CONTENTS

Chapter One

Fluctuations and the Nature of Mutations

1.1 HANDS-ON APPROACH TO MUTATIONS AND SELECTION

The goal of this lab is to simulate a growing bacterial population, including the ancestral "wild type" as well as mutants generated de novo during the growth process. The core techniques are straightforward: connecting the simplest model of exponential growth with stochastic events. To do so requires a few techniques, all centered on the ramifications of sampling from random distributions using Python. As you will see, learning how to sample from random distributions will be relevant in many biological systems. Indeed, being able to simulate stochastic dynamics is key for simulating biological systems at scales from molecules to organisms to ecosystems. Hence, this opening chapter introduces basic concepts that are used throughout the laboratory guide. This chapter also serves another function: to link the material in the textbook with the homework.

The laboratory will prepare you to build components of two categories of mutational models, as illustrated in a generalized schematic form in Figure 1.1. These initial components form the basis for the homework problems presented in the main text. In this figure, the left panel illustrates a branching process in which an individual bacterium in generation $g = 0$ divides so that there are two bacteria in generation $g = 1$, four bacteria in generation $g = 2$, and so on such that there are 2^g bacteria after g generations. Of these, a fraction of the offspring may be different than the ancestral wild type. These different bacteria are referred to as *mutants*. Notably, in this model, mutants give rise to mutant daughter cells and not to wild-type cells. The right panel illustrates an alternative model of mutation, in which many bacteria in a single generation undergo some stochastic change, i.e., a mutation, rendering a small number of bacteria into mutants. This latter case may be related to a phenotypic change, e.g., exposure to a virus or chemical agent. How to build models of both kinds, how to compare them, and how to reconcile the predictions of such models with experimental data from Luria and Delbrück form the core of this laboratory.

The key aim of this laboratory is to begin a process to relate the mechanism by which mutants are generated with signatures that can be measured. These signatures may include the mean as well as the variance in the number of mutants between parallel experiments.

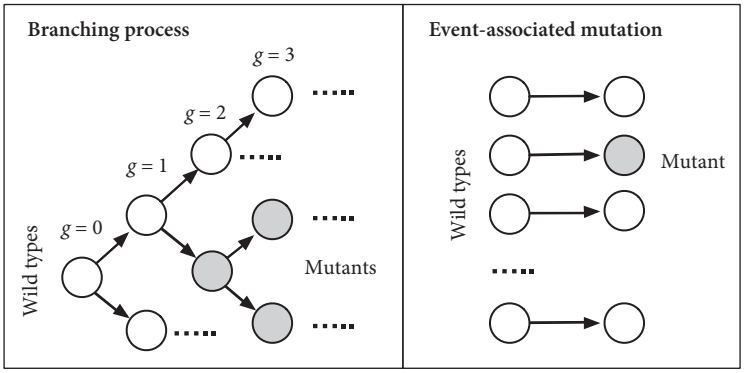


Figure 1.1: Stochastic models of mutation: Mutations are independent of selection (left) or dependent on selection (right). (Left) Branching process in which a single (or small) number of wild-type bacteria (empty cells) divide and occasionally mutate; the mutants (shaded) also divide. (Right) Mutation occurs randomly among a large population given interaction with a selective pressure, leading to a small fraction of mutants.

1.2 SAMPLING FROM PROVIDED DISTRIBUTIONS

In order to simulate stochastic processes, such as mutation in a population, one must repeatedly sample random numbers. Random numbers can be generated by any modern programming language. In doing so, it is possible to use built-in functions or to manipulate the generated random numbers to ensure they have a specified mean, variance, and higher-order moments. For example, to randomly sample a number between 0 and 1, use the command

```
import numpy as np
import matplotlib.pyplot as plt

np.random.rand()
```

Do this a few times. Each number is different. But generating multiple random numbers one at a time is unnecessary. Instead, generating multiple random numbers can be done automatically; e.g., use the following commands to randomly sample 100 points between 0 and 1:

```
randvec = np.random.rand(1,100)
```

or

```
randvec = np.random.rand(100,1)
```

These commands will generate a set of 100 random numbers in either a row or a column.

It is also possible, as shown in the introductory coding demos available on the book's website, to generate random matrices. Use the following command to generate a random matrix of size $m \times n$ array:

```
randarray = np.random.rand(m,n)
```

As is apparent, the shape of the matrix can be specified in terms of the number of rows `m` and columns `n`. If the code does not work, that is probably because you have not yet defined the size; do so a few times and see how easy it is to generate distinct random matrices. Note for future reference that two arrays must be the same size in order to perform element-wise operations (e.g., addition, subtraction, or element-by-element multiplication). Also note the names of variables—they tend to be descriptive. This is a good practice because it makes code easier to read, modify, and reuse. The first challenge problem should help get you more comfortable working with the core features of random distributions.

