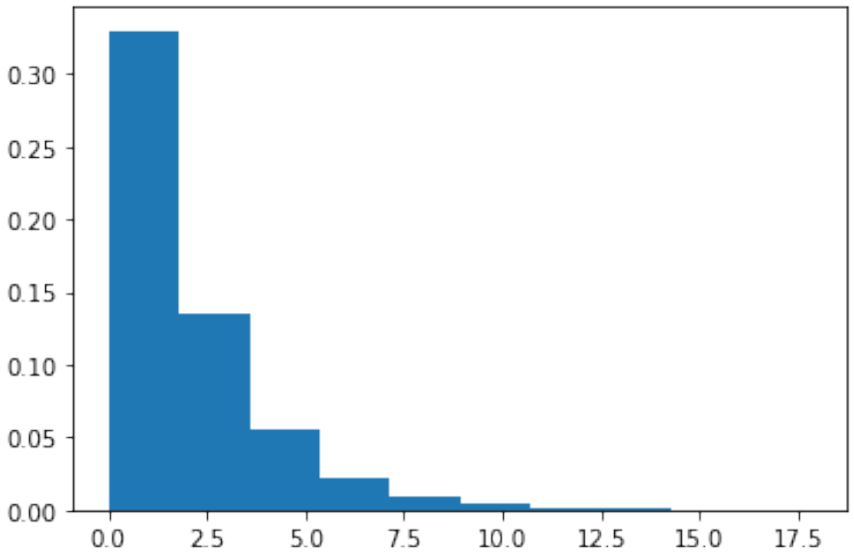


Fig. 3.3: 10000 random numbers from the exponential random variable with $\theta = 2$

Normal distribution in Python

The function stats.norm(μ, σ) means the normal distribution with mean μ and standard deviation σ. What we should be careful about is when we write, for example, $N(0, 4)$, four is the variance of the random variable, whereas in Python the same random variable is written as stats.norm$(0, 2)$, where two is its standard deviation.

After loading the stats subpackage from scipy and the PyPlot package, we define Z as the standard normal random variable:

```
In [1]: Z = stats.norm(0,1)
```

Let's evaluate the pdf and cdf of Z at 0:

```
In [2]: Z.pdf(0)
```

```
Out[2]: 0.3989422804014327
```

```
In [3]: Z.cdf(0)
```

```
Out[3]: 0.5
```

Next we plot the pdf and cdf of Z.

```
In [4]: xval = np.linspace(-4, 4, 100)
        y = Z.pdf(xval)
        plt.plot(xval, y);
```

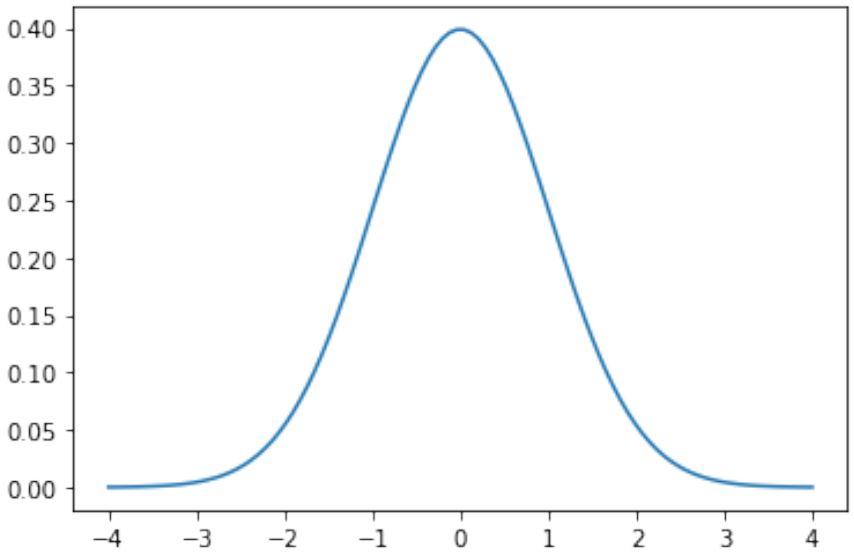


Fig. 3.4: The pdf of the standard normal random variable $N(0,1)$

```
In [5]: xval = np.linspace(-4, 4, 100)
        z = Z.cdf(xval)
        plt.plot(xval, z);
```

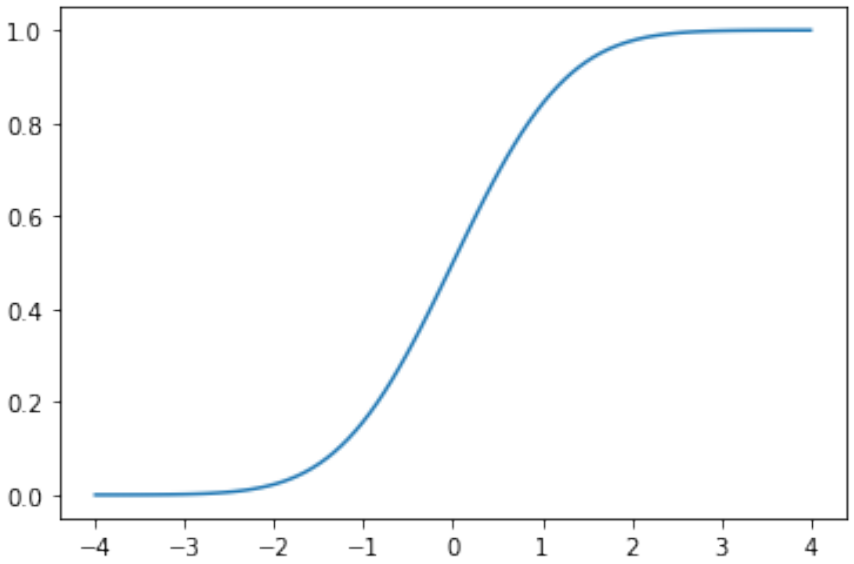


Fig. 3.5: The cdf of the standard normal random variable $N(0,1)$

Example 3.1 The actor Chris Hemsworth, better known as the superhero Thor, recounted on a TV show how he stuffed her daughter's shoes with candy bars so that she would meet the height requirement of 40 inches for a Disney ride. Do you wonder how many 5-year olds get disappointed at Disney World because they are shorter than 40 inches and cannot ride their favorite rides?

It is generally accepted that heights of people roughly follow a normal distribution. For example, the histogram below plots the heights of 746 children[2], and although not a perfect match, the shape of the data resembles the shape of the pdf of a normal random variable (see Figure 3.4).

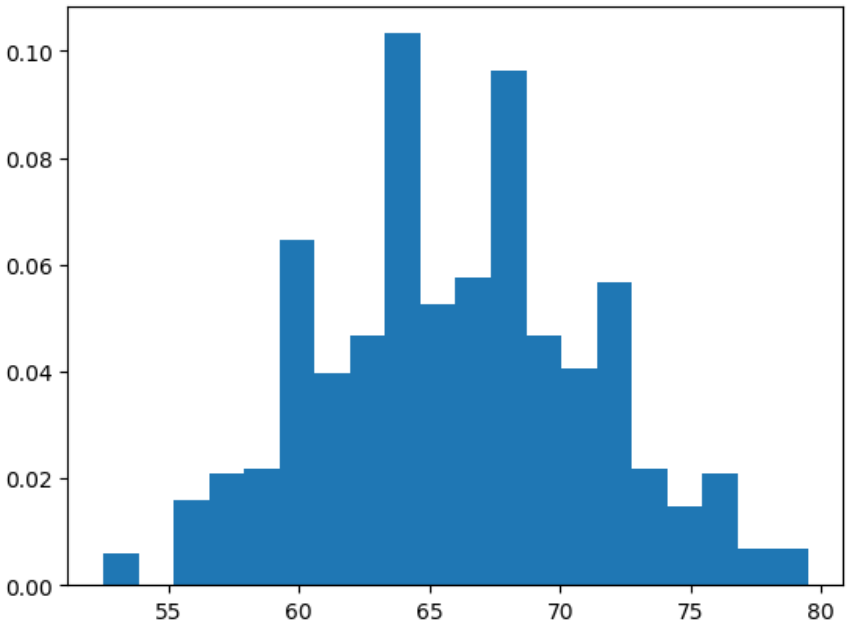


Fig. 3.6: Heights of 746 children

According to a report from the National Center for Health Statistics[3], the average height of 5-year olds in the U.S. is 44.8 inches, with a standard deviation of 3.1 inches. These values were obtained from a random survey of 205 kids in the age group. Assuming that a normal distribution is a good model for the heights of 5-year olds, we can find the proportion of them shorter than 40 inches. Let X be the height of a child, modeled by $N(44.8, 3.1^2)$. We have

$$P\{X \le 40\} = P\left\{\frac{X - 44.8}{3.1} \le \frac{40 - 44.8}{3.1}\right\} = P\{Z \le -1.5\} = \Phi(-1.5).$$

Recall that earlier we defined Z as a standard normal random variable in Python. Then we can compute $\Phi(-1.5)$ by using the cdf function of Z:

```
In [6]: Z.cdf(-1.5)
```

```
Out[6]: 0.06680720126885807
```

Therefore, about 7% of all 5-year olds won't be able to enjoy the rides with 40 inch minimum height requirement.

Generating random numbers from normal distribution

We generate 10000 random numbers from the standard normal distribution Z and plot their relative frequency histogram using 40 bins in Figure 3.7. Notice how the shape of the histogram follows the shape of the standard normal pdf in Figure 3.4.

