begindocument/before

QBioS Laboratory: Fluctuations and the Nature of Mutations Serrapilheira/ICTP-SAIFR training program

Joshua S. Weitz[1, 2, *]

[1] *School of Biology, Georgia Institute of Technology, Atlanta, GA, USA*
[2] *School of Physics, Georgia Institute of Technology, Atlanta, GA, USA*

(Dated: July 8, 2022)

Instructor: Adriana Lucia-Sanz – agarcia337@gatech.edu

I. HANDS-ON APPROACH TO MUTATIONS AND SELECTION

The goal of this lab is to simulate a growing bacterial population including both 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 R. 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 lab will introduce basic concepts that will be used throughout the laboratory guides. This chapter also serves another function, to link the main material in the textbook with the homework.

To do so, the laboratory will prepare you to build components of two categories of mutational models as illustrated in a generalized schematic form in Figure 1. These initial compoments 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 bacteria 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 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 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.

II. 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:

```
runif(1)
```

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

```
randvec = runif(100)
```

*Electronic address: `jsweitz@gatech.edu`; URL: `http://ecotheory.biology.gatech.edu`

Typeset by REVTEX

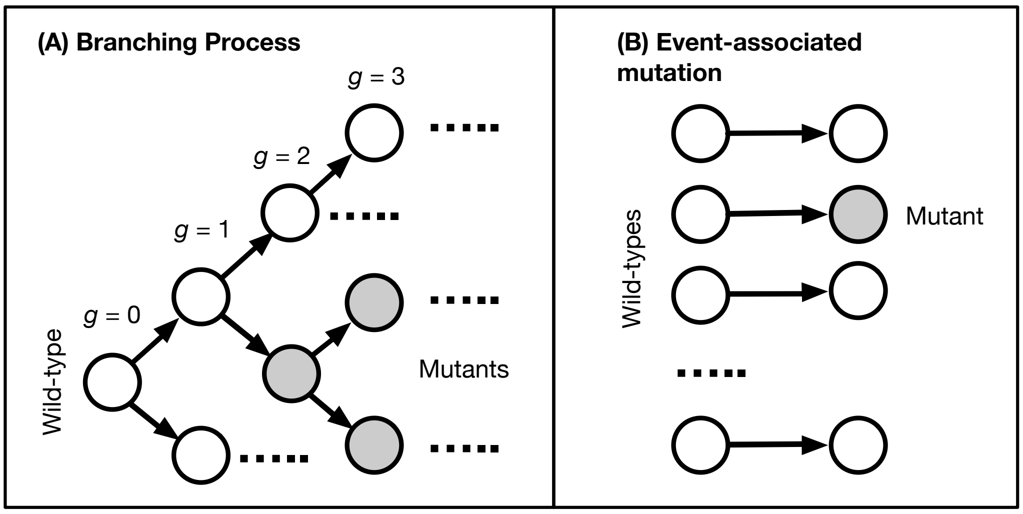


FIG. 1: Stochastic models of mutation: mutations are independent of selection (A) or dependent on selection (B). (A) Branching process in which a single (or small) number of wild-type bacteria (empty cells) divide and occasionally mutate, the mutants (shaded) also divide. (B) Mutation occur randomly amongst a large population given interaction with a selective pressure, leading to a small fraction of mutants.

This commands will generate a set of 100 random numbers.

It is also possible, as was shown in the Introduction, to generate random matrices. To generate a random matrix of size $m \times n$ array:

```
randarray = matrix(runif(100),nrow=m,ncol=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 npt 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, 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 re-use. 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 `runif`? Similarly, what is the variance? Once you have identified the mean and variance, plot the distribution of numbers generated by `runif` by sampling a large number of points (10^4) and then using the ggplot2 `geom_histogram` function to generate a histogram. What shape is the distribution? How does it change as you change the number of bins for the histogram?

R 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