CHALLENGE PROBLEM: Properties of Random Distributions

What is the mean value of a single instance of invoking `np.random.rand`? Similarly, what is the variance? Once you have identified the mean and variance, plot the distribution of numbers generated by `np.random.rand` by sampling a large number of points (10^4) and then using the `plt.hist` function to generate a histogram. What shape is the distribution? How does it change as you change the number of bins for the histogram?

Python also allows sampling different distributions than the uniform distribution. As one exercise, plot the distribution of the output for the following functions: (i) standard normal distribution with a mean of 20 and standard deviation of 5 using

```
np.random.normal
```

and (ii) the Poisson distribution with rate parameter $\lambda = 20$ using

```
np.random.poisson
```

Examples of the outputs can be seen in Figure 1.2.

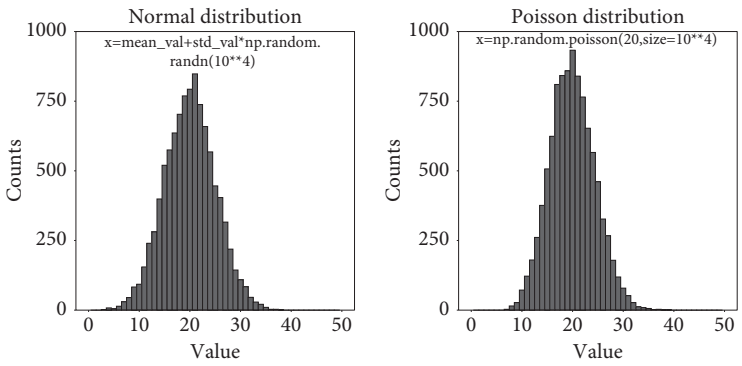


Figure 1.2: Sampling from random distributions, including the normal distribution (left) and the Poisson distribution (right).

It is possible to shift the range of randomly generated numbers using relatively simple operations, generating arbitrary variations (in range and location) of preexisting distributions. This may be useful in many circumstances, not only within the context of the LD problem. The following challenge problem provides an opportunity to build your intuition for manipulating and generating random numbers with distinct means and ranges.

CHALLENGE PROBLEM: Random Number Generation

This problem focuses on modifying the means and ranges of random numbers by modulating the output of built-in random number functions.

- Generate 1000 random numbers equally spaced between 0 and 5.
- Generate 1000 random numbers equally spaced between 2 and 7.
- Generate 1000 random numbers equally spaced between −5 and 5.

In each of these cases, use the built-in random number generator and then simple arithmetic (i.e., addition, subtraction, and multiplication) to transform the random numbers to specified ranges. You can do it!

1.3 SAMPLING FROM CUSTOM DISTRIBUTIONS

Python offers the option to generate specialized distributions. However, it is also possible to sample from "custom" distributions, i.e., both parametric and non-parametric distributions. One way to do so is to leverage the *cumulative distribution function*, or cdf. The cdf at a point, x, gives the probability of observing a value less than or equal to x. Formally, if $p(x)\mathrm{d}x$ is the probability of observing the random variable between x and $x + \mathrm{d}x$, then the cdf is

$$P(x) = \int_{-\infty}^{x} p(y)\mathrm{d}y \tag{1.1}$$

where y is a "dummy" variable used here for notational purposes of integrating over the probability distribution. The cdf is a monotonically increasing function with a range between 0 and 1. These constraints allow random sampling from arbitrary distributions if one is provided with the cdf in advance, by leveraging properties of the uniform distribution. An ideal way to illustrate this is via the exponential distribution.

The exponential distribution arises in many biological processes. For example, for processes that randomly occur with a constant rate λ, then the time of the first occurrence of an event is exponentially distributed such that $p(x) = \lambda e^{-\lambda x}$, given mean time $1/\lambda$. The cdf of the exponential distribution is $1 - e^{-\lambda x}$. Most numerical software tools have packages to sample exponential random numbers; this is precisely why it is instructive to compare the built-in solution to the custom solution. Indeed, one can think of the cdf of the exponential random distribution as having a one-to-one correspondence with the cdf of the uniform random distribution. That is, whereas half the values generated by a uniform random distribution will be < 0.5, that is not true for an exponential distribution. Instead, given the shape parameter λ, then half the values of an exponential distribution will have

values $x < x_u$ such that $1 - e^{-\lambda x_u} = 0.5$. This insight can help move from one distribution to the other.

To sample random numbers from the exponential distribution, first sample from the uniform distribution between 0 and 1.

```
probsamp = np.random.rand()
```

Think of this as a random value of P, which we denote as c_u. By randomly sampling the cdf of the uniform random distribution, the next question becomes: what value of the exponentially distributed random variable x_e corresponds to that point in the cdf? To answer this question requires that we invert the cdf, i.e., $P = 1 - e^{-\lambda x_e}$, to obtain an equation of x_e in terms of the cdf. To show this in action, denote c_u as the randomly selected value from the cdf of the uniform distribution. To map the cdf of the uniform distribution onto the cdf of the exponential distribution (our custom distribution) requires that $c_u = 1 - e^{-\lambda x_e}$. For x_e to be an exponentially distributed random number requires transforming the uniform random numbers into the exponentially distributed random numbers we would like to generate:

$$\begin{aligned} 1 - e^{-\lambda x_e} &= c_u \\ e^{-\lambda x_e} &= 1 - c_u \\ -\lambda x_e &= \log 1 - c_u \\ x_e &= \frac{-\log 1 - c_u}{\lambda} \end{aligned} \tag{1.2}$$