[2] Pearson and Lee's data on the heights of parents and children classified by gender, https://vincentarelbundock.github.io/Rdatasets/datasets.html

[3] Fryar CD, Gu Q, Ogden CL. Anthropometric reference data for children and adults: United States, 200-2010. National Center for Health Statistics. Vital Health Stat 11(252). 2012.

```
In [7]: plt.hist(Z.rvs(10000), 40, density=True);
```

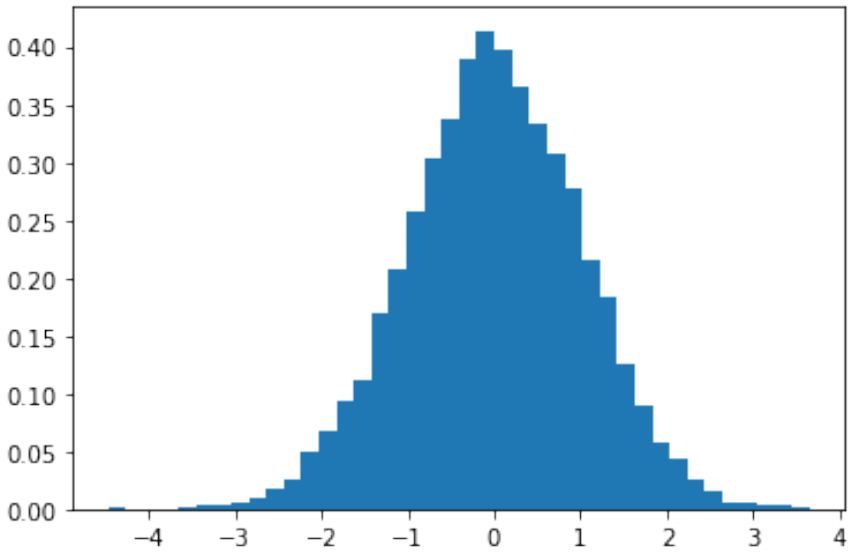


Fig. 3.7: 10000 random numbers from the standard normal random variable Z

Central limit theorem

The central limit theorem is one of the most important theorems in probability theory. Let $X_1, X_2, \ldots, X_N$ be independent random variables having the same distribution, and finite variance. Now consider their sample mean

$$S_N = \frac{X_1 + X_2 + \ldots + X_N}{N}$$

which is a random variable as well. We want to know the distribution of S_N.

Let's use Python to do some experiments. Assume $X_1, X_2, \ldots, X_N$ are independent uniform random variables on $(0, 1)$. We will try to figure out the distribution of S_N empirically in the following way. We will generate N random numbers $x_1, \ldots, x_N$ from $U(0, 1)$, and compute $s_N = \frac{x_1+x_2+\ldots+x_N}{N}$. We will then repeat this computation 1000 times, using independent random numbers, to obtain 1000 values for S_N

$$s_N^{(1)}, s_N^{(2)}, \ldots, s_N^{(1000)},$$

and plot the relative frequency histogram of these values.

First we load the packages we need.

```
In [1]: from scipy import stats
```

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

We can generate N random numbers from $U(0, 1)$ using the function **np.random.rand(N)**. To add these numbers, we can use the function **np.sum** and write **np.sum(np.random.rand(N))**. The following code, which is a modification of a code we discussed earlier, takes N as the input and returns the histogram.

```
In [3]: def sMean(N):
            means = np.zeros(1000)
```

```
            for i in range(1000):
                means[i] = np.sum(np.random.rand(N))/N
            plt.hist(means, density=True)
```

```
In [4]: sMean(500)
```

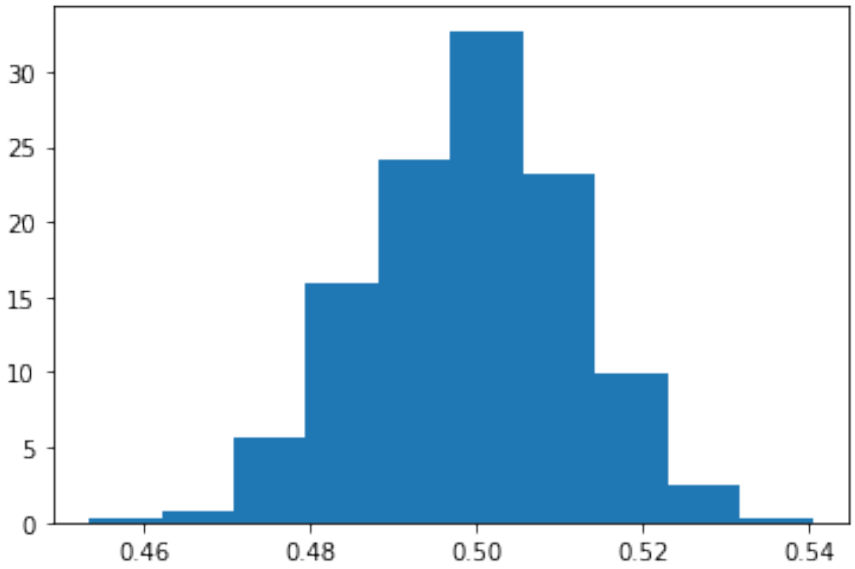


Fig. 3.8: Histogram of S_N when the X_n are uniform random variables on $(0, 1)$

What does the shape of the histogram in Fig. 3.8 remind you of? The normal distribution, isn't it? So it looks like the sample mean random variable S_N may be normally distributed. In our experiment we assumed $X_1, \ldots, X_N$ had the uniform distribution on $(0, 1)$. What would happen if we had picked X_n's from another distribution? Let's try and see.

Next we define X as an exponential random variable with $\theta = 2$:

```
In [5]: X = stats.expon(scale=2)
```

In the following code, the function **X.rvs(size=N)** generates N random numbers from the distribution X. The rest is similar to the previous code.

```
In [6]: def sMean(N):
            means = np.zeros(1000)
            for i in range(1000):
                means[i] = np.sum(X.rvs(size=N))/N
            plt.hist(means, density=True)
```

```
In [7]: sMean(500)
```

Surprise! The distribution of S_N still looks like a normal distribution. And this is precisely what the central limit theorem states! No matter what the distribution of $X_1, \ldots, X_N$ is, as long as the X_n have the same distribution and are independent, $S_N = \frac{X_1+X_2+\ldots+X_N}{N}$ will have an *approximately* normal distribution. The approximation gets better as $N \to \infty$. Here is the precise statement of the central limit theorem:

$$P\left\{\frac{S_N - \mu}{\sigma/\sqrt{N}} \leq a\right\} \to \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{a} e^{-x^2/2} dx \tag{3.1}$$

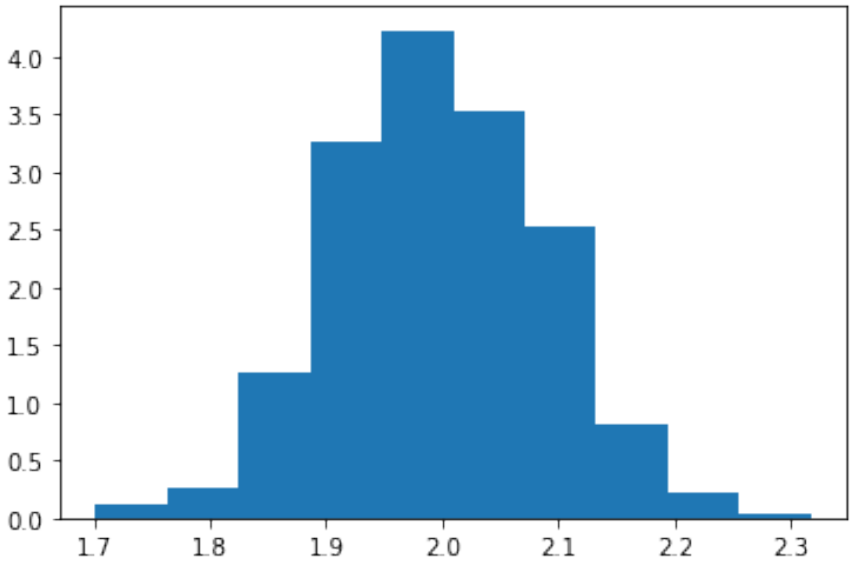


Fig. 3.9: Histogram of S_N when the X_n are exponential random variables with $\theta = 2$

as $N \to \infty$, where μ and σ^2 are the common expected value and variance of the random variables $X_1, \ldots, X_N$. Note that the limit in Eqn. (3.1) is the standard normal cdf. The expression on the left-side of (3.1) is not exactly $P\{S_N \leq a\}$, but the *standardized* version of S_N where we subtract from S_N its mean, μ, and divide the difference by its standard deviation, $\sigma/\sqrt{N}$. In this way the standardized random variables converge to the standard normal distribution. A proof of the central limit theorem can be found in more advanced textbooks on probability theory, such as Billingsley [2].

Let's make one final observation. Earlier we learned from the strong law of large numbers that $S_N = \frac{X_1+X_2+\ldots+X_N}{N}$ converges to $E[X]$ when the X_n are independent and have the same distribution as X. If we inspect Figs. 3.8 and 3.9, we see that the first histogram seems to be symmetric around 0.5, and the second one around 2. This indicates the sample mean of the numbers in the first histogram is about 0.5, and the other 2. Now recall that the expected value of $X \sim U(0, 1)$ is 0.5, and the expected value of $X \sim \exp(\theta = 2)$ is 2. The histograms in Figs. 3.8 and 3.9 illustrate both the strong law of large numbers and the central limit theorem. They show S_N have a bell-shaped curve, indicating normal distribution, and they are centered at the expected value of the common distribution X where the S_N are obtained from.

3.3 Project 9: Florida Panther

The Florida panther is one of two wild cat species native to Florida, the other being the bobcat. The current territory of Florida panther is southwest Florida. Florida panthers are endangered species, and according to Florida Fish and Wildlife Conservation Commission, there are approximately 120-230 adult panthers in the population as of 2019. Our task is to model the population growth of the panther and assess the likelihood of its extinction.