```
rnorm
```

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

```
rpois
```

Examples of the outputs can be seen in Figure 2.

It is possible to shift the range of randomly generated numbers using relatively simple operations, generating arbitrary variations (in range and location) of pre-existing 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.

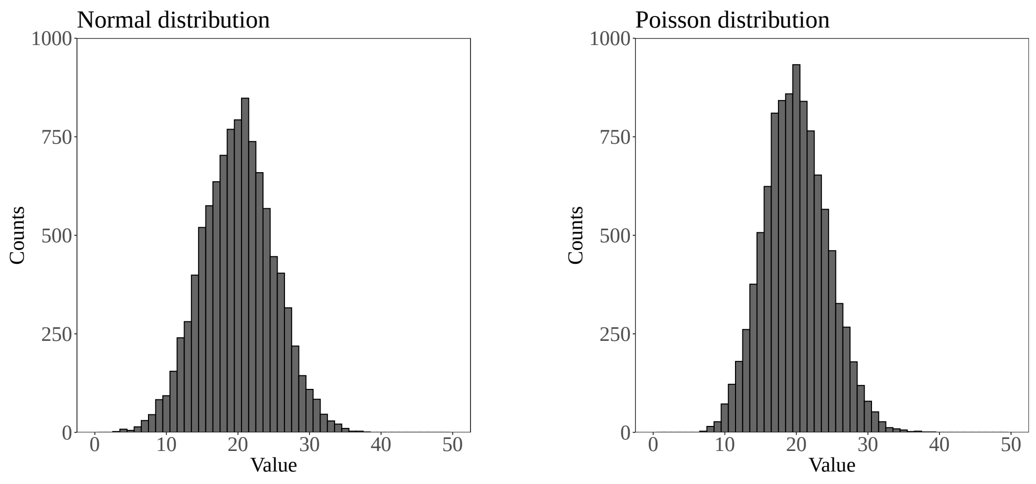


FIG. 2: Sampling from random distributions, including the normal distribution (left) and the Poisson distribution (right).

Challenge Problem: Random number generation

The following problems focus on modifying the mean and ranges of random numbers by modulating the output of built-in random number functions.

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

In each of these cases, use the command `rand(1,1000)` and then simple arithmetic (i.e., addition, subtraction, and multiplication) to transform the random numbers to specified ranges. You can do it!

III. SAMPLING FROM CUSTOM DISTRIBUTIONS

R 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', what we will term a '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

$$P(x) = \int_{-\infty}^{x} p(y)\mathrm{d}y \tag{1}$$

where y is a dummy variable used here for the purposes of integrating over the probability distribution. The cdf is a monotonically increasing function with a range between 0 and 1. These constraints allow randomly 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 occur at 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 that 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 to move between one distribution to the other.

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

```
probsamp = runif()
```

Think of this as a random value of P, which we will 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-s of the uniform distribution onto that of 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 randomly numbers we would like to generate:

$$\begin{aligned} 1 - e^{-\lambda x_e} &= c_u \\ e^{-\lambda x_e} &= 1 - c_u \\ -\lambda x_e &= \log 1 - c_u \\ x_e &= \frac{-\log 1 - c_u}{\lambda}. \end{aligned} \tag{2}$$

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.

```
rand2exp = function(probsamp,my_lambda){
    # function exprand = rand2exp(probsamp,lambda)
    # lambda is the rate of the Markov process
    # probsamp are uniformly randomly sampled numbers
  return(-1/my_lambda*log(1-probsamp))
}
```

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

```
my_lambda = 1/10
probsamp = runif(10**4)
expprobsamp = rand2exp(probsamp,my_lambda)
```

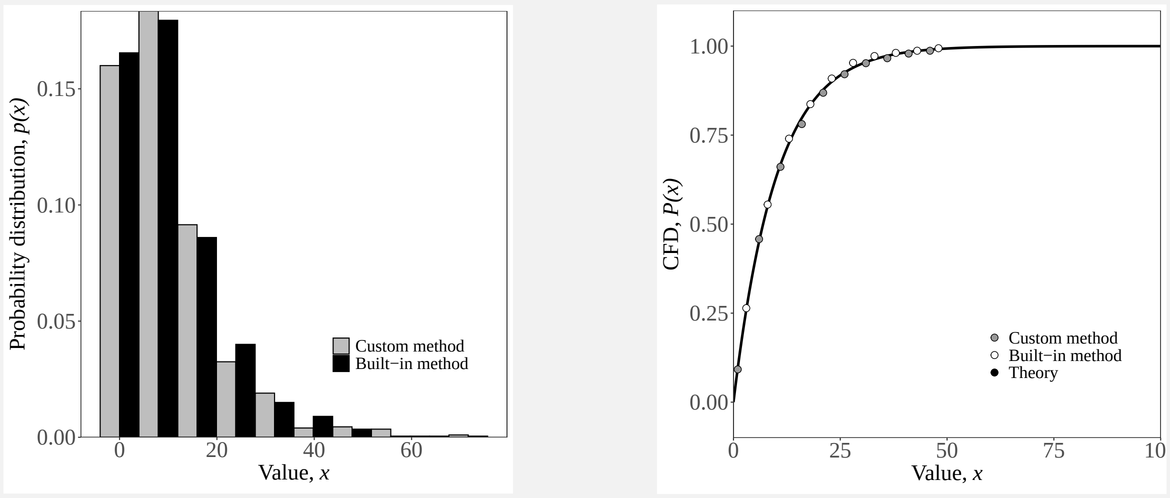
Now it is time to see if any of this works – via a Challenge Problem.

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 R command:

```
my_lambda = 1/10
Rexprnd = rexp(10**3,my_lambda)
```

Hint: the other useful function is the 'empirical cumulative distribution function' or `ecdf` from the `stats` package. This will be 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.

IV. 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 of selection, it would take a large number of cell divisions (or cells) for a mutant to appear. Using a binomial distribution one could, in theory, predict the expected 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:

$$P(k) = \binom{n}{k} p^k \left(1-p\right)^{n-k} \tag{3}$$

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
lambda = 20
p = lambda/n
numsamps = 10^3
binosamps = rbinom(numsamps,n,p)
```

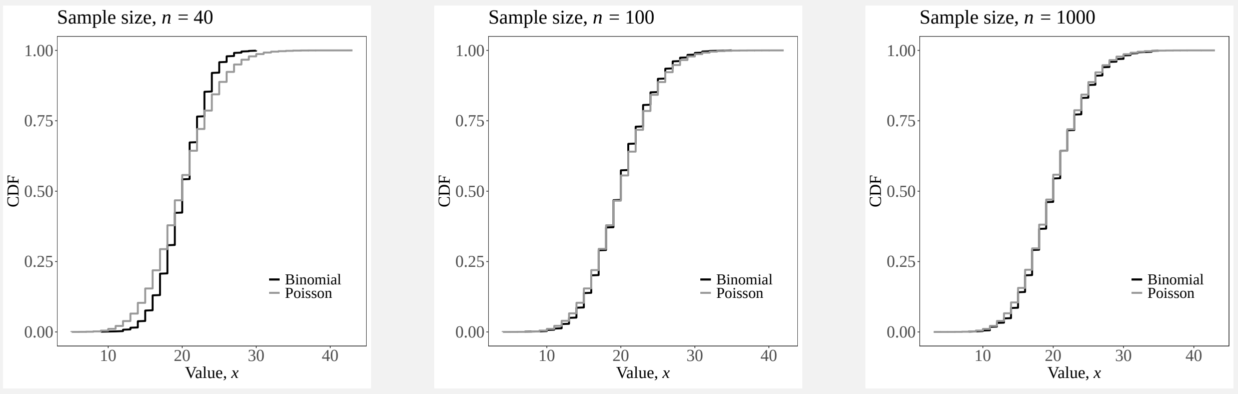
```
sortbino=sort(binosamps)
cdfbino=(1:numsamps)/numsamps


poissamps = rpois(n*numsamps,lambda)
sortpois = sort(poissamps)
cdfpois = (1:(numsamps*n))/(numsamps*n)
ggplot() +
  geom_line(aes(sortbino,cdfbino,col="Binomial")) +
  geom_line(aes(sortpois,cdfpois,col="Poisson"))
```

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 that there is expected number of 20 events; such that the probability per event decreases from 0.5 to 0.2 to 0.02. If your code is working, it should look something like:



Technical note: Keep in mind that the `binomial` function generates the outcome of `n` trials each with a `p` probability of success. Try and compare outcomes with the following `sum(np.random.rand(n)<p)`. Are the outcomes different in a substantive way than simply sampling from a Binomial? It is worth considering the fact 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 $rand < p$ will be 1 with probability p and 0 with probability $1 - p$. As such by invoking the `rand` command n times and comparing it to p, then 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.

V. THE START OF DYNAMICS

The schematics in Figure 1 illustrate two distinct mechansims 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 cells heritable acquire a mutation. The consequences of these two ideas are examined at length in the main text and then developed as the center-piece 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 towards dynamics. First, consider the case where mutations are dependent on selection. It should be apparent that manipulating probability distributions as desrcibed 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 change, in some way. Simply to illustrate this point, initialize a 1×5 array with a 0.5 in the second entry, e.g.,

```
x=rep(0,5)
x[2]=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 to illustrate ways to concatenate matrices together, e.g.,: `y=c(x,x)` Examine the output of `y` and notice that there are now instances of a 0.5 value, in the 2nd and and 7th positions. This is a direct result of stacking (or 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 the `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=c(1,0)
for(i in 1:4){
  x=c(x,x)
}
```

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

```
  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  1  0
```

This kind of approach loses track of the mother-daughter relationships (at least not explicitly). But it is possible to modify the arrays and then begin to change both the size and nature of the population.