This gives $x_e = -\frac{1}{\lambda}\log(1 - c_u)$. This converts the sampling distribution of `rand` (i.e., c_u) to an exponential distribution. In order to use this repeatedly, it will be convenient to make and save a function:

```
def rand2exp(probsamp,my_lambda):
    # Do not overwrite the lambda Python function
    # lambda is the rate of the Markov process
    # probsamp are uniformly random sampled numbers
    return -1/my_lambda*np.log(1-probsamp)
```

The following section leverages the prior code snippet for using uniform sampling to generate exponentially distributed numbers given a process with rate $\lambda = 1/10$:

```
my_lambda=1/10
probsamp=np.random.rand(10**3)
expprobsamp = rand2exp(probsamp,my_lambda)
```

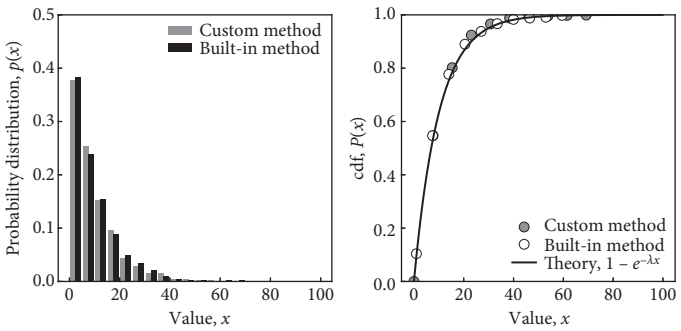
Now it is time to see if any of this works—via a challenge problem.

CHALLENGE PROBLEM: Comparing Exponential Random Sampling

Compare the distribution of 10^3 exponentially distributed random numbers using the cdf-based method to the distribution using the following built-in Python command:

```
my_lambda=1/10
pythonexprnd=np.random.exponential(1/my_lambda,10**3)
```

Note that the function `random` allows sampling from a number of different common distributions. (Hint: Another helpful function is the *empirical cumulative distribution function*, or `ECDF`, from `statsmodels.distributions.empirical_distribution`. This is useful in generating cumulative distributions.) If your code is working, it should look like the following:



These figures show a comparison of customized sampling and built-in exponential random sampling via probability distributions (left) and cumulative distributions (right). For the cdf, the expected distribution is shown as a solid black line.

1.4 COMPARING BINOMIAL AND POISSON DISTRIBUTIONS

Binomial distributions result from counting the number of occurrences given independent samples with probability of occurrence p. For example, consider a mutation probability of $p = 10^{-8}$. Irrespective of whether mutations are independent of or dependent on selection, it would typically take a large number of cell divisions (or cells) for a mutant to appear. Using a binomial distribution, one could, in theory, predict the number of mutants expected to occur in a single round of cell division or given an exposure of a large collection of n cells to a selective force. Formally, the binomial distribution denotes the probability that k events occur out of n trials given the per trial probability p. This distribution is

$$P(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{1.3}$$

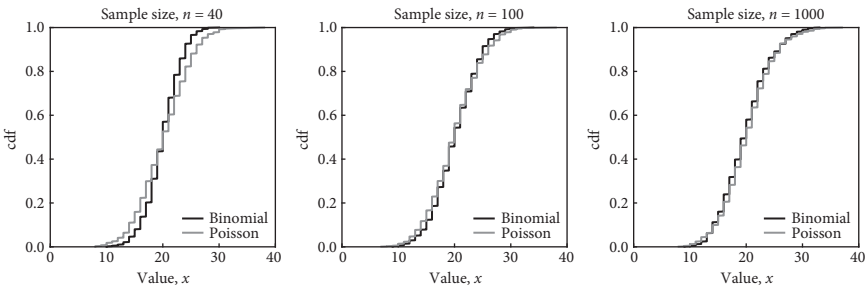
where $\binom{n}{k}$ denotes the number of unique ways of choosing k of n elements (i.e., the binomial coefficient). However, if occurrences are rare and the number of samples, n, is large, then the binomial distribution converges to the Poisson distribution with shape parameter $\lambda = np$ (this shape is the expected mean number of events in n trials). To see this computationally, compare the cdfs of the sample of repeated binomial sampling to repeated Poisson sampling with varying n. For example, use the following code to obtain and plot the cdf for binomial random numbers given 100 trials each with probability $p = 0.2$:

```
n=100
my_lambda = 20
p=my_lambda/n
numsamps=10**3
binosamps = np.random.binomial(n,p,numsamps)
sortbino = np.sort(binosamps)
cdfbino = np.arange(1,numsamps+1)/numsamps
plt.plot(sortbino,cdfbino,color='k',linewidth=2)
```

Here `binomial` samples binomial random numbers, and sorting the result allows for an explicit calculation of the empirical cdf (without using a built-in function).