We will use the following difference equation to model the growth of the Florida panther population:

$$x(n+1) - x(n) = (b-d)x(n)$$

where $x(n)$ denotes the population at time n, and b, d are the birth and death rates for the panther. The left-hand side of the equation is the change in the population from time n to $n+1$. The term $b-d$ on the right-hand side measures the net growth rate: for example, if this difference is 0.1, then the model tells us that the panther population will grow by 0.1 times the current population $x(n)$, as we go from time n to time $n+1$.

In deterministic models, the birth and death rates are taken as constants. A more realistic assumption would be to assume these rates are random quantities. Let's assume the birth rate and the death rate are random variables with a normal distribution:

$$b \sim N(\mu_b, \sigma_b^2)$$
$$d \sim N(\mu_d, \sigma_d^2)$$

The means μ_b, μ_d and variances σ_b^2, σ_d^2 have to be estimated from data.

1. The number of deaths and births of Florida panthers for recent years can be found at https://myfwc.com/wildlifehabitats/wildlife/panther/. Use the available data to estimate the parameters $\mu_b, \mu_d, \sigma_b^2, \sigma_d^2$, using the sample mean and variance of birth and death rate data. Recall that the sample variance of $\{x_1, \ldots, x_n\}$ is

$$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x}$ is the sample mean. The square root of the variance, σ, is the standard deviation. To compute the rate of birth or death, you have to know the current population, for which there are only estimates as cited on the aforementioned webpage. For example, if there are 2 births in a given year in a population of 100, then the birth rate is 2%.

You can also use Python to compute sample means and variances (or, standard deviations). The functions **numpy.mean** and **numpy.std** compute the sample mean and standard deviation of data.

```
In [1]: import numpy as np

In [2]: list = np.array([12, 34, 11, 4, 5, 9])
        list

Out[2]: array([12, 34, 11,  4,  5,  9])
```

```
In [3]: np.mean(list)

Out[3]: 12.5

In [4]: np.std(list, ddof=1)

Out[4]: 11.004544515789828
```

2. Write a Python code that computes the population $x(n)$ as $n = 1, 2, \ldots, 15$. Try generating a few population trajectories and plotting them.
Recursions can be coded easily in Python. For example, to code the recursion $x(1) = 1; x(n) = 2x(n-1) + 1$, we write:

```
In [1]: def x(n):
            if n==1:
                return 1
            else:
                return 2*x(n-1)+1

In [2]: [x(n) for n in range(1,7)]

Out[2]: [1, 3, 7, 15, 31, 63]
```

3. The Florida panther will become extinct if the number of breeding animals falls below a critical threshold. Assuming that this threshold corresponds to a population size of 10, estimate the probability that the Florida panther will become extinct in the next 15 years. The Python function **min** will be useful:

```
In [1]: a = np.array([3, 4, 9, 21, 2])
        a

Out[1]: array([ 3,  4,  9, 21,  2])

In [2]: a.min()

Out[2]: 2
```

Solution 3.2

The following data is from https://myfwc.com/wildlifehabitats/wildlife/panther/pulse/:

Year	2018	2017	2016	2015	2014
Births	9	19	14	15	32
Deaths	30	30	42	42	34

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

In [2]: b = np.array([9, 19, 14, 15, 32])
        d = np.array([30, 30, 42, 42, 34])

In [3]: brate = b/120
        print(brate)

[0.075      0.15833333 0.11666667 0.125      0.26666667]

In [4]: print(brate.mean())

0.14833333333333334
```

```
In [5]: print(brate.std(ddof=1))
```

```
0.07250478911385402
```

```
In [6]: drate = d/120
        print(drate)
```

```
[0.25       0.25       0.35       0.35       0.28333333]
```

```
In [7]: print(drate.mean())
```

```
0.2966666666666667
```

```
In [8]: print(drate.std(ddof=1))
```

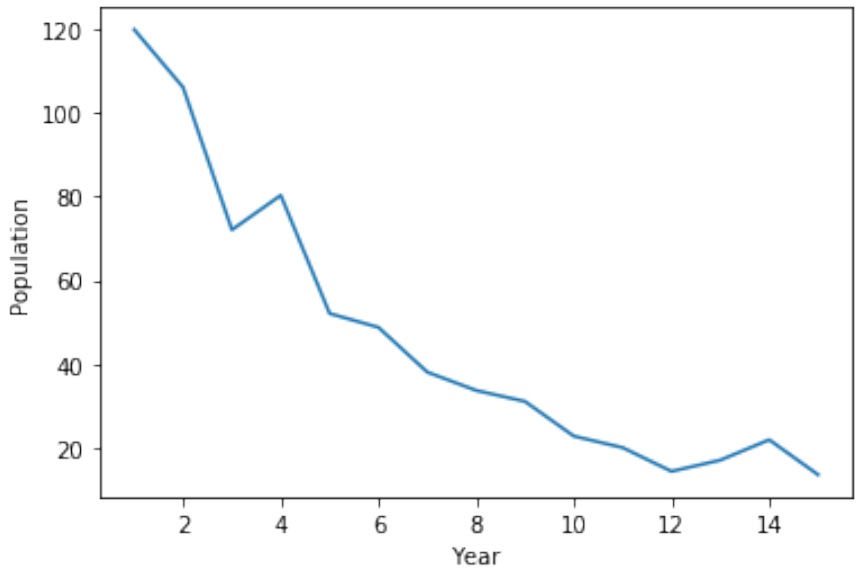
```
0.05055250296034366
```

```
In [9]: def x(n):
            b = 0.15 + 0.07*np.random.randn()
            d = 0.3 + 0.05*np.random.randn()
            if n == 1:
                return 120
            else:
                return (b-d+1)*x(n-1)
```

```
In [10]: pop=[x(n) for n in range(1, 16)]
         pop
```

```
Out[10]: [120,
          111.71412047324527,
          81.41803557593163,
          84.2434284707384,
          52.85080754681969,
          40.65679516393002,
          61.355367036155855,
          32.83638549761974,
          37.41872086228168,
          31.291732751626384,
          23.712836336046216,
          8.703567971544174,
          14.349051119652415,
          14.52722974942609,
          9.522202950276034]
```

```
In [11]: xvalues = np.arange(1, 16)
         plt.plot(xvalues, pop)
         plt.xlabel('Year')
         plt.ylabel('Population');
```




```
In [12]: def extinct(m):
             count = 0
             for i in range(m):
                 pop = np.array([x(n) for n in range(1, 16)])
                 if pop.min() < 10:
                     count += 1
             print('The probability of extinction is ', count/m)
```

```
In [13]: extinct(10000)
```

```
The probability of extinction is  0.5532
```

3.4 χ^2-distribution and χ^2-test

In Section 3.5 of *Probability and Simulation* tail probabilities of the χ^2-distribution were computed using Julia. Here is how we can compute them using Python. Let $X \sim \chi^2(9)$ and suppose we want to compute $P\{X > 20\}$. Observe that

$$P\{X > 20\} = 1 - P\{X \leq 20\} = 1 - F(20)$$

where F is the cdf of X. We can compute this using Python. After we load the subpackage **stats** from scipy, we define $X \sim \chi^2(9)$ by simply typing, X = stats.chi2(9):

```
In [1]: X = stats.chi2(9)
        1 - X.cdf(20)
```

```
Out[1]: 0.01791240452984333
```

Therefore the area under the density function of $\chi^2(9)$ to the right of 20 is about 0.018.

3.5 Project 10: Can humans generate random numbers?

It looks like the jury is still out on this question, at least in the medical community. Persaud [6] claims humans can generate random numbers, and Figurska et. al [3] claim they cannot.

In this project we will generate *random* digits 0 through 9, and then test how truly random these digits are. We will compare our performance with that of a machine: the random generator of Python.

1. Each student generates about 25 random digits, creating a total of at least 500 digits. As the digits are read out, input them in Python as an array, like:

```
In [1]: digs = np.array([1, 2, 0, 9, 3, 3, 6, 2])
        digs

Out[1]: array([1, 2, 0, 9, 3, 3, 6, 2])
```

   a. If the numbers in **digs** are truly random, then the relative frequency of each digit (0 through 9) should be 1/10. Test this hypothesis using the χ^2-test. As you write a code for the χ^2-statistic, here is how to count the number of 2's in the array **digs** in Python:

```
In [2]: np.where(digs == 2)[0].size
Out[2]: 2
```

   b. Next we will test the distribution of pairs of digits. If the numbers in **digs** are truly random, then the relative frequency of any consecutive pair of digits should be 1/100. Think of the random experiment as generating pairs of digits. There are 11 possible outcomes: 00, 11, 22, 33, 44, 55, 66, 77, 88, 99, and "all others". (We are grouping pairs that do not repeat in one category.) The probabilities of the outcomes are 1/100 for the equal-pair outcomes, and 9/10 for the last outcome "all others". Find the frequency of each outcome in your data and carry out the χ^2-test and check the randomness of the data. The following functions will be helpful:

```
In [3]: pairs = np.array([(digs[2*i],digs[2*i+1]) for i in range(4)])
        pairs
Out[3]: array([[1, 2],
               [0, 9],
               [3, 3],
               [6, 2]])
In [4]: np.sum(np.logical_and(pairs[:,0]==3, pairs[:,1]==3))
Out[4]: 1
```

2. Apply the χ^2-test as described in parts (a) and (b) to the random numbers generated by Python.
3. What are your conclusions? Do you think humans can generate random numbers? How about Python?

Solution 3.3

This is the application of the test to Python's random numbers.