How to build models of bacterial growth and mutation in detail are treated in the lecture notes (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 towards 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 Towards Bacterial Growth

Write a program to generate an *in silico* population of 100 bacteria, of which $\approx 90\%$ are wild-types and the rest are mutants (denote these as 1-s and 0-s 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.

VI. INFERRING PARAMETERS FROM DATA

Thus far, this laboratory has provide resources for sampling from and manipulating different probability distributions – with an eye towards developing dynamic simulations of growing and mutating bacterial populations. These can be used in a generative sense, as described in the main 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 distribution 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 the number of resistance colonies after a 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 will be the centerpiece of the homework. Hence, it's worthwhile taking some time to understand the *inverse* problem using a simpler example.

The central objective of parameter inference is to try and identify a value (or set of values) that are 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 probability density function (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, the Poisson distribution is defined over discrete values such that the sum of these must be 1, i.e., $\sum_{k=0}^{\infty} p(x=k|\lambda) = 1$.

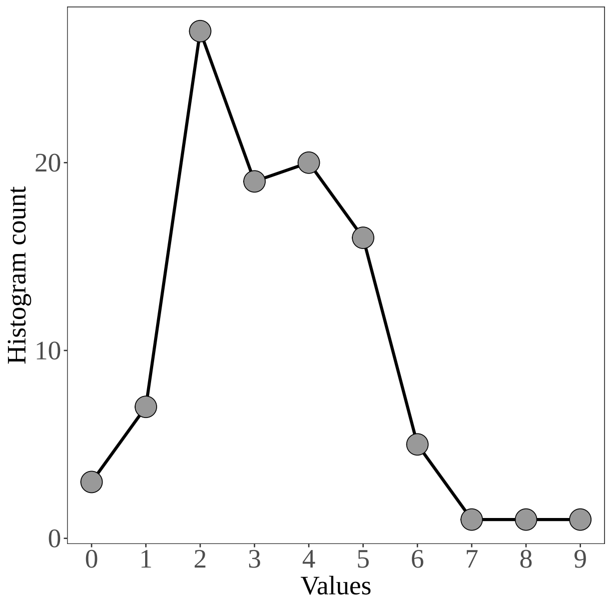
One of the features that Luria and Delbrück were interested in was simply the fraction of experiments in which nothing happened – meaning no mutant colonies formed on the agar plates after being exposed to viruses. This feature, the probability of zeros, can be used to infer a rate parameter. In the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

```
file = "poissdata.csv"
poissdata = scan(file, sep=",")
numberofzeros = sum(poissdata==0)
probzero = numberofzeros/(length(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.

```
lambda_estimator_zeros = function(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=sum(x==0)
  probzero=numberofzeros/length(x)
  lambda_est = -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 3 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 was $\lambda_{true} = 3$.

But you didn't know that in advance, did you? And that is the point.

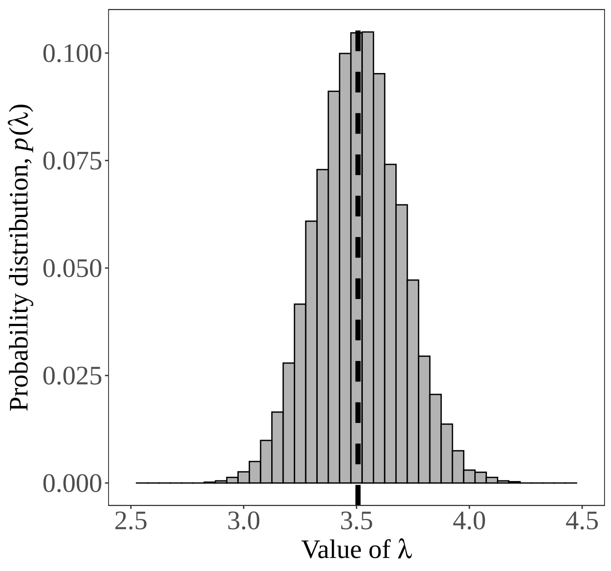
Indeed, Luria and Delbrück didn't know what the actual mutation rate was before the experiment (even if they had some ideas that it was small, and even some ideas on the level of smallness). In the case of any particular estimate, one may ask how confident is the estimate of the rate value? In other words, how often does sampling 100 points from a Poisson distribution with predetermined rate of λ_{set} lead to similar best estimates λ_{est}? One way to quantify similarity is asking 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 Poisson rate parameter it will be necessary to use a stable statistic – the mean value. Note that the expected value of the Poisson distribution is

$$\langle x \rangle = \sum_{k=0}^{\infty} kP(k) \tag{4}$$

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

```
numsamps = 10**4
lambdadist = rep(0,numsamps)
for(j in 1:numsamps){
  currdata = rpois(100,lambda_est)
  lambdadist[j] = mean(currdata) #The mean is the 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:

```
source("lambda_estimator_zeros.R")
require(ggplot2)
poissdata = scan("poissdata.csv", sep=",")
lambda_est = lambda_estimator_zeros(poissdata)

numsamps = 10**4
lambdadist = rep(0,numsamps)
for(j in 1:numsamps){
  currdata = rpois(100,lambda_est)
  lambdadist[j] = mean(currdata) #The mean is the best estimate of $\lambda$
}

ggplot(lambdadist_df,aes(lambdadist)) +
  geom_histogram(aes(y=..count../sum(..count..)),binwidth = .05,fill="gray") +
  xlim(c(2.5,4.5)) +
  xlab(expression(paste("Value of ", lambda))) +
  ylab(expression(paste("Probability distribution P(",lambda,")"))) +
  geom_vline(xintercept = lambda_est, linetype="dashed")
```

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 = sort(lambdadist)
lower025 = sortedlambdadist[floor(0.025*numsamps)]
upper975 = sortedlambdadist[floor(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 above analysis.

```
lambdavec = seq(0.5*lambda_est,1.5*lambda_est,0.01)
lower025 = rep(0,length(lambdavec))
upper975 = rep(0,length(lambdavec))
for(j in 1:length(lambdavec)){
  currlambdaset = lambdavec[j]
  poissfitdist = rep(0,numsamps)
  for(k in 1:numsamps){
    currdata = rpois(1000,currlambdaset)
    poissfitdist[k] = lambda_estimator_zeros(currdata)
  }
  sortedlambdadist = sort(poissfitdist)
  lower025[j] = sortedlambdadist[floor(0.025*numsamps)]
  upper975[j] = sortedlambdadist[floor(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} give observations.

VII. MODELING LURIA-DELBRÜCK EXPERIMENT

A central goal of this book is to help readers develop practical skills to quantitatively reason about living systems given uncertainty. However, each chapter is only part of this process (just like listening to lectures, paper discussions, and in-class work help solidify understanding). Moreover, for many readers, the mathematical and biological insights provide a partial guide. If seeing is believing, then coding and simulation will be a central path to build intuition and insight on the themes developed in this and subsequent chapters. The following homeworks operate in that spirit: help to leverage this ” in order to build intuition on the core ideas of Luria and Delbrück's seminal paper. The overall objective of these problems is to reproduce the ” of the number of resistant mutants, as observed by LD, and to begin to reach tentative conclusions regarding the confidence on estimated mutation rates and mechanisms in inferring the basis of mutation from resistant colony data. The problem set utilizes a common set of assumptions initially. That is, in these problems consider an experiment with C cultures each of which has N sensitive cells. Every time a cell divides there is a probability μ that one, and only one, of the daughter cells mutates to a resistant form. We will assume that the offspring of resistant cells are also resistant, i.e., there are no -mutations”. Good luck. And remember, you can do it!

BONUS Problem 1: Simulating the Luria-Delbruck Experiment over One Generation

Write a program to simulate just one generation of the LD experiment - stochastically. Simulate C = 500 cultures each of which has N = 1000 cells and $\mu = 10^{-3}$, i.e., a very high mutation rate. What is the distribution of resistant mutants that you observe across all the cultures? Are they similar or dissimilar to each other? Specify your measurement of c(m), i.e., the number of cultures with m resistant mutants. Is this distribution well fit by a Poisson distribution? If so, what is the best fit shape parameter of the Poisson density function and how does that relate to the microscopic value of mutation you used to generate the output? Finally, to what extent are the fluctuations ”large” or ”small”?

BONUS Problem 2: Simulating the Luria-Delbrück Experiment Forward One Generation at a Time

Extend the program in BONUS Problem 1 by setting C = 1000, N = 400 and $\lambda = 10^{-7}$, while having the population grow over $g = 15$ generations. What choice did you make with respect to modeling the population? If you decided to model each individual cell in each individual culture, explain your rationale? If you did so, instead,develop a model that represents the emergence of new resistant cells in each generation in each culture *en masse* (that is, all at once). *Hint: think about how prudent use of the Poisson random generation function could help. The objective here is to develop a working simulation, that is both accurate and efficient - in doing so, compare the speed when you use Poisson vs. Binomial random number generating functions*