CHALLENGE PROBLEM: Comparing the Binomial to the Poisson

Compare the binomial and Poisson cdfs for $n = 40$, 100, and 1000 in each case, assuming there is an expected number of 20 events such that the probability per event decreases from 0.5 to 0.2 to 0.02, respectively. If your code is working, it should look something like this:



Technical note: Keep in mind that the `binomial` function generates the outcome of `n` trials each with a `p` probability of success. Try to compare outcomes with `sum(np.random.rand(n)<p)`. Are the outcomes different in a substantive way than simply sampling from a binomial? It is worth considering that the binomial distribution is equivalent to running n trials each with a p probability of success and then reporting how many, m, were successful. By definition, $0 \leq m \leq n$. Hence, each trial is successful with probability p. Because `rand` returns uniformly distributed random numbers between 0 and 1,

then `rand < p` is 1 with probability p and 0 with probability $1 - p$. As such, by invoking the `rand` command n times and comparing it to p, it should return a 1 approximately np times; this is, by definition, λ from above. Hence, just as we used the uniform random distribution to generate exponentially distributed numbers, it is also possible to use the same distribution to generate random events that have precisely the same properties as binomial random numbers.

1.5 THE START OF DYNAMICS

The schematics in Figure 1.1 illustrate two distinct mechanisms by which mutant bacteria can increase in number in a population. Via the independent mutation hypothesis, mutations happen rarely during cell division and then selection acts upon them later. Via the dependent mutation hypothesis, mutations only occur when the cell experiences a selection pressure, and in that case a small fraction of heritable cells acquire a mutation. The consequences of these two ideas are examined at length in the main text and then developed as the centerpiece of the homework problems. Yet to get there requires that you develop a dynamic simulation.

Rather than giving away the homework (and the fun involved in doing this yourself), there is a way to start along the path toward dynamics. First, consider the case where mutations are dependent on selection. It should be apparent that manipulating probability distributions as described here can be used to generate a small number of mutants in a population. For example, consider the case where there are $n = 10^5$ cells and $p = 10^{-4}$. In that event, one expects approximately 10 mutational events, which can be generated as follows: `sum(np.random.rand(n)<p)`. Yet the case of the independent selection is more difficult.

As a start to a dynamic model, consider a two-step process. First, a population of cells, with certain features doubles in size. Second, a fraction of the cells changes in some way. Simply to illustrate this point, initialize a 1× 5 array with 0.5 in the second entry, e.g.,

```
x=np.zeros(5)
x[1]=0.5
```

Next, double the size of the array. How to do this is up to you. Indeed, doubling an array is perhaps a crude way to simulate a dynamic process, but it provides some intuition to the underlying changes in the system. It also helps illustrate ways to concatenate matrices together, e.g., `np.concatenate([x, x])` Examine the output of `y` and notice that there are now instances of a 0.5 value, in the second and seventh positions. This is a direct result of concatenating the matrices. Now, if the value of 0.5 was some property of a cell, then it is clear that two cells have that same property. If instead one used a value of 1 for a mutant and 0 for a wild type, then it is apparent that the process of cell division (which doubled the number of cells) also doubled the number of mutants. Of course, at this point it would be important to change the property of `y` in the event of a new mutation. In that case, you can use the random number generating methods already described to decide which, if any, of the array elements to change.

Of course, if you want to see what happens in a few instances, consider this loop:

```
x=[1,0]
for i in np.arange(4)
    x=np.concatenate((x,x),axis=None)
```

The result should be a growing list of 0s and 1s:

```
1  0
1  0  1  0
1  0  1  0  1  0  1  0
1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0 1 0 1 0 1 0 1 0 1 0 1 0 1 0...
```

This kind of approach loses track of the mother-daughter relationships (at least explicitly). But it is possible to modify the arrays and then begin to change both the size and the nature of the population.

How to build models of bacterial growth and mutation is treated in detail in the textbook (and associated homework). From a computational perspective, such models are built around a few simple ideas, including adding elements to an array and changing the value of an array. For example, here are a series of small exercises that illustrate core concepts toward building your own simulation model of bacterial growth and mutation. Type in each and modify them. Soon you may just be ready to tackle the question of whether mutations are dependent on or independent of selection.

CHALLENGE PROBLEM: A Step toward Bacterial Growth

Write a program to generate an in silico population of 100 bacteria, of which ≈ 90% are wild types and the rest are mutants (denote these as 1s and 0s, respectively). Then double the size of this population while retaining the properties of the original population. Finally, switch one element, either 0 to 1 or 1 to 0.