```
In [1]: import numpy as np
        from scipy import stats

In [2]: n = 500
        lsum = 0.
        digs = np.random.randint(0, 10, n)
        freq = np.empty(0)
        for i in range(10):
            freq = np.append(freq, np.where(digs == i)[0].size)
        for i in range(10):
            lsum += 10*(freq[i]-n/10)**2/n
        print(lsum)
```

```
8.12
```

```
In [3]: 1 - stats.chi2.cdf(lsum, 9)

Out[3]: 0.5220998805731258
```

The digits pass the χ^2-test.

```
In [4]: n = 500
        m = 250
        lsum = 0.
        digs = np.random.randint(0, 10, n)
        pairs = np.array([(digs[2*i],digs[2*i+1]) for i in range(m)])
        freq = np.empty(0)
        for i in range(10):
            freq = np.append(freq, np.sum(np.logical_and(pairs[:,0]==i, pairs[:,1]==i)))
        for i in range(10):
            lsum += 100*(freq[i]-m/100)**2/m
        rfreq = m-np.sum(freq)
        lsum += 10*(rfreq-9*m/10)**2/(9*m)
        print(lsum)
```

```
9.84
```

```
In [5]: 1 - stats.chi2.cdf(lsum, 10)
```

```
Out[5]: 0.4546411312258948
```

The pairs of digits pass the χ^2-test as well.

Chapter 4
Markov Chains

In Sections 4.1 and 4.2 of *Probability and Simulation*, powers of matrices are computed using Julia. In an example in Section 4.1, we wanted to compute P^4, where

$$P = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

Here is how to do this calculation using Python:

```
In [1]: import numpy as np
        P = np.array([[0.7, 0.3], [0.4, 0.6]])
        P

Out[1]: array([[0.7, 0.3],
               [0.4, 0.6]])

In [2]: np.linalg.matrix_power(P, 4)

Out[2]: array([[0.5749, 0.4251],
               [0.5668, 0.4332]])
```

Limiting behavior of the state vector

The two largest libraries at FSU are Dirac and Strozier. Assume Dirac has 30% of all the books, and Strozier has the remaining 70% , at the beginning of a new semester. When a student checks out a book, she can return it to any library. A survey of students during the first two weeks of the semester shows that 80% of students who check out a book from Dirac return the book to Dirac, and the rest is returned to Strozier. Students who check out from Strozier return their books to Strozier at the rate of 60%. We need to find how many books will end up in each library as the semester unwinds; Dirac has a smaller capacity for storing books, and if too many books are returned there, we need to transport books periodically from Dirac to Strozier.

We will model this problem using Markov chains. The states are D (Dirac) and S (Strozier). The initial proportions of books are given by $\pi_0 = [0.3, 0.7]$. The transition probabilities are

$$p_{DD} = 0.8, p_{DS} = 0.2, p_{SS} = 0.6, p_{SD} = 0.4,$$

and the probability transition matrix is:

```
In [1]: P = np.array([[0.8, 0.2], [0.4, 0.6]])
        P

Out[1]: array([[0.8, 0.2],
               [0.4, 0.6]])
```

In this example time is measured in terms of loan cycles. Let's simplify the loan policy and assume students must return their books exactly one week after they borrow them. Also assume all of the books are in circulation all the time. After n cycles, the proportion of books in Dirac and Strozier will be given by the state vector

$$\pi_n = [\pi_n(D), \pi_n(S)],$$

and from Lemma 4.1, $\pi_n = \pi_0 P^n$, which is coded in Python as:

```
In [2]: pi = lambda n : np.dot(np.array([[0.3, 0.7]]),
                               np.linalg.matrix_power(P, n))
```

Here is the state vector at $n = 5$

```
In [3]: pi(5)

Out[3]: array([[0.662912, 0.337088]])
```

and the state vector at $n = 20$:

```
In [4]: pi(20)

Out[4]: array([[0.66666666, 0.33333334]])
```

It looks like the state vector π_n is converging to the vector $\pi = [0.67, 0.33]$ as $n \to \infty$. In other words for large n, $\pi_n(D) \approx 0.67$ and $\pi_n(S) \approx 0.33$, which means 67% of the books end up in Dirac, and 33% in Strozier, and thus there will have to be periodic transportation of books from Dirac to Strozier.

Limiting behavior of the transition matrix

Let's compute some higher powers of the transition matrix P for the library example. Here are P^{10} and P^{20}:

```
In [5]: np.linalg.matrix_power(P, 10)

Out[5]: array([[0.66670162, 0.33329838],
               [0.66659676, 0.33340324]])

In [6]: np.linalg.matrix_power(P, 20)

Out[6]: array([[0.66666667, 0.33333333],
               [0.66666666, 0.33333334]])
```

It looks like the transition matrix P is converging to the following matrix:

$$P^\infty = \begin{bmatrix} 0.67 & 0.33 \\ 0.67 & 0.33 \end{bmatrix}.$$

What is interesting about P^∞ is that each row is equal to $\pi = [0.67, 0.33]$, the limit of the state vector π_n we discussed earlier. Here is a summary of our observations from the library example:

- The state vector π_n seems to converge to a vector: $\pi_n \to \pi$ as $n \to \infty$. This limit vector π is called the **steady state vector**.
- The transition probability matrix P^n converges to a matrix P^∞ where each row is the steady state vector π.

Finding the steady state vector when it exists

Section 4.3.1.2 of *Probability and Simulation* discussed how to find the steady state vector using matrix calculations. In the example that was considered, the following matrix equation had to be solved for x:

$$\underbrace{\begin{bmatrix} 1 & 1 \\ -0.2 & 0.4 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \pi(D) \\ \pi(S) \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{c}$$

To solve this in Python for x, we use "numpy.linalg.solve":

```
In [7]: A = np.array([[1,1], [-0.2, 0.4]])
        c = np.array([1, 0])

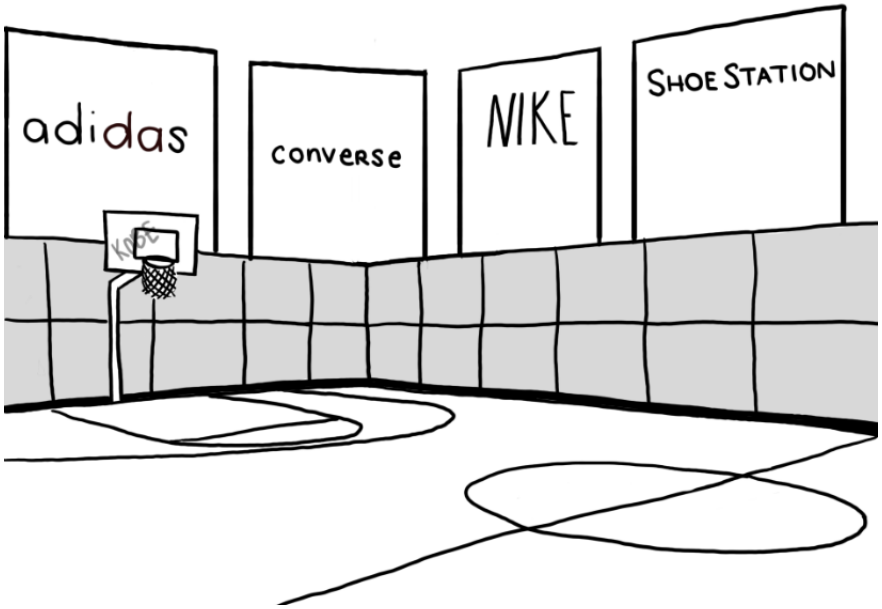
In [8]: np.linalg.solve(A, c)

Out[8]: array([0.66666667, 0.33333333])
```

Therefore $\pi(D) \approx 0.67$, $\pi(S) \approx 0.33$, and $\pi = [0.67, 0.33]$.

4.1 Project 12: Market share of shoe brands

There is a new development in Tallahassee - a strip mall with four shoe stores! The stores are Adidas, Nike, Converse, and Shoe Station. The Shoe Station will carry brands other than Adidas, Nike, and Converse. What is interesting about the strip mall is that the shoe stores are planning to build a basketball court in an attempt to support youth sports as well as promote their products. The initial cost of the basketball court is $50,000, and then there is the yearly maintenance of $10,000. The four stores agree to share the initial cost equally, but not the yearly maintenance cost. The manager of the Shoe Station wants the yearly cost to be divided in proportion to the market share of the stores.



We will model this problem using Markov chains, and find the percentage of customers shopping at each store in the long run, which is the information needed to divide the yearly cost of the basketball court among the stores in a proportional way. The states of the Markov chain are A (Adidas), N (Nike), C (Converse), and S (Shoe Station).

1. Think about the brand of the pair of shoes you own now, and what brand you want to purchase next. Collect this information from your friends.[1] We will use this data as a proxy for the real customers of the stores. Obtain transition probabilities using this data. For example, if 5 students own Nike shoes, and 2 of them want to buy Nike

[1] We survey YSP students in the class for this project.

again as their next pair of shoes, then $2/5 = 40\%$ is the transition probability from N to N, that is, $p_{NN} = 2/5$. (If unable to generate data in this way, use the artificial data given in Appendix B of *Probability and Simulation*.)
2. Find the steady state vector π, if it exists. Find how much each store should pay for the yearly maintenance.
3. Investigate the sensitivity of the steady state vector to the transition probabilities. For example, if the estimate for p_{AA} you had in part (1) changed a little bit, how much would the steady state vector change? Use Python to consider several scenarios and analyze the sensitivity empirically. (You can simplify this question by concentrating on the sensitivity of just one component of the steady state vector.)

Solution 4.1

We first enter the number of transitions data given in Appendix B of *Probability and Simulation* in Python:

```
In [1]: import numpy as np

In [2]: P = np.array([[5,7,2,1], [5,10,4,2], [8,2,3,3], [15,15,8,10]])
        P

Out[2]: array([[ 5,  7,  2,  1],
               [ 5, 10,  4,  2],
               [ 8,  2,  3,  3],
               [15, 15,  8, 10]])
```

To compute the transition probabilities, we need to divide each row of P by the row sum:

```
In [3]: Pt=np.array([[5/15,7/15, 2/15, 1/15], [5/21, 10/21, 4/21, 2/21],
                     [8/16, 2/16, 3/16, 3/16], [15/48, 15/48, 8/48, 10/48]])
        Pt

Out[3]: array([[0.33333333, 0.46666667, 0.13333333, 0.06666667],
               [0.23809524, 0.47619048, 0.19047619, 0.0952381 ],
               [0.5       , 0.125     , 0.1875    , 0.1875    ],
               [0.3125    , 0.3125    , 0.16666667, 0.20833333]])
```

The steady state vector π is the solution of $A\pi = c$ where A, c are given as (using two decimal digits):

```
In [4]: A = np.array([[1,1,1,1], [-0.67,0.24,0.5,0.31],
                      [0.13,0.19,-0.81,0.17],
                      [0.07,0.095,0.19,-0.79]])
        c = np.array([1,0,0,0])
```

We solve this equation for π next:

```
In [5]: np.linalg.solve(A, c)

Out[5]: array([0.32080838, 0.39440567, 0.16842426, 0.11636169])
```

Therefore Adidas will pay 32% of the yearly maintenance of \$10,000, Nike will pay 39% of it, etc.

Chapter 5
Brownian Motion

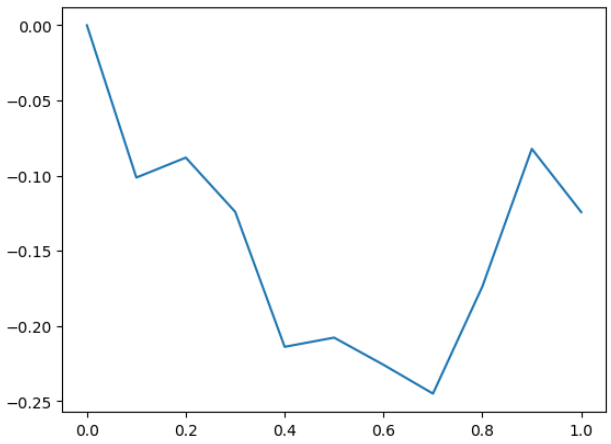
Section 5.1 of *Probability and Simulation* discussed how to simulate Brownian motion using Julia. Here is how the simulation can be done in Python.

Simulating Brownian motion

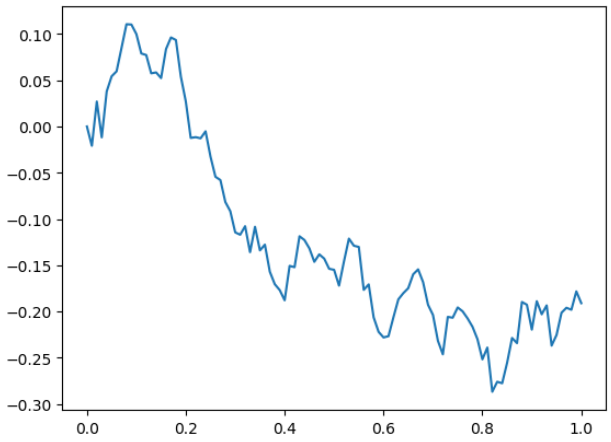
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: def BrownianMotion(mu, sigma, n):
            values = np.zeros(n+1)
            values[0] = 0
            for i in range(1, n+1):
                w = values[i-1]+mu/n+sigma*((1/n)**0.5)*np.random.randn()
                values[i] = w
            xvalues = np.linspace(0, 1, n+1)
            plt.plot(xvalues, values)
```

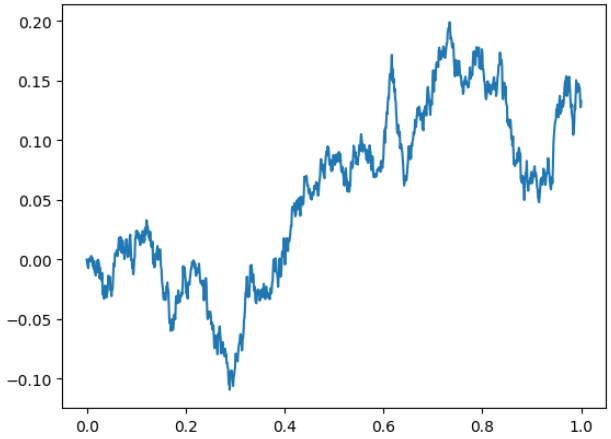
```
In [3]: BrownianMotion(0, 0.2, 10)
```




```
In [4]: BrownianMotion(0, 0.2, 100)
```




```
In [5]: BrownianMotion(0, 0.2, 1000)
```



Notice that as n gets larger and the time interval $(0, 1)$ is divided into smaller and smaller subintervals, the Brownian motion path becomes more wiggly.

5.1 Project 13: Modeling insect movement

An important research problem in ecology is developing mathematical models for the movement behavior of animals. A quantitative model predicting the location of an animal can be useful in conservation biology and pest control. In this project we will discuss a simple model based on two-dimensional Brownian motion. For a more realistic generalization of this model see Kareiva and Shigesada [7].

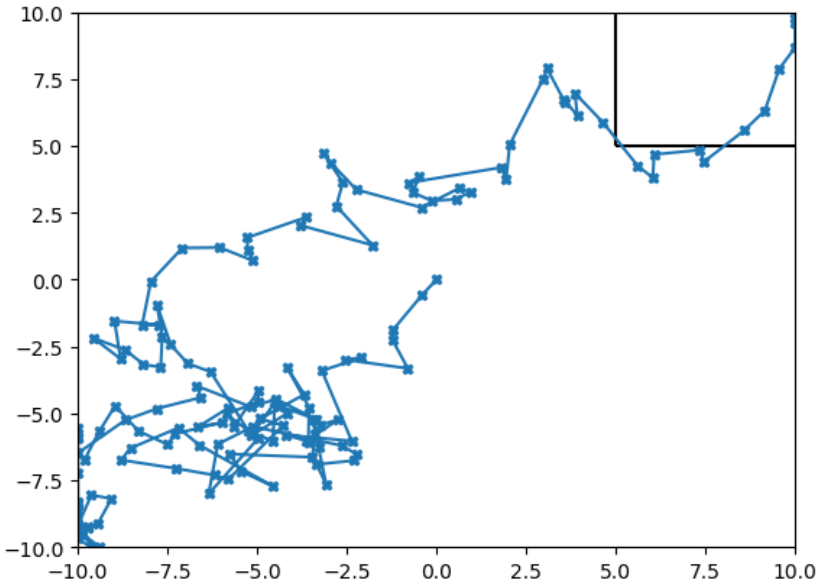
Imagine an insect in a two-dimensional grid, with initial position at the origin. The insect then starts moving in a way which would likely look erratic to our eyes. Let $(X(t), Y(t))$ be the location, that is, the x and y-coordinates of the insect at time t, with $X(0) = Y(0) = 0$. Our model assumes $X(t)$ and $Y(t)$ are two independent standard Brownian motions.

Our first task is to simulate the model and obtain trajectories for the insect. To do that we need to discretize time: let $t_0 = 0, t_1, t_2, \ldots$ be the discrete times of insect movement, and for simplicity let's set the time difference between two consecutive moves to one unit of time. We can then switch to a simpler notation and set $X(t_0) = X(0), X(t_1) = X(1)$ and so forth, and similarly set $Y(t_i) = X(i)$.

1. By updating Eqn (5.1) of *Probability and Simulation*, show that the movement of the insect can be simulated using the following equations, where $x(i), y(i)$ correspond to particular values the random variables $X(i), Y(i)$ take:

$$\begin{aligned} x(i) &= x(i-1) + z_1 \\ y(i) &= y(i-1) + z_2 \end{aligned} \tag{5.1}$$

   where z_1, z_2 are independent random numbers from the standard normal distribution. Consider the grid $[-10, 10] \times [-10, 10]$, with the insect located at the origin. Write a Python code that plots the movement of the insect. If the insect's x or y-coordinates go beyond the borders $x, y = \pm 10$, assume it stays on the border. For example, if the x-coordinate exceeds 10 at the fourth move, that is $x(4) > 10$, then set $x(4) = 10$. Your code should take the number of total moves n as an input, and plot $(x(0), y(0)), (x(1), y(1)), \ldots, (x(n), y(n))$ as the output. Plot two trajectories using $n = 200$ and $n = 400$ moves.
2. Let $L(i)$ be the random variable that gives the length of the ith move of the insect. We will denote a particular value from this random variable as $l(i)$. Update your Python code to estimate the average length of a move, $\frac{1}{n}\sum_{i=1}^{n} l(i)$, and the average length squared, $\frac{1}{n}\sum_{i=1}^{n} l^2(i)$. Do these averages seem to converge to a value as n gets larger?
3. The average length squared can be computed analytically. Use the properties of Brownian motion to prove $E[L(i)^2] = 2$ for any i.
4. Now assume there is food for the insect at the top right corner of the grid, at $(10, 10)$, and the insect can smell it once in enters the rectangle $[5, 10] \times [5, 10]$. Once in this rectangle, the insect will only move towards the top right corner. How can you modify Eqn. (5.1), in particular the random numbers in the equation, so that $x(i)$ and $y(i)$ will only increase once the insect is in the region? Plot new trajectories for the insect until it finds the food. Below is an example of such a trajectory.