1.6 INFERRING PARAMETERS FROM DATA

Thus far, this laboratory has provided resources for sampling from and manipulating different probability distributions—with an eye toward developing dynamic simulations of growing and mutating bacterial populations. These can be used in a generative sense, as described in the textbook, to compare and contrast the independent and acquired mutational hypotheses. However, there is another question that is relevant to hypothesis testing: how to infer process rates and parameters from data. To tackle such an approach, first download the file `poissdata.csv`, which contains 100 random samples from Poisson distributions with an unknown rate parameter. Or you can enter the following string of numbers into an array. Here it is—exciting, no?

3,4,2,5,2,2,5,0,5,2,4,4,4,1,4,3,3,2,3,2,2,6,3,4,4,5,2,2,5,0,1,2,2,2,4,3,
3,2,4,5,2,4,6,3,5,5,1,3,1,2,2,5,4,8,4,3,5,2,6,3,3,2,3,4,4,3,2,2,3,2,6,2,
2,0,2,5,4,5,4,5,3,9,3,5,2,6,3,5,1,1,2,1,4,2,5,7,4,3,4,4

Although this seems abstract, imagine that these numbers correspond to resistance colonies measured after a Luria-Delbrück (LD) experiment—it turns out that these have features quite distinct from the LD experiments, but they nonetheless provide a good basis for deeper exploration. The remainder of this lab is aimed at estimating the rate parameter, i.e., the unknown λ, from which one could estimate the unknown mutation rate. These steps are the centerpiece of the homework. Hence, it's worthwhile to take some time to understand the *inverse* problem using a simpler example.

The central objective of parameter inference is to try to identify a value (or set of values) that is compatible with observations. The degree of compatibility may depend on one's preference for the unexpected. In practice, most inference approaches try to ask the question: what is the probability that some unknown parameter θ is compatible with the observed data x, or $P(\theta|x)$? Yet, to answer that question, it is often critical to answer a related but different question: what is the likelihood of observing the data x given a parameter θ, or $P(x|\theta)$? These are not the same and, in fact, can be quite different (the literature on false positives in medical testing is an excellent example for study).

In this case, one way to estimate the rate parameter is to use features of the data—and find parameters that are expected to generate similar features. In this case, the pdf for the Poisson distribution is $p(x=k)=\frac{\lambda^k e^{-\lambda}}{k!}$. This means the probability of observing 0 occurrences is $p(0)=e^{-\lambda}$, the probability of observing 1 occurence is $\lambda e^{-\lambda}$, the probability of observing 2 occurences is $\frac{\lambda^2}{2}e^{-\lambda}$, etc. Note that the Poisson distribution is defined over discrete values such that the sum of these must be 1, i.e., $\sum_{k=0}^{\infty} p(x=k|\lambda)=1$.

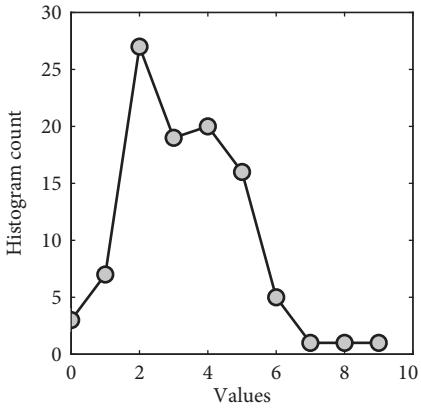
One of the features that Luria and Delbrück were interested in was simply the fraction of experiments in which nothing happened—meaning no mutant colonies formed on the agar plates after being exposed to viruses. This feature, the probability of zeros, can be used to infer a rate parameter. In the example, inverting the equation for $p(x=0)$ leads to an estimate of λ based on the data: $\lambda=-\log(P(x=0))$. To calculate the probability of observing 0 from the data, use

```
poissdata = np.genfromtxt('poissdata.csv',delimiter=',')
numberofzeros = np.sum(poissdata==0)
probzero = numberofzeros/len(poissdata)
```

where `sum(poissdata==0)` counts the number of zeros, which then can be used to infer the associated Poisson shape parameter as follows:

- Estimate λ and save the values as `lambda_est`.
- Write code that takes a vector of data as input and outputs the estimated λ based on the number of zero occurrences. Name this function `lambda_estimator_zeros`.

Of note, the data in `'poissdata.csv'` looks like the following, which you should plot to verify:



Try to write such a script on your own. However, for your reference, here is one solution script. The danger of course is that, if there are no zeros, then the estimator is undefined. Note that there is another way to estimate λ by using the average value of the output:

```
def lambda_estimator_zeros(x):
    # function lambda_est = lambda_estimator_zeros(x)
    # Estimates the Poisson rate parameter associated with a vector
    # of points x based on the zero
    numberofzeros=np.sum((x==0)*1)
    probzero=numberofzeros/len(x)
    lambda_est = -np.log(probzero)
    return lambda_est
```

What does all of this mean? According to the Poisson distribution, if λ_{true} is the true value, then we should observe an output of 0 a fraction $e^{-\lambda_{true}}$ of time. In the dataset, there are three zeros out of 100 trials. Hence, the observed probability of 0 is $P_{obs}(0) = 0.03$. Hence, our best estimate is $\hat{\lambda} = -log(P_{obs})$, or $\hat{\lambda} = 3.51$. It turns out that is not quite right, yet it is also not surprising; that is, such an output is expected given the true but unknown value λ_{true}. It turns out that the true value is $\lambda_{true} = 3$. But you didn't know that in advance, did you? And that is the point.

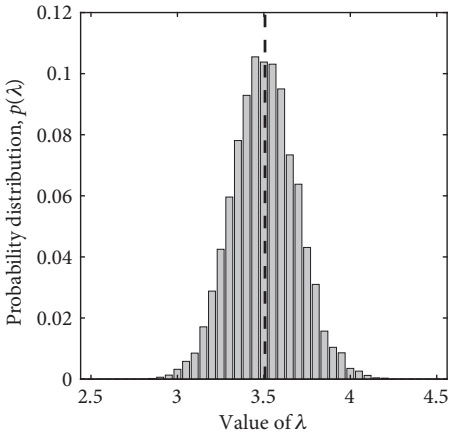
Indeed, Luria and Delbrück didn't know what the actual mutation rate was before the experiment (even if they had some idea that it was small, and even some idea of the level of smallness). In the case of any particular estimate, one may ask how confident the estimate of the rate value is. In other words, how often does sampling 100 points from a Poisson distribution with a predetermined rate of λ_{true} lead to similar best estimates of λ_{est}? One way to quantify similarity is to ask whether `lambda_est` lies within the middle 95% of a distribution of estimates obtained from a specified λ_{true}. To get a better sense, let's look at the distribution when we set λ_{true} equal to `lambda_est`. However, in estimating the Poission rate parameter, it is necessary to use a stable statistic—the mean value. Note that the expected value of the Poisson distribution is:

$$\langle x \rangle = \sum_{k=0}^{\infty} kP(k). \tag{1.4}$$

The mean, as it turns out, is simply $\langle x \rangle = \lambda$:

```
numsamps = 10**4
lambdadist = np.zeros(numsamps)
for j in range(numsamps):
    currdata = np.random.poisson(lambda_est,100)
    lambdadist[j] = np.mean(currdata) # Best estimate of $\lambda$
```

Next, plot a normalized histogram of this distribution and address whether `lambda_est` appears to be contained in the middle 95% of the distribution. As seen in the following plot—the answer is yes (as you should have expected):



This histogram can be generated using the following code:

```
# Main data goes here
x = np.genfromtxt('poissdata.csv',delimiter=',')
lambda_est = lambda_estimator_zeros(x)

numsamps = 10**4
lambdadist = np.zeros(numsamps)
for i in range(numsamps):
    currdata = np.random.poisson(lambda_est,100)
    lambdadist[i] = np.mean(currdata)

n, bin_edges = np.histogram(lambdadist, bins=30)
bin_probability = n/numsamps
bin_middles = (bin_edges[1:]+bin_edges[:-1])/2.
bin_width = bin_edges[1]-bin_edges[0]
plt.bar(bin_middles, bin_probability, width=bin_width,
        color=[0.75,0.75,0.75],edgecolor='k')
plt.plot([lambda_est, lambda_est],[0, 0.12],'k--',linewidth=3)
plt.ylabel(r'Probability Distribution, $p(\lambda)$', fontsize=14)
plt.xlabel(r'Value of $\lambda$', fontsize=14)
```

Although it is apparent that the first estimate of λ does lie within the center, you can formally identify the bounds to the middle 95% by sorting the distribution as follows:

```
sortedlambdadist = np.sort(lambdadist)
lower025 = sortedlambdadist[int(0.025*numsamps)]
upper975 = sortedlambdadist[int(0.975*numsamps)]
```

Such an outcome might not always be the case. What if you had set $\lambda_{true} = 1$ instead of `lambda_est` (which is equal to 3.51)? To estimate the confidence intervals of the estimated value of λ, one must establish the expected largest and smallest value of λ_{true} with associated distributions that contain `lambda_est` in the middle 95%. We can accomplish this by systematically looping over values of lambda and repeating the analysis above:

```
lambdavec = np.arange(0.5*lambda_est,1.5*lambda_est,0.01)
lower025 = np.zeros(len(lambdavec))
upper975 = np.zeros(len(lambdavec))

for j in range(len(lambdavec)):
    currlambdaset = lambdavec[j]
    poissfitdist = np.zeros(numsamps)
    for k in range(numsamps):
        currdata = np.random.poisson(currlambdaset,100)
        poissfitdist[k] = np.mean(currdata)
    sortedlambdadist = np.sort(poissfitdist)
    lower025[j] = sortedlambdadist[int(0.025*numsamps)]
    upper975[j] = sortedlambdadist[int(0.975*numsamps)]
```

CHALLENGE PROBLEM: Estimating the Confidence in Parameters

In this last problem, plot the lower and upper bounds of the realized values, λ_{obs}, given a range of true values for λ. Then use these forward likelihoods to answer two associated inference problems. First, what is the maximal value of λ with an upper bound below `lambda_est`? Second, what is the minimal value of λ with a lower bound above `lambda_est`? Interpret your findings with respect to the certainty you would have about the underlying value λ_{true} given observations.