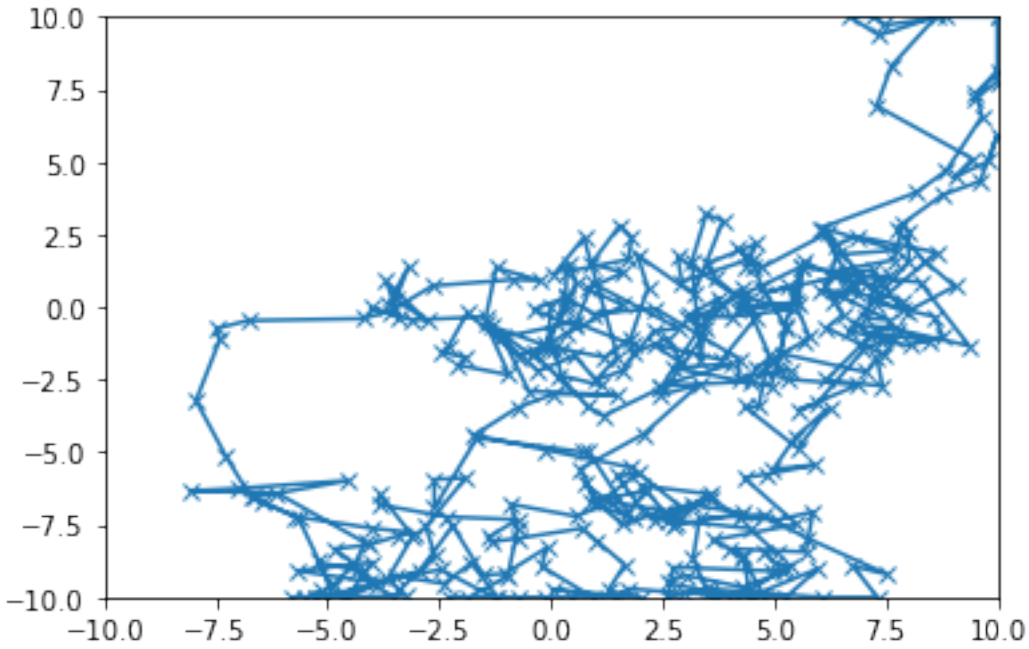
5. Update your code so that it will simulate N trajectories until the insect finds the food, compute the number of moves until the insect finds the food for each trajectory, and then return the average number of moves to find the food over N trajectories. Simulate $N = 1000$ and $N = 10000$ trajectories to find the average number of moves.

Solution 5.1

1. In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
def rwalk(n):
    k = 10
    x = np.zeros(n)
    y = np.zeros(n)
    for i in range(1,n):
        z1 = np.random.randn()
        z2 = np.random.randn()
        x[i] = x[i-1] + z1
        if x[i]>k:
            x[i] = k
        if x[i]<-k:
            x[i] = -k
        y[i] = y[i-1]+z2
        if y[i]>k:
            y[i] = k
        if y[i]<-k:
            y[i] = -k
    plt.plot(x, y, 'x-')
    plt.xlim((-k, k))
    plt.ylim((-k, k))
```

In [3]:
```
rwalk(400)
```



2. In [4]:
```
def rwalklength(n):
    k = 10
    x = np.zeros(n)
    y = np.zeros(n)
    length = 0
    lengthsq = 0
    for i in range(1,n):
        z1 = np.random.randn()
        z2 = np.random.randn()
        x[i] = x[i-1] + z1
        if x[i]>k:
            x[i] = k
        if x[i]<-k:
            x[i] = -k
        y[i] = y[i-1]+z2
        if y[i]>k:
            y[i] = k
        if y[i]<-k:
            y[i] = -k
        length += np.sqrt(z1**2+z2**2)
        lengthsq += z1**2+z2**2
    print('Average length: ', length/n)
    print('Average length squared: ', lengthsq/n)
```

In [5]:
```
rwalklength(1000)
```

```
Average length:  1.205052443505097
Average length squared:  1.85274094156512
```

In [6]:
```
rwalklength(100000)
```

```
Average length:  1.2539164716175908
Average length squared:  2.0010380269832644
```

3. First observe that the length of the ith move is $l(i) = \sqrt{z_1^2 + z_2^2}$ where z_1, z_2 are the random numbers from the standard normal distribution that were used in computing

$$x(i) = x(i-1) + z_1$$
$$y(i) = y(i-1) + z_2.$$

The notation $l(i)$ is for the specific length in one trajectory, and $L(i)$ for the random variable whose values are $l(i)$. Then

$$E[L(i)^2] = E[Z_1^2] + E[Z_2^2] = Var(Z_1) + Var(Z_2) = 2,$$

where Z_1, Z_2 are independent standard normal random variables.

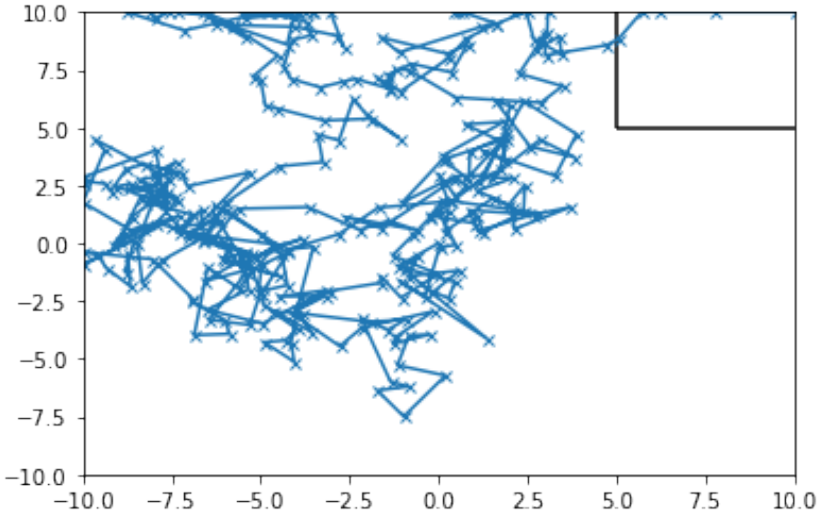
4. We can ensure the insect only moves towards the top right corner once it enters the rectangle $[5, 10] \times [5, 10]$ by replacing z_1, z_2 in Eqn. (5.1) with their absolute values, $|z_1|, |z_2|$, when the insect is in the rectangle.

```
In [7]: def dirwalk():
            k = 10
            x = np.array([0.])
            y = np.array([0.])
            while x[-1]<k/2 or y[-1]<k/2:
                z1 = np.random.randn()
                z2 = np.random.randn()
                xn = x[-1] + z1
                if xn > k:
                    xn = k
                if xn <= -k:
                    xn = -k
                x = np.append(x, xn)
                yn = y[-1] + z2
                if yn > k:
                    yn = k
                if yn < -k:
                    yn = -k
                y = np.append(y, yn)
            while x[-1]<k or y[-1]<k:
                z1 = np.random.randn()
                z2 = np.random.randn()
                if x[-1]+np.abs(z1) > k:
                    xn = k
                else:
                    xn = x[-1] + np.abs(z1)
                x = np.append(x, xn)
                if y[-1]+np.abs(z2) > k:
                    yn = k
                else:
                    yn = y[-1] + np.abs(z2)
                y = np.append(y, yn)

            plt.plot(x, y, 'x-', markersize=5)
            plt.xlim((-k, k))
            plt.ylim((-k, k))
```

```
            plt.hlines(5, 5, 10)
            plt.vlines(5, 5, 10)
```

```
In [8]: dirwalk()
```



5.
```
In [9]: def movestofood(m):
            k = 10
            count = 0
            for i in range(m):
                x = np.array([0.])
                y = np.array([0.])
                while x[-1]<k/2 or y[-1]<k/2:
                    z1 = np.random.randn()
                    z2 = np.random.randn()
                    xn = x[-1] + z1
                    if xn > k:
                        xn = k
                    if xn <= -k:
                        xn = -k
                    x = np.append(x, xn)
                    yn = y[-1] + z2
                    if yn > k:
                        yn = k
                    if yn < -k:
                        yn = -k
                    y = np.append(y, yn)
                while x[-1]<k or y[-1]<k:
                    z1 = np.random.randn()
                    z2 = np.random.randn()
                    if x[-1]+np.abs(z1) > k:
                        xn = k
                    else:
                        xn = x[-1] + np.abs(z1)
                    x = np.append(x, xn)
                    if y[-1]+np.abs(z2) > k:
                        yn = k
                    else:
```

```
                    yn = y[-1] + np.abs(z2)
                y = np.append(y, yn)
            count += x.size
        print('The avg number of moves till food is: ', count/m)
```

```
In [10]: movestofood(1000)
```

```
The avg number of moves till food is:  443.814
```

```
In [11]: movestofood(10000)
```

```
The avg number of moves till food is:  448.1667
```

5.2 Geometric Brownian motion

In Section 5.3 of *Probability and Simulation*, Julia was used to simulate stock prices that was modeled using a geometric Brownian motion. Two Julia functions were written: **stockprices** and **stockpricepath**. Here are the Python versions of these functions.

Simulating stock prices