SOLUTIONS TO CHALLENGE PROBLEMS

SOLUTION: Properties of Random Distributions

The mean value of the output of `rand` is 0.5 because the values are uniformly spaced between 0 and 1. The variance of a distribution is defined as $\text{Var} = \langle x^2 \rangle - \langle x \rangle^2$, in other words, the expectation of the sampled value squared minus the square of the expected value of the sampled value. For a uniform distribution such that $p(x) = 1$ for $0 \leq x < 1$,

the variance is

$$\mathrm{Var}_x = \int_0^1 x^2 p(x) - \left(\int_0^1 x p(x) \right)^2 \tag{1.5}$$

$$= \frac{1}{3}x^3\Big|_0^1 - \left(\frac{1}{2}x^2\Big|_0^1\right)^2 \tag{1.6}$$

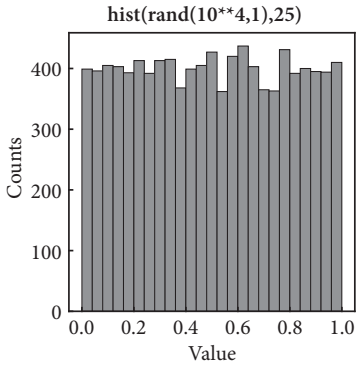
$$= 1/3 - (1/2)^2 \tag{1.7}$$

$$= 1/12 \tag{1.8}$$

This result can be verified entering the command `np.var(np.random.rand(10000))`, which returns the variance of 10,000 uniformly distributed random numbers and a number very close to 1/12, or 0.0833. A histogram can be used to visualize the random numbers. As is evident, the shape of the distribution seems largely "flat," though the noise increases with the number of bins. The expected number in a bin is itself a different sampling problem, i.e., a multinomial problem, a topic for a different day. The code to generate the histogram is

```
# Histogram with 25 bins
plt.hist(np.random.rand(10**4),25,\
         facecolor=[0.5,0.5,0.5],edgecolor='k')
plt.xlabel('Value',fontsize=20)
plt.ylabel('Counts',fontsize=20)
plt.title('hist(rand(10**4,1),25)',fontsize=20)
```

This code bins 10^4 random numbers into 25 bins and then visualizes them with labeled axes and a title.



SOLUTION: Random Number Generation

In order to generate this set of random numbers, recall that the `rand` command generates uniformly distributed numbers between 0 and 1. Hence, if you add or subtract a value, then you can shift the range. Moreover, if you multiply the output of `rand` by

a constant, you can expand the range. Judicious use of these techniques suggests the following solutions:

- Generate 1000 random numbers equally spaced between 0 and 5: `np.random.rand(1000)*5`
- Generate 1000 random numbers equally spaced between 2 and 7: `np.random.rand(1000)*5 + 2`
- Generate 1000 random numbers equally spaced between −5 and 5: `np.random.rand(1000)*10 - 5`

SOLUTION: Comparing Exponential Random Sampling

The two distributions can be compared using their pdfs (probability distribution functions) or cdfs (cumulative distribution functions). A comparison of the histograms of the pdfs reveals the exponential shape. Transforming the *y* axis to logarithmic scale would also reveal a linear decline (a bonus challenge for the interested reader). Second, if comparing cdfs, then both distributions map onto each other. In the case of the cdf, the theoretically expected functional form is plotted as an overlay. For plotting pdfs, the following code snippet is useful:

```
N=10**3
my_lambda=1/10
probsamp = np.random.rand(N)
expprobsamp = rand2exp(probsamp,my_lambda)
Pyexprnd = np.random.exponential(1/my_lambda,N)
# Get the histograms
binrange=np.arange(0,105,5)

# Overlay the histograms side by side
plt.hist([expprobsamp, Pyexprnd], binrange,
         weights = [np.ones(N)/N, np.ones(N)/N],
         label=['Custom method','Built-in method'],
         color=['gray', 'black'])
plt.legend(fontsize=12,frameon=False)
```

For plotting cdfs, the following code snippet is useful. In particular, note that this code samples the cumulative distribution at different points to help reveal the overlay of the two approaches to the theoretical distribution.

```
# Generate the random numbers
N=10**3
my_lambda=1/10
probsamp = np.random.rand(N)
```

```
expprobsamp = rand2exp(probsamp,my_lambda)
Pyexprnd = np.random.exponential(1/my_lambda,N)

# Get the empirical cdfs
from statsmodels.distributions.empirical_distribution import ECDF
f1=ECDF(expprobsamp)
f2=ECDF(Pyexprnd)
x1=f1(np.arange(0,max(expprobsamp),max(expprobsamp)/10))
x2=f2(np.arange(1,max(Pyexprnd)+1,(max(Pyexprnd)+1)/10))

# Overlay the cdfs
# First overlay
plt.plot(np.arange(0,max(expprobsamp),max(expprobsamp)/10),x1,'o',
         color=[0.5,0.5,0.5],markeredgecolor='black',markersize=10)
# Different overlay
plt.plot(np.arange(1,max(Pyexprnd)+1,(max(Pyexprnd)+1)/10),x2,'o',
color='white',markeredgecolor='black',markersize=10)
# Theory
x3=np.arange(0,100,0.1)
plt.plot(x3,1-np.exp(-my_lambda*x3),'-', color='black')
plt.legend(['Custom method','Built-in method',\
            r'Theory, $1-e^{-\lambda x}$'], fontsize=12,\
            loc='lower right',frameon=False)
```

SOLUTION: Comparing the Binomial to the Poisson

The comparison is facilitated by generating samples using the following command: `poisssamps = np.random.poisson(n*p,size=numsamps)`. The convergence between binomial and Poisson improves markedly with increasing n, as is apparent in the three sets of cdfs. In each of these cases, 1000 samples were taken given a process that should have an average value of 20. However, when the total number of trials increases from $n = 40$ where $p = 0.5$ to $n = 1000$ where $p = 0.02$, then the convergence to the conditions of the Poisson apply, i.e., large number of trials each with a low probability of success.

SOLUTION: A Step Towards Bacterial Growth

First, generate an array of 0s and 1s, using the `np.random.choice` function, in which ≈ 90% of them are 1s:

```
x = np.random.choice([1,0],size=(100,1),p=[0.9,0.1])
```

Next, try to change a specific type, e.g., by changing it from 0 to 1 or 1 to 0.

```
ind = 20
x[ind]=1-x[ind]
```

Check the value of `x` before and after to verify that the value did indeed flip. Now double the array in size, copying all elements. There are many ways to do this. In Python, the simplest way is to use the vectorized approach with `np.stack` (though a `for` loop could also work):

```
x.shape
x2 = np.concatenate((x,x),axis=0)
x2.shape
```

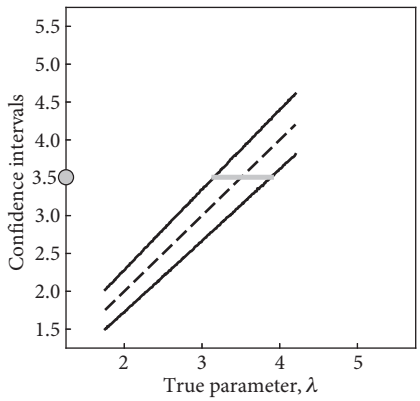
This simple command stacks the two column vectors atop each other. In contrast, to copy a column vector so that the resulting array has two (equal) column vectors, use `axis=1`:

```
x.shape
x2 = np.concatenate((x,x),axis=1)
x2.shape
```

Note in the example above that 2 values were given for the size when creating `x`. This allows Python to differentiate between a column vector and a row vector. If a single number is given, Python will default to creating a 1D array without any notion of rows or columns. In the two examples above, the `shape` attribute provides information on the number of columns and rows. Hence, if `x` did have information on the wild-type and/or mutant state of many bacteria, then a simple stacking command or direct modification of states could be used to explicitly modify the status of a population at a given time or from one generation to the next. How you use these and other techniques to build a simulation model of cell growth and mutation is up to you!

SOLUTION: Estimating the Confidence in Rates

The plot illustrates the bounds, calculated using a computational approach to generate estimates based on sampled data and then sorting the estimates to identify 95% confidence intervals. Yet to find compatibility requires that we look at what true value of λ could be compatible with the observation of an estimate of 3.51. By looking horizontally across the y axis, one can observe that if the true value had been 2.95, then the value of 3.51 would be a plausible upper limit. Similarly, if the true value had been 3.95, then the value of 3.51 would be a plausible lower limit. This provides a rationale for a confidence interval, as illustrated using the zeros method. See the image below, including a horizontal line at the `lambda_est` value.



The code to generate this plot is as follows:

```
## Plot commands for estimating lower and upper bounds
plt.plot(lambdavec,lower025,linewidth=3, color='k')
plt.plot(lambdavec,upper975, linewidth=3, color='k')
plt.plot(lambdavec,lambdavec,linewidth=2, linestyle='--', color='k')
tmpi=np.where(lower025 > lambda_est)[0]
llow=lambdavec[tmpi[0]]
tmpi=np.where(upper975 < lambda_est)[0]
lhigh=lambdavec[tmpi[-1]]
plt.plot([llow, lhigh],[lambda_est, lambda_est],
         linewidth=4,color=[0.75,0.75,0.75])
plt.plot(1.25,lambda_est,'ko',\
         markerfacecolor=[0.75,0.75,0.75],markersize=12)
plt.xlim([1.25, 5.75])
plt.ylim([1.25, 5.75])
```

It is possible to use different summary statistics other than the number of zero occurrences. If time permits, repeat this analysis using the mean of the Poisson distribution (equivalent to λ).