```
In [1]: def stockprices(mu, sigma, szero, m, T, N):
            finalprice = 0
            h = T/m
            for j in range(N):
                logprices = np.zeros(m+1)
                logprices[0] = np.log(szero)
                for i in range(1, m+1):
                    logprices[i] = logprices[i-1] + mu*h + \
                    sigma*(h**0.5)*np.random.randn()
                finalprice += np.exp(logprices[m])
            return finalprice/N
```

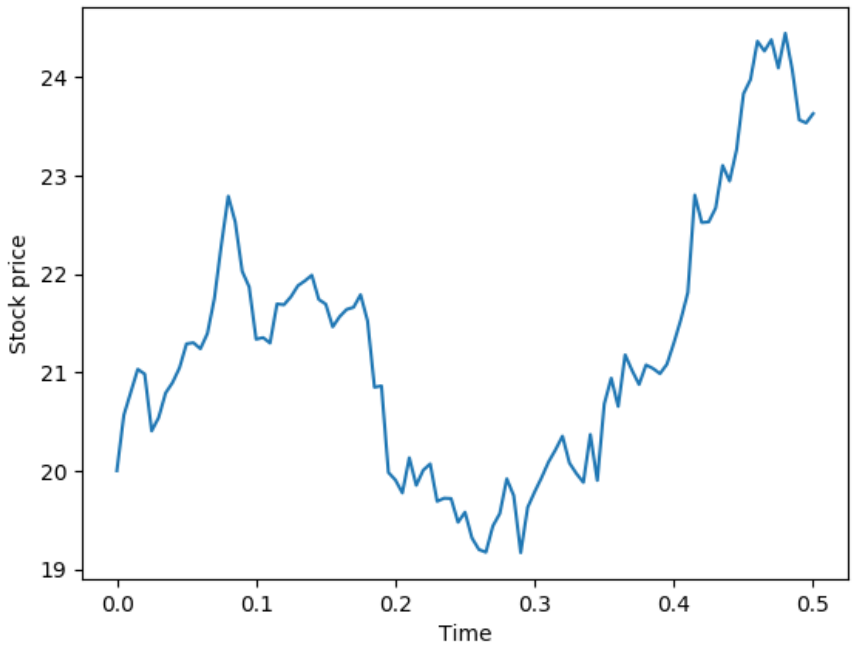
```
In [2]: stockprices(0.1, 0.2, 20, 10, 0.5, 10000)
```

```
Out[2]: 21.24961826954908
```

```
In [3]: def stockpricepath(mu, sigma, szero, m, T):
            h = T/m
            prices = np.zeros(m+1)
            prices[0] = szero
            for i in range(1, m+1):
                prices[i] = prices[i-1] * np.exp(mu*h+sigma*(h**0.5)*\
                                                 np.random.randn())
            xvalues = np.linspace(0, 0.5, m+1)
            plt.plot(xvalues, prices)
            plt.xlabel('Time')
            plt.ylabel('Stock price')
```

Here is the plot of a stock price path.

```
In [4]: stockpricepath(0.1, 0.2, 20, 100, 0.5)
```


Stock price
24
23
22
21
20
19
0.0
0.1
0.2
0.3
0.4
0.5
Time

5.3 Project 14: Option pricing

A European call option is an example of a *financial derivative*. You can buy and sell a European call option, just like a stock. A call option is tied to an *underlying*, which is usually a stock. For example, consider the Amazon.com stock which is traded for $1,733 today. We can buy a European call option on this stock, with a strike price of $1,715 and an expiry of four days, for the price of $30. At the end of the expiry (after four days), the call option gives us the right to purchase the Amazon.com stock for $1,715 (the strike price), no matter how much it is trading for on that day. For example, if the stock trades for $1,730 at the expiry, then we can make a profit of $15 by buying the stock at $1,715 and selling it for $1,730. If the stock trades for $1,700 at the expiry, then we *would not* strike the call option and purchase the stock for $1,715, since it is traded for a cheaper price. In that case we would be losing our initial investment of $30 that we used to purchase the call option. A famous question in financial mathematics was how to find the fair price of a European call option. The answer was given by the celebrated Black-Scholes-Merton formula, for which Robert Merton and Myron Scholes received the Nobel prize in Economics in 1997.

Let's introduce some notation. Let C be the price of the European call option, K the strike price, and $S(t)$ the underlying stock price at t. We denote the expiry by T. Let r denote the risk-free interest rate; this is like the interest rate a savings account pays. The payoff of the call option is $\max(S(T) - K, 0)$: that is, the option payoff is $S(T) - K$ if $S(T) > K$, and otherwise it is zero. We assume the stock price follows the geometric Brownian motion model with drift μ and volatility σ.

An important result from financial mathematics states that the fair price of the call option is

$$C = e^{-rT} E[\max(S(T) - K, 0)], \tag{5.2}$$

where $S(T)$ follows the geometric Brownian motion with drift $r - \sigma^2/2$ and volatility σ:

$$\log \frac{S(T)}{S(0)} \sim N((r - \sigma^2/2)T, \sigma^2 T). \tag{5.3}$$

Here are some interesting observations about this result:

- Eqn. (5.2) states the price of the call option is the expected value of its payoff, discounted to today's dollars: note that $1 today is e^{rT} dollars at time T since the money grows at risk-free interest rate r via continuous compounding. Equivalently, $1 at time T is worth e^{-rT} dollars today.
- Even though we originally assumed $S(T)$ followed geometric Brownian motion with drift μ and volatility σ, in Eqn. (5.3) the drift term was replaced by $r - \sigma^2/2$. The volatility term remained the same. This fascinating phenomenon is known as **risk-neutral pricing**, a topic covered in textbooks on financial mathematics.

In this project we will pick a call option that is traded in the markets, compute its fair price using the geometric Brownian motion model, and compare the price with the price it is trading for in the market.

1. Look up resources on the web such as Yahoo! Finance to find information about a call option on the General Electric (GE) stock[1]. Find out the following values: today's date t_0, expiry T, strike price K, the last price the option was traded for, $\tilde{C}$, and the last price the stock was traded for, S_0. Then find the interest rate for a one year certificate of deposit: we will use this value as the risk-free interest rate r.
2. We will estimate the volatility σ of the GE stock from historical data. Download historical daily stock price data for GE[2]. There are 252 trading days in a year, so one day amounts to 1/252 years. The geometric Brownian motion model (Eqn. (5.3) of *Probability and Simulation*) takes the following form when $t_1, t_2, \ldots,$ represent consecutive trading days:

$$\log \frac{S(t_i)}{S(t_{i-1})} \sim N(\mu/252, \sigma^2/252) \tag{5.4}$$

for $i = 1, 2, \ldots$. Let's label the downloaded stock price data as $S_1, S_2, \ldots, S_m$ for some m. Use this data to compute

[1] See, for example, the "options" link at https://finance.yahoo.com/quote/GE/.

[2] This can be obtained from the "historical data" link at https://finance.yahoo.com/quote/GE/.

$$\log\left(\frac{S_2}{S_1}\right), \log\left(\frac{S_3}{S_2}\right), \ldots, \log\left(\frac{S_m}{S_{m-1}}\right),$$

and then compute the sample variance $\hat{\sigma}^2$ of these ratios. We have

$$\hat{\sigma}^2 \approx \sigma^2/252$$

from Eqn. (5.4). Therefore we estimate the volatility by $\sigma = \sqrt{252}\hat{\sigma}$.

3. Use Monte Carlo simulation to estimate the price of the call option whose parameters you found in parts (1) and (2), using Eqns. (5.2) and (5.3). To do this, you will simulate N stock prices $S^{(1)}(T), \ldots, S^{(N)}(T)$ using the model described by Eqn. (5.3), and estimate C using the sample average

$$\frac{e^{-rT}}{N}\sum_{i=1}^{N}\max(S^{(i)}(T) - K, 0).$$

Use $N = 1000$ and $N = 10000$. Compare your estimates for C with the price the option is traded for, $\tilde{C}$. Answer this question using call options with different strike prices, fixing all other parameters. Are C and $\tilde{C}$ closer to each other for some strike prices than others?

Solution 5.2

1. From https://finance.yahoo.com/quote/GE/options?p=GE we pick: $t_0 = 0$ corresponds to today's date October 8, 2019, expiry $T = 10/252$ (the option expires in ten days), strike price $K = 4.5$, the last price the option was traded for, $\tilde{C} = 4.15$, and the last price the stock was traded for, $S_0 = 8.27$. The interest rate for a one year certificate of deposit is $r = 2.25\%$.
2. The sample standard deviation of $\log\frac{S(t_i)}{S(t_{i-1})}$ for the GE stock during 10/08/2018 to 10/08/2019 is 0.013, where $S(t_i)$ are the daily opening prices[3]. Then $\sigma = \sqrt{252}(0.013) = 0.21$.
3.
```
In [1]: import numpy as np
```

```
In [2]: def calloptionprice(r, sigma, szero, K, T, N):
            sum = 0
            for j in range(N):
                sfinal = szero*np.exp((r-sigma**2/2)*T+sigma*np.sqrt(T)*np.random.randn())
                sum += max(sfinal-K,0)
            return np.exp(-r*T)*sum/N
```

```
In [3]: calloptionprice(0.0225,0.21,8.27,4.5,10/252,10000)
```

```
Out[3]: 3.7752085014666314
```

[3] Data downloaded from https://finance.yahoo.com/quote/GE/

Appendix A
Benford's law

We will use a package called pandas to import data.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import pandas as pd
```

The data we will study is the US population by county. It was obtained from US Census Bureau (factfinder.census.gov). The data was saved in the same directory where this Python notebook is as a csv file. The name of the file is "population_county.csv". The function loadtable will import the data to a table. If there are any errors at this stage, open the csv file and look for entries that are not numbers.

```
In [2]: data = pd.read_csv('population_county.csv')
        data.dtypes
```

```
Out[2]: GEO.id               object
        GEO.id2               int64
        GEO.display-label    object
        rescen42010           int64
        resbase42010          int64
        respop72010           int64
        respop72011           int64
        respop72012           int64
        respop72013           int64
        respop72014           int64
        respop72015           int64
        respop72016           int64
        respop72017           int64
        respop72018           int64
        dtype: object
```

We now pick the column titled 'rescen42010':

```
In [3]: pop = data['rescen42010']
```

To extract the first number in the list, we type:

```
In [4]: pop[0]
```

```
Out[4]: 54571
```

We want the first digit of this number.

```
In [5]: int(str(pop[0])[0])
```

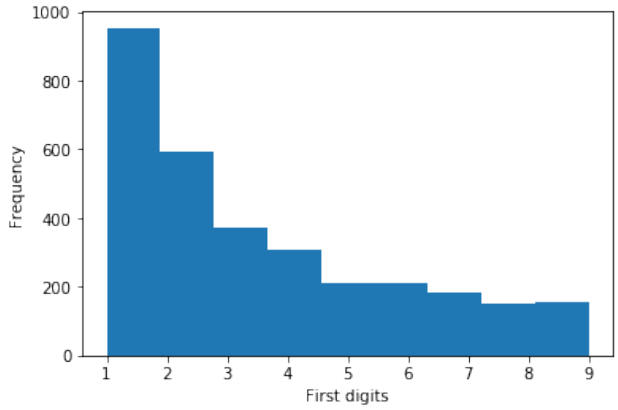
```
Out[5]: 5
```

The following for loop goes through the array pop and picks the first digit of each number.

```
In [6]: first_digs = [int(str(n)[0]) for n in pop]
```

A histogram of the first digits comes next:

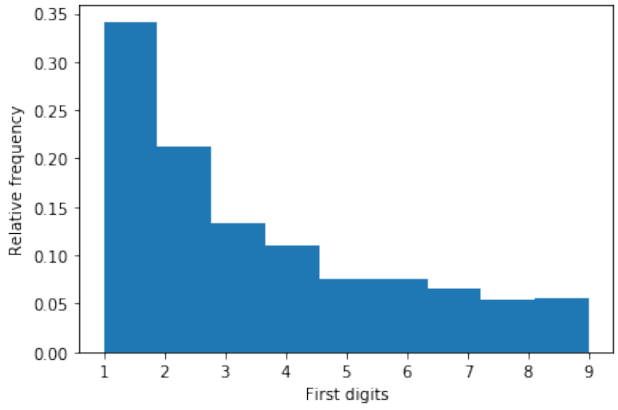
```
In [7]: plt.hist(first_digs, 9)
        plt.xlabel('First digits')
        plt.ylabel('Frequency');
```



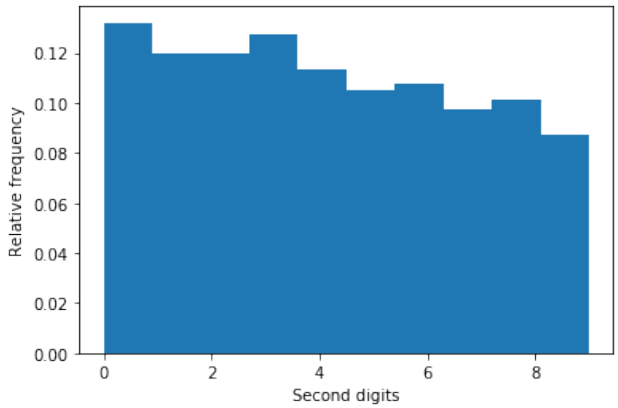
Clearly the earlier digits occur more often than the higher ones. This is what Benford observed in 1938, and Newcomb in 1881.

What about the distribution of other digits? Let's see what the distribution of third digits looks like. In the data, however, there are a few numbers with only two digits. We remove those numbers from the original data set, and call the new data set "population_county_threedigs.csv". Next we load this data, and plot a histogram for the third digits. Note that the third digit can be any integer between 0 and 9, so in the histogram we will specify 10 bins.

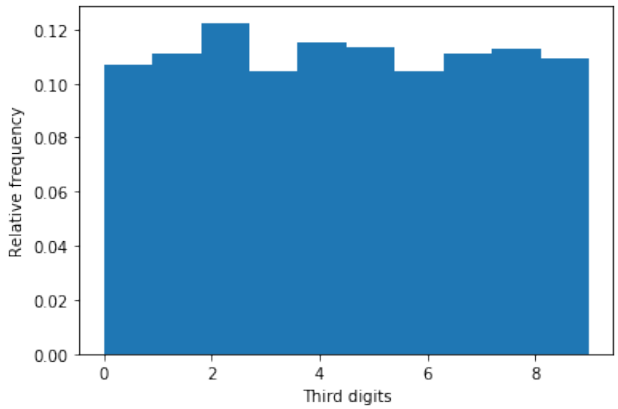
```
In [8]: data = pd.read_csv('population_county_threedigs.csv')
        pop = data['rescen42010']
        first_digs = [int(str(n)[0]) for n in pop]
        sec_digs = [int(str(n)[1]) for n in pop]
        third_digs = [int(str(n)[2]) for n in pop]
        plt.hist(first_digs, 9, density=True)
        plt.xlabel('First digits')
        plt.ylabel('Relative frequency');
```



```
In [9]: plt.hist(sec_digs, 10, density=True)
        plt.xlabel('Second digits')
        plt.ylabel('Relative frequency');
```




```
In [10]: plt.hist(third_digs, 10, density=True)
         plt.xlabel('Third digits')
         plt.ylabel('Relative frequency');
```



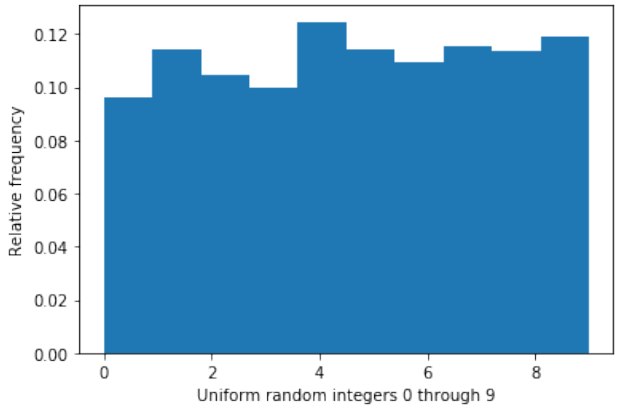
This histogram looks pretty uniform. Let's compare this with the histogram of 3140 random numbers (this is how many numbers we had in the array third_digs) from the discrete uniform distribution on the integers 0,1,...,9. The function **numpy.random.randint** can be used to generate random integers in a given range. Here is how to get 20 such random integers:

```
In [11]: np.random.randint(0, 10, 5)
```

```
Out[11]: array([5, 9, 7, 9, 9])
```

Here is a histogram for 3140 random numbers from the uniform distribution on 0,1,...,9:

```
In [12]: plt.hist(np.random.randint(0, 10, 3140), 10, density=True)
         plt.xlabel('Uniform random integers 0 through 9')
         plt.ylabel('Relative frequency');
```



This is visually indistinguishable from the histogram of the third digits. Although the distribution of the first digits is very different than uniform, as we look at the later digits, the distribution quickly becomes uniform. This observation was made by Newcomb as well.

Let's go back to the distribution of the first digits. Benford's law gives the probability of the first digit as:

$$\text{Prob}(\text{first digit } = d) = \log_{10}(1 + 1/d).$$

We write a Python code for this function.

```
In [13]: p = lambda d: np.log10(1+1/d)
```

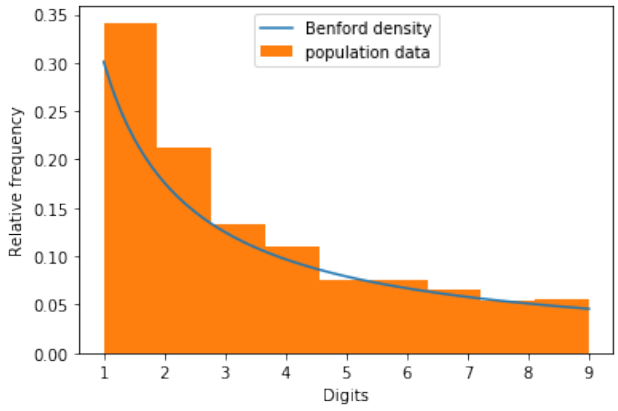
The probability that the first digit is a 1 is given by:

```
In [14]: p(1)
```

```
Out[14]: 0.3010299956639812
```

Let's plot the Benford probability mass function together with the histogram to see how good the fit is. There is one problem though. The y-axis of the histogram is the frequency of the digits, which goes as high as 1000. We need to normalize the heights so that they are between 0 and 1. This is done using the option "density=True" in the plt.hist function below.

```
In [15]: xaxis = np.linspace(1, 9, 81)
         yvals = p(xaxis)
         plt.plot(xaxis, yvals, label='Benford density')
         plt.xlabel('Digits')
         plt.ylabel('Relative frequency')
         plt.hist(first_digs, 9, density=True, label='population data')
         plt.legend(loc='upper center');
```



References

1. Benford, F., 1938. The Law of Anomalous Numbers. Proceedings of the American Philosophical Society 78(4), pp. 551-572.
2. Billingsley, P., 2008. Probability and Measure. John Wiley & Sons.
3. Figurska, M., Stańczyk, M. and Kulesza, K., 2008. Humans cannot consciously generate random numbers sequences: Polemic study. Medical Hypotheses, 70(1), pp.182-185.
4. Newcomb, S., 1881. Note on the Frequency of Use of the Different Digits in Natural Numbers. American Journal of Mathematics, 4(1/4), pp. 39-40.
5. Nigrini, M.J., 2012. Benford's Law: Applications for forensic accounting, auditing, and fraud detection (Vol. 586). John Wiley & Sons.
6. Persaud, N., 2005. Humans can consciously generate random number sequences: A possible test for artificial intelligence. Medical Hypotheses, 65(2), pp. 211-214.
7. Kareiva, P.M. and Shigesada, N., 1983. Analyzing insect movement as a correlated random walk. Oecologia, 56(2-3), pp. 234-238.

Index

http://www.springer.com/978-3-030-56069-0

Probability and Simulation

Ökten, G.

2020, X, 152 p. 50 illus., 39 illus. in color., Softcover
ISBN: 978-3-030-56069-0